The Mercury Library Reference Manual

Version 20.06.1

Table of Contents

The Mercury standard library contains a variety of modules which we hope may be of general usefulness. If you write a module that would be useful to others, and you would like us to include it as part of the Mercury standard library, please let us know.

The following documentation is simply the interface parts to those modules, automatically extracted from the source code. Some of the library modules are not very well documented; we apologize.

For many of the modules in the standard library, we have not yet had enough experience using them to be confident that the current interface is satisfactory; it is likely that the interfaces to many of the modules in the standard library will change somewhat in future releases of the Mercury system. Some modules are rather experimental modules that may even be removed in future releases. Of course, we wouldn't make changes gratuitously, but at the current time, preserving 100% backwards compatibility would be disadvantageous in the long run.

To help you protect yourself from depending on modules that are likely to change, each module has a comment "stability: low/medium/high" at the top which gives an indication of the likely stability of the interface to that module. For modules whose stability is "high", new functionality may be added to the interface, but we envisage very few if any changes to the interface of the sort that might break existing code. For modules whose stability is "medium", we expect that changes are more likely. For modules whose stability is "low", such changes are highly likely. If you want to minimize the possibility of your programs requiring modification to work with new releases of the Mercury system, we recommend that if possible you use only those modules whose stability is described as either "medium to high" or "high".

1 array

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 1993-1995, 1997-2012 The University of Melbourne.
% Copyright (C) 2013-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: array.m.
% Main authors: fjh, bromage.
% Stability: medium-low.
%
% This module provides dynamically-sized one-dimensional arrays.
% Array indices start at zero.
%
% WARNING!
%
% Arrays are currently not unique objects. until this situation is resolved,
% it is up to the programmer to ensure that arrays are used in ways that
% preserve correctness. In the absence of mode reordering, one should therefore
```

```
% assume that evaluation will take place in left-to-right order. For example,
% the following code will probably not work as expected (f is a function,
% A an array, I an index, and X an appropriate value):
%
%       Y = f(A ^ elem(I) := X, A ^ elem(I))
%
% The compiler is likely to compile this as
%
%       V0 = A ^ elem(I) := X,
%       V1 = A ^ elem(I),
%       Y  = f(V0, V1)
%
% and will be unaware that the first line should be ordered *after* the second.
% The safest thing to do is write things out by hand in the form
%
%       A0I = A0 ^ elem(I),
%       A1  = A0 ^ elem(I) := X,
%       Y   = f(A1, A0I)
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module array.
:- interface.

:- import_module list.
:- import_module maybe.
:- import_module pretty_printer.
:- import_module random.

:- type array(T).

:- inst array(I) == ground.
:- inst array == array(ground).

    % XXX the current Mercury compiler doesn't support 'ui' modes,
    % so to work-around that problem, we currently don't use
    % unique modes in this module.

% :- inst uniq_array(I) == unique.
% :- inst uniq_array == uniq_array(unique).
:- inst uniq_array(I) == array(I).          % XXX work-around
:- inst uniq_array == uniq_array(ground).   % XXX work-around

:- mode array_di == di(uniq_array).
:- mode array_uo == out(uniq_array).
:- mode array_ui == in(uniq_array).
```

```
% :- inst mostly_uniq_array(I) == mostly_unique).
% :- inst mostly_uniq_array == mostly_uniq_array(mostly_unique).
:- inst mostly_uniq_array(I) == array(I).    % XXX work-around
:- inst mostly_uniq_array == mostly_uniq_array(ground). % XXX work-around

:- mode array_mdi == mdi(mostly_uniq_array).
:- mode array_muo == out(mostly_uniq_array).
:- mode array_mui == in(mostly_uniq_array).

    % An 'index_out_of_bounds' is the exception thrown
    % on out-of-bounds array accesses. The string describes
    % the predicate or function reporting the error.
:- type index_out_of_bounds
    --->    index_out_of_bounds(string).

%-----------------------------------------------%

    % make_empty_array(Array) creates an array of size zero
    % starting at lower bound 0.
    %
:- pred make_empty_array(array(T)::array_uo) is det.

:- func make_empty_array = (array(T)::array_uo) is det.

    % init(Size, Init, Array) creates an array with bounds from 0
    % to Size-1, with each element initialized to Init.  Throws an
    % exception if Size < 0.
    %
:- pred init(int, T, array(T)).
:- mode init(in, in, array_uo) is det.

:- func init(int, T) = array(T).
:- mode init(in, in) = array_uo is det.

    % array/1 is a function that constructs an array from a list.
    % (It does the same thing as the predicate from_list/2.)
    % The syntax 'array([...])' is used to represent arrays
    % for io.read, io.write, term_to_type, and type_to_term.
    %
:- func array(list(T)) = array(T).
:- mode array(in) = array_uo is det.

    % generate(Size, Generate) = Array:
    % Create an array with bounds from 0 to Size - 1 using the function
    % Generate to set the initial value of each element of the array.
    % The initial value of the element at index K will be the result of
```

```
    % calling the function Generate(K). Throws an exception if Size < 0.
    %
:- func generate(int::in, (func(int) = T)::in) = (array(T)::array_uo)
    is det.

    % generate_foldl(Size, Generate, Array, !Acc):
    % As above, but using a predicate with an accumulator threaded through it
    % to generate the initial value of each element.
    %
:- pred generate_foldl(int, pred(int, T, A, A), array(T), A, A).
:- mode generate_foldl(in, in(pred(in, out, in, out) is det),
    array_uo, in, out) is det.
:- mode generate_foldl(in, in(pred(in, out, mdi, muo) is det),
    array_uo, mdi, muo) is det.
:- mode generate_foldl(in, in(pred(in, out, di, uo) is det),
    array_uo, di, uo) is det.
:- mode generate_foldl(in, in(pred(in, out, in, out) is semidet),
    array_uo, in, out) is semidet.
:- mode generate_foldl(in, in(pred(in, out, mdi, muo) is semidet),
    array_uo, mdi, muo) is semidet.
:- mode generate_foldl(in, in(pred(in, out, di, uo) is semidet),
    array_uo, di, uo) is semidet.

%-----------------------------------------------%

    % min returns the lower bound of the array.
    % Note: in this implementation, the lower bound is always zero.
    %
:- pred min(array(_T), int).
%:- mode min(array_ui, out) is det.
:- mode min(in, out) is det.

:- func min(array(_T)) = int.
%:- mode min(array_ui) = out is det.
:- mode min(in) = out is det.

    % least_index returns the lower bound of the array.
    % In future, it will be changed to behave like semidet_least_index,
    % failing on an empty array instead of returning an out-of-bounds index.
    %
:- pragma obsolete(least_index/1).
:- func least_index(array(T)) = int.
%:- mode least_index(array_ui) = out is det.
:- mode least_index(in) = out is det.

    % det_least_index returns the lower bound of the array.
    % Throws an exception if the array is empty.
```

```
    %
:- func det_least_index(array(T)) = int.
%:- mode det_least_index(array_ui) = out is det.
:- mode det_least_index(in) = out is det.

    % semidet_least_index returns the lower bound of the array,
    % or fails if the array is empty.
    %
:- func semidet_least_index(array(T)) = int.
%:- mode semidet_least_index(array_ui) = out is semidet.
:- mode semidet_least_index(in) = out is semidet.

    % max returns the upper bound of the array.
    % Returns lower bound - 1 for an empty array
    % (always -1 in this implementation).
    %
:- pred max(array(_T), int).
%:- mode max(array_ui, out) is det.
:- mode max(in, out) is det.

:- func max(array(_T)) = int.
%:- mode max(array_ui) = out is det.
:- mode max(in) = out is det.

    % greatest_index returns the upper bound of the array.
    % In future, it will be changed to behave like semidet_greatest_index,
    % failing on an empty array instead of returning an out-of-bounds index.
    %
:- pragma obsolete(greatest_index/1).
:- func greatest_index(array(T)) = int.
%:- mode greatest_index(array_ui) = out is det.
:- mode greatest_index(in) = out is det.

    % det_greatest_index returns the upper bound of the array.
    % Throws an exception if the array is empty.
    %
:- func det_greatest_index(array(T)) = int.
%:- mode det_greatest_index(array_ui) = out is det.
:- mode det_greatest_index(in) = out is det.

    % semidet_greatest_index returns the upper bound of the array,
    % or fails if the array is empty.
    %
:- func semidet_greatest_index(array(T)) = int.
%:- mode semidet_greatest_index(array_ui) = out is semidet.
:- mode semidet_greatest_index(in) = out is semidet.
```

```
    % size returns the length of the array,
    % i.e. upper bound - lower bound + 1.
    %
:- pred size(array(_T), int).
%:- mode size(array_ui, out) is det.
:- mode size(in, out) is det.

:- func size(array(_T)) = int.
%:- mode size(array_ui) = out is det.
:- mode size(in) = out is det.

    % bounds(Array, Min, Max) returns the lower and upper bounds of an array.
    % The upper bound will be lower bound - 1 for an empty array.
    % Note: in this implementation, the lower bound is always zero.
    %
:- pred bounds(array(_T), int, int).
%:- mode bounds(array_ui, out, out) is det.
:- mode bounds(in, out, out) is det.

    % in_bounds checks whether an index is in the bounds of an array.
    %
:- pred in_bounds(array(_T), int).
%:- mode in_bounds(array_ui, in) is semidet.
:- mode in_bounds(in, in) is semidet.

    % is_empty(Array):
    % True iff Array is an array of size zero.
    %
:- pred is_empty(array(_T)).
%:- mode is_empty(array_ui) is semidet.
:- mode is_empty(in) is semidet.

%-----------------------------------------------%

    % lookup returns the N'th element of an array.
    % Throws an exception if the index is out of bounds.
    %
:- pred lookup(array(T), int, T).
%:- mode lookup(array_ui, in, out) is det.
:- mode lookup(in, in, out) is det.

:- func lookup(array(T), int) = T.
%:- mode lookup(array_ui, in) = out is det.
:- mode lookup(in, in) = out is det.

    % semidet_lookup returns the N'th element of an array.
    % It fails if the index is out of bounds.
```

```
    %
:- pred semidet_lookup(array(T), int, T).
%:- mode semidet_lookup(array_ui, in, out) is semidet.
:- mode semidet_lookup(in, in, out) is semidet.

    % unsafe_lookup returns the N'th element of an array.
    % It is an error if the index is out of bounds.
    %
:- pred unsafe_lookup(array(T), int, T).
%:- mode unsafe_lookup(array_ui, in, out) is det.
:- mode unsafe_lookup(in, in, out) is det.

    % set sets the N'th element of an array, and returns the
    % resulting array (good opportunity for destructive update ;-).
    % Throws an exception if the index is out of bounds.
    %
:- pred set(int, T, array(T), array(T)).
:- mode set(in, in, array_di, array_uo) is det.

:- func set(array(T), int, T) = array(T).
:- mode set(array_di, in, in) = array_uo is det.

    % semidet_set sets the nth element of an array, and returns
    % the resulting array. It fails if the index is out of bounds.
    %
:- pred semidet_set(int, T, array(T), array(T)).
:- mode semidet_set(in, in, array_di, array_uo) is semidet.

    % unsafe_set sets the nth element of an array, and returns the
    % resulting array. It is an error if the index is out of bounds.
    %
:- pred unsafe_set(int, T, array(T), array(T)).
:- mode unsafe_set(in, in, array_di, array_uo) is det.

    % slow_set sets the nth element of an array, and returns the
    % resulting array. The initial array is not required to be unique,
    % so the implementation may not be able to use destructive update.
    % It is an error if the index is out of bounds.
    %
:- pred slow_set(int, T, array(T), array(T)).
%:- mode slow_set(in, in, array_ui, array_uo) is det.
:- mode slow_set(in, in, in, array_uo) is det.

:- func slow_set(array(T), int, T) = array(T).
%:- mode slow_set(array_ui, in, in) = array_uo is det.
:- mode slow_set(in, in, in) = array_uo is det.
```

```
    % semidet_slow_set sets the nth element of an array, and returns
    % the resulting array. The initial array is not required to be unique,
    % so the implementation may not be able to use destructive update.
    % It fails if the index is out of bounds.
    %
:- pred semidet_slow_set(int, T, array(T), array(T)).
%:- mode semidet_slow_set(in, in, array_ui, array_uo) is semidet.
:- mode semidet_slow_set(in, in, in, array_uo) is semidet.

    % Field selection for arrays.
    % Array ^ elem(Index) = lookup(Array, Index).
    %
:- func elem(int, array(T)) = T.
%:- mode elem(in, array_ui) = out is det.
:- mode elem(in, in) = out is det.

    % As above, but omit the bounds check.
    %
:- func unsafe_elem(int, array(T)) = T.
%:- mode unsafe_elem(in, array_ui) = out is det.
:- mode unsafe_elem(in, in) = out is det.

    % Field update for arrays.
    % (Array ^ elem(Index) := Value) = set(Array, Index, Value).
    %
:- func 'elem :='(int, array(T), T) = array(T).
:- mode 'elem :='(in, array_di, in) = array_uo is det.

    % As above, but omit the bounds check.
    %
:- func 'unsafe_elem :='(int, array(T), T) = array(T).
:- mode 'unsafe_elem :='(in, array_di, in) = array_uo is det.

    % swap(I, J, !Array):
    % Swap the item in the I'th position with the item in the J'th position.
    % Throws an exception if either of I or J is out-of-bounds.
    %
:- pred swap(int, int, array(T), array(T)).
:- mode swap(in, in, array_di, array_uo) is det.

    % As above, but omit the bounds checks.
    %
:- pred unsafe_swap(int, int, array(T), array(T)).
:- mode unsafe_swap(in, in, array_di, array_uo) is det.

    % Returns every element of the array, one by one.
    %
```

```
:- pred member(array(T)::in, T::out) is nondet.

%-----------------------------------------------%

    % copy(Array0, Array):
    % Makes a new unique copy of an array.
    %
:- pred copy(array(T), array(T)).
%:- mode copy(array_ui, array_uo) is det.
:- mode copy(in, array_uo) is det.

:- func copy(array(T)) = array(T).
%:- mode copy(array_ui) = array_uo is det.
:- mode copy(in) = array_uo is det.

    % resize(Size, Init, Array0, Array):
    % The array is expanded or shrunk to make it fit the new size `Size'.
    % Any new entries are filled with `Init'. Throws an exception if
    % `Size' < 0.
    %
:- pred resize(int, T, array(T), array(T)).
:- mode resize(in, in, array_di, array_uo) is det.

    % resize(Array0, Size, Init) = Array:
    % The array is expanded or shrunk to make it fit the new size `Size'.
    % Any new entries are filled with `Init'. Throws an exception if
    % `Size' < 0.
    %
:- func resize(array(T), int, T) = array(T).
:- mode resize(array_di, in, in) = array_uo is det.

    % shrink(Size, Array0, Array):
    % The array is shrunk to make it fit the new size `Size'.
    % Throws an exception if `Size' is larger than the size of `Array0' or
    % if `Size' < 0.
    %
:- pred shrink(int, array(T), array(T)).
:- mode shrink(in, array_di, array_uo) is det.

    % shrink(Array0, Size) = Array:
    % The array is shrunk to make it fit the new size `Size'.
    % Throws an exception if `Size' is larger than the size of `Array0' or
    % if `Size' < 0.
    %
:- func shrink(array(T), int) = array(T).
:- mode shrink(array_di, in) = array_uo is det.
```

```
    % fill(Item, Array0, Array):
    % Sets every element of the array to 'Elem'.
    %
:- pred fill(T::in, array(T)::array_di, array(T)::array_uo) is det.

    % fill_range(Item, Lo, Hi, !Array):
    % Sets every element of the array with index in the range Lo..Hi
    % (inclusive) to Item. Throws a software_error/1 exception if Lo > Hi.
    % Throws an index_out_of_bounds/0 exception if Lo or Hi is out of bounds.
    %
:- pred fill_range(T::in, int::in, int::in,
     array(T)::array_di, array(T)::array_uo) is det.

    % from_list takes a list, and returns an array containing those
    % elements in the same order that they occurred in the list.
    %
:- func from_list(list(T)::in) = (array(T)::array_uo) is det.
:- pred from_list(list(T)::in, array(T)::array_uo) is det.

    % from_reverse_list takes a list, and returns an array containing
    % those elements in the reverse order that they occurred in the list.
    %
:- func from_reverse_list(list(T)::in) = (array(T)::array_uo) is det.

    % to_list takes an array and returns a list containing the elements
    % of the array in the same order that they occurred in the array.
    %
:- pred to_list(array(T), list(T)).
%:- mode to_list(array_ui, out) is det.
:- mode to_list(in, out) is det.

:- func to_list(array(T)) = list(T).
%:- mode to_list(array_ui) = out is det.
:- mode to_list(in) = out is det.

    % fetch_items(Array, Lo, Hi, List):
    % Returns a list containing the items in the array with index in the range
    % Lo..Hi (both inclusive) in the same order that they occurred in the
    % array. Returns an empty list if Hi < Lo. Throws an index_out_of_bounds/0
    % exception if either Lo or Hi is out of bounds, *and* Hi >= Lo.
    %
    % If Hi < Lo, we do not generate an exception even if either or both
    % are out of bounds, for two reasons. First, there is no need; if Hi < Lo,
    % we can return the empty list without accessing any element of the array.
    % Second, without this rule, some programming techniques for accessing
    % consecutive contiguous regions of an array would require explicit
    % bound checks in the *caller* of fetch_items, which would duplicate
```

```
    % the checks inside fetch_items itself.
    %
:- pred fetch_items(array(T), int, int, list(T)).
:- mode fetch_items(in, in, in, out) is det.

:- func fetch_items(array(T), int, int) = list(T).
%:- mode fetch_items(array_ui, in, in) = out is det.
:- mode fetch_items(in, in, in) = out is det.

    % XXX We prefer users to call the new binary_search predicate
    % instead of bsearch, which is why bsearch is marked as obsolete.
    %
    % bsearch takes an array, an element to be matched and a comparison
    % predicate and returns the position of the first occurrence in the array
    % of an element which is equivalent to the given one in the ordering
    % provided. Assumes the array is sorted according to this ordering.
    %
:- pred bsearch(array(T), T, comparison_pred(T), maybe(int)).
%:- mode bsearch(array_ui, in, in(comparison_pred), out) is det.
:- mode bsearch(in, in, in(comparison_pred), out) is det.
:- pragma obsolete(bsearch/4).

:- func bsearch(array(T), T, comparison_func(T)) = maybe(int).
%:- mode bsearch(array_ui, in, in(comparison_func)) = out is det.
:- mode bsearch(in, in, in(comparison_func)) = out is det.
:- pragma obsolete(bsearch/3).

    % binary_search(A, X, I) does a binary search for the element X
    % in the array A. If there is an element with that value in the array,
    % it returns its index I; otherwise, it fails.
    %
    % The array A must be sorted into ascending order with respect to the
    % the builtin Mercury order on terms for binary_search/3, and with respect
    % to supplied comparison predicate for binary_search/4.
    %
    % The array may contain duplicates. If it does, and a search looks for
    % a duplicated value, the search will return the index of one of the
    % copies, but it is not specified *which* copy's index it will return.
    %
:- pred binary_search(array(T)::array_ui,
    T::in, int::out) is semidet.
:- pred binary_search(comparison_func(T)::in, array(T)::array_ui,
    T::in, int::out) is semidet.

    % approx_binary_search(A, X, I) does a binary search for the element X
    % in the array A. If there is an element with that value in the array,
    % it returns its index I. If there is no element with that value in the
```

```
    % array, it returns an index whose slot contains the highest value in the
    % array that is less than X, as measured by the builtin Mercury order
    % on terms for approx_binary_search/3, and as measured by the supplied
    % ordering for approx_binary_search/4. It will fail only if there is
    % no value smaller than X in the array.
    %
    % The array A must be sorted into ascending order with respect to the
    % the builtin Mercury order on terms for approx_binary_search/3, and
    % with respect to supplied comparison predicate for approx_binary_search/4.
    %
    % The array may contain duplicates. If it does, and if either the
    % searched-for value or (if that does not exist) the highest value
    % smaller than the searched-for value is duplicated, the search will return
    % the index of one of the copies, but it is not specified *which* copy's
    % index it will return.
    %
:- pred approx_binary_search(array(T)::array_ui,
    T::in, int::out) is semidet.
:- pred approx_binary_search(comparison_func(T)::in, array(T)::array_ui,
    T::in, int::out) is semidet.

    % map(Closure, OldArray, NewArray) applies `Closure' to
    % each of the elements of `OldArray' to create `NewArray'.
    %
:- pred map(pred(T1, T2), array(T1), array(T2)).
%:- mode map(pred(in, out) is det, array_ui, array_uo) is det.
:- mode map(pred(in, out) is det, in, array_uo) is det.

:- func map(func(T1) = T2, array(T1)) = array(T2).
%:- mode map(func(in) = out is det, array_ui) = array_uo is det.
:- mode map(func(in) = out is det, in) = array_uo is det.

:- func array_compare(array(T), array(T)) = comparison_result.
:- mode array_compare(in, in) = uo is det.

    % sort(Array) returns a version of Array sorted into ascending
    % order.
    %
    % This sort is not stable. That is, elements that compare/3 decides are
    % equal will appear together in the sorted array, but not necessarily
    % in the same order in which they occurred in the input array. This is
    % primarily only an issue with types with user-defined equivalence for
    % which `equivalent' objects are otherwise distinguishable.
    %
:- func sort(array(T)) = array(T).
:- mode sort(array_di) = array_uo is det.
```

```
    % array.sort was previously buggy. This symbol provides a way to ensure
    % that you are using the fixed version.
    %
:- pred array.sort_fix_2014 is det.

    % foldl(Fn, Array, X) is equivalent to
    %   list.foldl(Fn, to_list(Array), X)
    % but more efficient.
    %
:- func foldl(func(T1, T2) = T2, array(T1), T2) = T2.
%:- mode foldl(func(in, in) = out is det, array_ui, in) = out is det.
:- mode foldl(func(in, in) = out is det, in, in) = out is det.
%:- mode foldl(func(in, di) = uo is det, array_ui, di) = uo is det.
:- mode foldl(func(in, di) = uo is det, in, di) = uo is det.

    % foldl(Pr, Array, !X) is equivalent to
    %   list.foldl(Pr, to_list(Array), !X)
    % but more efficient.
    %
:- pred foldl(pred(T1, T2, T2), array(T1), T2, T2).
:- mode foldl(pred(in, in, out) is det, in, in, out) is det.
:- mode foldl(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode foldl(pred(in, di, uo) is det, in, di, uo) is det.
:- mode foldl(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode foldl(pred(in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode foldl(pred(in, di, uo) is semidet, in, di, uo) is semidet.

    % foldl2(Pr, Array, !X, !Y) is equivalent to
    %   list.foldl2(Pr, to_list(Array), !X, !Y)
    % but more efficient.
    %
:- pred foldl2(pred(T1, T2, T2, T3, T3), array(T1), T2, T2, T3, T3).
:- mode foldl2(pred(in, in, out, in, out) is det, in, in, out, in, out)
    is det.
:- mode foldl2(pred(in, in, out, mdi, muo) is det, in, in, out, mdi, muo)
    is det.
:- mode foldl2(pred(in, in, out, di, uo) is det, in, in, out, di, uo)
    is det.
:- mode foldl2(pred(in, in, out, in, out) is semidet, in,
    in, out, in, out) is semidet.
:- mode foldl2(pred(in, in, out, mdi, muo) is semidet, in,
    in, out, mdi, muo) is semidet.
:- mode foldl2(pred(in, in, out, di, uo) is semidet, in,
    in, out, di, uo) is semidet.

    % As above, but with three accumulators.
    %
```

```
:- pred foldl3(pred(T1, T2, T2, T3, T3, T4, T4), array(T1),
    T2, T2, T3, T3, T4, T4).
:- mode foldl3(pred(in, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out) is det.
:- mode foldl3(pred(in, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, mdi, muo) is det.
:- mode foldl3(pred(in, in, out, in, out, di, uo) is det,
    in, in, out, in, out, di, uo) is det.
:- mode foldl3(pred(in, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out) is semidet.
:- mode foldl3(pred(in, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, mdi, muo) is semidet.
:- mode foldl3(pred(in, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, di, uo) is semidet.

    % As above, but with four accumulators.
    %
:- pred foldl4(pred(T1, T2, T2, T3, T3, T4, T4, T5, T5), array(T1),
    T2, T2, T3, T3, T4, T4, T5, T5).
:- mode foldl4(pred(in, in, out, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out, in, out) is det.
:- mode foldl4(pred(in, in, out, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, in, out, mdi, muo) is det.
:- mode foldl4(pred(in, in, out, in, out, in, out, di, uo) is det,
    in, in, out, in, out, in, out, di, uo) is det.
:- mode foldl4(pred(in, in, out, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out, in, out) is semidet.
:- mode foldl4(pred(in, in, out, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode foldl4(pred(in, in, out, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, in, out, di, uo) is semidet.

    % As above, but with five accumulators.
    %
:- pred foldl5(pred(T1, T2, T2, T3, T3, T4, T4, T5, T5, T6, T6),
    array(T1), T2, T2, T3, T3, T4, T4, T5, T5, T6, T6).
:- mode foldl5(
    pred(in, in, out, in, out, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out, in, out, in, out) is det.
:- mode foldl5(
    pred(in, in, out, in, out, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, in, out, in, out, mdi, muo) is det.
:- mode foldl5(
    pred(in, in, out, in, out, in, out, in, out, di, uo) is det,
    in, in, out, in, out, in, out, in, out, di, uo) is det.
:- mode foldl5(
    pred(in, in, out, in, out, in, out, in, out, in, out) is semidet,
```

```
    in, in, out, in, out, in, out, in, out, in, out) is semidet.
:- mode foldl5(
    pred(in, in, out, in, out, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode foldl5(
    pred(in, in, out, in, out, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, in, out, in, out, di, uo) is semidet.

%-----------------------------------------------%

    % foldr(Fn, Array, X) is equivalent to
    %   list.foldr(Fn, to_list(Array), X)
    % but more efficient.
    %
:- func foldr(func(T1, T2) = T2, array(T1), T2) = T2.
%:- mode foldr(func(in, in) = out is det, array_ui, in) = out is det.
:- mode foldr(func(in, in) = out is det, in, in) = out is det.
%:- mode foldr(func(in, di) = uo is det, array_ui, di) = uo is det.
:- mode foldr(func(in, di) = uo is det, in, di) = uo is det.

    % foldr(P, Array, !Acc) is equivalent to
    %   list.foldr(P, to_list(Array), !Acc)
    % but more efficient.
    %
:- pred foldr(pred(T1, T2, T2), array(T1), T2, T2).
:- mode foldr(pred(in, in, out) is det, in, in, out) is det.
:- mode foldr(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode foldr(pred(in, di, uo) is det, in, di, uo) is det.
:- mode foldr(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode foldr(pred(in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode foldr(pred(in, di, uo) is semidet, in, di, uo) is semidet.

    % As above, but with two accumulators.
    %
:- pred foldr2(pred(T1, T2, T2, T3, T3), array(T1), T2, T2, T3, T3).
:- mode foldr2(pred(in, in, out, in, out) is det, in, in, out, in, out)
    is det.
:- mode foldr2(pred(in, in, out, mdi, muo) is det, in, in, out, mdi, muo)
    is det.
:- mode foldr2(pred(in, in, out, di, uo) is det, in, in, out, di, uo)
    is det.
:- mode foldr2(pred(in, in, out, in, out) is semidet, in,
    in, out, in, out) is semidet.
:- mode foldr2(pred(in, in, out, mdi, muo) is semidet, in,
    in, out, mdi, muo) is semidet.
:- mode foldr2(pred(in, in, out, di, uo) is semidet, in,
    in, out, di, uo) is semidet.
```

```
    % As above, but with three accumulators.
    %
:- pred foldr3(pred(T1, T2, T2, T3, T3, T4, T4), array(T1),
    T2, T2, T3, T3, T4, T4).
:- mode foldr3(pred(in, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out) is det.
:- mode foldr3(pred(in, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, mdi, muo) is det.
:- mode foldr3(pred(in, in, out, in, out, di, uo) is det, in,
    in, out, in, out, di, uo) is det.
:- mode foldr3(pred(in, in, out, in, out, in, out) is semidet, in,
    in, out, in, out, in, out) is semidet.
:- mode foldr3(pred(in, in, out, in, out, mdi, muo) is semidet, in,
    in, out, in, out, mdi, muo) is semidet.
:- mode foldr3(pred(in, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, di, uo) is semidet.

    % As above, but with four accumulators.
    %
:- pred foldr4(pred(T1, T2, T2, T3, T3, T4, T4, T5, T5), array(T1),
    T2, T2, T3, T3, T4, T4, T5, T5).
:- mode foldr4(pred(in, in, out, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out, in, out) is det.
:- mode foldr4(pred(in, in, out, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, in, out, mdi, muo) is det.
:- mode foldr4(pred(in, in, out, in, out, in, out, di, uo) is det,
    in, in, out, in, out, in, out, di, uo) is det.
:- mode foldr4(pred(in, in, out, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out, in, out) is semidet.
:- mode foldr4(pred(in, in, out, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode foldr4(pred(in, in, out, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, in, out, di, uo) is semidet.

    % As above, but with five accumulators.
    %
:- pred foldr5(pred(T1, T2, T2, T3, T3, T4, T4, T5, T5, T6, T6),
    array(T1), T2, T2, T3, T3, T4, T4, T5, T5, T6, T6).
:- mode foldr5(
    pred(in, in, out, in, out, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out, in, out, in, out) is det.
:- mode foldr5(
    pred(in, in, out, in, out, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, in, out, in, out, mdi, muo) is det.
:- mode foldr5(
    pred(in, in, out, in, out, in, out, in, out, di, uo) is det,
```

```
    in, in, out, in, out, in, out, in, out, di, uo) is det.
:- mode foldr5(
    pred(in, in, out, in, out, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out, in, out, in, out) is semidet.
:- mode foldr5(
    pred(in, in, out, in, out, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode foldr5(
    pred(in, in, out, in, out, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, in, out, in, out, di, uo) is semidet.

%---------------------------------------------------%

    % foldl_corresponding(P, A, B, !Acc):
    %
    % Does the same job as foldl, but works on two arrays in parallel.
    % Throws an exception if the array arguments differ in size.
    %
:- pred foldl_corresponding(pred(T1, T2, T3, T3), array(T1), array(T2),
    T3, T3).
:- mode foldl_corresponding(in(pred(in, in, in, out) is det), in, in,
    in, out) is det.
:- mode foldl_corresponding(in(pred(in, in, mdi, muo) is det), in, in,
    mdi, muo) is det.
:- mode foldl_corresponding(in(pred(in, in, di, uo) is det), in, in,
    di, uo) is det.
:- mode foldl_corresponding(in(pred(in, in, in, out) is semidet), in, in,
    in, out) is semidet.
:- mode foldl_corresponding(in(pred(in, in, mdi, muo) is semidet), in, in,
    mdi, muo) is semidet.
:- mode foldl_corresponding(in(pred(in, in, di, uo) is semidet), in, in,
    di, uo) is semidet.

    % As above, but with two accumulators.
    %
:- pred foldl2_corresponding(pred(T1, T2, T3, T3, T4, T4),
    array(T1), array(T2), T3, T3, T4, T4).
:- mode foldl2_corresponding(in(pred(in, in, in, out, in, out) is det),
    in, in, in, out, in, out) is det.
:- mode foldl2_corresponding(in(pred(in, in, in, out, mdi, muo) is det),
    in, in, in, out, mdi, muo) is det.
:- mode foldl2_corresponding(in(pred(in, in, in, out, di, uo) is det),
    in, in, in, out, di, uo) is det.
:- mode foldl2_corresponding(in(pred(in, in, in, out, in, out) is semidet),
    in, in, in, out, in, out) is semidet.
:- mode foldl2_corresponding(in(pred(in, in, in, out, mdi, muo) is semidet),
    in, in, in, out, mdi, muo) is semidet.
```

```
:- mode foldl2_corresponding(in(pred(in, in, in, out,  di, uo) is semidet),
    in, in, in, out, di, uo) is semidet.

%---------------------------------------------------%

    % map_foldl(P, A, B, !Acc):
    % Invoke P(Aelt, Belt, !Acc) on each element of the A array,
    % and construct array B from the resulting values of Belt.
    %
:- pred map_foldl(pred(T1, T2, T3, T3), array(T1), array(T2), T3, T3).
:- mode map_foldl(in(pred(in, out, in, out) is det),
    in, array_uo, in, out) is det.
:- mode map_foldl(in(pred(in, out, mdi, muo) is det),
    in, array_uo, mdi, muo) is det.
:- mode map_foldl(in(pred(in, out, di, uo) is det),
    in, array_uo, di, uo) is det.
:- mode map_foldl(in(pred(in, out, in, out) is semidet),
    in, array_uo, in, out) is semidet.

%---------------------------------------------------%

    % map_corresponding_foldl(P, A, B, C, !Acc):
    %
    % Given two arrays A and B, invoke P(Aelt, Belt, Celt, !Acc) on
    % each corresponding pair of elements Aelt and Belt. Build up the ar-
ray C
    % from the result Celt values. Return C and the final value of the
    % accumulator.
    %
    % Throws an exception if A and B differ in size.
    %
:- pred map_corresponding_foldl(pred(T1, T2, T3, T4, T4),
    array(T1), array(T2), array(T3), T4, T4).
:- mode map_corresponding_foldl(
    in(pred(in, in, out, in, out) is det),
    in, in, array_uo, in, out) is det.
:- mode map_corresponding_foldl(
    in(pred(in, in, out, mdi, muo) is det),
    in, in, array_uo, mdi, muo) is det.
:- mode map_corresponding_foldl(
    in(pred(in, in, out, di, uo) is det),
    in, in, array_uo, di, uo) is det.
:- mode map_corresponding_foldl(
    in(pred(in, in, out, in, out) is semidet),
    in, in, array_uo, in, out) is semidet.
:- mode map_corresponding_foldl(
    in(pred(in, in, out, mdi, muo) is semidet),
```

```
    in, in, array_uo, mdi, muo) is semidet.
:- mode map_corresponding_foldl(
    in(pred(in, in, out, di, uo) is semidet),
    in, in, array_uo, di, uo) is semidet.

%---------------------------------------------------%

    % all_true(Pred, Array):
    % True iff Pred is true for every element of Array.
    %
:- pred all_true(pred(T), array(T)).
%:- mode all_true(in(pred(in) is semidet), array_ui) is semidet.
:- mode all_true(in(pred(in) is semidet), in) is semidet.

    % all_false(Pred, Array):
    % True iff Pred is false for every element of Array.
    %
:- pred all_false(pred(T), array(T)).
%:- mode all_false(in(pred(in) is semidet), array_ui) is semidet.
:- mode all_false(in(pred(in) is semidet), in) is semidet.

    % append(A, B) = C:
    %
    % Make C a concatenation of the arrays A and B.
    %
:- func append(array(T)::in, array(T)::in) = (array(T)::array_uo) is det.

    % random_permutation(A0, A, RS0, RS) permutes the elements in
    % A0 given random seed RS0 and returns the permuted array in A
    % and the next random seed in RS.
    %
:- pred random_permutation(array(T)::array_di, array(T)::array_uo,
    random.supply::mdi, random.supply::muo) is det.

    % Convert an array to a pretty_printer.doc for formatting.
    %
:- func array_to_doc(array(T)) = pretty_printer.doc.
:- mode array_to_doc(array_ui) = out is det.

%---------------------------------------------------%
%---------------------------------------------------%
```

2 array2d

```
%---------------------------------------------------%
```

```
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 2003, 2005-2007, 2011-2012 The University of Melbourne.
% Copyright (C) 2013-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: array2d.m.
% Author: Ralph Becket <rafe@cs.mu.oz.au>.
% Stability: medium-low.
%
% Two-dimensional rectangular (i.e. not ragged) array ADT.
%
% XXX The same caveats re: uniqueness of arrays apply to array2ds.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module array2d.
:- interface.

:- import_module array.
:- import_module list.

%-----------------------------------------------------%

    % An array2d is a two-dimensional array stored in row-major order
    % (that is, the elements of the first row in left-to-right
    % order, followed by the elements of the second row and so forth.)
    %
:- type array2d(T).

:- inst array2d for array2d/1
    --->    array2d(ground, ground, array).

    % XXX These are work-arounds until we get nested uniqueness working.
    %
:- mode array2d_di == di(array2d).
:- mode array2d_ui == in(array2d).
:- mode array2d_uo == out(array2d).

    % init(M, N, X) = array2d([[X11, ..., X1N], ..., [XM1, ..., XMN]])
    % where each XIJ = X.  An exception is thrown if M < 0 or N < 0.
    %
:- func init(int, int, T) = array2d(T).
:- mode init(in, in, in) = array2d_uo is det.
```

```
    % array2d([[X11, ..., X1N], ..., [XM1, ..., XMN]]) constructs an array2d
    % of size M * N, with the special case that bounds(array2d([]), 0, 0).
    %
    % An exception is thrown if the sublists are not all the same length.
    %
:- func array2d(list(list(T))) = array2d(T).
:- mode array2d(in) = array2d_uo is det.

    % A synonym for the above.
    %
:- func from_lists(list(list(T))) = array2d(T).
:- mode from_lists(in) = array2d_uo is det.

    % from_array(M, N, Array) constructs an array2d of size M * N where the
    % elements are taken from Array in row-major order, i.e. the element at row
    % I column J is taken from Array at index (I * N + J). Indices start from
    % zero. Throws an exception if M < 0 or N < 0, or if the number of elements
    % in Array does not equal M * N.
    %
:- func from_array(int, int, array(T)) = array2d(T).
:- mode from_array(in, in, array_di) = array2d_uo is det.

    % is_empty(Array):
    % True iff Array contains zero elements.
    %
:- pred is_empty(array2d(T)).
%:- mode is_empty(array2d_ui) is semidet.
:- mode is_empty(in) is semidet.

    % bounds(array2d([[X11, ..., X1N], ..., [XM1, ..., XMN]), M, N)
    %
:- pred bounds(array2d(T), int, int).
%:- mode bounds(array2d_ui, out, out) is det.
:- mode bounds(in,       out, out) is det.

    % in_bounds(array2d([[X11, ..., X1N], ..., [XM1, ..., XMN]), I, J)
    % succeeds iff 0 =< I < M, 0 =< J < N.
    %
:- pred in_bounds(array2d(T), int, int).
%:- mode in_bounds(array2d_ui, in,  in ) is semidet.
:- mode in_bounds(in,       in,  in ) is semidet.

    % array2d([[X11, ..., X1N], ..., [XM1, ..., XMN]]) ^ elem(I, J) = X
    % where X is the J+1'th element of the I+1'th row (that is, indices
    % start from zero.)
    %
    % An exception is thrown unless 0 =< I < M, 0 =< J < N.
```

```
    %
:- func array2d(T) ^ elem(int, int) = T.
%:- mode array2d_ui ^ elem(in,  in ) = out is det.
:- mode in       ^ elem(in,  in ) = out is det.

    % T ^ unsafe_elem(I, J) is the same as T ^ elem(I, J) except that
    % behaviour is undefined if not in_bounds(T, I, J).
    %
:- func array2d(T) ^ unsafe_elem(int, int) = T.
%:- mode array2d_ui ^ unsafe_elem(in,  in ) = out is det.
:- mode in       ^ unsafe_elem(in,  in ) = out is det.

    % ( T0 ^ elem(I, J) := X ) = T
    % where T ^ elem(II, JJ) = X                 if I = II, J = JJ
    % and   T ^ elem(II, JJ) = T0 ^ elem(II, JJ) otherwise.
    %
    % An exception is thrown unless 0 =< I < M, 0 =< J < N.
    %
:- func ( array2d(T) ^ elem(int, int) := T  ) = array2d(T).
:- mode ( array2d_di ^ elem(in,  in)  := in ) = array2d_uo is det.

    % Pred version of the above.
    %
:- pred set(int, int, T,  array2d(T), array2d(T)).
:- mode set(in,  in,  in, array2d_di, array2d_uo) is det.

    % T ^ unsafe_elem(I, J) := X is the same as T ^ elem(I, J) := X except
    % that behaviour is undefined if not in_bounds(T, I, J).
    %
:- func ( array2d(T) ^ unsafe_elem(int, int) := T  ) = array2d(T).
:- mode ( array2d_di ^ unsafe_elem(in,  in)  := in ) = array2d_uo is det.

    % Pred version of the above.
    %
:- pred unsafe_set(int, int, T,  array2d(T), array2d(T)).
:- mode unsafe_set(in,  in,  in, array2d_di, array2d_uo) is det.

    % lists(array2d([[X11, ..., X1N], ..., [XM1, ..., XMN])) =
    %     [[X11, ..., X1N], ..., [XM1, ..., XMN]]
    %
:- func lists(array2d(T)) = list(list(T)).
%:- mode lists(array2d_ui) = out is det.
:- mode lists(in       ) = out is det.

    % fill(Item, !Array2d):
    % Sets every element of the array to Item.
    %
```

```
:- pred fill(T::in, array2d(T)::array2d_di, array2d(T)::array2d_uo) is det.

%---------------------------------------------------%
%---------------------------------------------------%
```

3 assoc_list

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 1995-1997, 1999-2001, 2004-2006, 2010-2011 The University of Melbourne
% Copyright (C) 2013-2020 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: assoc_list.m.
% Main authors: fjh, zs.
% Stability: medium to high.
%
% This file contains the definition of the type assoc_list(K, V)
% and some predicates which operate on those types.
%
%---------------------------------------------------%
%---------------------------------------------------%

:- module assoc_list.
:- interface.

:- import_module list.
:- import_module pair.

%---------------------------------------------------%

:- type assoc_list(K, V) == list(pair(K, V)).

:- type assoc_list(T) == list(pair(T, T)).

%---------------------------------------------------%
%
% Creating assoc_lists from lists of keys and values.
%

    % Zip together a list of keys and a list of values.
    % Throw an exception if they are of different lengths.
    %
```

```
:- func from_corresponding_lists(list(K), list(V)) = assoc_list(K, V).
:- pred from_corresponding_lists(list(K)::in, list(V)::in,
    assoc_list(K, V)::out) is det.

%-----------------------------------------------------%
%
% Operations on lists of keys and/or values.
%

    % Swap the two sides of the pairs in each member of the list.
    %
:- func reverse_members(assoc_list(K, V)) = assoc_list(V, K).
:- pred reverse_members(assoc_list(K, V)::in, assoc_list(V, K)::out) is det.

    % Return the first member of each pair.
    %
:- func keys(assoc_list(K, V)) = list(K).
:- pred keys(assoc_list(K, V)::in, list(K)::out) is det.

    % Return the second member of each pair.
    %
:- func values(assoc_list(K, V)) = list(V).
:- pred values(assoc_list(K, V)::in, list(V)::out) is det.

    % Return two lists containing respectively the first and the second member
    % of each pair in the assoc_list.
    %
:- pred keys_and_values(assoc_list(K, V)::in,
    list(K)::out, list(V)::out) is det.

%-----------------------------------------------------%
%
% Searching assoc_lists.
%

    % Find the first element of the association list that matches
    % the given key, and return the associated value.
    % Fail if there is no matching key.
    %
:- pred search(assoc_list(K, V)::in, K::in, V::out) is semidet.

    % Find the first element of the association list that matches
    % the given key, and return the associated value.
    % Throw an exception if there is no matching key.
    %
:- pred lookup(assoc_list(K, V)::in, K::in, V::out) is det.
```

```
    % A field access version of search.
    %
:- func assoc_list(K, V) ^ elem(K) = V is semidet.

    % A field access version of lookup.
    %
:- func assoc_list(K, V) ^ det_elem(K) = V is det.

%-----------------------------------------------------%
%
% Updating elements in assoc_lists.
%

    % Find the first element of the assoc_list list that matches
    % the given key, and update the associated value.
    % Fail if there is no matching key.
    %
:- pred update(K::in, V::in, assoc_list(K, V)::in, assoc_list(K, V)::out)
    is semidet.

%-----------------------------------------------------%
%
% Removing elements from assoc_lists.
%

    % Find the first element of the association list that matches
    % the given key. Return the associated value, and the original list
    % with the selected element removed.
    %
:- pred remove(assoc_list(K, V)::in, K::in, V::out, assoc_list(K, V)::out)
    is semidet.

    % As above, but with an argument ordering that is more conducive to
    % the use of state variable notation.
    %
:- pred svremove(K::in, V::out, assoc_list(K, V)::in, assoc_list(K, V)::out)
    is semidet.

%-----------------------------------------------------%
%
% Mapping keys or values.
%

:- func map_keys_only(func(K) = L, assoc_list(K, V)) = assoc_list(L, V).
:- pred map_keys_only(pred(K, L), assoc_list(K, V), assoc_list(L, V)).
:- mode map_keys_only(pred(in, out) is det, in, out) is det.
```

```
:- func map_values_only(func(V) = W, assoc_list(K, V)) = assoc_list(K, W).
:- pred map_values_only(pred(V, W), assoc_list(K, V), assoc_list(K, W)).
:- mode map_values_only(pred(in, out) is det, in, out) is det.

:- func map_values(func(K, V) = W, assoc_list(K, V)) = assoc_list(K, W).
:- pred map_values(pred(K, V, W), assoc_list(K, V), assoc_list(K, W)).
:- mode map_values(pred(in, in, out) is det, in, out) is det.

%-----------------------------------------------------%
%
% Filtering elements in assoc_lists.
%

    % filter(Pred, List, TrueList) takes a closure with one input argument,
    % and for each key-value pair in List, calls the closure on the key K.
    % The key-value pair is included in TrueList iff Pred(K) is true.
    %
:- func filter(pred(K)::in(pred(in) is semidet),
    assoc_list(K, V)::in) = (assoc_list(K, V)::out) is det.
:- pred filter(pred(K)::in(pred(in) is semidet),
    assoc_list(K, V)::in, assoc_list(K, V)::out) is det.

    % negated_filter(Pred, List, FalseList) takes a closure with one
    % input argument, and for each key-value pair in List, calls the closure
    % on the key K. The key-value pair is included in TrueList iff
    % Pred(K) is false.
    %
:- func negated_filter(pred(K)::in(pred(in) is semidet),
    assoc_list(K, V)::in) = (assoc_list(K, V)::out) is det.
:- pred negated_filter(pred(K)::in(pred(in) is semidet),
    assoc_list(K, V)::in, assoc_list(K, V)::out) is det.

    % filter(Pred, List, TrueList, FalseList) takes a closure with
    % one input argument, and for each key-value pair in List,
    % calls the closure on the key K. If Pred(K) is true, the key-value pair
    % is included in TrueList; otherwise, it is included in FalseList.
    %
:- pred filter(pred(K)::in(pred(in) is semidet),
    assoc_list(K, V)::in, assoc_list(K, V)::out, assoc_list(K, V)::out) is det.

%-----------------------------------------------------%
%
% Merging assoc_lists.
%

    % merge(L1, L2, L):
    %
```

```
    % L is the result of merging the elements of L1 and L2, in ascending order.
    % L1 and L2 must be sorted on the keys.
    %
:- func merge(assoc_list(K, V), assoc_list(K, V)) = assoc_list(K, V).
:- pred merge(assoc_list(K, V)::in, assoc_list(K, V)::in,
    assoc_list(K, V)::out) is det.

%---------------------------------------------------%
%
% Folding over assoc_lists.
%

    % foldl(Pred, List, Start End) calls Pred
    % with each key-value pair in List, working left-to-right,
    % and an accumulator whose initial value is Start,
    % and returns the final value in End.
    %
:- pred foldl(pred(K, V, A, A), assoc_list(K, V), A, A).
:- mode foldl(pred(in, in, in, out) is det, in, in, out) is det.
:- mode foldl(pred(in, in, mdi, muo) is det, in, mdi, muo) is det.
:- mode foldl(pred(in, in, di, uo) is det, in, di, uo) is det.
:- mode foldl(pred(in, in, in, out) is semidet, in, in, out) is semidet.
:- mode foldl(pred(in, in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode foldl(pred(in, in, di, uo) is semidet, in, di, uo) is semidet.
:- mode foldl(pred(in, in, in, out) is nondet, in, in, out) is nondet.

    % foldl_keys(Func List, Start) = End calls Func
    % with each key in List, working left-to-right, and an accumulator
    % whose initial value is Start, and returns the final value in End.
    %
:- func foldl_keys(func(K, A) = A, assoc_list(K, V), A) = A.

    % foldl_keys(Pred, List, Start End) calls Pred
    % with each key in List, working left-to-right, and an accumulator
    % whose initial value is Start, and returns the final value in End.
    %
:- pred foldl_keys(pred(K, A, A), assoc_list(K, V), A, A).
:- mode foldl_keys(pred(in, in, out) is det, in, in, out) is det.
:- mode foldl_keys(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode foldl_keys(pred(in, di, uo) is det, in, di, uo) is det.
:- mode foldl_keys(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode foldl_keys(pred(in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode foldl_keys(pred(in, di, uo) is semidet, in, di, uo) is semidet.
:- mode foldl_keys(pred(in, in, out) is multi, in, in, out) is multi.
:- mode foldl_keys(pred(in, in, out) is nondet, in, in, out) is nondet.

    % foldl_values(Func List, Start) = End calls Func
```

```
    % with each value in List, working left-to-right, and an accumulator
    % whose initial value is Start, and returns the final value in End.
    %
:- func foldl_values(func(V, A) = A, assoc_list(K, V), A) = A.

    % foldl_values(Pred, List, Start End) calls Pred
    % with each value in List, working left-to-right, and an accumulator
    % whose initial value is Start, and returns the final value in End.
    %
:- pred foldl_values(pred(V, A, A), assoc_list(K, V), A, A).
:- mode foldl_values(pred(in, in, out) is det, in,
    in, out) is det.
:- mode foldl_values(pred(in, mdi, muo) is det, in,
    mdi, muo) is det.
:- mode foldl_values(pred(in, di, uo) is det, in,
    di, uo) is det.
:- mode foldl_values(pred(in, in, out) is semidet, in,
    in, out) is semidet.
:- mode foldl_values(pred(in, mdi, muo) is semidet, in,
    mdi, muo) is semidet.
:- mode foldl_values(pred(in, di, uo) is semidet, in,
    di, uo) is semidet.
:- mode foldl_values(pred(in, in, out) is multi, in,
    in, out) is multi.
:- mode foldl_values(pred(in, in, out) is nondet, in,
    in, out) is nondet.

    % As foldl, but with two accumulators.
    %
:- pred foldl2(pred(K, V, A, A, B, B), assoc_list(K, V), A, A, B, B).
:- mode foldl2(pred(in, in, in, out, in, out) is det, in, in, out,
    in, out) is det.
:- mode foldl2(pred(in, in, in, out, mdi, muo) is det, in, in, out,
    mdi, muo) is det.
:- mode foldl2(pred(in, in, in, out, di, uo) is det, in, in, out,
    di, uo) is det.
:- mode foldl2(pred(in, in, in, out, in, out) is semidet, in, in, out,
    in, out) is semidet.
:- mode foldl2(pred(in, in, in, out, mdi, muo) is semidet, in,in, out,
    mdi, muo) is semidet.
:- mode foldl2(pred(in, in, in, out, di, uo) is semidet, in, in, out,
    di, uo) is semidet.
:- mode foldl2(pred(in, in, in, out, in, out) is nondet, in, in, out,
    in, out) is nondet.

    % As foldl_values, but with two accumulators.
    %
```

```
:- pred foldl2_values(pred(V, A, A, B, B), assoc_list(K, V),
    A, A, B, B).
:- mode foldl2_values(pred(in, in, out, in, out) is det, in,
    in, out, in, out) is det.
:- mode foldl2_values(pred(in, in, out, mdi, muo) is det, in,
    in, out, mdi, muo) is det.
:- mode foldl2_values(pred(in, in, out, di, uo) is det, in,
    in, out, di, uo) is det.
:- mode foldl2_values(pred(in, in, out, in, out) is semidet, in,
    in, out, in, out) is semidet.
:- mode foldl2_values(pred(in, in, out, mdi, muo) is semidet, in,
    in, out, mdi, muo) is semidet.
:- mode foldl2_values(pred(in, in, out, di, uo) is semidet, in,
    in, out, di, uo) is semidet.
:- mode foldl2_values(pred(in, in, out, in, out) is multi, in,
    in, out, in, out) is multi.
:- mode foldl2_values(pred(in, in, out, in, out) is nondet, in,
    in, out, in, out) is nondet.

    % As foldl, but with three accumulators.
    %
:- pred foldl3(pred(K, V, A, A, B, B, C, C), assoc_list(K, V),
    A, A, B, B, C, C).
:- mode foldl3(pred(in, in, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out) is det.
:- mode foldl3(pred(in, in, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, mdi, muo) is det.
:- mode foldl3(pred(in, in, in, out, in, out, di, uo) is det, in,
    in, out, in, out, di, uo) is det.
:- mode foldl3(pred(in, in, in, out, in, out, in, out) is semidet, in,
    in, out, in, out, in, out) is semidet.
:- mode foldl3(pred(in, in, in, out, in, out, mdi, muo) is semidet, in,
        in, out, in, out, mdi, muo) is semidet.
:- mode foldl3(pred(in, in, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, di, uo) is semidet.
:- mode foldl3(pred(in, in, in, out, in, out, in, out) is nondet, in,
    in, out, in, out, in, out) is nondet.

    % As foldl_values, but with three accumulators.
    %
:- pred foldl3_values(pred(V, A, A, B, B, C, C), assoc_list(K, V),
    A, A, B, B, C, C).
:- mode foldl3_values(pred(in, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out) is det.
:- mode foldl3_values(pred(in, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, mdi, muo) is det.
:- mode foldl3_values(pred(in, in, out, in, out, di, uo) is det,
```

```
    in, in, out, in, out, di, uo) is det.
:- mode foldl3_values(pred(in, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out) is semidet.
:- mode foldl3_values(pred(in, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, mdi, muo) is semidet.
:- mode foldl3_values(pred(in, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, di, uo) is semidet.
:- mode foldl3_values(pred(in, in, out, in, out, in, out) is multi,
    in, in, out, in, out, in, out) is multi.
:- mode foldl3_values(pred(in, in, out, in, out, in, out) is nondet,
    in, in, out, in, out, in, out) is nondet.

%---------------------------------------------------%
%---------------------------------------------------%
```

4 bag

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 1994-1999, 2003-2007, 2011 The University of Melbourne.
% Copyright (C) 2013-2015, 2017-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: bag.m.
% Main authors: conway, crs.
% Stability: medium.
%
% An implementation of multisets.
%
%---------------------------------------------------%
%---------------------------------------------------%

:- module bag.
:- interface.

:- import_module assoc_list.
:- import_module list.
:- import_module set.

%---------------------------------------------------%

:- type bag(T).
```

```
%---------------------------------------------------%

    % Create an empty bag.
    %
:- func init = bag(T).
:- pred init(bag(T)::out) is det.

    % Create a bag containing the given item.
    %
:- func singleton(T) = bag(T).

    % Check whether a bag is empty.
    %
:- pred is_empty(bag(T)::in) is semidet.

%---------------------------------------------------%

    % contains(Bag, X):
    %
    % Check whether Bag contains X.
    %
:- pred contains(bag(T)::in, T::in) is semidet.

    % count_value(Bag, X):
    %
    % Return how many occurrences of X Bag contains.
    % Return 0 if X is not in Bag.
    %
:- func count_value(bag(T), T) = int.
:- pred count_value(bag(T)::in, T::in, int::out) is det.

    % member(X, Bag):
    %
    % True iff 'Bag' contains at least one occurrence of 'X'.
    %
:- pred member(T::in, bag(T)::in) is semidet.

    % member(X, Bag, BagMinusX):
    %
    % Nondeterministically returns all values X from Bag, and the corresponding
    % bag after X has been removed. Duplicate values are returned as
    % many times as they occur in the Bag.
    %
:- pred member(T::out, bag(T)::in, bag(T)::out) is nondet.

%---------------------------------------------------%
```

```
    % Insert a particular value into a bag.
    %
:- func insert(bag(T), T) = bag(T).
:- pred insert(T::in, bag(T)::in, bag(T)::out) is det.

    % Insert a list of values into a bag.
    %
:- func insert_list(bag(T), list(T)) = bag(T).
:- pred insert_list(list(T)::in, bag(T)::in, bag(T)::out) is det.

    % Insert N copies of a particular value into a bag.
    % Fails if N < 0.
    %
:- pred insert_duplicates(int::in, T::in, bag(T)::in, bag(T)::out)
    is semidet.

    % As above, but throws an exception if N < 0.
    %
:- func det_insert_duplicates(bag(T), int, T) = bag(T).
:- pred det_insert_duplicates(int::in, T::in, bag(T)::in, bag(T)::out) is det.

    % Insert a set of values into a bag.
    %
:- func insert_set(bag(T), set(T)) = bag(T).
:- pred insert_set(set(T)::in, bag(T)::in, bag(T)::out) is det.

%-----------------------------------------------%

    % Remove one occurrence of the smallest value from a bag.
    % Fails if the bag is empty.
    %
:- pred remove_smallest(T::out, bag(T)::in, bag(T)::out) is semidet.

    % Remove one occurrence of a particular value from a bag.
    % Fail if the item does not exist in the bag.
    %
:- pred remove(T::in, bag(T)::in, bag(T)::out) is semidet.

    % Remove one occurrence of a particular value from a bag.
    % Throw an exception if the item does not exist in the bag.
    %
:- func det_remove(bag(T), T) = bag(T).
:- pred det_remove(T::in, bag(T)::in, bag(T)::out) is det.

    % Remove a list of values from a bag. Duplicates are removed from the bag
    % the appropriate number of times. Fail if any of the items in the list
    % do not exist in the bag.
```

```
    %
    % This call is logically equivalent to:
    %
    %   remove_list(Bag0, RemoveList, Bag) :-
    %       from_list(RemoveList, RemoveBag),
    %       is_subbag(RemoveBag, Bag0),
    %       subtract(Bag0, RemoveBag, Bag).
    %
:- pred remove_list(list(T)::in, bag(T)::in, bag(T)::out) is semidet.

    % Remove a list of values from a bag. Duplicates are removed from the bag
    % the appropriate number of times. Throw an exception if any of the items
    % in the list do not exist in the bag.
    %
:- func det_remove_list(bag(T), list(T)) = bag(T).
:- pred det_remove_list(list(T)::in, bag(T)::in, bag(T)::out) is det.

    % Remove a set of values from a bag. Each value is removed once.
    % Fail if any of the items in the set do not exist in the bag.
    %
:- pred remove_set(set(T)::in, bag(T)::in, bag(T)::out) is semidet.

    % Remove a set of values from a bag. Each value is removed once.
    % Throw an exception if any of the items in the set do not exist in the
    % bag.
    %
:- func det_remove_set(bag(T), set(T)) = bag(T).
:- pred det_remove_set(set(T)::in, bag(T)::in, bag(T)::out) is det.

    % Delete one occurrence of a particular value from a bag.
    % If the key is not present, leave the bag unchanged.
    %
:- func delete(bag(T), T) = bag(T).
:- pred delete(T::in, bag(T)::in, bag(T)::out) is det.

    % Remove all occurrences of a particular value from a bag.
    % Fail if the item does not exist in the bag.
    %
:- pred remove_all(T::in, bag(T)::in, bag(T)::out) is semidet.

    % Delete all occurrences of a particular value from a bag.
    %
:- func delete_all(bag(T), T) = bag(T).
:- pred delete_all(T::in, bag(T)::in, bag(T)::out) is det.

%---------------------------------------------------%
```

```
    % Make a bag from a list.
    %
:- func bag(list(T)) = bag(T).
:- func from_list(list(T)) = bag(T).
:- pred from_list(list(T)::in, bag(T)::out) is det.

    % Make a bag from a sorted list.
    %
:- func from_sorted_list(list(T)) = bag(T).
:- pred from_sorted_list(list(T)::in, bag(T)::out) is det.

    % Make a bag from a set.
    %
:- func from_set(set(T)) = bag(T).
:- pred from_set(set(T)::in, bag(T)::out) is det.

    % Given a bag, produce a sorted list containing all the values in the bag.
    % Each value will appear in the list the same number of times that it
    % appears in the bag.
    %
:- func to_list(bag(T)) = list(T).
:- pred to_list(bag(T)::in, list(T)::out) is det.

    % Given a bag, produce a sorted list containing all the values in the bag.
    % Each value will appear in the list once, with the associated integer
    % giving the number of times that it appears in the bag.
    %
:- func to_assoc_list(bag(T)) = assoc_list(T, int).
:- pred to_assoc_list(bag(T)::in, assoc_list(T, int)::out) is det.

    % Given a bag, produce a sorted list with no duplicates containing
    % all the values in the bag.
    %
:- func to_list_without_duplicates(bag(T)) = list(T).
:- pred to_list_without_duplicates(bag(T)::in, list(T)::out) is det.

    % Given a bag, produce a sorted list containing one copy each
    % of all the values that have *more* than one copy in the bag.
    %
:- func to_list_only_duplicates(bag(T)) = list(T).
:- pred to_list_only_duplicates(bag(T)::in, list(T)::out) is det.

    % Given a bag, produce a set containing all the values in the bag.
    %
:- func to_set(bag(T)) = set(T).
:- func to_set_without_duplicates(bag(T)) = set(T).
:- pred to_set_without_duplicates(bag(T)::in, set(T)::out) is det.
```

```
:- pragma obsolete(to_set_without_duplicates/1).
:- pragma obsolete(to_set_without_duplicates/2).

%-----------------------------------------------------%

    % subtract(BagA, BagB, BagAmB):
    %
    % Subtracts BagB from BagA to produce BagAmB. Each element in BagB is
    % removed from BagA to produce BagAmB.
    %
    % An example:
    % subtract({1, 1, 2, 2, 3 }, {1, 1, 2, 3, 3, 3}, {2}).
    %
    % Use one of the subtract_small variants if BagB is expected to be
    % significantly smaller than BagA.
    %
:- func subtract(bag(T), bag(T)) = bag(T).
:- pred subtract(bag(T)::in, bag(T)::in, bag(T)::out) is det.
:- func subtract_small(bag(T), bag(T)) = bag(T).
:- pred subtract_small(bag(T)::in, bag(T)::in, bag(T)::out) is det.

    % least_upper_bound(BagA, BagB, BagAlubB):
    %
    % BagAlubB is the least upper bound of BagA and BagB.
    % It is the smallest bag that contains at least as many copies
    % of each element as BagA, and at least as many copies as BagB.
    % If an element X is present AXN in BagA and BXN times in BagB,
    % X will be present int.max(AXN, BXN) times in BagAlubB.
    %
    % An example:
    % least_upper_bound({1, 1, 2}, {2, 2, 3}, {1, 1, 2, 2, 3}).
    %
    % Use one of the least_upper_bound_small variants if BagB is expected
    % to be significantly smaller than BagA. (If BagA is expected to be
    % significantly smaller than BagB, then switch the operands around.)
    %
:- func least_upper_bound(bag(T), bag(T)) = bag(T).
:- pred least_upper_bound(bag(T)::in, bag(T)::in, bag(T)::out) is det.
:- func least_upper_bound_small(bag(T), bag(T)) = bag(T).
:- pred least_upper_bound_small(bag(T)::in, bag(T)::in, bag(T)::out) is det.

    % union(BagA, BagB, BagAuB):
    %
    % BagAuB is the union of BagA and BagB.
    %
    % An example:
    % e.g. {1, 1, 2, 2} U {2, 2, 3, 3} = {1, 1, 2, 2, 2, 2, 3, 3}.
```

```
    %
    % Use one of the union_small variants if BagB is expected to be
    % significantly smaller than BagA. (If BagA is expected to be
    % significantly smaller than BagB, then switch the operands around.)
    %
:- func union(bag(T), bag(T)) = bag(T).
:- pred union(bag(T)::in, bag(T)::in, bag(T)::out) is det.
:- func union_small(bag(T), bag(T)) = bag(T).
:- pred union_small(bag(T)::in, bag(T)::in, bag(T)::out) is det.

    % intersect(BagA, BagB, BagAuB):
    %
    % BagAiB is the intersection of BagA and BagB.
    %
    % An example:
    % intersect({1, 2, 2, 3, 3}, {2, 2, 3, 4}, {2, 2, 3}).
    %
    % Use one of the intersect_small variants if BagB is expected to be
    % significantly smaller than BagA. (If BagA is expected to be
    % significantly smaller than BagB, then switch the operands around.)
    %
:- func intersect(bag(T), bag(T)) = bag(T).
:- pred intersect(bag(T)::in, bag(T)::in, bag(T)::out) is det.
:- func intersect_small(bag(T), bag(T)) = bag(T).
:- pred intersect_small(bag(T)::in, bag(T)::in, bag(T)::out) is det.

    % Fails if there is no intersection between the 2 bags.
    % intersect(A, B) :- intersect(A, B, C), not is_empty(C).
    %
:- pred intersect(bag(T)::in, bag(T)::in) is semidet.

%-----------------------------------------------%

    % Tests whether the first bag is a subbag of the second.
    % is_subbag(BagA, BagB) implies that every element in the BagA
    % is also in the BagB. If an element is in BagA multiple times,
    % it must be in BagB at least as many times.
    % e.g. is_subbag({1, 1, 2}, {1, 1, 2, 2, 3}).
    % e.g. is_subbag({1, 1, 2}, {1, 2, 3}) :- fail.
    %
:- pred is_subbag(bag(T)::in, bag(T)::in) is semidet.

    % Compares the two bags, and returns whether the first bag is a
    % subset (<), is equal (=), or is a superset (>) of the second.
    % Fails if the two bags are incomparable.
    %
    % Examples:
```

```
    % subset_compare(<, {apple, orange}, {apple, apple, orange}).
    % subset_compare(=, {apple, orange}, {apple, orange}).
    % subset_compare(>, {apple, apple, orange}, {apple, orange}).
    % subset_compare(_, {apple, apple}, {orange, orange}) :- fail.
    %
:- pred subset_compare(comparison_result::out, bag(T)::in, bag(T)::in)
    is semidet.

%---------------------------------------------------%

    % Perform a traversal of the bag, applying an accumulator predicate
    % to each value - count pair.
    %
:- pred foldl(pred(T, int, A, A), bag(T), A, A).
:- mode foldl(pred(in, in, in, out) is det, in, in, out) is det.
:- mode foldl(pred(in, in, mdi, muo) is det, in, mdi, muo) is det.
:- mode foldl(pred(in, in, di, uo) is det, in, di, uo) is det.
:- mode foldl(pred(in, in, in, out) is semidet, in, in, out) is semidet.
:- mode foldl(pred(in, in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode foldl(pred(in, in, di, uo) is semidet, in, di, uo) is semidet.

    % As above, but with two accumulators.
    %
:- pred foldl2(pred(T, int, A, A, B, B), bag(T), A, A, B, B).
:- mode foldl2(pred(in, in, in, out, in, out) is det, in, in, out,
    in, out) is det.
:- mode foldl2(pred(in, in, in, out, mdi, muo) is det, in, in, out,
    mdi, muo) is det.
:- mode foldl2(pred(in, in, in, out, di, uo) is det, in, in, out,
    di, uo) is det.
:- mode foldl2(pred(in, in, in, out, in, out) is semidet, in, in, out,
    in, out) is semidet.
:- mode foldl2(pred(in, in, in, out, mdi, muo) is semidet, in, in, out,
    mdi, muo) is semidet.
:- mode foldl2(pred(in, in, in, out, di, uo) is semidet, in, in, out,
    di, uo) is semidet.

%---------------------------------------------------%

    % Return the number of values in a bag.
    % If an element X is present N times, count it N times.
    %
:- func count(bag(T)) = int.

    % Return the number of unique values in a bag.
    % Even if an element X is present N times, count it just one.
    %
```

```
:- func count_unique(bag(T)) = int.

%---------------------------------------------------%
%---------------------------------------------------%
```

5 benchmarking

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 1994-2011 The University of Melbourne.
% Copyright (C) 2014-2016, 2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: benchmarking.m.
% Main author: zs.
% Stability: medium.
%
% This module contains predicates that deal with the CPU time requirements
% of (various parts of) the program.
%
%---------------------------------------------------%
%---------------------------------------------------%

:- module benchmarking.
:- interface.

:- import_module bool.
:- import_module io.
:- import_module maybe.

    % 'report_stats' is a non-logical procedure intended for use in profiling
    % the performance of a program. It has the side-effect of reporting
    % some memory and time usage statistics about the time period since
    % the last call to report_stats to stderr.
    %
    % Note: in Java, this reports usage of the calling thread. You will get
    % nonsensical results if the previous call to 'report_stats' was from a
    % different thread.
    %
    % Note: in Erlang, the benchmark_* procedures will change the appar-
ent time
    % of the last call to report_stats.
    %
```

```
:- impure pred report_stats is det.

    % 'report_full_memory_stats' is a non-logical procedure intended for use
    % in profiling the memory usage of a program. It has the side-effect
    % of reporting a full memory profile to stderr.
    %
:- impure pred report_full_memory_stats is det.

    % report_memory_attribution(Label, Collect, !IO) is a procedure intended
    % for use in profiling the memory usage by a program. It is supported in
    % 'memprof.gc' grades only, in other grades it is a no-op. It reports a
    % summary of the objects on the heap to a data file. See ''Using mprof -
s
    % for profiling memory retention'' in the Mercury User's Guide. The label
    % is for your reference. If Collect is yes it has the effect of forc-
ing a
    % garbage collection before building the report.
    %
:- pred report_memory_attribution(string::in, bool::in, io::di, io::uo) is det.
:- impure pred report_memory_attribution(string::in, bool::in) is det.

    % report_memory_attribution(Label, !IO) is the same as
    % report_memory_attribution/4 above, except that it always forces a
    % collection (in 'memprof.gc' grades).
    %
:- pred report_memory_attribution(string::in, io::di, io::uo) is det.
:- impure pred report_memory_attribution(string::in) is det.

    % benchmark_det(Pred, In, Out, Repeats, Time) is for benchmarking the det
    % predicate Pred. We call Pred with the input In and the output Out, and
    % return Out so that the caller can check the correctness of the
    % benchmarked predicate. Since most systems do not have good facilities
    % for measuring small times, the Repeats parameter allows the caller
    % to specify how many times Pred should be called inside the timed
    % interval. The number of milliseconds required to execute Pred with input
    % In this many times is returned as Time.
    %
    % benchmark_func(Func, In, Out, Repeats, Time) does for functions
    % exactly what benchmark_det does for predicates.
    %
:- pred benchmark_det(pred(T1, T2), T1, T2, int, int).
:- mode benchmark_det(pred(in, out) is det, in, out, in, out) is cc_multi.
:- mode benchmark_det(pred(in, out) is cc_multi, in, out, in, out) is cc_multi.

:- pred benchmark_func(func(T1) = T2, T1, T2, int, int).
:- mode benchmark_func(func(in) = out is det, in, out, in, out) is cc_multi.
```

```
:- pred benchmark_det_io(pred(T1, T2, T3, T3), T1, T2, T3, T3, int, int).
:- mode benchmark_det_io(pred(in, out, di, uo) is det, in, out, di, uo,
    in, out) is cc_multi.

    % benchmark_nondet(Pred, In, Count, Repeats, Time) is for benchmarking
    % the nondet predicate Pred. benchmark_nondet is similar to benchmark_det,
    % but it returns only a count of the solutions, rather than solutions
    % themselves. The number of milliseconds required to generate all
    % solutions of Pred with input In Repeats times is returned as Time.
    %
:- pred benchmark_nondet(pred(T1, T2), T1, int, int, int).
:- mode benchmark_nondet(pred(in, out) is nondet, in, out, in, out)
    is cc_multi.

%-----------------------------------------------------%
%-----------------------------------------------------%

    % Turn off or on the collection of all profiling statistics.
    %
:- pred turn_off_profiling(io::di, io::uo) is det.
:- pred turn_on_profiling(io::di, io::uo) is det.

:- impure pred turn_off_profiling is det.
:- impure pred turn_on_profiling is det.

    % Turn off or on the collection of call graph profiling statistics.
    %
:- pred turn_off_call_profiling(io::di, io::uo) is det.
:- pred turn_on_call_profiling(io::di, io::uo) is det.

:- impure pred turn_off_call_profiling is det.
:- impure pred turn_on_call_profiling is det.

    % Turn off or on the collection of time spent in each procedure
    % profiling statistics.
    %
:- pred turn_off_time_profiling(io::di, io::uo) is det.
:- pred turn_on_time_profiling(io::di, io::uo) is det.

:- impure pred turn_off_time_profiling is det.
:- impure pred turn_on_time_profiling is det.

    % Turn off or on the collection of memory allocated in each procedure
    % profiling statistics.
    %
:- pred turn_off_heap_profiling(io::di, io::uo) is det.
:- pred turn_on_heap_profiling(io::di, io::uo) is det.
```

```
:- impure pred turn_off_heap_profiling is det.
:- impure pred turn_on_heap_profiling is det.

%-----------------------------------------------------%
%-----------------------------------------------------%

    % write_out_trace_counts(FileName, MaybeErrorMsg, !IO):
    %
    % Write out the trace counts accumulated so far in this program's execution
    % to FileName. If successful, set MaybeErrorMsg to "no". If unsuccessful,
    % e.g. because the program wasn't compiled with debugging enabled or
    % because trace counting isn't turned on, then set MaybeErrorMsg to a "yes"
    % wrapper around an error message.
    %
:- pred write_out_trace_counts(string::in, maybe(string)::out,
    io::di, io::uo) is det.

%-----------------------------------------------------%
%-----------------------------------------------------%

    % Place a log message in the threadscope event stream. The event will be
    % logged as being generated by the current Mercury Engine. This is a no-
op
    % when threadscope is not available.
    %
:- pred log_threadscope_message(string::in, io::di, io::uo) is det.

%-----------------------------------------------------%
%-----------------------------------------------------%
```

6 bimap

```
%-----------------------------------------------------%
% vim: ts=4 sw=4 et ft=mercury
%-----------------------------------------------------%
% Copyright (C) 1994-1995,1997,1999,2004-2006,2008,2011-2012 The Univer-
sity of Melbourne.
% Copyright (C) 2013-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: bimap.m.
% Main author: conway.
% Stability: medium.
```

```
%
% This file provides a bijective map ADT.
% A map (also known as a dictionary or an associative array) is a collection
% of (Key, Data) pairs which allows you to look up any Data item given the
% Key. A bimap also allows you to efficiently look up the Key given the Data.
% This time efficiency comes at the expense of using twice as much space.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module bimap.
:- interface.

:- import_module assoc_list.
:- import_module list.
:- import_module map.
:- import_module maybe.

%-----------------------------------------------------%

:- type bimap(K, V).

%-----------------------------------------------------%

    % Initialize an empty bimap.
    %
:- func init = bimap(K, V).
:- pred init(bimap(K, V)::out) is det.

    % Initialize a bimap with the given key-value pair.
    %
:- func singleton(K, V) = bimap(K, V).

    % Check whether a bimap is empty.
    %
:- pred is_empty(bimap(K, V)::in) is semidet.

    % True if both bimaps have the same set of key-value pairs, regard-
less of
    % how the bimaps were constructed.
    %
    % Unifying bimaps does not work as one might expect because the internal
    % structures of two bimaps that contain the same set of key-value pairs
    % may be different.
    %
:- pred equal(bimap(K, V)::in, bimap(K, V)::in) is semidet.
```

```
    % Search the bimap. The first mode searches for a value given a key
    % and the second mode searches for a key given a value.
    %
:- pred search(bimap(K, V), K, V).
:- mode search(in, in, out) is semidet.
:- mode search(in, out, in) is semidet.

    % Search the bimap for the value corresponding to a given key.
    %
:- func forward_search(bimap(K, V), K) = V is semidet.
:- pred forward_search(bimap(K, V)::in, K::in, V::out) is semidet.

    % Search the bimap for the key corresponding to the given value.
    %
:- func reverse_search(bimap(K, V), V) = K is semidet.
:- pred reverse_search(bimap(K, V)::in, K::out, V::in) is semidet.

    % Look up the value in the bimap corresponding to the given key.
    % Throws an exception if the key is not present in the bimap.
    %
:- func lookup(bimap(K, V), K) = V.
:- pred lookup(bimap(K, V)::in, K::in, V::out) is det.

    % Look up the key in the bimap corresponding to the given value.
    % Throws an exception if the value is not present in the bimap.
    %
:- func reverse_lookup(bimap(K, V), V) = K.
:- pred reverse_lookup(bimap(K, V)::in, K::out, V::in) is det.

    % Succeeds iff the bimap contains the given key.
    %
:- pred contains_key(bimap(K, V)::in, K::in) is semidet.

    % Succeeds iff the bimap contains the given value.
    %
:- pred contains_value(bimap(K, V)::in, V::in) is semidet.

    % Given a bimap, return a list of all the keys in the bimap.
    %
:- func ordinates(bimap(K, V)) = list(K).
:- pred ordinates(bimap(K, V)::in, list(K)::out) is det.

    % Given a bimap, return a list of all the data values in the bimap.
    %
:- func coordinates(bimap(K, V)) = list(V).
:- pred coordinates(bimap(K, V)::in, list(V)::out) is det.
```

```
    % Insert a new key-value pair into the bimap.
    % Fails if either the key or value already exists.
    %
:- func insert(bimap(K, V), K, V) = bimap(K, V) is semidet.
:- pred insert(K::in, V::in, bimap(K, V)::in, bimap(K, V)::out)
    is semidet.

    % As above but throws an exception if the key or value already
    % exists.
    %
:- func det_insert(bimap(K, V), K, V) = bimap(K, V).
:- pred det_insert(K::in, V::in, bimap(K, V)::in, bimap(K, V)::out)
    is det.

    % search_insert(K, V, MaybeOldV, !Bimap):
    %
    % Search for the key K in the bimap. If the key is already in the bimap,
    % with corresponding value OldV, set MaybeOldV to yes(OldV). If it
    % is not in the bimap, then insert it with value V, and set MaybeOldV
    % to no. The value of V should be guaranteed to be different to
    % all the values already in !.Bimap. If it isn't, this predicate
    % will throw an exception.
    %
:- pred search_insert(K::in, V::in, maybe(V)::out,
    bimap(K, V)::in, bimap(K, V)::out) is det.

    % Update the key and value if already present, otherwise insert the
    % new key and value.
    %
    % NOTE: setting the key-value pair (K, V) will remove the key-value pairs
    % (K, V1) and (K1, V) if they exist.
    %
:- func set(bimap(K, V), K, V) = bimap(K, V).
:- pred set(K::in, V::in, bimap(K, V)::in, bimap(K, V)::out) is det.

    % Insert key-value pairs from an association list into the given bimap.
    % Fails if the contents of the association list and the initial bimap
    % do not implicitly form a bijection.
    %
:- func insert_from_assoc_list(assoc_list(K, V), bimap(K, V)) =
    bimap(K, V) is semidet.
:- pred insert_from_assoc_list(assoc_list(K, V)::in,
    bimap(K, V)::in, bimap(K, V)::out) is semidet.

    % As above but throws an exception if the association list and
    % initial bimap are not implicitly bijective.
    %
```

```
:- func det_insert_from_assoc_list(assoc_list(K, V), bimap(K, V))
    = bimap(K, V).
:- pred det_insert_from_assoc_list(assoc_list(K, V)::in,
    bimap(K, V)::in, bimap(K, V)::out) is det.

    % Insert key-value pairs from a pair of corresponding lists.
    % Throws an exception if the lists are not of equal lengths
    % or if they do not implicitly define a bijection.
    %
:- func det_insert_from_corresponding_lists(list(K), list(V),
    bimap(K, V)) = bimap(K, V).
:- pred det_insert_from_corresponding_lists(list(K)::in, list(V)::in,
    bimap(K, V)::in, bimap(K, V)::out) is det.

    % Apply set to each key-value pair in the association list.
    % The key-value pairs from the association list may update existing keys
    % and values in the bimap.
    %
:- func set_from_assoc_list(assoc_list(K, V), bimap(K, V))
    = bimap(K, V).
:- pred set_from_assoc_list(assoc_list(K, V)::in,
    bimap(K, V)::in, bimap(K, V)::out) is det.

    % As above but with a pair of corresponding lists in place of an
    % association list. Throws an exception if the lists are not of
    % equal length.
    %
:- func set_from_corresponding_lists(list(K), list(V),
    bimap(K, V)) = bimap(K, V).
:- pred set_from_corresponding_lists(list(K)::in, list(V)::in,
    bimap(K, V)::in, bimap(K, V)::out) is det.

    % Delete a key-value pair from a bimap. If the key is not present,
    % leave the bimap unchanged.
    %
:- func delete_key(bimap(K, V), K) = bimap(K, V).
:- pred delete_key(K::in, bimap(K, V)::in, bimap(K, V)::out) is det.

    % Delete a key-value pair from a bimap. If the value is not present,
    % leave the bimap unchanged.
    %
:- func delete_value(bimap(K, V), V) = bimap(K, V).
:- pred delete_value(V::in, bimap(K, V)::in, bimap(K, V)::out) is det.

    % Apply delete_key to a list of keys.
    %
:- func delete_keys(bimap(K, V), list(K)) = bimap(K, V).
```

```
:- pred delete_keys(list(K)::in, bimap(K, V)::in, bimap(K, V)::out)
    is det.

    % Apply delete_value to a list of values.
    %
:- func delete_values(bimap(K, V), list(V)) = bimap(K, V).
:- pred delete_values(list(V)::in, bimap(K, V)::in, bimap(K, V)::out)
    is det.

    % overlay(BIMapA, BIMapB, BIMap):
    % Apply map.overlay to the forward maps of BIMapA and BIMapB,
    % and compute the reverse map from the resulting map.
    %
:- func overlay(bimap(K, V), bimap(K, V)) = bimap(K, V).
:- pred overlay(bimap(K, V)::in, bimap(K, V)::in, bimap(K, V)::out)
    is det.

    % Count the number of key-value pairs in the bimap.
    %
:- func count(bimap(K, V)) = int.

    % Convert a bimap to an association list.
    %
:- func to_assoc_list(bimap(K, V)) = assoc_list(K, V).
:- pred to_assoc_list(bimap(K, V)::in, assoc_list(K, V)::out) is det.

    % Convert an association list to a bimap. Fails if the association list
    % does not implicitly define a bijection, i.e. a key or value occurs
    % multiple times in the association list.
    %
:- func from_assoc_list(assoc_list(K, V)) = bimap(K, V) is semidet.
:- pred from_assoc_list(assoc_list(K, V)::in, bimap(K, V)::out)
    is semidet.

    % As above but throws an exception instead of failing if the
    % association list does not implicitly define a bijection.
    %
:- func det_from_assoc_list(assoc_list(K, V)) = bimap(K, V).
:- pred det_from_assoc_list(assoc_list(K, V)::in, bimap(K, V)::out)
    is det.

    % Convert a pair of lists into a bimap. Fails if the lists do not
    % implicitly define a bijection or if the lists are of unequal length.
    %
:- func from_corresponding_lists(list(K), list(V)) = bimap(K, V)
    is semidet.
:- pred from_corresponding_lists(list(K)::in, list(V)::in,
```

```
    bimap(K, V)::out) is semidet.

    % As above but throws an exception instead of failing if the lists
    % do not implicitly define a bijection or are of unequal length.
    %
:- func det_from_corresponding_lists(list(K), list(V)) = bimap(K, V).
:- pred det_from_corresponding_lists(list(K)::in, list(V)::in,
    bimap(K, V)::out) is det.

:- func apply_forward_map_to_list(bimap(K, V), list(K)) = list(V).
:- pred apply_forward_map_to_list(bimap(K, V)::in, list(K)::in,
    list(V)::out) is det.

:- func apply_reverse_map_to_list(bimap(K, V), list(V)) = list(K).
:- pred apply_reverse_map_to_list(bimap(K, V)::in, list(V)::in,
    list(K)::out) is det.

    % Apply a transformation predicate to all the keys.
    % Throws an exception if the resulting bimap is not bijective.
    %
:- func map_keys(func(V, K) = L, bimap(K, V)) = bimap(L, V).
:- pred map_keys(pred(V, K, L)::in(pred(in, in, out) is det),
    bimap(K, V)::in, bimap(L, V)::out) is det.

    % Apply a transformation predicate to all the values.
    % Throws an exception if the resulting bimap is not bijective.
    %
:- func map_values(func(K, V) = W, bimap(K, V)) = bimap(K, W).
:- pred map_values(pred(K, V, W)::in(pred(in, in, out) is det),
    bimap(K, V)::in, bimap(K, W)::out) is det.

    % Perform an inorder traversal, by key, of the bimap, applying an
    % accumulator predicate for each key-value pair.
    %
:- func foldl(func(K, V, A) = A, bimap(K, V), A) = A.
:- pred foldl(pred(K, V, A, A), bimap(K, V), A, A).
:- mode foldl(pred(in, in, in, out) is det, in, in, out) is det.
:- mode foldl(pred(in, in, mdi, muo) is det, in, mdi, muo) is det.
:- mode foldl(pred(in, in, di, uo) is det, in, di, uo) is det.
:- mode foldl(pred(in, in, in, out) is semidet, in, in, out) is semidet.
:- mode foldl(pred(in, in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode foldl(pred(in, in, di, uo) is semidet, in, di, uo) is semidet.

    % Perform a traversal of the bimap, applying an accumulator predicate
    % with two accumulators for each key-value pair. (Although no more
    % expressive than foldl, this is often a more convenient format,
    % and a little more efficient).
```

```
    %
:- pred foldl2(pred(K, V, A, A, B, B), bimap(K, V), A, A, B, B).
:- mode foldl2(pred(in, in, in, out, in, out) is det,
    in, in, out, in, out) is det.
:- mode foldl2(pred(in, in, in, out, mdi, muo) is det,
    in, in, out, mdi, muo) is det.
:- mode foldl2(pred(in, in, in, out, di, uo) is det,
    in, in, out, di, uo) is det.
:- mode foldl2(pred(in, in, di, uo, di, uo) is det,
    in, di, uo, di, uo) is det.
:- mode foldl2(pred(in, in, in, out, in, out) is semidet,
    in, in, out, in, out) is semidet.
:- mode foldl2(pred(in, in, in, out, mdi, muo) is semidet,
    in, in, out, mdi, muo) is semidet.
:- mode foldl2(pred(in, in, in, out, di, uo) is semidet,
    in, in, out, di, uo) is semidet.

    % Perform a traversal of the bimap, applying an accumulator predicate
    % with three accumulators for each key-value pair. (Although no more
    % expressive than foldl, this is often a more convenient format,
    % and a little more efficient).
    %
:- pred foldl3(pred(K, V, A, A, B, B, C, C), bimap(K, V),
    A, A, B, B, C, C).
:- mode foldl3(pred(in, in, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out) is det.
:- mode foldl3(pred(in, in, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, mdi, muo) is det.
:- mode foldl3(pred(in, in, in, out, in, out, di, uo) is det,
    in, in, out, in, out, di, uo) is det.
:- mode foldl3(pred(in, in, in, out, di, uo, di, uo) is det,
    in, in, out, di, uo, di, uo) is det.
:- mode foldl3(pred(in, in, di, uo, di, uo, di, uo) is det,
    in, di, uo, di, uo, di, uo) is det.
:- mode foldl3(pred(in, in, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out) is semidet.
:- mode foldl3(pred(in, in, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, mdi, muo) is semidet.
:- mode foldl3(pred(in, in, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, di, uo) is semidet.

    % Extract a the forward map from the bimap, the map from key to value.
    %
:- func forward_map(bimap(K, V)) = map(K, V).

    % Extract the reverse map from the bimap, the map from value to key.
    %
```

```
:- func reverse_map(bimap(K, V)) = map(V, K).

%---------------------------------------------------%
%---------------------------------------------------%
```

7 bit_buffer

```
%---------------------------------------------------%
% vim: ts=4 sw=4 et ft=mercury
%---------------------------------------------------%
% Copyright (C) 2007, 2009 The University of Melbourne
% Copyright (C) 2014, 2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
% File: bit_buffer.m.
% Main author: stayl.
% Stability: low.
%
% A bit buffer provides an interface between bit-oriented I/O requests
% and byte-oriented streams.  The useful part of the interface is defined
% in bit_buffer.read and bit_buffer.write.
%
% CAVEAT: the user is referred to the documentation in the header
% of array.m regarding programming with unique objects (the compiler
% does not currently recognise them, hence we are forced to use
% non-unique modes until the situation is rectified; this places
% a small burden on the programmer to ensure the correctness of his
% code that would otherwise be assured by the compiler.)
%
%---------------------------------------------------%
%---------------------------------------------------%

:- module bit_buffer.
:- interface.

:- import_module bitmap.
:- import_module stream.

:- include_module bit_buffer.read.
:- include_module bit_buffer.write.

    % An error_stream throws an ‘error_stream_error’ exception if any of
    % its output methods are called, or returns an ‘error_stream_error’
    % if any of its input methods are called.
    %
```

```
:- type error_stream ---> error_stream.
:- type error_state ---> error_state.
:- type error_stream_error ---> error_stream_error.
:- instance stream.error(error_stream_error).
:- instance stream.stream(error_stream, error_state).
:- instance stream.input(error_stream, error_state).
:- instance stream.bulk_reader(error_stream, byte_index, bitmap,
        error_state, error_stream_error).

:- instance stream.output(error_stream, error_state).
:- instance stream.writer(error_stream, bitmap.slice, error_state).

%-----------------------------------------------------%
```

8 bit_buffer.read

```
%-----------------------------------------------------%
% vim: ts=4 sw=4 et ft=mercury
%-----------------------------------------------------%
% Copyright (C) 2007, 2010-2011 The University of Melbourne
% Copyright (C) 2014-2015, 2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
% File: bit_buffer.read.m.
% Main author: stayl.
% Stability: low.
%
% A bit buffer provides an interface between bit-oriented input requests
% and byte-oriented streams, getting a large chunk of bits with one call
% to 'bulk_get', then satisfying bit-oriented requests from the buffer.
%
% Return values of 'error(...)' are only used for errors in the stream
% being read.  Once an error value has been returned, all future calls
% will return that error.
%
% Bounds errors or invalid argument errors (for example a read request
% for a negative number of bits) will result in an exception being thrown.
% Requests triggering an exception in this way will not change the state
% of the stream.
%
% CAVEAT: the user is referred to the documentation in the header
% of array.m regarding programming with unique objects (the compiler
% does not currently recognise them, hence we are forced to use
% non-unique modes until the situation is rectified; this places
% a small burden on the programmer to ensure the correctness of his
```

```
% code that would otherwise be assured by the compiler.)
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module bit_buffer.read.
:- interface.

:- import_module io.
:- import_module bitmap.

:- type read_buffer(Stream, State, Error).
        % <= stream.bulk_reader(Stream, byte_index, bitmap, State, Error).

:- type read_buffer ==
            read_buffer(error_stream, error_state, error_stream_error).

:- type io_read_buffer ==
            read_buffer(io.binary_input_stream, io.state, io.error).

:- inst uniq_read_buffer == ground.   % XXX Should be unique.
:- mode read_buffer_di == in(uniq_read_buffer).
:- mode read_buffer_ui == in(uniq_read_buffer).
:- mode read_buffer_uo == out(uniq_read_buffer).

    % new(NumBytes, Stream, State) creates a buffer which will read from
    % the stream specified by Stream and State in chunks of NumBytes bytes.
    % 'NumBytes' must at least the size of a Mercury int, given by
    % int.bits_per_int.  If it is less, the size of an int will be used
    % instead.
    %
:- func new(num_bytes, Stream, State) = read_buffer(Stream, State, Error)
    <= stream.bulk_reader(Stream, byte_index, bitmap, State, Error).
:- mode new(in, in, di) = read_buffer_uo is det.

    % new(BitIndex, StartIndex, NumBits)
    % Create a buffer which reads bits from a bitmap, not from a stream.
    %
:- func new_bitmap_reader(bitmap, bit_index, num_bits) = read_buffer.
:- mode new_bitmap_reader(in, in, in) = read_buffer_uo is det.

:- func new_bitmap_reader(bitmap) = read_buffer.
:- mode new_bitmap_reader(in) = read_buffer_uo is det.

    % How many bits to be read does the buffer contain.
    %
:- func num_buffered_bits(read_buffer(_, _, _)) = num_bits.
```

```
:- mode num_buffered_bits(read_buffer_ui) = out is det.

    % How many bits need to be read to get to the next byte boundary.
    %
:- func num_bits_to_byte_boundary(read_buffer(_, _, _)) = num_bits.
:- mode num_bits_to_byte_boundary(read_buffer_ui) = out is det.

    % Find out whether there are bits left in the stream or an error
    % has been found.
    %
:- pred buffer_status(stream.result(Error),
    read_buffer(Stream, State, Error),
    read_buffer(Stream, State, Error))
    <= stream.bulk_reader(Stream, byte_index, bitmap, State, Error).
:- mode buffer_status(out, read_buffer_di, read_buffer_uo) is det.

    % Read a bit from the buffer.
    %
    % This implements the get/4 method of class stream.reader.
    %
:- pred get_bit(stream.result(bool, Error), read_buffer(Stream, State, Error),
    read_buffer(Stream, State, Error))
    <= stream.bulk_reader(Stream, byte_index, bitmap, State, Error).
:- mode get_bit(out, read_buffer_di, read_buffer_uo) is det.

    % get_bits(Index, NumBits, !Word, NumBitsRead, Result, !Buffer).
    %
    % Read NumBits bits from the buffer into a word starting at Index,
    % where the highest order bit is bit zero.
    % 0 =< NumBits =< int.bits_per_int.
    %
    % This implements the bulk_get/9 method of stream.bulk_reader.
    %
    % To read into the lower order bits of the word, use
    % 'get_bits(bits_per_int - NumBits, NumBits, ...)'.
    %
:- pred get_bits(bit_index, num_bits, word, word, num_bits,
    stream.res(Error), read_buffer(Stream, State, Error),
    read_buffer(Stream, State, Error))
    <= stream.bulk_reader(Stream, byte_index, bitmap, State, Error).
:- mode get_bits(in, in, di, uo, out, out,
    read_buffer_di, read_buffer_uo) is det.

    % get_bitmap(!Bitmap, NumBitsRead, Result, !Buffer)
    %
    % Fill a bitmap from the buffered stream, returning the number
    % of bits read.
```

```
    %
    % Note that this is much more efficient if the initial position in
    % the buffer is at a byte boundary (for example after a call to
    % skip_padding_to_byte).
    %
:- pred get_bitmap(bitmap, bitmap, num_bits,
    stream.res(Error), read_buffer(Stream, State, Error),
    read_buffer(Stream, State, Error))
    <= stream.bulk_reader(Stream, byte_index, bitmap, State, Error).
:- mode get_bitmap(bitmap_di, bitmap_uo, out, out,
    read_buffer_di, read_buffer_uo) is det.

    % get_bitmap(Index, NumBits, !Bitmap, NumBitsRead, Result, !Buffer)
    %
    % Note that this is much more efficient if both Index and the initial
    % position in the buffer are both at a byte boundary (for example after
    % a call to skip_padding_to_byte).
    %
    % This implements the bulk_get method of stream.bulk_reader.
    %
:- pred get_bitmap(bit_index, num_bits, bitmap, bitmap, num_bits,
    stream.res(Error), read_buffer(Stream, State, Error),
    read_buffer(Stream, State, Error))
    <= stream.bulk_reader(Stream, byte_index, bitmap, State, Error).
:- mode get_bitmap(in, in, bitmap_di, bitmap_uo, out, out,
    read_buffer_di, read_buffer_uo) is det.

    % finalize(Buffer, Stream, State, BufferBM,
    %   IndexInBufferBM, NumBitsInBufferBM)
    %
    % Returns the stream, state and the unread buffered bits.
    %
:- pred finalize(read_buffer(Stream, State, Error), Stream, State,
    bitmap, bit_index, num_bits)
    <= stream.bulk_reader(Stream, byte_index, bitmap, State, Error).
:- mode finalize(read_buffer_di, out, uo, bitmap_uo, out, out) is det.

%-----------------------------------------------------------------------%
```

9 bit_buffer.write

```
%-----------------------------------------------------------------------%
% vim: ts=4 sw=4 et ft=mercury
%-----------------------------------------------------------------------%
% Copyright (C) 2007, 2011 The University of Melbourne
% Copyright (C) 2014-2016, 2018 The Mercury team.
```

```
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
% File: bit_buffer.write.m.
% Main author: stayl.
% Stability: low.
%
% A bit buffer provides an interface between bit-oriented output requests
% and byte-array-oriented streams, storing bits until there are enough bytes
% to make calling the 'put' method on the stream worthwhile.
%
% CAVEAT: the user is referred to the documentation in the header
% of array.m regarding programming with unique objects (the compiler
% does not currently recognise them, hence we are forced to use
% non-unique modes until the situation is rectified; this places
% a small burden on the programmer to ensure the correctness of his
% code that would otherwise be assured by the compiler.)
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module bit_buffer.write.
:- interface.

:- import_module io.

:- type write_buffer(Stream, State).
        % <= stream.writer(Stream, bitmap.slice, State).

:- type write_buffer == write_buffer(error_stream, error_state).
:- type io_write_buffer == write_buffer(io.binary_output_stream, io.state).

:- inst uniq_write_buffer == ground.   % XXX Should be unique.
:- mode write_buffer_di == in(uniq_write_buffer).
:- mode write_buffer_ui == in(uniq_write_buffer).
:- mode write_buffer_uo == out(uniq_write_buffer).

    % new(NumBytes, Stream, State) creates a buffer which will write to
    % the stream specified by Stream and State in chunks of NumBytes bytes.
    % If NumBytes is less than the size of an integer (given by
    % int.bits_per_int), the size of an integer will be used instead.
    %
:- func new(num_bytes, Stream, State) = write_buffer(Stream, State)
    <= stream.writer(Stream, byte_index, State).
:- mode new(in, in, di) = write_buffer_uo is det.

    % new(NumBytes):
    %
```

```
    % Create a buffer which collects all of the bits written, and does
    % not write them to a stream.  The bits are collected in chunks of
    % size NumBytes bytes, and are written to a bitmap by
    % 'finalize_to_bitmap/1'.
    %
:- func new_bitmap_builder(num_bytes) = write_buffer.
:- mode new_bitmap_builder(in) = out is det.

    % How many bits to be written does the buffer contain?
    %
:- func num_buffered_bits(write_buffer(_, _)) = num_bits.
:- mode num_buffered_bits(write_buffer_ui) = out is det.

    % Return how many bits need to be written to get to a byte boundary
    % in the output stream.
    %
:- func num_bits_to_byte_boundary(write_buffer(_, _)) = num_bits.
:- mode num_bits_to_byte_boundary(write_buffer_ui) = out is det.

    % Write a bit to the buffer.
    %
:- pred put_bit(bool, write_buffer(Stream, State), write_buffer(Stream, State))
    <= stream.writer(Stream, bitmap.slice, State).
:- mode put_bit(in, write_buffer_di, write_buffer_uo) is det.

    % Write the given number of low-order bits from an int to the buffer.
    % The number of bits must be less than int.bits_per_int.
    %
:- pred put_bits(word, num_bits, write_buffer(Stream, State),
    write_buffer(Stream, State))
    <= stream.writer(Stream, bitmap.slice, State).
:- mode put_bits(in, in, write_buffer_di, write_buffer_uo) is det.

    % Write the eight low-order bits from an int to the buffer.
    % The number of bits must be less than int.bits_per_int.
    %
:- pred put_byte(word, write_buffer(Stream, State),
    write_buffer(Stream, State))
    <= stream.writer(Stream, bitmap.slice, State).
:- mode put_byte(in, write_buffer_di, write_buffer_uo) is det.

    % Write bits from a bitmap to the buffer.
    % The buffer does not keep a reference to the bitmap.
    %
:- pred put_bitmap(bitmap, write_buffer(Stream, State),
    write_buffer(Stream, State))
    <= stream.writer(Stream, bitmap.slice, State).
```

```
:- mode put_bitmap(bitmap_ui, write_buffer_di, write_buffer_uo) is det.

:- pred put_bitmap(bitmap, bit_index, num_bits,
    write_buffer(Stream, State), write_buffer(Stream, State))
    <= stream.writer(Stream, bitmap.slice, State).
:- mode put_bitmap(bitmap_ui, in, in, write_buffer_di, write_buffer_uo) is det.

    % Flush all complete bytes in the buffer to the output stream.
    % If there is an incomplete final byte it will remain unwritten
    % in the buffer.
    %
:- pred flush(write_buffer(Stream, State), write_buffer(Stream, State))
    <= stream.writer(Stream, bitmap.slice, State).
:- mode flush(write_buffer_di, write_buffer_uo) is det.

    % Pad the buffered data out to a byte boundary, flush it to
    % the output stream, then return the Stream and State.
    %
:- pred finalize(write_buffer(Stream, State), Stream, State)
    <= stream.writer(Stream, bitmap.slice, State).
:- mode finalize(write_buffer_di, out, uo) is det.

    % Copy the data from a non-streamed write_buffer to a bitmap.
    % The output is not padded to an even number of bits.
    %
:- func finalize_to_bitmap(write_buffer) = bitmap.
:- mode finalize_to_bitmap(write_buffer_di) = bitmap_uo is det.

%---------------------------------------------------%
```

10 bitmap

```
%---------------------------------------------------%
% vim: ts=4 sw=4 et ft=mercury
%---------------------------------------------------%
% Copyright (C) 2001-2002, 2004-2007, 2009-2011 The University of Melbourne
% Copyright (C) 2013-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: bitmap.m.
% Main author: rafe, stayl.
% Stability: low.
%
% Efficient bitmap implementation.
%
```

```
% CAVEAT: the user is referred to the documentation in the header
% of array.m regarding programming with unique objects (the compiler
% does not currently recognise them, hence we are forced to use
% non-unique modes until the situation is rectified; this places
% a small burden on the programmer to ensure the correctness of his
% code that would otherwise be assured by the compiler.)
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module bitmap.
:- interface.

:- import_module bool.
:- import_module list.

%-----------------------------------------------------%

    % Type 'bitmap' is equivalent to 'array(bool)', but is implemented
    % much more efficiently. Accessing bitmaps as if they were an array
    % of eight bit bytes is especially efficient.
    %
    % See runtime/mercury_types.h for the definition of MR_BitmapPtr for
    % use in foreign code.
    %
    % Comparison of bitmaps first compares the size, if the size is equal
    % then each bit in turn is compared starting from bit zero.
    %
:- type bitmap.

:- inst bitmap == ground.
:- inst uniq_bitmap == bitmap.  % XXX should be unique
:- mode bitmap_di == in(uniq_bitmap). % XXX should be di
:- mode bitmap_uo == out(uniq_bitmap).
:- mode bitmap_ui == in(uniq_bitmap).

    % The exception thrown for any error.
    %
:- type bitmap_error
    --->    bitmap_error(string).

%-----------------------------------------------------%

:- type bit_index == int.
:- type byte_index == int.
:- type num_bits == int.
:- type num_bytes == int.
```

```
    % 8 bits stored in the least significant bits of the integer.
    %
:- type byte == int.

    % An integer interpreted as a vector of int.bits_per_int bits.
    %
:- type word == int.

%-----------------------------------------------------%

    % init(N, B) creates a bitmap of size N (indexed 0 .. N-1)
    % setting each bit if B = yes and clearing each bit if B = no.
    % An exception is thrown if N is negative.
    %
:- func init(num_bits::in, bool::in) = (bitmap::bitmap_uo) is det.

    % A synonym for init(N, no).
    %
:- func init(num_bits::in) = (bitmap::bitmap_uo) is det.

    % Is the given bit number in range?
    %
:- pred in_range(bitmap, bit_index).
% :- mode in_range(bitmap_ui, in) is semidet.
:- mode in_range(in, in) is semidet.

    % Is the given byte number in range?
    %
:- pred byte_in_range(bitmap, byte_index).
% :- mode byte_in_range(bitmap_ui, in) is semidet.
:- mode byte_in_range(in, in) is semidet.

    % Return the number of bits in a bitmap.
    %
:- func num_bits(bitmap) = num_bits.
% :- mode num_bits(bitmap_ui) = out is det.
:- mode num_bits(in) = out is det.

    % Return the number of bytes in a bitmap, failing if the bitmap
    % has a partial final byte.
    %
:- func num_bytes(bitmap) = num_bytes.
% :- mode num_bytes(bitmap_ui) = out is semidet.
:- mode num_bytes(in) = out is semidet.

    % As above, but throw an exception if the bitmap has a partial final byte.
```

```
    %
:- func det_num_bytes(bitmap) = num_bytes.
% :- mode det_num_bytes(bitmap_ui) = out is det.
:- mode det_num_bytes(in) = out is det.

    % Return the number of bits in a byte (always 8).
    %
:- func bits_per_byte = int.

    % is_empty(Bitmap):
    % True iff Bitmap is a bitmap containing zero bits.
    %
:- pred is_empty(bitmap).
%:- mode is_empty(bitmap_ui) is semidet.
:- mode is_empty(in) is semidet.

%-----------------------------------------------%
%
% Get or set the given bit.
% The unsafe versions do not check whether the bit is in range.
%

:- func bitmap      ^ bit(bit_index)    = bool.
% :- mode bitmap_ui ^ bit(in)           = out is det.
:- mode in          ^ bit(in)           = out is det.

:- func bitmap      ^ unsafe_bit(bit_index) = bool.
% :- mode bitmap_ui ^ unsafe_bit(in)        = out is det.
:- mode in          ^ unsafe_bit(in)        = out is det.

:- func (bitmap     ^ bit(bit_index)    := bool)    = bitmap.
:- mode (bitmap_di  ^ bit(in)           := in)      = bitmap_uo is det.

:- func (bitmap     ^ unsafe_bit(bit_index) := bool) = bitmap.
:- mode (bitmap_di  ^ unsafe_bit(in)        := in)   = bitmap_uo is det.

%-----------------------------------------------%
%
% Bitmap ^ bits(OffSet, NumBits) = Word.
% The low order bits of Word contain the NumBits bits of BitMap
% starting at OffSet.
% 0 =< NumBits =< int.bits_per_int.
%

:- func bitmap      ^ bits(bit_index, num_bits) = word.
% :- mode bitmap_ui ^ bits(in, in)              = out is det.
:- mode in          ^ bits(in, in)              = out is det.
```

```
:- func bitmap      ^ unsafe_bits(bit_index, num_bits)  = word.
% :- mode bitmap_ui ^ unsafe_bits(in, in)               = out is det.
:- mode in          ^ unsafe_bits(in, in)               = out is det.

:- func (bitmap     ^ bits(bit_index, num_bits) := word) = bitmap.
:- mode (bitmap_di  ^ bits(in, in)              := in)   = bitmap_uo is det.

:- func (bitmap     ^ unsafe_bits(bit_index, num_bits) := word) = bitmap.
:- mode (bitmap_di  ^ unsafe_bits(in, in)              := in)   = bitmap_uo
    is det.

%-----------------------------------------------%
%
% BM ^ byte(ByteNumber)
% Get or set the given numbered byte (multiply ByteNumber by
% bits_per_byte to get the bit index of the start of the byte).
%
% The bits are stored in or taken from the least significant bits of an int.
% The safe versions will throw an exception if the given ByteNumber is out of
% bounds.  Final partial bytes are out of bounds.  The unsafe versions do not
% check whether the byte is in range.
%

:- func bitmap      ^ byte(byte_index) = byte.
% :- mode bitmap_ui ^ byte(in) = out is det.
:- mode in          ^ byte(in) = out is det.

:- func bitmap      ^ unsafe_byte(byte_index)   = byte.
% :- mode bitmap_ui ^ unsafe_byte(in)           = out is det.
:- mode in          ^ unsafe_byte(in)           = out is det.

:- func (bitmap     ^ byte(byte_index)  := byte) = bitmap.
:- mode (bitmap_di  ^ byte(in)          := in)   = bitmap_uo is det.

:- func (bitmap     ^ unsafe_byte(byte_index)   := byte) = bitmap.
:- mode (bitmap_di  ^ unsafe_byte(in)           := in)   = bitmap_uo is det.

%
% Versions of the above that set or take uint8 values instead of a byte stored
% in the least significant bits of an int.
%

:- func get_uint8(bitmap, byte_index) = uint8.
%:- mode get_uint8(bitmap_ui, in) = out is det.
:- mode get_uint8(in, in) = out is det.
```

```
:- func unsafe_get_uint8(bitmap, byte_index) = uint8.
%:- mode unsafe_get_uint8(bitmap_ui, in) = out is det.
:- mode unsafe_get_uint8(in, in) = out is det.

:- pred set_uint8(byte_index::in, uint8::in,
    bitmap::bitmap_di, bitmap::bitmap_uo) is det.

:- pred unsafe_set_uint8(byte_index::in, uint8::in,
    bitmap::bitmap_di, bitmap::bitmap_uo) is det.

%-----------------------------------------------%

    % Flip the given bit.
    %
:- func flip(bitmap, bit_index) = bitmap.
:- mode flip(bitmap_di, in) = bitmap_uo is det.

:- pred flip(bit_index, bitmap, bitmap).
:- mode flip(in, bitmap_di, bitmap_uo) is det.

%-----------------------------------------------%
%
% Variations that might be slightly more efficient by not
% converting bits to bool.
%

:- func set(bitmap, bit_index) = bitmap.
:- mode set(bitmap_di, in) = bitmap_uo is det.

:- pred set(bit_index, bitmap, bitmap).
:- mode set(in, bitmap_di, bitmap_uo) is det.

:- func clear(bitmap, bit_index) = bitmap.
:- mode clear(bitmap_di, in) = bitmap_uo is det.

:- pred clear(bit_index, bitmap, bitmap).
:- mode clear(in, bitmap_di, bitmap_uo) is det.

    % is_set(BM, I) and is_clear(BM, I) succeed iff bit I in BM
    % is set or clear respectively.
    %
:- pred is_set(bitmap, bit_index).
% :- mode is_set(bitmap_ui, in) is semidet.
:- mode is_set(in, in) is semidet.

:- pred is_clear(bitmap, bit_index).
% :- mode is_clear(bitmap_ui, in) is semidet.
```

```
:- mode is_clear(in, in) is semidet.

%-----------------------------------------------------%
%
% Unsafe versions of the above: if the index is out of range
% then behaviour is undefined and bad things are likely to happen.
%

:- func unsafe_flip(bitmap, bit_index) = bitmap.
:- mode unsafe_flip(bitmap_di, in) = bitmap_uo is det.

:- pred unsafe_flip(bit_index, bitmap, bitmap).
:- mode unsafe_flip(in, bitmap_di, bitmap_uo) is det.

:- func unsafe_set(bitmap, bit_index) = bitmap.
:- mode unsafe_set(bitmap_di, in) = bitmap_uo is det.

:- pred unsafe_set(bit_index, bitmap, bitmap).
:- mode unsafe_set(in, bitmap_di, bitmap_uo) is det.

:- func unsafe_clear(bitmap, bit_index) = bitmap.
:- mode unsafe_clear(bitmap_di, in) = bitmap_uo is det.

:- pred unsafe_clear(bit_index, bitmap, bitmap).
:- mode unsafe_clear(in, bitmap_di, bitmap_uo) is det.

:- pred unsafe_is_set(bitmap, bit_index).
% :- mode unsafe_is_set(bitmap_ui, in) is semidet.
:- mode unsafe_is_set(in, in) is semidet.

:- pred unsafe_is_clear(bitmap, bit_index).
% :- mode unsafe_is_clear(bitmap_ui, in) is semidet.
:- mode unsafe_is_clear(in, in) is semidet.

%-----------------------------------------------------%

    % Create a new copy of a bitmap.
    %
:- func copy(bitmap) = bitmap.
% :- mode copy(bitmap_ui) = bitmap_uo is det.
:- mode copy(in) = bitmap_uo is det.

    % resize(BM, N, B) resizes bitmap BM to have N bits; if N is
    % smaller than the current number of bits in BM then the excess
    % are discarded. If N is larger than the current number of bits
    % in BM then the new bits are set if B = yes and cleared if
    % B = no.
```

```
    %
:- func resize(bitmap, num_bits, bool) = bitmap.
:- mode resize(bitmap_di, in, in) = bitmap_uo is det.

    % Shrink a bitmap without copying it into a smaller memory allocation.
    %
:- func shrink_without_copying(bitmap, num_bits) = bitmap.
:- mode shrink_without_copying(bitmap_di, in) = bitmap_uo is det.

%-----------------------------------------------------%

    % Slice = slice(BM, StartIndex, NumBits)
    %
    % A bitmap slice represents the sub-range of a bitmap of NumBits bits
    % starting at bit index StartIndex. Throws an exception if the slice
    % is not within the bounds of the bitmap.
    %
:- type slice.
:- func slice(bitmap, bit_index, num_bits) = bitmap.slice.

    % As above, but use byte indices.
    %
:- func byte_slice(bitmap, byte_index, num_bytes) = slice.

    % Access functions for slices.
    %
:- func slice ^ slice_bitmap = bitmap.
:- func slice ^ slice_start_bit_index = bit_index.
:- func slice ^ slice_num_bits = num_bits.

    % As above, but return byte indices, throwing an exception if
    % the slice doesn't start and end on a byte boundary.
    %
:- func slice ^ slice_start_byte_index = byte_index.
:- func slice ^ slice_num_bytes = num_bytes.

%-----------------------------------------------------%
%
% Set operations; for binary operations the second argument is altered
% in all cases. The input bitmaps must have the same size.
%

:- func complement(bitmap) = bitmap.
:- mode complement(bitmap_di) = bitmap_uo is det.

:- func union(bitmap, bitmap) = bitmap.
% :- mode union(bitmap_ui, bitmap_di) = bitmap_uo is det.
```

```
:- mode union(in, bitmap_di) = bitmap_uo is det.

:- func intersect(bitmap, bitmap) = bitmap.
% :- mode intersect(bitmap_ui, bitmap_di) = bitmap_uo is det.
:- mode intersect(in, bitmap_di) = bitmap_uo is det.

:- func difference(bitmap, bitmap) = bitmap.
% :- mode difference(bitmap_ui, bitmap_di) = bitmap_uo is det.
:- mode difference(in, bitmap_di) = bitmap_uo is det.

:- func xor(bitmap, bitmap) = bitmap.
% :- mode xor(bitmap_ui, bitmap_di) = bitmap_uo is det.
:- mode xor(in, bitmap_di) = bitmap_uo is det.

%-----------------------------------------------------%

    % Condense a list of bitmaps into a single bitmap.
    %
:- func append_list(list(bitmap)) = bitmap.
:- mode append_list(in) = bitmap_uo is det.

%-----------------------------------------------------%
%
% Operations to copy part of a bitmap.
%

    % copy_bits(SrcBM, SrcStartBit, DestBM, DestStartBit, NumBits)
    %
    % Overwrite NumBits bits in DestBM starting at DestStartBit with
    % the NumBits bits starting at SrcStartBit in SrcBM.
    %
:- func copy_bits(bitmap, bit_index, bitmap, bit_index, num_bits) = bitmap.
% :- mode copy_bits(bitmap_ui, in, bitmap_di, in, in) = bitmap_uo is det.
:- mode copy_bits(in, in, bitmap_di, in, in) = bitmap_uo is det.

    % copy_bits_in_bitmap(BM, SrcStartBit, DestStartBit, NumBits)
    %
    % Overwrite NumBits bits starting at DestStartBit with the NumBits
    % bits starting at SrcStartBit in the same bitmap.
    %
:- func copy_bits_in_bitmap(bitmap, bit_index, bit_index, num_bits) = bitmap.
:- mode copy_bits_in_bitmap(bitmap_di, in, in, in) = bitmap_uo is det.

    % copy_bytes(SrcBM, SrcStartByte, DestBM, DestStartByte, NumBytes)
    %
    % Overwrite NumBytes bytes in DestBM starting at DestStartByte with
    % the NumBytes bytes starting at SrcStartByte in SrcBM.
```

```
    %
:- func copy_bytes(bitmap, byte_index, bitmap, byte_index, num_bytes) = bitmap.
% :- mode copy_bytes(bitmap_ui, in, bitmap_di, in, in) = bitmap_uo is det.
:- mode copy_bytes(in, in, bitmap_di, in, in) = bitmap_uo is det.

    % copy_bytes_in_bitmap(BM, SrcStartByte, DestStartByte, NumBytes)
    %
    % Overwrite NumBytes bytes starting at DestStartByte with the NumBytes
    % bytes starting at SrcStartByte in the same bitmap.
    %
:- func copy_bytes_in_bitmap(bitmap, byte_index,
    byte_index, num_bytes) = bitmap.
:- mode copy_bytes_in_bitmap(bitmap_di, in, in, in) = bitmap_uo is det.

%-----------------------------------------------%

    % Convert a bitmap to a string of the form "<length:hex digits>",
    % e.g. "<24:10AFBD>".
    %
:- func to_string(bitmap) = string.
% :- mode to_string(bitmap_ui) = out is det.
:- mode to_string(in) = out is det.

    % Convert a string created by to_string back into a bitmap.
    % Fails if the string is not of the form created by to_string.
    %
:- func from_string(string) = bitmap.
:- mode from_string(in) = bitmap_uo is semidet.

    % As above, but throws an exception instead of failing.
    %
:- func det_from_string(string) = bitmap.
:- mode det_from_string(in) = bitmap_uo is det.

    % Convert a bitmap to a string of '1' and '0' characters, where
    % the bytes are separated by '.'.
    %
:- func to_byte_string(bitmap) = string.
% :- mode to_byte_string(bitmap_ui) = out is det.
:- mode to_byte_string(in) = out is det.

%-----------------------------------------------%

    % Compute a hash function for a bitmap.
    %
:- func hash(bitmap) = int.
% :- mode hash(bitmap_ui) = out is det.
```

```
:- mode hash(in) = out is det.

%-----------------------------------------------------%
%-----------------------------------------------------%
```

11 bool

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 1996-1997,2000,2002-2007,2009-2010 The University of Melbourne.
% Copyright (C) 2014-2016, 2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: bool.m.
% Main authors: fjh, zs.
% Stability: medium to high.
%
% This module exports the boolean type 'bool' and some operations on bools.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module bool.
:- interface.

:- import_module enum.
:- import_module list.

%-----------------------------------------------------%

    % The boolean type.
    % Unlike most languages, we use 'yes' and 'no' as boolean constants
    % rather than 'true' and 'false'.  This is to avoid confusion
    % with the predicates 'true' and 'fail'.
:- type bool
    --->    no
    ;       yes.

:- instance enum(bool).

    % not(A) = yes iff A = no.
    %
:- func not(bool) = bool.
```

```
:- pred not(bool::in, bool::out) is det.

    % or(A, B) = yes iff A = yes, or B = yes, or both.
    %
:- func or(bool, bool) = bool.
:- pred or(bool::in, bool::in, bool::out) is det.

    % xor(A, B) = yes iff A = yes, or B = yes, but not both.
    %
:- func xor(bool, bool) = bool.

    % and(A, B) = yes iff A = yes and B = yes.
    %
:- func and(bool, bool) = bool.
:- pred and(bool::in, bool::in, bool::out) is det.

    % or_list(As) = yes iff there exists an element of As equal to yes.
    % (Note that or_list([]) = no.)
    %
:- func or_list(list(bool)) = bool.
:- pred or_list(list(bool)::in, bool::out) is det.

    % and_list(As) = yes iff every element of As is equal to yes.
    % (Note that and_list([]) = yes.)
    %
:- func and_list(list(bool)) = bool.
:- pred and_list(list(bool)::in, bool::out) is det.

    % pred_to_bool(P) = (if P then yes else no).
    %
:- func pred_to_bool((pred)::((pred) is semidet)) = (bool::out) is det.

%---------------------------------------------------%
%---------------------------------------------------%
```

12 bt_array

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 1997,1999-2000,2002-2003,2005-2006 The University of Melbourne.
% Copyright (C) 2014-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
```

```
% File: bt_array.m
% Main author: bromage.
% Stability: medium-low
%
% This file contains a set of predicates for generating an manipulating a
% bt_array data structure. This implementation allows O(log n) access and
% update time, and does not require the bt_array to be unique. If you need
% O(1) access/update time, use the array datatype instead. (‘bt_array’ is
% supposed to stand for either "binary tree array" or "backtrackable array".)
%
% Implementation obscurity: This implementation is biased towards larger
% indices. The access/update time for a bt_array of size N with index I is
% actually O(log(N-I)). The reason for this is so that the resize operations
% can be optimised for a (possibly very) common case, and to exploit
% accumulator recursion in some operations. See the documentation of resize
% and shrink for more details.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module bt_array.
:- interface.
:- import_module list.

:- type bt_array(T).

%-----------------------------------------------------%

    % make_empty_array(Low, Array) is true iff Array is a
    % bt_array of size zero starting at index Low.
    %
:- pred make_empty_array(int::in, bt_array(T)::out) is det.
:- func make_empty_array(int) = bt_array(T).

    % init(Low, High, Init, Array) is true iff Array is a
    % bt_array with bounds from Low to High whose elements each equal Init.
    %
:- pred init(int::in, int::in, T::in, bt_array(T)::out) is det.
:- func init(int, int, T) = bt_array(T).

%-----------------------------------------------------%

    % Returns the lower bound of the array.
    %
:- pred min(bt_array(_T)::in, int::out) is det.
:- func min(bt_array(_T)) = int.
```

```
    % Returns the upper bound of the array.
    % Returns lower bound - 1 for an empty array.
    %
:- pred max(bt_array(_T)::in, int::out) is det.
:- func max(bt_array(_T)) = int.

    % Returns the length of the array,
    % i.e. upper bound - lower bound + 1.
    %
:- pred size(bt_array(_T)::in, int::out) is det.
:- func size(bt_array(_T)) = int.

    % bounds(Array, Min, Max) returns the lower and upper bounds of a bt_array.
    % The upper bound will be the lower bound - 1 for an empty array.
    %
:- pred bounds(bt_array(_T)::in, int::out, int::out) is det.

    % in_bounds checks whether an index is in the bounds
    % of a bt_array.
    %
:- pred in_bounds(bt_array(_T)::in, int::in) is semidet.

%-----------------------------------------------%

    % lookup returns the Nth element of a bt_array.
    % It is an error if the index is out of bounds.
    %
:- pred lookup(bt_array(T)::in, int::in, T::out) is det.
:- func lookup(bt_array(T), int) = T.

    % semidet_lookup is like lookup except that it fails if the index is out of
    % bounds.
    %
:- pred semidet_lookup(bt_array(T)::in, int::in, T::out) is semidet.

    % set sets the nth element of a bt_array, and returns the resulting
    % bt_array. It is an error if the index is out of bounds.
    %
:- pred set(bt_array(T)::in, int::in, T::in, bt_array(T)::out) is det.
:- func set(bt_array(T), int, T) = bt_array(T).

    % set sets the nth element of a bt_array, and returns the
    % resulting bt_array (good opportunity for destructive update ;-).
    % It fails if the index is out of bounds.
    %
:- pred semidet_set(bt_array(T)::in, int::in, T::in, bt_array(T)::out)
    is semidet.
```

```
    % 'resize(BtArray0, Lo, Hi, Item, BtArray)' is true if BtArray
    % is a bt_array created by expanding or shrinking BtArray0 to fit the
    % bounds (Lo, Hi). If the new bounds are not wholly contained within
    % the bounds of BtArray0, Item is used to fill out the other places.
    %
    % Note: This operation is optimised for the case where the lower bound
    % of the new bt_array is the same as that of the old bt_array. In that
    % case, the operation takes time proportional to the absolute difference
    % in size between the two bt_arrays. If this is not the case, it may take
    % time proportional to the larger of the two bt_arrays.
    %
:- pred resize(bt_array(T)::in, int::in, int::in, T::in,
    bt_array(T)::out) is det.
:- func resize(bt_array(T), int, int, T) = bt_array(T).

    % shrink(BtArray0, Lo, Hi, Item, BtArray) is true if BtArray
    % is a bt_array created by shrinking BtArray0 to fit the bounds (Lo, Hi).
    % It is an error if the new bounds are not wholly within the bounds of
    % BtArray0.
    %
    % Note: This operation is optimised for the case where the lower bound
    % of the new bt_array is the same as that of the old bt_array. In that
    % case, the operation takes time proportional to the absolute difference
    % in size between the two bt_arrays. If this is not the case, it may take
    % time proportional to the larger of the two bt_arrays.
    %
:- pred shrink(bt_array(T)::in, int::in, int::in, bt_array(T)::out)
    is det.
:- func shrink(bt_array(T), int, int) = bt_array(T).

    % from_list(Low, List, BtArray) takes a list (of possibly zero
    % length), and returns a bt_array containing % those elements in the same
    % order that they occurred in the list. The lower bound of the new array
    % is 'Low'.
:- pred from_list(int::in, list(T)::in, bt_array(T)::out) is det.
:- func from_list(int, list(T)) = bt_array(T).

    % to_list takes a bt_array and returns a list containing
    % the elements of the bt_array in the same order that they occurred
    % in the bt_array.
    %
:- pred to_list(bt_array(T)::in, list(T)::out) is det.
:- func to_list(bt_array(T)) = list(T).

    % fetch_items takes a bt_array and a lower and upper index,
    % and places those items in the bt_array between these indices into a list.
```

```
    % It is an error if either index is out of bounds.
    %
:- pred fetch_items(bt_array(T)::in, int::in, int::in, list(T)::out)
    is det.
:- func fetch_items(bt_array(T), int, int) = list(T).

    % bsearch takes a bt_array, an element to be matched and a
    % comparison predicate and returns the position of the first occurrence
    % in the bt_array of an element which is equivalent to the given one
    % in the ordering provided. Assumes the bt_array is sorted according
    % to this ordering. Fails if the element is not present.
    %
:- pred bsearch(bt_array(T)::in, T::in,
    comparison_pred(T)::in(comparison_pred), int::out) is semidet.

    % Field selection for arrays.
    % Array ^ elem(Index) = lookup(Array, Index).
    %
:- func elem(int, bt_array(T)) = T.

    % Field update for arrays.
    % (Array ^ elem(Index) := Value) = set(Array, Index, Value).
    %
:- func 'elem :='(int, bt_array(T), T) = bt_array(T).

%-----------------------------------------------------------------------%
```

13 builtin

```
%-----------------------------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------------------------%
% Copyright (C) 1994-2007, 2010-2012 The University of Melbourne.
% Copyright (C) 2014-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------------------------%
%
% File: builtin.m.
% Main author: fjh.
% Stability: low.
%
% This file is automatically imported into every module.
% It is intended for things that are part of the language,
% but which are implemented just as normal user-level code
% rather than with special coding in the compiler.
```

```
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module builtin.
:- interface.

%-----------------------------------------------------%
%
% Types.
%

% The types ‘character’, ‘int’, ‘int8’, ‘int16’, ‘int32’, ‘int64’,
% ‘uint’, ‘uint8’, ‘uint16’, ‘uint32’, ‘uint64’, ‘float’, and ‘string’,
% and tuple types ‘{}’, ‘{T}’, ‘{T1, T2}’, ...
% and the types ‘pred’, ‘pred(T)’, ‘pred(T1, T2)’, ‘pred(T1, T2, T3)’, ...
% and ‘func(T1) = T2’, ‘func(T1, T2) = T3’, ‘func(T1, T2, T3) = T4’, ...
% are builtin and are implemented using special code in the type-checker.

    % The type c_pointer can be used by predicates that use the C interface.
    %
    % NOTE: We *strongly* recommend using a ‘foreign_type’ pragma instead
    % of using this type.
    %
:- type c_pointer.

%-----------------------------------------------------%
%
% Insts.
%

% The standard insts ‘free’, ‘ground’, and ‘bound(...)’ are builtin and are
% implemented using special code in the parser and mode-checker.
%
% So are the standard unique insts ‘unique’, ‘unique(...)’, ‘mostly_unique’,
% ‘mostly_unique(...)’, and ‘clobbered’.
%
% Higher-order predicate insts ‘pred(<modes>) is <detism>’
% and higher-order function insts ‘func(<modes>) = <mode> is <detism>’
% are also builtin.
%
% The ‘any’ inst used for constraint solver interfaces is builtin and so are
% its higher-order variants: ‘any_pred(<modes>) is <detism>’ and
% ‘any_func(<modes>) = <mode> is <detism>’.

    % The name ‘dead’ is allowed as a synonym for ‘clobbered’.
    % Similarly, ‘mostly_dead’ is a synonym for ‘mostly_clobbered’.
```

```
    %
:- inst dead == clobbered.
:- inst mostly_dead == mostly_clobbered.

%-----------------------------------------------%
%
% Standard modes.
%

:- mode unused == free >> free.

    % This mode is deprecated, use 'out' instead.
    %
:- mode output == free >> ground.

    % This mode is deprecated, use 'in' instead.
    %
:- mode input  == ground >> ground.

:- mode in  == ground >> ground.
:- mode out == free >> ground.

:- mode in(Inst)  == Inst >> Inst.
:- mode out(Inst) == free >> Inst.
:- mode di(Inst)  == Inst >> clobbered.
:- mode mdi(Inst) == Inst >> mostly_clobbered.

%-----------------------------------------------%
%
% Unique modes.
%

% XXX These are still not fully implemented.

    % unique output
    %
:- mode uo == free >> unique.

    % unique input
    %
:- mode ui == unique >> unique.

    % destructive input
    %
:- mode di == unique >> clobbered.

%-----------------------------------------------%
```

```
%
% "Mostly" unique modes.
%

% Unique except that they may be referenced again on backtracking.

    % mostly unique output
    %
:- mode muo == free >> mostly_unique.

    % mostly unique input
    %
:- mode mui == mostly_unique >> mostly_unique.

    % mostly destructive input
    %
:- mode mdi == mostly_unique >> mostly_clobbered.

%-----------------------------------------------%
%
% Dynamic modes.
%

    % Solver type modes.
    %
:- mode ia == any >> any.
:- mode oa == free >> any.

%-----------------------------------------------%
%
% Predicates.
%

    % copy/2 makes a deep copy of a data structure.
    % The resulting copy is a 'unique' value, so you can use
    % destructive update on it.
    %
:- pred copy(T, T).
:- mode copy(ui, uo) is det.
:- mode copy(in, uo) is det.

    % unsafe_promise_unique/2 is used to promise the compiler that you
    % have a 'unique' copy of a data structure, so that you can use
    % destructive update. It is used to work around limitations in
    % the current support for unique modes.
    % 'unsafe_promise_unique(X, Y)' is the same as 'Y = X' except that
    % the compiler will assume that 'Y' is unique.
```

```
    %
    % Note that misuse of this predicate may lead to unsound results:
    % if there is more than one reference to the data in question,
    % i.e. it is not 'unique', then the behaviour is undefined.
    % (If you lie to the compiler, the compiler will get its revenge!)
    %
:- func unsafe_promise_unique(T::in) = (T::uo) is det.
:- pred unsafe_promise_unique(T::in, T::uo) is det.

    % A synonym for fail/0; this name is more in keeping with Mercury's
    % declarative style rather than its Prolog heritage.
    %
:- pred false is failure.

%-----------------------------------------------%

    % This function is useful for converting polymorphic non-solver type
    % values with inst any to inst ground (the compiler recognises that
    % inst any is equivalent to ground for non-polymorphic non-solver
    % type values.)
    %
    % Do not call this on solver type values unless you are *absolutely sure*
    % that they are semantically ground.
    %
:- func unsafe_cast_any_to_ground(T::ia) = (T::out) is det.

%-----------------------------------------------%

    % A call to the function 'promise_only_solution(Pred)' constitutes a
    % promise on the part of the caller that 'Pred' has at most one
    % solution, i.e. that 'not some [X1, X2] (Pred(X1), Pred(X2), X1 \=
    % X2)'. 'promise_only_solution(Pred)' presumes that this assumption is
    % satisfied, and returns the X for which Pred(X) is true, if there is
    % one.
    %
    % You can use 'promise_only_solution' as a way of introducing
    % 'cc_multi' or 'cc_nondet' code inside a 'det' or 'semidet' procedure.
    %
    % Note that misuse of this function may lead to unsound results: if the
    % assumption is not satisfied, the behaviour is undefined. (If you lie
    % to the compiler, the compiler will get its revenge!)
    %
    % NOTE: This function is deprecated and will be removed in a future
    % release. Use a 'promise_equivalent_solutions' goal instead.
    %
:- pragma obsolete(promise_only_solution/1).
:- func promise_only_solution(pred(T)) = T.
```

```
:- mode promise_only_solution(pred(out) is cc_multi) = out is det.
:- mode promise_only_solution(pred(uo) is cc_multi) = uo is det.
:- mode promise_only_solution(pred(out) is cc_nondet) = out is semidet.
:- mode promise_only_solution(pred(uo) is cc_nondet) = uo is semidet.

    % 'promise_only_solution_io' is like 'promise_only_solution', but for
    % procedures with unique modes (e.g. those that do IO).
    %
    % A call to 'promise_only_solution_io(P, X, IO0, IO)' constitutes a
    % promise on the part of the caller that for the given IO0, there is
    % only one value of 'X' and 'IO' for which 'P(X, IO0, IO)' is true.
    % 'promise_only_solution_io(P, X, IO0, IO)' presumes that this
    % assumption is satisfied, and returns the X and IO for which 'P(X,
    % IO0, IO)' is true.
    %
    % Note that misuse of this predicate may lead to unsound results: if
    % the assumption is not satisfied, the behaviour is undefined. (If you
    % lie to the compiler, the compiler will get its revenge!)
    %
    % NOTE: This predicate is deprecated and will be removed in a future
    % release. Use a 'promise_equivalent_solutions' goal instead.
    %
:- pragma obsolete(promise_only_solution_io/4).
:- pred promise_only_solution_io(
    pred(T, IO, IO)::in(pred(out, di, uo) is cc_multi), T::out,
    IO::di, IO::uo) is det.

%-----------------------------------------------------%

    % unify(X, Y) is true iff X = Y.
    %
:- pred unify(T::in, T::in) is semidet.

    % For use in defining user-defined unification predicates.
    % The relation defined by a value of type 'unify', must be an
    % equivalence relation; that is, it must be symmetric, reflexive,
    % and transitive.
    %
:- type unify(T) == pred(T, T).
:- inst unify == (pred(in, in) is semidet).

:- type comparison_result
    --->    (=)
    ;       (<)
    ;       (>).

    % compare(Res, X, Y) binds Res to =, <, or > depending on whether
```

```
    % X is =, <, or > Y in the standard ordering.
    %
:- pred compare(comparison_result, T, T).
:- mode compare(uo, in, in) is det.
:- mode compare(uo, ui, ui) is det.
:- mode compare(uo, ui, in) is det.
:- mode compare(uo, in, ui) is det.

    % For use in defining user-defined comparison predicates.
    % For a value ‘ComparePred’ of type ‘compare’, the following
    % conditions must hold:
    %
    % - the relation
    %   compare_eq(X, Y) :- ComparePred((=), X, Y).
    %   must be an equivalence relation; that is, it must be symmetric,
    %   reflexive, and transitive.
    %
    % - the relations
    %   compare_leq(X, Y) :-
    %       ComparePred(R, X, Y), (R = (=) ; R = (<)).
    %   compare_geq(X, Y) :-
    %       ComparePred(R, X, Y), (R = (=) ; R = (>)).
    %   must be total order relations: that is they must be antisymmetric,
    %   reflexive and transitive.
    %
:- type compare(T) == pred(comparison_result, T, T).
:- inst compare == (pred(uo, in, in) is det).

    % ordering(X, Y) = R <=> compare(R, X, Y)
    %
:- func ordering(T, T) = comparison_result.

    % The standard inequalities defined in terms of compare/3.
    % XXX The ui modes are commented out because they don’t yet work properly.
    %
:- pred T  @<  T.
:- mode in @< in is semidet.
% :- mode ui @< in is semidet.
% :- mode in @< ui is semidet.
% :- mode ui @< ui is semidet.

:- pred T  @=<  T.
:- mode in @=< in is semidet.
% :- mode ui @=< in is semidet.
% :- mode in @=< ui is semidet.
% :- mode ui @=< ui is semidet.
```

```
:- pred T  @>  T.
:- mode in @> in is semidet.
% :- mode ui @> in is semidet.
% :- mode in @> ui is semidet.
% :- mode ui @> ui is semidet.

:- pred T  @>=  T.
:- mode in @>= in is semidet.
% :- mode ui @>= in is semidet.
% :- mode in @>= ui is semidet.
% :- mode ui @>= ui is semidet.

    % Values of types comparison_pred/1 and comparison_func/1 are used
    % by predicates and functions which depend on an ordering on a given
    % type, where this ordering is not necessarily the standard ordering.
    % In addition to the type, mode and determinism constraints, a
    % comparison predicate C is expected to obey two other laws.
    % For all X, Y and Z of the appropriate type, and for all
    % comparison_results R:
    %   1) C(X, Y, (>)) if and only if C(Y, X, (<))
    %   2) C(X, Y, R) and C(Y, Z, R) implies C(X, Z, R).
    % Comparison functions are expected to obey analogous laws.
    %
    % Note that binary relations <, > and = can be defined from a
    % comparison predicate or function in an obvious way. The following
    % facts about these relations are entailed by the above constraints:
    % = is an equivalence relation (not necessarily the usual equality),
    % and the equivalence classes of this relation are totally ordered
    % with respect to < and >.
    %
:- type comparison_pred(T) == pred(T, T, comparison_result).
:- inst comparison_pred(I) == (pred(in(I), in(I), out) is det).
:- inst comparison_pred == comparison_pred(ground).

:- type comparison_func(T) == (func(T, T) = comparison_result).
:- inst comparison_func(I) == (func(in(I), in(I)) = out is det).
:- inst comparison_func == comparison_func(ground).

% In addition, the following predicate-like constructs are builtin:
%
%   :- pred (T = T).
%   :- pred (T \= T).
%   :- pred (pred , pred).
%   :- pred (pred ; pred).
%   :- pred (\+ pred).
%   :- pred (not pred).
%   :- pred (pred -> pred).
```

```
%   :- pred (if pred then pred).
%   :- pred (if pred then pred else pred).
%   :- pred (pred => pred).
%   :- pred (pred <= pred).
%   :- pred (pred <=> pred).
%
%   (pred -> pred ; pred).
%   some Vars pred
%   all Vars pred
%   call/N

%-----------------------------------------------------%

    % 'semidet_succeed' is exactly the same as 'true', except that
    % the compiler thinks that it is semi-deterministic. You can use
    % calls to 'semidet_succeed' to suppress warnings about determinism
    % declarations that could be stricter.
    %
:- pred semidet_succeed is semidet.

    % 'semidet_fail' is like 'fail' except that its determinism is semidet
    % rather than failure.
    %
:- pred semidet_fail is semidet.

    % A synonym for semidet_succeed/0.
    %
:- pred semidet_true is semidet.

    % A synonym for semidet_fail/0.
    %
:- pred semidet_false is semidet.

    % 'cc_multi_equal(X, Y)' is the same as 'X = Y' except that it
    % is cc_multi rather than det.
    %
:- pred cc_multi_equal(T, T).
:- mode cc_multi_equal(di, uo) is cc_multi.
:- mode cc_multi_equal(in, out) is cc_multi.

    % 'impure_true' is like 'true' except that it is impure.
    %
:- impure pred impure_true is det.

    % 'semipure_true' is like 'true' except that it is semipure.
    %
:- semipure pred semipure_true is det.
```

```
%-----------------------------------------------------%

    % dynamic_cast(X, Y) succeeds with Y = X iff X has the same ground type
    % as Y (so this may succeed if Y is of type list(int), say, but not if
    % Y is of type list(T)).
    %
:- pred dynamic_cast(T1::in, T2::out) is semidet.

%-----------------------------------------------------%
%-----------------------------------------------------%
```

14 calendar

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 2009-2010 The University of Melbourne.
% Copyright (C) 2013-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: calendar.m.
% Main authors: maclarty
% Stability: low.
%
% Proleptic Gregorian calendar utilities.
%
% The Gregorian calendar is the calendar that is currently used by most of
% the world. In this calendar, a year is a leap year if it is divisible by
% 4, but not divisible by 100. The only exception is if the year is divisible
% by 400, in which case it is a leap year. For example 1900 is not leap year,
% while 2000 is. The proleptic Gregorian calendar is an extension of the
% Gregorian calendar backward in time to before it was first introduced in
% 1582.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module calendar.
:- interface.

:- import_module io.

%-----------------------------------------------------%
```

```
    % A point on the Proleptic Gregorian calendar, to the nearest microsecond.
    %
:- type date.

    % A more meaningful name for the above.
    %
:- type date_time == date.

    % Date components.
    %
:- type year == int.         % Year 0 is 1 BC, -1 is 2 BC, etc.
:- type day_of_month == int. % 1..31 depending on the month and year
:- type hour == int.         % 0..23
:- type minute == int.       % 0..59
:- type second == int.       % 0..61 (60 and 61 are for leap seconds)
:- type microsecond == int.  % 0..999999

:- type month
    --->    january
    ;       february
    ;       march
    ;       april
    ;       may
    ;       june
    ;       july
    ;       august
    ;       september
    ;       october
    ;       november
    ;       december.

:- type day_of_week
    --->    monday
    ;       tuesday
    ;       wednesday
    ;       thursday
    ;       friday
    ;       saturday
    ;       sunday.

%-----------------------------------------------------%

    % Functions to retrieve the components of a date.
    %
:- func year(date) = year.
:- func month(date) = month.
```

```
:- func day_of_month(date) = day_of_month.
:- func day_of_week(date) = day_of_week.
:- func hour(date) = hour.
:- func minute(date) = minute.
:- func second(date) = second.
:- func microsecond(date) = microsecond.

    % int_to_month(Int, Month):
    % Int is the number of Month where months are numbered from 1-12.
    %
:- pred int_to_month(int, month).
:- mode int_to_month(in, out) is semidet.
:- mode int_to_month(out, in) is det.

    % det_int_to_month(Int) returns the month corresponding to Int.
    % Throws an exception if Int is not in 1-12.
    %
:- func det_int_to_month(int) = month.

    % int_to_month(Int, Month):
    % Int is the number of Month where months are numbered from 0-11.
    %
:- pred int0_to_month(int, month).
:- mode int0_to_month(in, out) is semidet.
:- mode int0_to_month(out, in) is det.

    % det_int0_to_month(Int) returns the month corresponding to Int.
    % Throws an exception if Int is not in 0-11.
    %
:- func det_int0_to_month(int) = month.

    % month_to_int(Month) returns the number of Month where months are
    % numbered from 1-12.
    %
:- func month_to_int(month) = int.

    % month_to_int0(Month) returns the number of Month where months are
    % numbered from 0-11.
    %
:- func month_to_int0(month) = int.

%-----------------------------------------------%

    % init_date(Year, Month, Day, Hour, Minute, Second, MicroSecond, Date):
    % Initialize a new date. Fails if the given date is invalid.
    %
:- pred init_date(year::in, month::in, day_of_month::in, hour::in,
```

```
    minute::in, second::in, microsecond::in, date::out) is semidet.

    % Same as above, but throws an exception if the date is invalid.
    %
:- func det_init_date(year, month, day_of_month, hour, minute, second,
    microsecond) = date.

    % Retrieve all the components of a date.
    %
:- pred unpack_date(date::in,
    year::out, month::out, day_of_month::out, hour::out, minute::out,
    second::out, microsecond::out) is det.

%-----------------------------------------------------%

    % Convert a string of the form "YYYY-MM-DD HH:MM:SS.mmmmmm" to a date.
    % The microseconds component (.mmmmmm) is optional.
    %
:- pred date_from_string(string::in, date::out) is semidet.

    % Same as above, but throws an exception if the string is not a valid date.
    %
:- func det_date_from_string(string) = date.

    % Convert a date to a string of the form "YYYY-MM-DD HH:MM:SS.mmmmmm".
    % If the microseconds component of the date is zero, then the
    % ".mmmmmm" part is omitted.
    %
:- func date_to_string(date) = string.

%-----------------------------------------------------%

    % Get the current local time.
    %
:- pred current_local_time(date::out, io::di, io::uo) is det.

    % Get the current UTC time.
    %
:- pred current_utc_time(date::out, io::di, io::uo) is det.

    % Calculate the Julian day number for a date on the Gregorian calendar.
    %
:- func julian_day_number(date) = int.

    % Returns 1970/01/01 00:00:00.
    %
:- func unix_epoch = date.
```

```
    % same_date(A, B):
    % True iff A and B are equal with respect to only their date components.
    % The time components are ignored.
    %
:- pred same_date(date::in, date::in) is semidet.

%-----------------------------------------------%
%
% Durations.
%

    % A period of time measured in years, months, days, hours, minutes,
    % seconds and microseconds. Internally a duration is represented
    % using only months, days, seconds and microseconds components.
    %
:- type duration.

    % Duration components.
    %
:- type years == int.
:- type months == int.
:- type days == int.
:- type hours == int.
:- type minutes == int.
:- type seconds == int.
:- type microseconds == int.

    % Functions to retrieve duration components.
    %
:- func years(duration) = years.
:- func months(duration) = months.
:- func days(duration) = days.
:- func hours(duration) = hours.
:- func minutes(duration) = minutes.
:- func seconds(duration) = seconds.
:- func microseconds(duration) = microseconds.

    % init_duration(Years, Months, Days, Hours, Minutes,
    %   Seconds, MicroSeconds) = Duration.
    % Create a new duration. All of the components should either be
    % non-negative or non-positive (they can all be zero).
    %
:- func init_duration(years, months, days, hours, minutes, seconds,
    microseconds) = duration.

    % Retrieve all the components of a duration.
```

```
    %
:- pred unpack_duration(duration::in, years::out, months::out,
    days::out, hours::out, minutes::out, seconds::out, microseconds::out)
    is det.

    % Return the zero length duration.
    %
:- func zero_duration = duration.

    % Negate a duration.
    %
:- func negate(duration) = duration.

%-----------------------------------------------------%

    % Parse a duration string.
    %
    % The string should be of the form "PnYnMnDTnHnMnS" where each "n" is a
    % non-negative integer representing the number of years (Y), months (M),
    % days (D), hours (H), minutes (M) or seconds (S). The duration string
    % always starts with 'P' and the 'T' separates the date and time components
    % of the duration. A component may be omitted if it is zero, and the 'T'
    % separator is not required if all the time components are zero.
    % The second component may include a fraction component using a period.
    % This fraction component should not have a resolution higher than a
    % microsecond.
    %
    % For example the duration 1 year, 18 months, 100 days, 10 hours, 15
    % minutes 90 seconds and 300 microseconds can be written as:
    %   P1Y18M100DT10H15M90.0003S
    % while the duration 1 month and 2 days can be written as:
    %   P1M2D
    %
    % Note that internally the duration is represented using only months,
    % days, seconds and microseconds, so that
    % duration_to_string(det_duration_from_string("P1Y18M100DT10H15M90.0003S"))
    % will result in the string "P2Y6M100DT10H16M30.0003S".
    %
:- pred duration_from_string(string::in, duration::out) is semidet.

    % Same as above, but throws an exception if the duration string is invalid.
    %
:- func det_duration_from_string(string) = duration.

    % Convert a duration to a string using the same representation
    % parsed by duration_from_string.
    %
```

```
:- func duration_to_string(duration) = string.

%-----------------------------------------------%

    % Add a duration to a date.
    %
    % First the years and months are added to the date.
    % If this causes the day to be out of range (e.g. April 31), then it is
    % decreased until it is in range (e.g. April 30). Next the remaining
    % days, hours, minutes and seconds components are added. These could
    % in turn cause the month and year components of the date to change again.
    %
:- pred add_duration(duration::in, date::in, date::out) is det.

    % This predicate implements a partial order relation on durations.
    % DurationA is less than or equal to DurationB iff for all of the
    % dates list below, adding DurationA to the date results in a date
    % less than or equal to the date obtained by adding DurationB.
    %
    %    1696-09-01 00:00:00
    %    1697-02-01 00:00:00
    %    1903-03-01 00:00:00
    %    1903-07-01 00:00:00
    %
    % There is only a partial order on durations, because some durations
    % cannot be said to be less than, equal to or greater than another duration
    % (e.g. 1 month vs. 30 days).
    %
:- pred duration_leq(duration::in, duration::in) is semidet.

    % Get the difference between local and UTC time as a duration.
    %
    % local_time_offset(TZ, !IO) is equivalent to:
    %   current_local_time(Local, !IO),
    %   current_utc_time(UTC, !IO),
    %   TZ = duration(UTC, Local)
    % except that it is as if the calls to current_utc_time and
    % current_local_time occurred at the same instant.
    %
    % To convert UTC time to local time, add the result of local_time_offset/3
    % to UTC (using add_duration/3). To compute UTC given the local time,
    % first negate the result of local_time_offset/3 (using negate/1) and then
    % add it to the local time.
    %
:- pred local_time_offset(duration::out, io::di, io::uo) is det.

    % duration(DateA, DateB) = Duration.
```

```
    % Find the duration between two dates using a "greedy" algorithm.
    % The algorithm is greedy in the sense that it will try to maximise each
    % component in the returned duration in the following order: years, months,
    % days, hours, minutes, seconds, microseconds.
    % The returned duration is positive if DateB is after DateA and negative
    % if DateB is before DateA.
    % Any leap seconds that occurred between the two dates are ignored.
    % The dates should be in the same timezone and in the same daylight
    % savings phase. To work out the duration between dates in different
    % timezones or daylight savings phases, first convert the dates to UTC.
    %
    % If the seconds components of DateA and DateB are < 60 then
    % add_duration(DateA, duration(DateA, DateB), DateB) will hold, but
    % add_duration(DateB, negate(duration(DateA, DateB)), DateA) may not hold.
    % For example if:
    %   DateA = 2001-01-31
    %   DateB = 2001-02-28
    %   Duration = 1 month
    % then the following holds:
    %   add_duration(duration(DateA, DateB), DateA, DateB)
    % but the following does not:
    %   add_duration(negate(duration(DateA, DateB), DateB, DateA)
    % (Adding -1 month to 2001-02-28 will yield 2001-01-28).
    %
:- func duration(date, date) = duration.

    % Same as above, except that the year and month components of the
    % returned duration will always be zero. The duration will be in terms
    % of days, hours, minutes, seconds and microseconds only.
    %
:- func day_duration(date, date) = duration.

%-----------------------------------------------%
%
% Folds over ranges of dates.
%

    % foldl_days(Pred, Start, End, !Acc):
    % Calls Pred for each day in the range of dates from Start to End
    % with an accumulator.
    % Each date in the range is generated by adding a duration of one day
    % to the previous date using the add_duration/3 predicate.
    % Consequently, the time components of the dates in the range may
    % differ if the time components of the given start and end times
    % include leap seconds.
    %
:- pred foldl_days(pred(date, A, A), date, date, A, A).
```

```
:- mode foldl_days(pred(in, in, out) is det, in, in, in, out) is det.
:- mode foldl_days(pred(in, mdi, muo) is det, in, in, mdi, muo) is det.
:- mode foldl_days(pred(in, di, uo) is det, in, in, di, uo) is det.
:- mode foldl_days(pred(in, in, out) is semidet, in, in, in, out) is semidet.
:- mode foldl_days(pred(in, mdi, muo) is semidet, in, in, mdi, muo) is semidet.
:- mode foldl_days(pred(in, di, uo) is semidet, in, in, di, uo) is semidet.

    % foldl2_days(Pred, Start, End, !Acc1, !Acc2):
    % As above, but with two accumulators.
    %
:- pred foldl2_days(pred(date, A, A, B, B), date, date, A, A, B, B).
:- mode foldl2_days(pred(in, in, out, in, out) is det, in, in, in, out,
    in, out) is det.
:- mode foldl2_days(pred(in, in, out, mdi, muo) is det, in, in, in, out,
    mdi, muo) is det.
:- mode foldl2_days(pred(in, in, out, di, uo) is det, in, in, in, out,
    di, uo) is det.
:- mode foldl2_days(pred(in, in, out, in, out) is semidet, in, in, in, out,
    in, out) is semidet.
:- mode foldl2_days(pred(in, in, out, mdi, muo) is semidet, in, in, in, out,
    mdi, muo) is semidet.
:- mode foldl2_days(pred(in, in, out, di, uo) is semidet, in, in, in, out,
    di, uo) is semidet.

    % foldl3_days(Pred, Start, End, !Acc1, !Acc2, !Acc3):
    % As above, but with three accumulators.
    %
:- pred foldl3_days(pred(date, A, A, B, B, C, C), date, date,
    A, A, B, B, C, C).
:- mode foldl3_days(pred(in, in, out, in, out, in, out) is det, in, in,
    in, out, in, out, in, out) is det.
:- mode foldl3_days(pred(in, in, out, in, out, mdi, muo) is det, in, in,
    in, out, in, out, mdi, muo) is det.
:- mode foldl3_days(pred(in, in, out, in, out, di, uo) is det, in, in,
    in, out, in, out, di, uo) is det.
:- mode foldl3_days(pred(in, in, out, in, out, in, out) is semidet, in, in,
    in, out, in, out, in, out) is semidet.
:- mode foldl3_days(pred(in, in, out, in, out, mdi, muo) is semidet, in, in,
    in, out, in, out, mdi, muo) is semidet.
:- mode foldl3_days(pred(in, in, out, in, out, di, uo) is semidet, in, in,
    in, out, in, out, di, uo) is semidet.

%-----------------------------------------------%
%-----------------------------------------------%
```

15 char

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 1994-2008, 2011 The University of Melbourne.
% Copyright (C) 2013-2015, 2017-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: char.m.
% Main author: fjh.
% Stability: high.
%
% This module defines some predicates that manipulate characters.
%
% Originally we used ‘character’ rather than ‘char’ for the type name
% because ‘char’ was used by NU-Prolog to mean something different.
% But now we use ‘char’ and the use of ‘character’ is discouraged.
%
% All predicates and functions exported by this module that deal with
% Unicode conform to version 10 of the Unicode standard.
%
%---------------------------------------------------%
%---------------------------------------------------%

:- module char.
:- interface.

:- import_module enum.
:- import_module list.
:- import_module pretty_printer.

%---------------------------------------------------%

    % A Unicode code point.
    %
:- type char == character.

:- instance enum(character).

    % ‘to_int’/1 and ‘to_int(in, out)’ convert a character to its
    % corresponding numerical code (integer value).
    %
    % ‘to_int(out, in)’ converts an integer value to a character value.
    % It fails for integer values outside of the Unicode range.
    %
```

```
    % Be aware that there is no guarantee that characters can be written to
    % files or to the standard output or standard error streams. Files us-
ing an
    % 8-bit national character set would only be able to represent a sub-
set of
    % all possible code points. Currently, the Mercury standard library can
    % only read and write UTF-8 text files, so the entire range is supported
    % (excluding surrogate and noncharacter code points).
    %
    % Note that '\0' is not accepted as a Mercury null character literal.
    % Instead, a null character can be created using `det_from_int(0)'.
    % Null characters are not allowed in Mercury strings in C grades.
    %
:- func to_int(char) = int.
:- pred to_int(char, int).
:- mode to_int(in, out) is det.
:- mode to_int(in, in) is semidet.    % implied
:- mode to_int(out, in) is semidet.

    % Converts an integer to its corresponding character, if any.
    % A more expressive name for the reverse mode of to_int.
    %
:- pred from_int(int::in, char::out) is semidet.

    % Converts an integer to its corresponding character.
    % Throws an exception if there isn't one.
    %
:- func det_from_int(int) = char.
:- pred det_from_int(int::in, char::out) is det.

    % Returns the minimum numerical character code.
    %
:- func min_char_value = int.
:- pred min_char_value(int::out) is det.

    % Returns the maximum numerical character code.
    %
:- func max_char_value = int.
:- pred max_char_value(int::out) is det.

%-----------------------------------------------%

    % True iff the character is a lowercase letter (a-z) in the ASCII range.
    %
:- pred is_lower(char::in) is semidet.

    % True iff the character is an uppercase letter (A-Z) in the ASCII range.
```

```
    %
:- pred is_upper(char::in) is semidet.

    % Convert a character to lowercase.
    % Note that this only converts letters (A-Z) in the ASCII range.
    %
:- func to_lower(char) = char.
:- pred to_lower(char::in, char::out) is det.

    % Convert a character to uppercase.
    % Note that this only converts letters (a-z) in the ASCII range.
    %
:- func to_upper(char) = char.
:- pred to_upper(char::in, char::out) is det.

    % lower_upper(Lower, Upper) is true iff
    % Lower is a lowercase letter (a-z) and Upper is the corresponding
    % uppercase letter (A-Z) in the ASCII range.
    %
:- pred lower_upper(char, char).
:- mode lower_upper(in, out) is semidet.
:- mode lower_upper(out, in) is semidet.

%-----------------------------------------------%

    % True iff the character is in the ASCII range (0-127).
    %
:- pred is_ascii(char::in) is semidet.

    % True iff the character is a whitespace character in the ASCII range:
    %
    %   U+0020  space
    %   U+0009  character tabulation (horizontal tab)
    %   U+000A  line feed
    %   U+000B  line tabulation (vertical tab)
    %   U+000C  form feed
    %   U+000D  carriage return
    %
:- pred is_whitespace(char::in) is semidet.

    % True iff the character is a letter (A-Z, a-z) in the ASCII range.
    %
:- pred is_alpha(char::in) is semidet.

    % True iff the character is a letter (A-Z, a-z) or digit (0-9)
    % in the ASCII range.
    %
```

```
:- pred is_alnum(char::in) is semidet.

    % True iff the character is a letter (A-Z, a-z) or an underscore (_)
    % in the ASCII range.
    %
:- pred is_alpha_or_underscore(char::in) is semidet.

    % True iff the character is a letter (A-Z, a-z), a digit (0-9) or an
    % underscore (_) in the ASCII range.
    %
:- pred is_alnum_or_underscore(char::in) is semidet.

%-----------------------------------------------%

    % True iff the character is a decimal digit (0-9) in the ASCII range.
    %
:- pred is_digit(char::in) is semidet.

    % True iff the character is a binary digit (0 or 1) in the ASCII range.
    %
:- pred is_binary_digit(char::in) is semidet.

    % True iff the character is an octal digit (0-7) in the ASCII range.
    %
:- pred is_octal_digit(char::in) is semidet.

    % True iff the character is a decimal digit (0-9) in the ASCII range.
    % Synonym for is_digit/1.
    %
:- pred is_decimal_digit(char::in) is semidet.

    % True iff the character is a hexadecimal digit (0-9, a-f, A-F) in the
    % ASCII range.
    %
:- pred is_hex_digit(char::in) is semidet.

    % is_base_digit(Base, Digit):
    % True iff Digit is a digit in the given Base (0-9, a-z, A-Z).
    % Throws an exception if Base < 2 or Base > 36.
    %
:- pred is_base_digit(int::in, char::in) is semidet.

%-----------------------------------------------%

    % binary_digit_to_int(Char, Int):
    % True iff Char is a binary digit (0-1) representing the value Int.
    %
```

```
:- pred binary_digit_to_int(char::in, int::out) is semidet.

    % As above, but throws an exception instead of failing.
    %
:- func det_binary_digit_to_int(char) = int.

    % octal_digit_to_int(Char, Int):
    % True iff Char is an octal digit (0-7) representing the value Int.
    %
:- pred octal_digit_to_int(char::in, int::out) is semidet.

    % As above, but throws an exception instead of failing.
    %
:- func det_octal_digit_to_int(char) = int.

    % decimal_digit_to_int(Char, Int):
    % True iff Char is a decimal digit (0-9) representing the value Int.
    %
:- pred decimal_digit_to_int(char::in, int::out) is semidet.

    % As above, but throws an exception instead of failing.
    %
:- func det_decimal_digit_to_int(char) = int.

    % hex_digit_to_int(Char, Int):
    % True iff Char is a hexadecimal digit (0-9, a-z or A-F) represent-
ing the
    % value Int.
    %
:- pred hex_digit_to_int(char::in, int::out) is semidet.

    % As above, but throws an exception instead of failing.
    %
:- func det_hex_digit_to_int(char) = int.

    % base_digit_to_int(Base, Char, Int):
    % True iff Char is a decimal digit (0-9) or a letter (a-z, A-Z)
    % representing the value Int (0-35) in the given base.
    % Throws an exception if Base < 2 or Base > 36.
    %
:- pred base_digit_to_int(int::in, char::in, int::out) is semidet.

    % As above, but throws an exception instead of failing.
    %
:- func det_base_digit_to_int(int, char) = int.

%-----------------------------------------------%
```

```
    % Convert an integer in the range 0-1 to a binary digit (0 or 1) in the
    % ASCII range.
    %
:- pred int_to_binary_digit(int::in, char::out) is semidet.

    % As above, but throw an exception instead of failing.
    %
:- func det_int_to_binary_digit(int) = char.

    % Convert an integer 0-7 to an octal digit (0-7) in the ASCII range.
    %
:- pred int_to_octal_digit(int::in, char::out) is semidet.

    % As above, but throw an exception instead of failing.
    %
:- func det_int_to_octal_digit(int) = char.

    % Convert an integer 0-9 to a decimal digit (0-9) in the ASCII range.
    %
:- pred int_to_decimal_digit(int::in, char::out) is semidet.

    % As above, but throw an exception in instead of failing.
    %
:- func det_int_to_decimal_digit(int) = char.

    % Convert an integer 0-15 to an uppercase hexadecimal digit (0-9, A-
F) in
    % the ASCII range.
    %
:- pred int_to_hex_digit(int::in, char::out) is semidet.

    % As above, but throw an exception in instead of failing.
    %
:- func det_int_to_hex_digit(int) = char.

    % base_int_to_digit(Base, Int, Char):
    % True iff Char is a decimal digit (0-9) or an uppercase letter (A-
Z)
    % representing the value Int (0-35) in the given base.
    % Throws an exception if Base < 2 or Base > 36.
    %
:- pred base_int_to_digit(int::in, int::in, char::out) is semidet.

    % As above, but throw an exception instead of failing.
    %
:- func det_base_int_to_digit(int, int) = char.
```

```
%-----------------------------------------------------%

    % Encode a Unicode code point in UTF-8.
    % Fails for surrogate code points.
    %
:- pred to_utf8(char::in, list(int)::out) is semidet.

    % Encode a Unicode code point in UTF-16 (native endianness).
    % Fails for surrogate code points.
    %
:- pred to_utf16(char::in, list(int)::out) is semidet.

    % True iff the character is a Unicode Surrogate code point, that is a code
    % point in General Category `Other,surrogate' (`Cs').
    % In UTF-16, a code point with a scalar value greater than 0xffff is
    % encoded with a pair of surrogate code points.
    %
:- pred is_surrogate(char::in) is semidet.

    % True iff the character is a Unicode leading surrogate code point.
    % A leading surrogate code point is in the inclusive range from
    % 0xd800 to 0xdbff.
    %
:- pred is_leading_surrogate(char::in) is semidet.

    % True iff the character is a Unicode trailing surrogate code point.
    % A trailing surrogate code point is in the inclusive range from
    % 0xdc00 to 0xdfff.
    %
:- pred is_trailing_surrogate(char::in) is semidet.

    % True iff the character is a Unicode Noncharacter code point.
    % Sixty-six code points are not used to encode characters.
    % These code points should not be used for interchange, but may be used
    % internally.
    %
:- pred is_noncharacter(char::in) is semidet.

    % True iff the character is a Unicode Control code point, that is a code
    % point in General Category `Other,control' (`Cc').
    %
:- pred is_control(char::in) is semidet.

    % True iff the character is a Unicode Space Separator code point, that is a
    % code point in General Category `Separator,space' (`Zs').
    %
```

```
:- pred is_space_separator(char::in) is semidet.

    % True iff the character is a Unicode Line Separator code point, that is a
    % code point in General Category ‘Separator,line’ (‘Zl’).
    %
:- pred is_line_separator(char::in) is semidet.

    % True iff the character is a Unicode Paragraph Separator code point, that
    % is a code point in General Category ‘Separator,paragraph’ (‘Zp’).
    %
:- pred is_paragraph_separator(char::in) is semidet.

    % True iff the character is a Unicode Private-use code point, that is a
    % code point in General Category ‘Other,private use’ (‘Co’).
    %
:- pred is_private_use(char::in) is semidet.

%-----------------------------------------------------%

    % Convert a char to a pretty_printer.doc for formatting.
    %
:- func char_to_doc(char) = pretty_printer.doc.

%-----------------------------------------------------%

% The following have all been deprecated.

    % Use hex_digit_to_int/2 instead.
    %
:- pred is_hex_digit(char, int).
:- mode is_hex_digit(in, out) is semidet.

    % Convert an integer 0-15 to a hexadecimal digit (0-9, A-F) in the ASCII
    % range.
    %
    % Use int_to_hex_digit/2 instead.
    %
:- pred int_to_hex_char(int, char).
:- mode int_to_hex_char(in, out) is semidet.

    % True iff the characters is a decimal digit (0-9) or letter (a-z or A-
Z).
    % Returns the character’s value as a digit (0-9 or 10-35).
    %
:- pragma obsolete(digit_to_int/2).
:- pred digit_to_int(char::in, int::out) is semidet.
```

```
    % int_to_digit(Int, Char):
    %
    % True iff Int is an integer in the range 0-35 and Char is a
    % decimal digit or uppercase letter whose value as a digit is Int.
    %
    % Use whichever of int_to_binary_digit/2, int_to_octal_digit/2,
    % int_to_decimal_digit/2, int_to_hex_digit/2 or base_int_to_digit/3 is
    % appropriate instead of the (in, out) mode
    %
    % Use whichever of binary_digit_to_int/2, octal_digit_to_int/2,
    % decimal_digit_to_int/2, hex_digit_to_int/2 or base_digit_to_int/3
    % is appropriate instead of the (out, in) mode.
    %
:- pragma obsolete(int_to_digit/2).
:- pred int_to_digit(int, char).
:- mode int_to_digit(in, out) is semidet.
:- mode int_to_digit(out, in) is semidet.

    % Returns a decimal digit or uppercase letter corresponding to the value.
    % Calls error/1 if the integer is not in the range 0-35.
    %
    % Use whichever of det_int_to_binary_digit/1, det_int_to_octal_digit/1
    % det_int_to_decimal_digit/1, det_int_to_hex_digit/1 or
    % det_base_int_to_digit/2 is appropriate instead.
    %
:- pragma obsolete(det_int_to_digit/1).
:- func det_int_to_digit(int) = char.
:- pragma obsolete(det_int_to_digit/2).
:- pred det_int_to_digit(int::in, char::out) is det.

%-----------------------------------------------------%
%
% Computing hashes of chars.
%

    % Compute a hash value for a char.
    %
:- func hash(char) = int.
:- pred hash(char::in, int::out) is det.

%-----------------------------------------------------%
%-----------------------------------------------------%
```

16 construct

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 2002-2009, 2011 The University of Melbourne.
% Copyright (C) 2014-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: construct.m.
% Main author: zs.
% Stability: low.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module construct.
:- interface.

:- import_module list.
:- import_module maybe.
:- import_module univ.
:- import_module type_desc.

%-----------------------------------------------------%

    % The functors of a discriminated union type are numbered from
    % zero to N-1, where N is the value returned by num_functors.
    % The functors are numbered in lexicographic order. If two
    % functors have the same name, the one with the lower arity
    % will have the lower number.
    %
:- type functor_number_ordinal == int.
:- type functor_number_lex == int.

    % num_functors(Type).
    %
    % Returns the number of different functors for the top-level
    % type constructor of the type specified by Type.
    % Fails if the type is not a discriminated union type.
    %
    % deconstruct.functor_number/3, deconstruct.deconstruct_du/5
    % and the semidet predicates and functions in this module will
    % only succeed for types for which num_functors/1 succeeds.
    %
:- func num_functors(type_desc) = int is semidet.
```

```
:- func det_num_functors(type_desc) = int.

    % get_functor(Type, FunctorNumber, FunctorName, Arity, ArgTypes).
    %
    % Binds FunctorName and Arity to the name and arity of functor number
    % FunctorNumber for the specified type, and binds ArgTypes to the
    % type_descs for the types of the arguments of that functor.
    % Fails if the type is not a discriminated union type, or if
    % FunctorNumber is out of range.
    %
:- pred get_functor(type_desc::in, functor_number_lex::in,
    string::out, int::out, list(pseudo_type_desc)::out) is semidet.

    % get_functor_with_names(Type, FunctorNumber, FunctorName, Arity, ArgTypes,
    %   ArgNames).
    %
    % Binds FunctorName and Arity to the name and arity of functor number
    % FunctorNumber for the specified type, ArgTypes to the type_descs
    % for the types of the arguments of that functor, and ArgNames to the
    % field name of each functor argument, if any. Fails if the type is
    % not a discriminated union type, or if FunctorNumber is out of range.
    %
:- pred get_functor_with_names(type_desc::in, functor_number_lex::in,
    string::out, int::out, list(pseudo_type_desc)::out,
    list(maybe(string))::out) is semidet.

    % get_functor_ordinal(Type, I) = Ordinal.
    %
    % Returns Ordinal, where Ordinal is the position in declaration order
    % for the specified type of the function symbol that is in position I
    % in lexicographic order. Fails if the type is not a discriminated
    % union type, or if I is out of range.
    %
:- func get_functor_ordinal(type_desc, functor_number_lex) =
    functor_number_ordinal is semidet.
:- pred get_functor_ordinal(type_desc::in, functor_number_lex::in,
    functor_number_ordinal::out) is semidet.

    % get_functor_lex(Type, Ordinal) = I.
    %
    % Returns I, where I is the position in lexicographic order for the
    % specified type of the function symbol that is in position Ordinal
    % in declaration order. Fails if the type is not a discriminated
    % union type, or if Ordinal is out of range.
    %
:- func get_functor_lex(type_desc, functor_number_ordinal) =
```

```
    functor_number_lex is semidet.

    % find_functor(Type, FunctorName, Arity, FunctorNumber, ArgTypes).
    %
    % Given a type descriptor, a functor name and arity, finds the functor
    % number and the types of its arguments. It thus serves as the converse
    % to get_functor/5.
    %
:- pred find_functor(type_desc::in, string::in, int::in,
    functor_number_lex::out, list(type_desc)::out) is semidet.

    % construct(Type, I, Args) = Term.
    %
    % Returns a term of the type specified by Type whose functor
    % is functor number I of the type given by Type, and whose
    % arguments are given by Args. Fails if the type is not a
    % discriminated union type, or if I is out of range, or if the
    % number of arguments supplied doesn't match the arity of the selected
    % functor, or if the types of the arguments do not match
    % the expected argument types of that functor.
    %
:- func construct(type_desc, functor_number_lex, list(univ)) = univ is semidet.

    % construct_tuple(Args) = Term.
    %
    % Returns a tuple whose arguments are given by Args.
    %
:- func construct_tuple(list(univ)) = univ.

%---------------------------------------------------%
%---------------------------------------------------%
```

17 cord

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 2002-2011 The University of Melbourne.
% Copyright (C) 2013-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: cord.m.
% Author: Ralph Becket <rafe@cs.mu.oz.au>
% Stability: medium.
```

```
%
% A cord is a sequence type supporting O(1) consing and concatenation.
% A cord is essentially a tree structure with data stored in the leaf nodes.
% Joining two cords together to construct a new cord is therefore an O(1)
% operation.
%
% This data type is intended for situations where efficient, linearised
% collection of data is required.
%
% While this data type presents a list-like interface, calls to list/1 and
% head_tail/3 in particular are O(n) in the size of the cord.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module cord.
:- interface.

:- import_module list.

%-----------------------------------------------------%

    % Cords that contain the same members in the same order will not
    % necessarily have the same representation and will, therefore,
    % not necessarily either unify or compare as equal.
    %
    % The exception to this rule is that the empty cord does have a
    % unique representation.
    %
:- type cord(T).

    % Return the empty cord.
    %
:- func init = cord(T).

    % The unique representation for the empty cord:
    %
    %   list(empty) = []
    %
:- func empty = cord(T).

    % Succeed iff the given cord is empty.
    %
:- pred is_empty(cord(T)::in) is semidet.

    % list(singleton(X)) = [X]
    %
```

```
:- func singleton(T) = cord(T).

    % list(from_list(Xs)) = Xs
    % An O(1) operation.
    %
:- func from_list(list(T)) = cord(T).

    % The list of data in a cord:
    %
    %   list(empty        ) = []
    %   list(from_list(Xs)) = Xs
    %   list(cons(X, C)   ) = [X | list(C)]
    %   list(TA ++ TB     ) = list(TA) ++ list(TB)
    %
:- func list(cord(T)) = list(T).

    % A synonym for the list/1.
    %
:- func to_list(cord(T)) = list(T).

    % rev_list(Cord) = list.reverse(list(Cord).
    %
:- func rev_list(cord(T)) = list(T).

    % A synonym for rev_list/1.
    %
:- func to_rev_list(cord(T)) = list(T).

    % Cord = condense(CordOfCords):
    %
    % ‘Cord’ is the result of concatenating all the elements of ‘CordOfCords’.
    %
:- func condense(cord(cord(T))) = cord(T).

    % list(cons(X, C)) = [X | list(C)]
    % An O(1) operation.
    %
:- func cons(T, cord(T)) = cord(T).
:- pred cons(T::in, cord(T)::in, cord(T)::out) is det.

    % list(snoc(C, X)) = list(C) ++ [X]
    % An O(1) operation.
    %
:- func snoc(cord(T), T) = cord(T).
:- pred snoc(T::in, cord(T)::in, cord(T)::out) is det.

    % list(CA ++ CB) = list(CA) ++ list(CB)
```

```
    % An O(1) operation.
    %
:- func cord(T) ++ cord(T) = cord(T).

    % Append together a list of cords.
    %
:- func cord_list_to_cord(list(cord(T))) = cord(T).

    % Reverse the given list (of cords), and then append together
    % the resulting list of cords.
    %
:- func rev_cord_list_to_cord(list(cord(T))) = cord(T).

    % Append together a list of cords, and return the result as a list.
    %
:- func cord_list_to_list(list(cord(T))) = list(T).

    % Reverse the given list (of cords), and then append together
    % the resulting list of cords, and return it as a list.
    %
:- func rev_cord_list_to_list(list(cord(T))) = list(T).

    %     head_tail(C0, X, C)  =>  list(C0) = [X | list(C)]
    % not head_tail(C0, _, _)  =>  C0 = empty
    % An O(n) operation, although traversing an entire cord with
    % head_tail/3 gives O(1) amortized cost for each call.
    %
:- pred head_tail(cord(T)::in, T::out, cord(T)::out) is semidet.

    %     split_last(C0, C, X)  =>  list(C0) = C ++ [X].
    % not split_last(C0, _, _)  =>  C0 = empty
    % An O(n) operation, although traversing an entire cord with
    % split_last/3 gives O(1) amortized cost for each call.
    %
:- pred split_last(cord(T)::in, cord(T)::out, T::out) is semidet.

    %     get_first(C0, X)  =>  some [C]: list(C0) = [X] ++ C.
    % not get_first(C0, _)  =>  C0 = empty
    %
:- pred get_first(cord(T)::in, T::out) is semidet.

    %     get_last(C0, X)  =>  some [C]: list(C0) = C ++ [X].
    % not get_last(C0, _)  =>  C0 = empty
    %
:- pred get_last(cord(T)::in, T::out) is semidet.

    % length(C) = list.length(list(C))
```

```
    % An O(n) operation.
    %
:- func length(cord(T)) = int.

    % member(X, C) <=> list.member(X, list(C)).
    %
:- pred member(T::out, cord(T)::in) is nondet.

    % list(map(F, C)) = list.map(F, list(C))
    %
:- func map(func(T) = U, cord(T)) = cord(U).
:- pred map_pred(pred(T, U)::in(pred(in, out) is det),
    cord(T)::in, cord(U)::out) is det.

    % filter(Pred, Cord, TrueCord):
    %
    % Pred is a closure with one input argument.
    % For each member X of Cord,
    % - if Pred(X) is true, then X is included in TrueCord.
    %
:- pred filter(pred(T)::in(pred(in) is semidet),
    cord(T)::in, cord(T)::out) is det.

    % filter(Pred, Cord, TrueCord, FalseCord):
    %
    % Pred is a closure with one input argument.
    % For each member X of Cord,
    % - if Pred(X) is true, then X is included in TrueCord.
    % - if Pred(X) is false, then X is included in FalseCord.
    %
:- pred filter(pred(T)::in(pred(in) is semidet),
    cord(T)::in, cord(T)::out, cord(T)::out) is det.

    % foldl(F, C, A) = list.foldl(F, list(C), A).
    %
:- func foldl(func(T, U) = U, cord(T), U) = U.
:- pred foldl_pred(pred(T, U, U), cord(T), U, U).
:- mode foldl_pred(in(pred(in, in, out) is det), in, in, out) is det.
:- mode foldl_pred(in(pred(in, mdi, muo) is det), in, mdi, muo) is det.
:- mode foldl_pred(in(pred(in, di, uo) is det), in, di, uo) is det.
:- mode foldl_pred(in(pred(in, in, out) is semidet), in, in, out) is semidet.
:- mode foldl_pred(in(pred(in, mdi, muo) is semidet), in, mdi, muo) is semidet.
:- mode foldl_pred(in(pred(in, di, uo) is semidet), in, di, uo) is semidet.

    % foldr(F, C, A) = list.foldr(F, list(C), A).
    %
:- func foldr(func(T, U) = U, cord(T), U) = U.
```

```
:- pred foldr_pred(pred(T, U, U), cord(T), U, U).
:- mode foldr_pred(in(pred(in, in, out) is det), in, in, out) is det.
:- mode foldr_pred(in(pred(in, mdi, muo) is det), in, mdi, muo) is det.
:- mode foldr_pred(in(pred(in, di, uo) is det), in, di, uo) is det.
:- mode foldr_pred(in(pred(in, in, out) is semidet), in, in, out) is semidet.
:- mode foldr_pred(in(pred(in, mdi, muo) is semidet), in, mdi, muo) is semidet.
:- mode foldr_pred(in(pred(in, di, uo) is semidet), in, di, uo) is semidet.

    % map_foldl(P, CA, CB, !Acc):
    %
    % This predicate calls P on each element of the input cord, working
    % left to right. Each call to P transforms an element of the input cord
    % to the corresponding element of the output cord, and updates the
    % accumulator.
    %
:- pred map_foldl(pred(A, B, C, C), cord(A), cord(B), C, C).
:- mode map_foldl(in(pred(in, out, in, out) is det), in, out, in, out)
    is det.
:- mode map_foldl(in(pred(in, out, mdi, muo) is det), in, out, mdi, muo)
    is det.
:- mode map_foldl(in(pred(in, out, di, uo) is det), in, out, di, uo)
    is det.
:- mode map_foldl(in(pred(in, out, in, out) is semidet), in, out, in, out)
    is semidet.
:- mode map_foldl(in(pred(in, out, mdi, muo) is semidet), in, out, mdi, muo)
    is semidet.
:- mode map_foldl(in(pred(in, out, di, uo) is semidet), in, out, di, uo)
    is semidet.

    % As above, but with two accumulators.
    %
:- pred map_foldl2(pred(A, B, C, C, D, D)::
    in(pred(in, out, in, out, in, out) is det),
    cord(A)::in, cord(B)::out, C::in, C::out, D::in, D::out) is det.

    % As above, but with three accumulators.
    %
:- pred map_foldl3(pred(A, B, C, C, D, D, E, E)::
    in(pred(in, out, in, out, in, out, in, out) is det),
    cord(A)::in, cord(B)::out, C::in, C::out, D::in, D::out, E::in, E::out)
    is det.

    % find_first_match(Pred, List, FirstMatch) takes a closure with one
    % input argument. It returns the first element X of the cord (if any)
    % for which Pred(X) is true.
    %
:- pred find_first_match(pred(X)::in(pred(in) is semidet),
```

```
    cord(X)::in, X::out) is semidet.

    % equal(CA, CB)  <=>  list(CA) = list(CB).
    % An O(n) operation where n = length(CA) + length(CB).
    %
    % (Note: the current implementation works exactly this way.)
    %
:- pred equal(cord(T)::in, cord(T)::in) is semidet.

%---------------------------------------------------%
%---------------------------------------------------%
```

18 counter

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 2000, 2005-2006, 2011 The University of Melbourne.
% Copyright (C) 2014-2016, 2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: counter.m.
% Author: zs.
% Stability: high.
%
% Predicates for dealing with counters, which are mechanisms for allocating
% consecutively numbered integers. The abstraction barrier eliminates the
% possibility of confusion along the lines of "does this counter record
% the next number to be handed out, or the last number that was handed out?".
%
%---------------------------------------------------%
%---------------------------------------------------%

:- module counter.
:- interface.

%---------------------------------------------------%

:- type counter.

    % init(N, Counter) returns a counter whose first allocation will be the
    % integer N.
    %
:- pred init(int::in, counter::out) is det.
```

```
    % A function version of init/2.
    %
:- func init(int) = counter.

    % allocate(N, Counter0, Counter) takes a counter, and returns (a) the next
    % integer to be allocated from that counter, and (b) the updated state of
    % the counter.
    %
:- pred allocate(int::out, counter::in, counter::out) is det.

%---------------------------------------------------%
%---------------------------------------------------%
```

19 deconstruct

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 2002-2007 The University of Melbourne.
% Copyright (C) 2014-2015, 2017-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: deconstruct.m.
% Main author: zs.
% Stability: low.
%
%---------------------------------------------------%
%---------------------------------------------------%

:- module deconstruct.
:- interface.

:- import_module construct.
:- import_module list.
:- import_module maybe.
:- import_module univ.

%---------------------------------------------------%

    % Values of type noncanon_handling are intended to control how
    % predicates that deconstruct terms behave when they find that
    % the term they are about to deconstruct is of a noncanonical type,
    % i.e. of a type in which a single logical value may have more than one
```

```
    % concrete representation.
    %
    % The value ‘do_not_allow’ means that in such circumstances the
    % predicate should cause a runtime abort.
    %
    % The value ‘canonicalize’ means that in such circumstances the
    % predicate should return a constant giving the identity of the type,
    % regardless of the actual value of the term.
    %
    % The value ‘include_details_cc’ means that in such circumstances
    % the predicate should proceed as if the term were of a canonical type.
    % Use of this option requires a committed choice context.

:- type noncanon_handling
    --->    do_not_allow
    ;       canonicalize
    ;       include_details_cc.

:- inst do_not_allow for noncanon_handling/0
    --->    do_not_allow.
:- inst canonicalize for noncanon_handling/0
    --->    canonicalize.
:- inst include_details_cc for noncanon_handling/0
    --->    include_details_cc.
:- inst canonicalize_or_do_not_allow for noncanon_handling/0
    --->    do_not_allow
    ;       canonicalize.
:- inst do_not_allow_or_include_details_cc for noncanon_handling/0
    --->    do_not_allow
    ;       include_details_cc.

    % functor, argument and deconstruct and their variants take any type
    % (including univ), and return representation information for that type.
    %
    % The string representation of the functor that these predicates
    % return is:
    %
    %   - for user defined types with standard equality, the functor
    %     that is given in the type definition. For lists, this means
    %     the functors [|]/2 and []/0 are used, even if the list uses
    %     the [....] shorthand.
    %   - for user-defined types with user-defined equality, the
    %     functor will be of the form <<module.type/arity>>, except
    %     with include_details_cc, in which case the type will be
    %     handled as if it had standard equality.
    %   - for integers, the string is a base 10 number;
    %     positive integers have no sign.
```

```
%   - for finite floats, the string is a base 10 floating point number;
%     positive floating point numbers have no sign;
%     for infinite floats, the string "infinity" or "-infinity".
%   - for strings, the string, inside double quotation marks using
%     backslash escapes if necessary and backslash or octal escapes for
%     all characters for which char.is_control/1 is true.
%   - for characters, the character inside single quotation marks using
%     a backslash escape if necessary and a backslash or octal escape for
%     for all characters for which char.is_control/1 is true.
%   - for predicates, the string <<predicate>>, and for functions,
%     the string <<function>>, except with include_details_cc,
%     in which case it will be the predicate or function name.
%     (The predicate or function name will be artificial for
%     predicate and function values created by lambda expressions.)
%   - for tuples, the string {}.
%   - for arrays, the string <<array>>.
%   - for c_pointers, the string c_pointer(0xXXXX) where XXXX is the
%     hexadecimal representation of the pointer.
%   - for foreign types, a string of the form <<foreign(Name, Rep)>> where
%     Name is the type's Mercury name and Rep is a target language specific
%     representation of the term's value.
%   - for bitmaps, the bitmap converted to a length and a
%     hexadecimal string inside angle brackets and quotes of the
%     form """<[0-9]:[0-9A-F]*>""".
%
% The arity that these predicates return is:
%
%   - for user defined types with standard equality, the arity
%     of the functor.
%   - for user defined types with user-defined equality, zero,
%     except with include_details_cc, in which case the type
%     will be handled as if it had standard equality.
%   - for integers, zero.
%   - for floats, zero.
%   - for strings, zero.
%   - for characters, zero.
%   - for predicates and functions, zero, except with
%     include_details_cc, in which case it will be the number of
%     arguments hidden in the closure.
%   - for tuples, the number of elements in the tuple.
%   - for arrays, the number of elements in the array.
%   - for c_pointers, zero.
%   - for foreign types, zero.
%   - for bitmaps, zero.
%
% Note that in the current University of Melbourne implementation,
% the implementations of these predicates depart from the above
```

```
    % specification in that with --high-level-code, they do not
    % deconstruct predicate- and function-valued terms even with
    % include_details_cc; instead, they return <<predicate>> or
    % <<function>> (in both cases with arity zero) as appropriate.

    % functor(Data, NonCanon, Functor, Arity)
    %
    % Given a data item (Data), binds Functor to a string representation
    % of the functor and Arity to the arity of this data item.
    %
:- pred functor(T, noncanon_handling, string, int).
:- mode functor(in, in(do_not_allow), out, out) is det.
:- mode functor(in, in(canonicalize), out, out) is det.
:- mode functor(in, in(include_details_cc), out, out) is cc_multi.
:- mode functor(in, in, out, out) is cc_multi.

    % functor_number(Data, FunctorNumber, Arity)
    %
    % Given a data item, return the number of the functor,
    % suitable for use by construct.construct, and the arity.
    % Fail if the item does not have a discriminated union type.
    % Cause a runtime abort if the type has user-defined equality.
    %
:- pred functor_number(T::in, functor_number_lex::out, int::out) is semidet.

    % functor_number_cc(Data, FunctorNumber, Arity)
    %
    % Given a data item, return the number of the functor,
    % suitable for use by construct.construct, and the arity.
    % Fail if the item does not have a discriminated union type.
    % Do not cause a runtime abort if the type has user-defined equality.
    %
:- pred functor_number_cc(T::in, functor_number_lex::out,
    int::out) is cc_nondet.

    % arg(Data, NonCanon, Index, Argument)
    %
    % Given a data item (Data) and an argument index (Index), starting
    % at 0 for the first argument, binds Argument to that argument of
    % the functor of the data item. If the argument index is out of range
    % -- that is, greater than or equal to the arity of the functor or
    % lower than 0 -- then the call fails.
    %
    % Note that this predicate only returns an answer when NonCanon is
    % do_not_allow or canonicalize.  If you need the include_details_cc
    % behaviour use deconstruct.arg_cc/3.
    %
```

```
:- some [ArgT] pred arg(T, noncanon_handling, int, ArgT).
:- mode arg(in, in(do_not_allow), in, out) is semidet.
:- mode arg(in, in(canonicalize), in, out) is semidet.
:- mode arg(in, in(canonicalize_or_do_not_allow), in, out) is semidet.

:- type maybe_arg
    --->    some [T] arg(T)
    ;       no_arg.

    % arg_cc/3 is similar to arg/4, except that it handles arguments with
    % non-canonical types.  The possible non-existence of an argument is
    % encoded using a maybe type.
    %
:- pred arg_cc(T::in, int::in, maybe_arg::out) is cc_multi.

    % named_arg(Data, NonCanon, Name, Argument)
    %
    % Same as arg/4, except the chosen argument is specified by giving
    % its name rather than its position. If Data has no argument with that
    % name, named_arg fails.
    %
:- some [ArgT] pred named_arg(T, noncanon_handling, string, ArgT).
:- mode named_arg(in, in(do_not_allow), in, out) is semidet.
:- mode named_arg(in, in(canonicalize), in, out) is semidet.
:- mode named_arg(in, in(canonicalize_or_do_not_allow), in, out) is semidet.

    % named_arg_cc/3 is similar to named_arg/4, except that it handles
    % arguments with non-canonical types.
    %
:- pred named_arg_cc(T::in, string::in, maybe_arg::out) is cc_multi.

    % det_arg(Data, NonCanon, Index, Argument)
    %
    % Same as arg/4, except that for cases where arg/4 would fail,
    % det_arg/4 will throw an exception.
    %
:- some [ArgT] pred det_arg(T, noncanon_handling, int, ArgT).
:- mode det_arg(in, in(do_not_allow), in, out) is det.
:- mode det_arg(in, in(canonicalize), in, out) is det.
:- mode det_arg(in, in(include_details_cc), in, out) is cc_multi.
:- mode det_arg(in, in, in, out) is cc_multi.

    % det_named_arg(Data, NonCanon, Name, Argument)
    %
    % Same as named_arg/4, except that for cases where named_arg/4 would fail,
    % det_named_arg/4 will throw an exception.
    %
```

```
:- some [ArgT] pred det_named_arg(T, noncanon_handling, string, ArgT).
:- mode det_named_arg(in, in(do_not_allow), in, out) is det.
:- mode det_named_arg(in, in(canonicalize), in, out) is det.
:- mode det_named_arg(in, in(include_details_cc), in, out) is cc_multi.
:- mode det_named_arg(in, in, in, out) is cc_multi.

    % deconstruct(Data, NonCanon, Functor, Arity, Arguments)
    %
    % Given a data item (Data), binds Functor to a string representation
    % of the functor, Arity to the arity of this data item, and Arguments
    % to a list of arguments of the functor. The arguments in the list
    % are each of type univ.
    %
    % The cost of calling deconstruct depends greatly on how many arguments
    % Data has. If Data is an array, then each element of the array is
    % considered one of its arguments. Therefore calling deconstruct
    % on large arrays can take a very large amount of memory and a very
    % long time. If you call deconstruct in a situation in which you may
    % pass it a large array, you should probably use limited_deconstruct
    % instead.
    %
:- pred deconstruct(T, noncanon_handling, string, int, list(univ)).
:- mode deconstruct(in, in(do_not_allow), out, out, out) is det.
:- mode deconstruct(in, in(canonicalize), out, out, out) is det.
:- mode deconstruct(in, in(include_details_cc), out, out, out) is cc_multi.
:- mode deconstruct(in, in, out, out, out) is cc_multi.

    % deconstruct_du(Data, NonCanon, FunctorNumber, Arity, Arguments)
    %
    % Given a data item (Data) which has a discriminated union type, binds
    % FunctorNumber to the number of the functor in lexicographic order,
    % Arity to the arity of this data item, and Arguments to a list of
    % arguments of the functor. The arguments in the list are each of type
    % univ.
    %
    % Fails if Data does not have discriminated union type.
    %
:- pred deconstruct_du(T, noncanon_handling, functor_number_lex,
    int, list(univ)).
:- mode deconstruct_du(in, in(do_not_allow), out, out, out) is semidet.
:- mode deconstruct_du(in, in(include_details_cc), out, out, out) is cc_nondet.
:- mode deconstruct_du(in, in, out, out, out) is cc_nondet.

    % limited_deconstruct(Data, NonCanon, MaxArity,
    %   Functor, Arity, Arguments)
    %
    % limited_deconstruct works like deconstruct, but if the arity of T is
```

```
    % greater than MaxArity, limited_deconstruct fails. This is useful in
    % avoiding bad performance in cases where Data may be a large array.
    %
    % Note that this predicate only returns an answer when NonCanon is
    % do_not_allow or canonicalize.  If you need the include_details_cc
    % behaviour use deconstruct.limited_deconstruct_cc/3.
    %
:- pred limited_deconstruct(T, noncanon_handling, int, string, int,
    list(univ)).
:- mode limited_deconstruct(in, in(do_not_allow), in, out, out, out)
    is semidet.
:- mode limited_deconstruct(in, in(canonicalize), in, out, out, out)
    is semidet.

:- pred limited_deconstruct_cc(T::in, int::in,
    maybe({string, int, list(univ)})::out) is cc_multi.

%---------------------------------------------------%
%---------------------------------------------------%
```

20 diet

```
%---------------------------------------------------%
% vim: ts=4 sw=4 et ft=mercury
%---------------------------------------------------%
% Copyright (C) 2012-2014 YesLogic Pty. Ltd.
% Copyright (C) 2014-2015, 2017-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: diet.m.
% Author: wangp.
% Stability: medium.
%
% Discrete Interval Encoding Trees are a highly efficient set implementation
% for fat sets, i.e. densely populated sets over a discrete linear order.
%
% M. Erwig: Diets for Fat Sets,
% Journal of Functional Programming, Vol. 8, No. 6, pp. 627-632.
%
% O. Friedmann, M. Lange: More on Balanced Diets,
% Journal of Functional Programming, volume 21, issue 02, pp. 135-157.
%
%---------------------------------------------------%
```

```
:- module diet.
:- interface.

:- import_module bool.
:- import_module enum.
:- import_module list.

%-----------------------------------------------------%

:- type diet(T). % <= diet_element(T).

:- typeclass diet_element(T) where [
    % less_than(X, Y) succeeds iff X < Y.
    pred less_than(T::in, T::in) is semidet,

    % successor(X) returns the successor of X, e.g. X + 1.
    func successor(T) = T,

    % predecessor(X) returns the predecessor of X, e.g. X - 1.
    func predecessor(T) = T
].

:- instance diet_element(int).

%-----------------------------------------------------%
%
% Initial creation of sets.
%

    % Return an empty set.
    %
:- func init = diet(T).
:- pred init(diet(T)::out) is det.

    % 'make_singleton_set(Elem)' returns a set containing just the single
    % element 'Elem'.
    %
:- func make_singleton_set(T) = diet(T) <= diet_element(T).

    % 'make_interval_set(X, Y)' returns a set containing just the elements in
    % the interval [X, Y]. Throws an exception if Y < X.
    %
:- func make_interval_set(T, T) = diet(T) <= diet_element(T).

%-----------------------------------------------------%
%
% Emptiness and singleton-ness tests.
```

```
%

:- pred empty(diet(T)).
:- mode empty(in) is semidet.
:- mode empty(out) is det.
:- pragma obsolete(empty/1, [init/0, init/1, is_empty/1]).

:- pred is_empty(diet(T)::in) is semidet.

:- pred is_non_empty(diet(T)::in) is semidet.

    % 'is_singleton(Set, X)' is true iff 'Set' is a singleton containing the
    % element 'X'.
    %
:- pred is_singleton(diet(T)::in, T::out) is semidet <= diet_element(T).

%-----------------------------------------------%
%
% Membership tests.
%

    % 'member(X, Set)' is true iff 'X' is a member of 'Set'.
    %
:- pred member(T, diet(T)) <= diet_element(T).
:- mode member(in, in) is semidet.
:- mode member(out, in) is nondet.

    % 'contains(Set, X)' is true iff 'X' is a member of 'Set'.
    %
:- pred contains(diet(T)::in, T::in) is semidet <= diet_element(T).

%-----------------------------------------------%
%
% Insertions and deletions.
%

    % 'insert(X, Set0, Set)' is true iff 'Set' is the union of
    % 'Set0' and the set containing only 'X'.
    %
:- func insert(diet(T), T) = diet(T) <= diet_element(T).
:- pred insert(T::in, diet(T)::in, diet(T)::out) is det <= diet_element(T).

    % 'insert_interval(X, Y, Set0, Set)' is true iff 'Set' is the union of
    % 'Set0' and the set containing only the elements of the interval [X, Y].
    % Throws an exception if Y < X.
    %
:- pred insert_interval(T::in, T::in, diet(T)::in, diet(T)::out) is det
```

```
        <= diet_element(T).

        % 'insert_new(X, Set0, Set)' is true iff 'Set0' does not contain
        % 'X', and 'Set' is the union of 'Set0' and the set containing only 'X'.
        %
    :- pred insert_new(T::in, diet(T)::in, diet(T)::out) is semidet
        <= diet_element(T).

        % 'insert_list(Xs, Set0, Set)' is true iff 'Set' is the union of
        % 'Set0' and the set containing only the members of 'Xs'.
        %
    :- func insert_list(diet(T), list(T)) = diet(T) <= diet_element(T).
    :- pred insert_list(list(T)::in, diet(T)::in, diet(T)::out) is det
        <= diet_element(T).

        % 'delete(X, Set0, Set)' is true iff 'Set' is the relative
        % complement of 'Set0' and the set containing only 'X', i.e.
        % if 'Set' is the set which contains all the elements of 'Set0'
        % except 'X'.
        %
    :- func delete(diet(T), T) = diet(T) <= diet_element(T).
    :- pred delete(T::in, diet(T)::in, diet(T)::out) is det <= diet_element(T).

        % 'delete_list(Set, X)' returns the difference of 'Set' and the set
        % containing only the members of 'X'. Same as
        % 'difference(Set, list_to_set(X))', but may be more efficient.
        %
    :- func delete_list(diet(T), list(T)) = diet(T) <= diet_element(T).
    :- pred delete_list(list(T)::in, diet(T)::in, diet(T)::out) is det
        <= diet_element(T).

        % 'remove(X, Set0, Set)' is true iff 'Set0' contains 'X',
        % and 'Set' is the relative complement of 'Set0' and the set
        % containing only 'X', i.e.  if 'Set' is the set which contains
        % all the elements of 'Set0' except 'X'.
        %
    :- pred remove(T::in, diet(T)::in, diet(T)::out) is semidet <= diet_element(T).

        % 'remove_list(X, Set0, Set)' returns in 'Set' the difference of 'Set0'
        % and the set containing all the elements of 'X', failing if any element
        % of 'X' is not in 'Set0'. Same as 'subset(list_to_set(X), Set0),
        % difference(Set0, list_to_set(X), Set)', but may be more efficient.
        %
    :- pred remove_list(list(T)::in, diet(T)::in, diet(T)::out) is semidet
        <= diet_element(T).

        % 'remove_least(X, Set0, Set)' is true iff 'X' is the least element in
```

```
    % ‘Set0’, and ‘Set’ is the set which contains all the elements of ‘Set0’
    % except ‘X’.
    %
:- pred remove_least(T::out, diet(T)::in, diet(T)::out) is semidet
    <= diet_element(T).

%-----------------------------------------------------%
%
% Comparisons between sets.
%

    % ‘equal(SetA, SetB)’ is true iff ‘SetA’ and ‘SetB’ contain the same
    % elements.
    %
:- pred equal(diet(T)::in, diet(T)::in) is semidet <= diet_element(T).

    % ‘subset(Subset, Set)’ is true iff ‘Subset’ is a subset of ‘Set’.
    %
:- pred subset(diet(T)::in, diet(T)::in) is semidet <= diet_element(T).

    % ‘superset(Superset, Set)’ is true iff ‘Superset’ is a superset of ‘Set’.
    %
:- pred superset(diet(T)::in, diet(T)::in) is semidet <= diet_element(T).

%-----------------------------------------------------%
%
% Operations on two or more sets.
%

    % ‘union(SetA, SetB, Set)’ is true iff ‘Set’ is the union of
    % ‘SetA’ and ‘SetB’.
    %
:- func union(diet(T), diet(T)) = diet(T) <= diet_element(T).
:- pred union(diet(T)::in, diet(T)::in, diet(T)::out) is det
    <= diet_element(T).

    % ‘union_list(Sets, Set)’ returns the union of all the sets in Sets.
    %
:- func union_list(list(diet(T))) = diet(T) <= diet_element(T).
:- pred union_list(list(diet(T))::in, diet(T)::out) is det <= diet_element(T).

    % ‘intersect(SetA, SetB, Set)’ is true iff ‘Set’ is the
    % intersection of ‘SetA’ and ‘SetB’.
    %
:- func intersect(diet(T), diet(T)) = diet(T) <= diet_element(T).
:- pred intersect(diet(T)::in, diet(T)::in, diet(T)::out) is det
    <= diet_element(T).
```

```
    % ‘intersect_list(Sets, Set)’ returns the intersection of all the sets
    % in Sets.
    %
:- func intersect_list(list(diet(T))) = diet(T) <= diet_element(T).
:- pred intersect_list(list(diet(T))::in, diet(T)::out) is det
    <= diet_element(T).

    % ‘difference(SetA, SetB)’ returns the set containing all the elements
    % of ‘SetA’ except those that occur in ‘SetB’.
    %
:- func difference(diet(T), diet(T)) = diet(T) <= diet_element(T).
:- pred difference(diet(T)::in, diet(T)::in, diet(T)::out) is det
    <= diet_element(T).

%-----------------------------------------------%
%
% Operations that divide a set into two parts.
%

    % ‘split(X, Set, Lesser, IsPresent, Greater)’ is true iff
    % ‘Lesser’ is the set of elements in ‘Set’ which are less than ‘X’ and
    % ‘Greater’ is the set of elements in ‘Set’ which are greater than ‘X’.
    % ‘IsPresent’ is ‘yes’ if ‘Set’ contains ‘X’, and ‘no’ otherwise.
    %
:- pred split(T::in, diet(T)::in, diet(T)::out, bool::out, diet(T)::out) is det
    <= diet_element(T).

    % divide(Pred, Set, InPart, OutPart):
    % InPart consists of those elements of Set for which Pred succeeds;
    % OutPart consists of those elements of Set for which Pred fails.
    %
:- pred divide(pred(T)::in(pred(in) is semidet), diet(T)::in,
    diet(T)::out, diet(T)::out) is det <= diet_element(T).

    % divide_by_set(DivideBySet, Set, InPart, OutPart):
    % InPart consists of those elements of Set which are also in DivideBySet;
    % OutPart consists of those elements of Set which are not in DivideBySet.
    %
:- pred divide_by_set(diet(T)::in, diet(T)::in, diet(T)::out, diet(T)::out)
    is det <= diet_element(T).

%-----------------------------------------------%
%
% Converting lists to sets.
%
```

```
    % ‘list_to_set(List)’ returns a set containing only the members of ‘List’.
    %
:- func list_to_set(list(T)) = diet(T) <= diet_element(T).
:- pred list_to_set(list(T)::in, diet(T)::out) is det <= diet_element(T).

:- func from_list(list(T)) = diet(T) <= diet_element(T).
:- pred from_list(list(T)::in, diet(T)::out) is det <= diet_element(T).

    % ‘from_interval_list(Intervals, Set)’ returns a Set containing the
    % elements of all intervals [X, Y] in Intervals, where each interval is
    % represented by a tuple. Throws an exception if any interval has Y < X.
    % The intervals may overlap.
    %
:- pred from_interval_list(list({T, T})::in, diet(T)::out) is det
    <= diet_element(T).

    % ‘sorted_list_to_set(List)’ returns a set containing only the members
    % of ‘List’. ‘List’ must be sorted.
    %
:- func sorted_list_to_set(list(T)) = diet(T) <= diet_element(T).
:- pred sorted_list_to_set(list(T)::in, diet(T)::out) is det
    <= diet_element(T).

%-----------------------------------------------------%
%
% Converting sets to lists.
%

    % ‘to_sorted_list(Set)’ returns a list containing all the members of ‘Set’,
    % in sorted order.
    %
:- func to_sorted_list(diet(T)) = list(T) <= diet_element(T).
:- pred to_sorted_list(diet(T)::in, list(T)::out) is det <= diet_element(T).

    % ‘to_sorted_interval_list(Set)’ returns a list of intervals in ‘Set’
    % in sorted order, where each interval is represented by a tuple.
    % The intervals do not overlap.
    %
:- pred to_sorted_interval_list(diet(T)::in, list({T, T})::out) is det
    <= diet_element(T).

%-----------------------------------------------------%
%
% Counting.
%

    % ‘count(Set)’ returns the number of elements in Set.
```

```
    %
:- func count(diet(T)) = int <= enum(T).

%-----------------------------------------------------%
%
% Standard higher order functions on collections.
%

    % all_true(Pred, Set) succeeds iff Pred(Element) succeeds
    % for all the elements of Set.
    %
:- pred all_true(pred(T)::in(pred(in) is semidet), diet(T)::in) is semidet
    <= diet_element(T).

    % 'filter(Pred, Set) = TrueSet' returns the elements of Set for which
    % Pred succeeds.
    %
:- func filter(pred(T), diet(T)) = diet(T) <= diet_element(T).
:- mode filter(pred(in) is semidet, in) = out is det.

    % 'filter(Pred, Set, TrueSet, FalseSet)' returns the elements of Set
    % for which Pred succeeds, and those for which it fails.
    %
:- pred filter(pred(T), diet(T), diet(T), diet(T)) <= diet_element(T).
:- mode filter(pred(in) is semidet, in, out, out) is det.

    % 'foldl_intervals(Pred, Set, Start)' calls Pred with each interval of
    % 'Set' (in sorted order) and an accumulator (with the initial value of
    % 'Start'), and returns the final value.
    %
:- pred foldl_intervals(pred(T, T, A, A), diet(T), A, A) <= diet_element(T).
:- mode foldl_intervals(pred(in, in, in, out) is det, in, in, out) is det.
:- mode foldl_intervals(pred(in, in, di, uo) is det, in, di, uo) is det.
:- mode foldl_intervals(pred(in, in, in, out) is semidet, in, in, out)
    is semidet.

    % 'foldr_intervals(Pred, Set, Start)' calls Pred with each interval of
    % 'Set' (in reverse sorted order) and an accumulator (with the initial
    % value of 'Start'), and returns the final value.
    %
:- pred foldr_intervals(pred(T, T, A, A), diet(T), A, A) <= diet_element(T).
:- mode foldr_intervals(pred(in, in, in, out) is det, in, in, out) is det.
:- mode foldr_intervals(pred(in, in, di, uo) is det, in, di, uo) is det.
:- mode foldr_intervals(pred(in, in, in, out) is semidet, in, in, out)
    is semidet.

    % 'foldl(Func, Set, Start)' calls Func with each element of 'Set'
```

```
    % (in sorted order) and an accumulator (with the initial value of 'Start'),
    % and returns the final value.
    %
:- func foldl(func(T, A) = A, diet(T), A) = A <= diet_element(T).

:- pred foldl(pred(T, A, A), diet(T), A, A) <= diet_element(T).
:- mode foldl(pred(in, in, out) is det, in, in, out) is det.
:- mode foldl(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode foldl(pred(in, di, uo) is det, in, di, uo) is det.
:- mode foldl(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode foldl(pred(in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode foldl(pred(in, di, uo) is semidet, in, di, uo) is semidet.

:- pred foldl2(pred(T, A, A, B, B), diet(T), A, A, B, B) <= diet_element(T).
:- mode foldl2(pred(in, in, out, in, out) is det, in,
    in, out, in, out) is det.
:- mode foldl2(pred(in, in, out, mdi, muo) is det, in,
    in, out, mdi, muo) is det.
:- mode foldl2(pred(in, in, out, di, uo) is det, in,
    in, out, di, uo) is det.
:- mode foldl2(pred(in, in, out, in, out) is semidet, in,
    in, out, in, out) is semidet.
:- mode foldl2(pred(in, in, out, mdi, muo) is semidet, in,
    in, out, mdi, muo) is semidet.
:- mode foldl2(pred(in, in, out, di, uo) is semidet, in,
    in, out, di, uo) is semidet.

:- pred foldl3(pred(T, A, A, B, B, C, C), diet(T),
    A, A, B, B, C, C) <= diet_element(T).
:- mode foldl3(pred(in, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out) is det.
:- mode foldl3(pred(in, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, mdi, muo) is det.
:- mode foldl3(pred(in, in, out, in, out, di, uo) is det, in,
    in, out, in, out, di, uo) is det.
:- mode foldl3(pred(in, in, out, in, out, in, out) is semidet, in,
    in, out, in, out, in, out) is semidet.
:- mode foldl3(pred(in, in, out, in, out, mdi, muo) is semidet, in,
    in, out, in, out, mdi, muo) is semidet.
:- mode foldl3(pred(in, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, di, uo) is semidet.

:- pred foldl4(pred(T, A, A, B, B, C, C, D, D), diet(T),
    A, A, B, B, C, C, D, D) <= diet_element(T).
:- mode foldl4(pred(in, in, out, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out, in, out) is det.
:- mode foldl4(pred(in, in, out, in, out, in, out, mdi, muo) is det, in,
```

```
    in, out, in, out, in, out, mdi, muo) is det.
:- mode foldl4(pred(in, in, out, in, out, in, out, di, uo) is det, in,
    in, out, in, out, in, out, di, uo) is det.
:- mode foldl4(pred(in, in, out, in, out, in, out, in, out) is semidet, in,
    in, out, in, out, in, out, in, out) is semidet.
:- mode foldl4(pred(in, in, out, in, out, in, out, mdi, muo) is semidet, in,
    in, out, in, out, in, out, mdi, muo) is semidet.
:- mode foldl4(pred(in, in, out, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, in, out, di, uo) is semidet.

:- pred foldl5(pred(T, A, A, B, B, C, C, D, D, E, E), diet(T),
    A, A, B, B, C, C, D, D, E, E) <= diet_element(T).
:- mode foldl5(
    pred(in, in, out, in, out, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out, in, out, in, out) is det.
:- mode foldl5(
    pred(in, in, out, in, out, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, in, out, in, out, mdi, muo) is det.
:- mode foldl5(
    pred(in, in, out, in, out, in, out, in, out, di, uo) is det,
    in, in, out, in, out, in, out, in, out, di, uo) is det.
:- mode foldl5(
    pred(in, in, out, in, out, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out, in, out, in, out) is semidet.
:- mode foldl5(
    pred(in, in, out, in, out, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode foldl5(
    pred(in, in, out, in, out, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, in, out, in, out, di, uo) is semidet.

:- func foldr(func(T, A) = A, diet(T), A) = A <= diet_element(T).

:- pred foldr(pred(T, A, A), diet(T), A, A) <= diet_element(T).
:- mode foldr(pred(in, in, out) is det, in, in, out) is det.
:- mode foldr(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode foldr(pred(in, di, uo) is det, in, di, uo) is det.
:- mode foldr(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode foldr(pred(in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode foldr(pred(in, di, uo) is semidet, in, di, uo) is semidet.

%-----------------------------------------------%
%-----------------------------------------------%
```

21 digraph

```
%----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%----------------------------------------------------%
% Copyright (C) 1995-1999,2002-2007,2010-2012 The University of Melbourne.
% Copyright (C) 2014-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%----------------------------------------------------%
%
% File: digraph.m
% Main author: bromage, petdr
% Stability: medium
%
% This module defines a data type representing directed graphs. A directed
% graph of type digraph(T) is logically equivalent to a set of vertices of
% type T, and a set of edges of type pair(T). The endpoints of each edge
% must be included in the set of vertices; cycles and loops are allowed.
%
%----------------------------------------------------%
%----------------------------------------------------%

:- module digraph.
:- interface.

:- import_module assoc_list.
:- import_module enum.
:- import_module list.
:- import_module map.
:- import_module pair.
:- import_module set.
:- import_module sparse_bitset.

%----------------------------------------------------%

    % The type of directed graphs with vertices in T.
    %
:- type digraph(T).

    % The abstract type that indexes vertices in a digraph. Each key is only
    % valid with the digraph it was created from -- predicates and functions
    % in this module may throw an exception if an invalid key is used.
    %
:- type digraph_key(T).

:- instance enum(digraph_key(T)).
```

```
:- type digraph_key_set(T) == sparse_bitset(digraph_key(T)).

    % init creates an empty digraph.
    %
:- func init = digraph(T).
:- pred init(digraph(T)::out) is det.

    % add_vertex adds a vertex to the domain of a digraph.
    % Returns the old key if one already exists for this vertex,
    % otherwise it allocates a new key.
    %
:- pred add_vertex(T::in, digraph_key(T)::out,
    digraph(T)::in, digraph(T)::out) is det.

    % search_key returns the key associated with a vertex.
    % Fails if the vertex is not in the graph.
    %
:- pred search_key(digraph(T)::in, T::in, digraph_key(T)::out) is semidet.

    % lookup_key returns the key associated with a vertex.
    % Throws an exception if the vertex is not in the graph.
    %
:- func lookup_key(digraph(T), T) = digraph_key(T).
:- pred lookup_key(digraph(T)::in, T::in, digraph_key(T)::out) is det.

    % lookup_vertex returns the vertex associated with a key.
    %
:- func lookup_vertex(digraph(T), digraph_key(T)) = T.
:- pred lookup_vertex(digraph(T)::in, digraph_key(T)::in, T::out) is det.

    % add_edge adds an edge to the digraph if it doesn't already
    % exist, and leaves the digraph unchanged otherwise.
    %
:- func add_edge(digraph_key(T), digraph_key(T), digraph(T)) = digraph(T).
:- pred add_edge(digraph_key(T)::in, digraph_key(T)::in,
    digraph(T)::in, digraph(T)::out) is det.

    % add_vertices_and_edge adds a pair of vertices and an edge
    % between them to the digraph.
    %
    % add_vertices_and_edge(X, Y, !G) :-
    %    add_vertex(X, XKey, !G),
    %    add_vertex(Y, YKey, !G),
    %    add_edge(XKey, YKey, !G).
    %
:- func add_vertices_and_edge(T, T, digraph(T)) = digraph(T).
:- pred add_vertices_and_edge(T::in, T::in,
```

```
    digraph(T)::in, digraph(T)::out) is det.

    % As above, but takes a pair of vertices in a single argument.
    %
:- func add_vertex_pair(pair(T), digraph(T)) = digraph(T).
:- pred add_vertex_pair(pair(T)::in, digraph(T)::in, digraph(T)::out) is det.

    % add_assoc_list adds a list of edges to a digraph.
    %
:- func add_assoc_list(assoc_list(digraph_key(T), digraph_key(T)),
    digraph(T)) = digraph(T).
:- pred add_assoc_list(assoc_list(digraph_key(T), digraph_key(T))::in,
    digraph(T)::in, digraph(T)::out) is det.

    % delete_edge deletes an edge from the digraph if it exists,
    % and leaves the digraph unchanged otherwise.
    %
:- func delete_edge(digraph_key(T), digraph_key(T), digraph(T)) = digraph(T).
:- pred delete_edge(digraph_key(T)::in, digraph_key(T)::in,
    digraph(T)::in, digraph(T)::out) is det.

    % delete_assoc_list deletes a list of edges from a digraph.
    %
:- func delete_assoc_list(assoc_list(digraph_key(T), digraph_key(T)),
    digraph(T)) = digraph(T).
:- pred delete_assoc_list(
    assoc_list(digraph_key(T), digraph_key(T))::in,
    digraph(T)::in, digraph(T)::out) is det.

    % is_edge checks to see if an edge is in the digraph.
    %
:- pred is_edge(digraph(T), digraph_key(T), digraph_key(T)).
:- mode is_edge(in, in, out) is nondet.
:- mode is_edge(in, in, in) is semidet.

    % is_edge_rev is equivalent to is_edge, except that
    % the nondet mode works in the reverse direction.
    %
:- pred is_edge_rev(digraph(T), digraph_key(T), digraph_key(T)).
:- mode is_edge_rev(in, out, in) is nondet.
:- mode is_edge_rev(in, in, in) is semidet.

    % Given key x, lookup_from returns the set of keys y such that
    % there is an edge (x,y) in the digraph.
    %
:- func lookup_from(digraph(T), digraph_key(T)) = set(digraph_key(T)).
:- pred lookup_from(digraph(T)::in, digraph_key(T)::in,
```

```
    set(digraph_key(T))::out) is det.

    % As above, but returns a digraph_key_set.
    %
:- func lookup_key_set_from(digraph(T), digraph_key(T)) = digraph_key_set(T).
:- pred lookup_key_set_from(digraph(T)::in, digraph_key(T)::in,
    digraph_key_set(T)::out) is det.

    % Given a key y, lookup_to returns the set of keys x such that
    % there is an edge (x,y) in the digraph.
    %
:- func lookup_to(digraph(T), digraph_key(T)) = set(digraph_key(T)).
:- pred lookup_to(digraph(T)::in, digraph_key(T)::in,
    set(digraph_key(T))::out) is det.

    % As above, but returns a digraph_key_set.
    %
:- func lookup_key_set_to(digraph(T), digraph_key(T)) = digraph_key_set(T).
:- pred lookup_key_set_to(digraph(T)::in, digraph_key(T)::in,
    digraph_key_set(T)::out) is det.

%-----------------------------------------------------%

    % to_assoc_list turns a digraph into a list of pairs of vertices,
    % one for each edge.
    %
:- func to_assoc_list(digraph(T)) = assoc_list(T, T).
:- pred to_assoc_list(digraph(T)::in, assoc_list(T, T)::out) is det.

    % to_key_assoc_list turns a digraph into a list of pairs of keys,
    % one for each edge.
    %
:- func to_key_assoc_list(digraph(T)) =
    assoc_list(digraph_key(T), digraph_key(T)).
:- pred to_key_assoc_list(digraph(T)::in,
    assoc_list(digraph_key(T), digraph_key(T))::out) is det.

    % from_assoc_list turns a list of pairs of vertices into a digraph.
    %
:- func from_assoc_list(assoc_list(T, T)) = digraph(T).
:- pred from_assoc_list(assoc_list(T, T)::in, digraph(T)::out) is det.

%-----------------------------------------------------%

    % dfs(G, Key, Dfs) is true if Dfs is a depth-first sorting of G
    % starting at Key. The set of keys in the list Dfs is equal to the
    % set of keys reachable from Key.
```

```
    %
:- func dfs(digraph(T), digraph_key(T)) = list(digraph_key(T)).
:- pred dfs(digraph(T)::in, digraph_key(T)::in,
    list(digraph_key(T))::out) is det.

    % dfsrev(G, Key, DfsRev) is true if DfsRev is a reverse
    % depth-first sorting of G starting at Key. The set of keys in the
    % list DfsRev is equal to the set of keys reachable from Key.
    %
:- func dfsrev(digraph(T), digraph_key(T)) = list(digraph_key(T)).
:- pred dfsrev(digraph(T)::in, digraph_key(T)::in,
    list(digraph_key(T))::out) is det.

    % dfs(G, Dfs) is true if Dfs is a depth-first sorting of G.
    % If one considers each edge to point from a parent node to a child node,
    % then Dfs will be a list of all the keys in G such that all keys for
    % the children of a vertex are placed in the list before the parent key.
    %
    % If the digraph is cyclic, the position in which cycles are broken
    % (that is, in which a child is placed *after* its parent) is undefined.
    %
:- func dfs(digraph(T)) = list(digraph_key(T)).
:- pred dfs(digraph(T)::in, list(digraph_key(T))::out) is det.

    % dfsrev(G, DfsRev) is true if DfsRev is a reverse depth-first
    % sorting of G. That is, DfsRev is the reverse of Dfs from dfs/2.
    %
:- func dfsrev(digraph(T)) = list(digraph_key(T)).
:- pred dfsrev(digraph(T)::in, list(digraph_key(T))::out) is det.

    % dfs(G, Key, !Visit, Dfs) is true if Dfs is a depth-first
    % sorting of G starting at Key, assuming we have already visited !.Visit
    % vertices. That is, Dfs is a list of vertices such that all the
    % unvisited children of a vertex are placed in the list before the
    % parent. !.Visit allows us to initialise a set of previously visited
    % vertices. !:Visit is Dfs + !.Visit.
    %
:- pred dfs(digraph(T)::in, digraph_key(T)::in, digraph_key_set(T)::in,
    digraph_key_set(T)::out, list(digraph_key(T))::out) is det.

    % dfsrev(G, Key, !Visit, DfsRev) is true if DfsRev is a
    % reverse depth-first sorting of G starting at Key providing we have
    % already visited !.Visit nodes, ie the reverse of Dfs from dfs/5.
    % !:Visit is !.Visit + DfsRev.
    %
:- pred dfsrev(digraph(T)::in, digraph_key(T)::in,
    digraph_key_set(T)::in, digraph_key_set(T)::out,
```

```
    list(digraph_key(T))::out) is det.

%-----------------------------------------------%

    % vertices returns the set of vertices in a digraph.
    %
:- func vertices(digraph(T)) = set(T).
:- pred vertices(digraph(T)::in, set(T)::out) is det.

    % inverse(G, G') is true iff the domains of G and G' are equal,
    % and for all x, y in this domain, (x,y) is an edge in G iff (y,x) is
    % an edge in G'.
    %
:- func inverse(digraph(T)) = digraph(T).
:- pred inverse(digraph(T)::in, digraph(T)::out) is det.

    % compose(G1, G2, G) is true if G is the composition
    % of the digraphs G1 and G2. That is, there is an edge (x,y) in G iff
    % there exists vertex m such that (x,m) is in G1 and (m,y) is in G2.
    %
:- func compose(digraph(T), digraph(T)) = digraph(T).
:- pred compose(digraph(T)::in, digraph(T)::in, digraph(T)::out)
    is det.

    % is_dag(G) is true iff G is a directed acyclic graph.
    %
:- pred is_dag(digraph(T)::in) is semidet.

    % components(G, Comp) is true if Comp is the set of the
    % connected components of G.
    %
:- func components(digraph(T)) = set(set(digraph_key(T))).
:- pred components(digraph(T)::in, set(set(digraph_key(T)))::out)
    is det.

    % cliques(G, Cliques) is true if Cliques is the set of the
    % cliques (strongly connected components) of G.
    %
:- func cliques(digraph(T)) = set(set(digraph_key(T))).
:- pred cliques(digraph(T)::in, set(set(digraph_key(T)))::out) is det.

    % reduced(G, R) is true if R is the reduced digraph (digraph of cliques)
    % obtained from G.
    %
:- func reduced(digraph(T)) = digraph(set(T)).
:- pred reduced(digraph(T)::in, digraph(set(T))::out) is det.
```

```
    % As above, but also return a map from each key in the original digraph
    % to the key for its clique in the reduced digraph.
    %
:- pred reduced(digraph(T)::in, digraph(set(T))::out,
    map(digraph_key(T), digraph_key(set(T)))::out) is det.

    % tsort(G, TS) is true if TS is a topological sorting of G.
    %
    % If we view each edge in the digraph as representing a <from, to>
    % relationship, then TS will contain a vertex "from" *before*
    % all the other vertices "to" for which a <from, to> edge exists
    % in the graph. In other words, TS will be in from-to order.
    %
    % tsort fails if G is cyclic.
    %
:- pred tsort(digraph(T)::in, list(T)::out) is semidet.

    % Both these predicates do a topological sort of G.
    %
    % return_vertices_in_from_to_order(G, TS) is a synonym for tsort(G, TS).
    % return_vertices_in_to_from_order(G, TS) is identical to both
    % except for the fact that it returns the vertices in the opposite order.
    %
:- pred return_vertices_in_from_to_order(digraph(T)::in, list(T)::out)
    is semidet.
:- pred return_vertices_in_to_from_order(digraph(T)::in, list(T)::out)
    is semidet.

    % atsort(G, ATS) is true if ATS is a topological sorting
    % of the strongly connected components (SCCs) in G.
    %
    % If we view each edge in the digraph as representing a <from, to>
    % relationship, then ATS will contain SCC A before all SCCs B
    % for which there is a vertex <from, to> with "from" being in SCC A
    % and "to" being in SCC B. In other words, ATS will be in from-to order.
    %
:- func atsort(digraph(T)) = list(set(T)).
:- pred atsort(digraph(T)::in, list(set(T))::out) is det.

    % Both these predicates do a topological sort of the strongly connected
    % components (SCCs) of G.
    %
    % return_sccs_in_from_to_order(G) = ATS is a synonym for atsort(G) = ATS.
    % return_sccs_in_to_from_order(G) = ATS is identical to both
    % except for the fact that it returns the SCCs in the opposite order.
    %
:- func return_sccs_in_from_to_order(digraph(T)) = list(set(T)).
```

```
:- func return_sccs_in_to_from_order(digraph(T)) = list(set(T)).

    % sc(G, SC) is true if SC is the symmetric closure of G.
    % That is, (x,y) is in SC iff either (x,y) or (y,x) is in G.
    %
:- func sc(digraph(T)) = digraph(T).
:- pred sc(digraph(T)::in, digraph(T)::out) is det.

    % tc(G, TC) is true if TC is the transitive closure of G.
    %
:- func tc(digraph(T)) = digraph(T).
:- pred tc(digraph(T)::in, digraph(T)::out) is det.

    % rtc(G, RTC) is true if RTC is the reflexive transitive closure of G.
    %
:- func rtc(digraph(T)) = digraph(T).
:- pred rtc(digraph(T)::in, digraph(T)::out) is det.

    % traverse(G, ProcessVertex, ProcessEdge) will traverse a digraph
    % calling ProcessVertex for each vertex in the digraph and ProcessEdge for
    % each edge in the digraph. Each vertex is processed followed by all the
    % edges originating at that vertex, until all vertices have been processed.
    %
:- pred traverse(digraph(T), pred(T, A, A), pred(T, T, A, A), A, A).
:- mode traverse(in, pred(in, di, uo) is det,
    pred(in, in, di, uo) is det, di, uo) is det.
:- mode traverse(in, pred(in, in, out) is det,
    pred(in, in, in, out) is det, in, out) is det.

%---------------------------------------------------%
%---------------------------------------------------%
```

22 dir

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 1994-1995,1997,1999-2000,2002-2012 The University of Melbourne.
% Copyright (C) 2016-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: dir.m.
% Main authors: fjh, stayl.
% Stability: high.
```

```
%
% Filename and directory handling.
%
% Note that the predicates and functions in this module change directory
% separators in paths passed to them to the normal separator for the platform,
% if that doesn't change the meaning of the path name.
%
% Duplicate directory separators and trailing separators are also removed
% where that doesn't change the meaning of the path name.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module dir.
:- interface.

:- import_module bool.
:- import_module io.
:- import_module list.

%-----------------------------------------------------%
%
% Predicates to isolate system dependencies
%

    % Returns the default separator between components of a pathname --

    % '/' on Unix systems and '\\' on Microsoft Windows systems.
    %
:- func directory_separator = character.
:- pred directory_separator(character::out) is det.

    % Is the character a directory separator.
    % On Microsoft Windows systems this will succeed for '/'
    % as well as '\\'.
    %
:- pred is_directory_separator(character).
:- mode is_directory_separator(in) is semidet.
:- mode is_directory_separator(out) is multi.

    % Returns ".".
    %
:- func this_directory = string.
:- pred this_directory(string::out) is det.

    % Returns "..".
    %
```

```
:- func parent_directory = string.
:- pred parent_directory(string::out) is det.

    % split_name(PathName, DirName, BaseName).
    %
    % Split a filename into a directory part and a filename part.
    %
    % Fails for root directories or relative filenames not containing
    % directory information.
    %
    % Trailing slashes are removed from PathName before splitting,
    % if that doesn't change the meaning of PathName.
    %
    % Trailing slashes are removed from DirName after splitting,
    % if that doesn't change the meaning of DirName.
    %
    % On Windows, drive current directories are handled correctly,
    % for example `split_name("C:foo", "C:", "foo")'.
    % (`X:' is the current directory on drive `X').
    % Note that Cygwin doesn't support drive current directories,
    % so `split_name("C:foo, _, _)' will fail when running under Cygwin.
    %
:- pred split_name(string::in, string::out, string::out) is semidet.

    % basename(PathName) = BaseName.
    %
    % Returns the non-directory part of a filename.
    %
    % Fails when given a root directory, ".", ".." or a Windows path
    % such as "X:".
    %
    % Trailing slashes are removed from PathName before splitting,
    % if that doesn't change the meaning of PathName.
    %
:- func basename(string) = string is semidet.
:- pred basename(string::in, string::out) is semidet.

    % As above, but throws an exception instead of failing.
    %
:- func det_basename(string) = string.

    % dirname(PathName) = DirName.
    %
    % Returns the directory part of a filename.
    %
    % Returns PathName if it specifies a root directory.
    %
```

```
    % Returns PathName for Windows paths such as "X:".
    %
    % Returns ‘this_directory’ when given a filename
    % without any directory information (e.g. "foo").
    %
    % Trailing slashes in PathName are removed first, if that doesn’t change
    % the meaning of PathName.
    %
    % Trailing slashes are removed from DirName after splitting,
    % if that doesn’t change the meaning of DirName.
    %
:- func dirname(string) = string.
:- pred dirname(string::in, string::out) is det.

    % path_name_is_root_directory(PathName)
    %
    % On Unix, ’/’ is the only root directory.
    % On Windows, a root directory is one of the following:
    %   ’X:\’, which specifies the root directory of drive X,
    %       where X is any letter.
    %   ’\’, which specifies the root directory of the current drive.
    %   ’\\server\share\’, which specifies a UNC (Universal Naming
    %       Convention) root directory for a network drive.
    %
    % Note that ’X:’ is not a Windows root directory -- it specifies the
    % current directory on drive X, where X is any letter.
    %
:- pred path_name_is_root_directory(string::in) is semidet.

    % path_name_is_absolute(PathName)
    %
    % Is the path name syntactically an absolute path
    % (this doesn’t check whether the path exists).
    %
    % An path is absolute iff it begins with a root directory
    % (see path_name_is_root_directory).
    %
:- pred path_name_is_absolute(string::in) is semidet.

    % PathName = DirName / FileName
    %
    % Given a directory name and a filename, return the pathname of that
    % file in that directory.
    %
    % Duplicate directory separators will not be introduced if
    % DirName ends with a directory separator.
    %
```

```
    % On Windows, a call such as `"C:"/"foo"' will return "C:foo".
    %
    % Throws an exception if FileName is an absolute path name.
    % Throws an exception on Windows if FileName is a current
    % drive relative path such as "C:".
    %
:- func string / string = string.
:- func make_path_name(string, string) = string.

    % relative_path_name_from_components(List) = PathName.
    %
    % Return the relative pathname from the components in the list.
    % The components of the list must not contain directory separators.
    %
:- func relative_path_name_from_components(list(string)) = string.

%---------------------------------------------------%

    % current_directory(Result)
    % Return the current working directory.
    %
:- pred current_directory(io.res(string)::out, io::di, io::uo) is det.

%---------------------------------------------------%

    % Make the given directory, and all parent directories.
    % This will also succeed if the directory already exists
    % and is readable and writable by the current user.
    %
:- pred make_directory(string::in, io.res::out, io::di, io::uo) is det.

    % Make only the given directory.
    % Fails if the directory already exists, or the parent directory doesn't.
    %
:- pred make_single_directory(string::in, io.res::out, io::di, io::uo)
    is det.

%---------------------------------------------------%

    % FoldlPred(DirName, BaseName, FileType, Continue, !Data, !IO).
    %
    % A predicate passed to foldl2 to process each entry in a directory.
    % Processing will stop if Continue is bound to `no'.
    %
:- type foldl_pred(T) ==
    pred(string, string, io.file_type, bool, T, T, io, io).
:- inst foldl_pred == (pred(in, in, in, out, in, out, di, uo) is det).
```

```
    % foldl2(P, DirName, InitialData, Result, !IO).
    %
    % Apply 'P' to all files and directories in the given directory.
    % Directories are not processed recursively.
    % Processing will stop if the boolean (Continue) output of P is bound
    % to 'no'.
    % The order in which the entries are processed is unspecified.
    %
:- pred foldl2(foldl_pred(T)::in(foldl_pred), string::in,
    T::in, io.maybe_partial_res(T)::out, io::di, io::uo) is det.

    % recursive_foldl2(P, DirName, FollowSymLinks, InitialData, Result, !IO).
    %
    % As above, but recursively process subdirectories.
    % Subdirectories are processed depth-first, processing the directory itself
    % before its contents. If 'FollowSymLinks' is 'yes', recursively process
    % the directories referenced by symbolic links.
    %
:- pred recursive_foldl2(foldl_pred(T)::in(foldl_pred),
    string::in, bool::in, T::in, io.maybe_partial_res(T)::out,
    io::di, io::uo) is det.

%---------------------------------------------------%

    % Implement brace expansion, as in sh: return the sequence of strings
    % generated from the given input string. Throw an exception if the
    % input string contains mismatched braces.
    %
    % The following is the documentation of brace expansion from the sh manual:
    %
    %   Brace expansion is a mechanism by which arbitrary strings may be
    %   generated. This mechanism is similar to pathname expansion, but the
    %   filenames generated need not exist. Patterns to be brace expanded
    %   take the form of an optional preamble, followed by a series of
    %   comma-separated strings between a pair of braces, followed by an
    %   optional postscript. The preamble is prefixed to each string contained
    %   within the braces, and the postscript is then appended to each
    %   resulting string, expanding left to right.
    %
    %   Brace expansions may be nested. The results of each expanded string
    %   are not sorted; left to right order is preserved. For example,
    %   a{d,c,b}e expands into 'ade ace abe'.
    %
:- func expand_braces(string) = list(string).

%---------------------------------------------------%
```

```
%-----------------------------------------------------%
```

23 edit_seq

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 2019 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: edit_seq.m.
% Stability: medium.
%
% This module finds an edit sequence, which means that given two sequences
% of items, it finds the shortest sequence of edit operations (deletes,
% inserts and/or replaces) that will transform the first sequence
% into the second.
%
% The code is a naive implementation of the Wagner-Fischer algorithm,
% which is documented on its own wikipedia page.
%
% Given two lists of length M and N, its time complexity is O(MN),
% so it is suitable for use only on reasonably short lists.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module edit_seq.
:- interface.

:- import_module list.

%-----------------------------------------------------%
%-----------------------------------------------------%

    % Given two item sequences A and B, the edit sequence is the sequence
    % of edit operations that transforms sequence A into sequence B.
    %
    % Item numbers start at 1. The item numbers in edit operations reflect
    % the *original* position of the relevant item, i.e. they are not affected
    % by any edit operations that take place before that position.
    %
:- type edit_seq(T) == list(edit(T)).
:- type edit(T)
```

```
    --->    delete(int)
            % Delete item #N in sequence A.

    ;       insert(int, T)
            % Insert the given item from sequence B
            % after item #N in sequence A.

    ;       replace(int, T).
            % Replace item #N in sequence A with the given item
            % from sequence B.

:- type edit_params
    --->    edit_params(
                % The cost of delete, insert and replace operations
                % respectively. Only the *relative* values of the costs matter;
                % if these are fixed, their *absolute* values are irrelevant
                % (unless they are so high that they cause arithmetic
                % overflows).
                cost_of_delete      :: int,
                cost_of_insert      :: int,
                cost_of_replace     :: int
            ).

    % find_shortest_edit_seq(Params, SeqA, SeqB, Edits):
    %
    % Compute Edits as the cheapest sequence of edit operations
    % that will transform SeqA into SeqB, where the cost of each kind of
    % edit operation is specified by Params.
    %
:- pred find_shortest_edit_seq(edit_params::in, list(T)::in, list(T)::in,
    edit_seq(T)::out) is det.

%-----------------------------------------------%

    % A diff_seq represents a unified diff with unlimited context,
    % such as the output of "diff -u --context=MAXINT".
    %
    % Each line (or in general, one item) in it can be an item from SeqA
    % that is left unchanged, an item from SeqA that is to be deleted, or
    % an item (from SeqB) that is to be inserted.
:- type diff_seq(T) == list(diff(T)).
:- type diff(T)
    --->    unchanged(T)
    ;       deleted(T)
    ;       inserted(T).

    % Given an edit sequence computed by find_shortest_edit_seq, return
```

```
    % the unified diff representing that edit sequence.
    %
    % The main difference between the edit sequence and the diff sequence
    % is that given several consecutive replace edits, a naive representation
    % of those edit operations would output interleaved pairs of items
    % to be deleted and inserted, while the diff sequence would output
    % *all* of the items to be deleted by those replace operations *before*
    % printing the insertions of their replacements.
    %
:- pred find_diff_seq(list(T)::in, edit_seq(T)::in, diff_seq(T)::out) is det.

%-----------------------------------------------------%

    % This type and its fields are documented below.
:- type change_hunk(T)
    --->    change_hunk(
                ch_seq_a_start      :: int,
                ch_seq_a_length     :: int,
                ch_seq_b_start      :: int,
                ch_seq_b_length     :: int,
                ch_diff             :: diff_seq(T)
            ).

    % find_change_hunks(ContextSize, DiffSeq, ChangeHunks):
    %
    % A diff_seq may contain long sequences of unchanged items, which are
    % often not of interest. This predicate computes from a diff sequence
    % a list of its *change hunks*, which are its interesting parts,
    % the parts that contain insertions and/or deletions.
    %
    % A change hunk looks like this, using the syntax of "diff -u".
    % The ContextSize of this example is 3.
    %
    %   @@ -25,6 +25,7 @@
    %    Roosevelt
    %    Taft
    %    Wilson
    %   +Pershing
    %    Harding
    %    Coolidge
    %    Hoover
    %
    % This change hunk shows the insertion of one line containing "Pershing"
    % into a list of US presidents. The "-25,6" part of the header shows that
    % the part of the original sequence (sequence A) covered by this change
    % hunk contains six lines, starting at line 25. The "+25,7" part shows that
    % the part of the updated sequence (sequence B) contains seven lines,
```

```
    % starting at line at 25 in that sequence as well. The first four fields
    % of the change_hunk type contain these two pairs of numbers.
    %
    % A change hunk consists of three parts, of which the first and/or last
    % may be empty.
    %
    % - The first part is a sequence of up to ContextSize unchanged items
    %   (the initial context).
    % - The second part is a sequence of unchanged, insertion or deletion
    %   items that
    %       * starts with an insertion or deletion item,
    %       * ends with an insertion or deletion item, and
    %       * contains at most 2 * ContextSize consecutive unchanged items.
    %   The start and end item may be the same, as in the example above.
    % - The third part is a sequence of up to ContextSize unchanged items
    %   (the trailing context).
    %
    % The idea is to surround regions of changes with ContextSize unchanged
    % items to provide context (hence the name ContextSize). The first and
    % third parts will always contain *exactly* ContextSize unchanged items,
    % unless the changed region occurs so close to the start or to the end
    % of the item sequence that there are fewer than ContextSize unchanged
    % items there.
    %
    % The reason why there may be up to 2 * ContextSize consecutive unchanged
    % items in the middle of a change hunk is that if the limit were any lower,
    % then some of those unchanged items would end up *both* in the trailing
    % context of one change hunk and the initial context of the next change
    % hunk.
    %
    % To make sense, ContextSize must be least one. This predicate throws
    % an exception if ContextSize is zero or negative.
    %
:- pred find_change_hunks(int::in, diff_seq(T)::in,
    list(change_hunk(T))::out) is det.

%-----------------------------------------------------%
%-----------------------------------------------------%
```

24 enum

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 2000, 2005-2006 The University of Melbourne.
```

```
% Copyright (C) 2014-2015, 2017-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: enum.m.
% Author: stayl.
% Stability: medium
%
% This module provides the typeclass ‘enum’, which describes types
% which can be converted to and from integers without loss of information.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module enum.
:- interface.

    % A type T can be declared to be a member of the enum typeclass if
    %
    % - all values X of type T can be converted to an int N using
    %   the instance's to_int member function, with each distinct X
    %   being translated by to_int to a distinct N; and
    %
    % - for all values N of type int that are equal to to_int(X) for some X,
    %   from_int(N) = X.
    %
    % - for all values N of type int that are not equal to to_int(X) for any X,
    %   from_int(N) should fail.
    %
    % In mathematical notation, the following must hold:
    %
    %   all [X] (X = from_int(to_int(X)))
    %   all [X, Y] (to_int(X) = to_int(Y)) => X = Y)
    %   all [N] (some [X] N = to_int(X) => from_int(N) = X)
    %   all [N] (not (some [X] N = to_int(X))) => from_int(N) fails
    %
:- typeclass enum(T) where [
    func to_int(T) = int,
    func from_int(int) = T is semidet
].

    % ‘det_from_int(I)’ returns the result of ‘from_int(I)’, but throws an
    % exception if ‘from_int’ fails.
    %
:- func det_from_int(int) = T <= enum(T).

%-----------------------------------------------------%
```

```
%-----------------------------------------------------%
```

25 eqvclass

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 1995-1997, 1999, 2003-2006, 2011-2012 The University of Melbourne.
% Copyright (C) 2014-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: eqvclass.m.
% Author: zs.
% Stability: low.
%
% A module for handling equivalence classes.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module eqvclass.
:- interface.

:- import_module list.
:- import_module set.

%-----------------------------------------------------%

:- type eqvclass(T).
:- type partition_id.

    % Create an empty equivalence class.
    %
:- func init = eqvclass(T).
:- pred init(eqvclass(T)::out) is det.

    % Is this item known to the equivalence class?
    %
:- pred is_member(eqvclass(T)::in, T::in) is semidet.

    % If this item is known to the equivalence class, return the id of its
    % partition. The only use that the caller can make of the partition id
    % is to check whether two items in the same equivalence calls have the
    % same partition id; that test will succeed if and only if the two
```

```
    % items are in the same partition. Partition ids are not guaranteed
    % to stay the same as an eqvclass is updated, so such comparisons will
    % work only against the same eqvclass.
    %
    % If you want to check whether two items are in the same equivalence class,
    % using same_eqvclass is more expressive than calling
    % partition_id on both items and comparing the results.
    % However, if you want to perform this check on X and Y1, on X and Y2,
    % ... X and Yn, then calling partition_id on X just once and
    % comparing this with the partition_ids of the Yi will be more efficient.
    %
:- pred partition_id(eqvclass(T)::in, T::in, partition_id::out)
    is semidet.

    % Make an element known to the equivalence class.
    % The element may already be known to the class;
    % if it isn't, it is created without any equivalence relationships.
    %
:- func ensure_element(eqvclass(T), T) = eqvclass(T).
:- pred ensure_element(T::in, eqvclass(T)::in, eqvclass(T)::out)
    is det.

    % Make this item known to the equivalence class if it isn't already,
    % and return the id of its partition. The same proviso applies with
    % respect to partition_ids as with partition_id.
    %
:- pred ensure_element_partition_id(T::in, partition_id::out,
    eqvclass(T)::in, eqvclass(T)::out) is det.

    % Make an element known to the equivalence class.
    % The element must not already be known to the class;
    % it is created without any equivalence relationships.
    %
:- func new_element(eqvclass(T), T) = eqvclass(T).
:- pred new_element(T::in, eqvclass(T)::in, eqvclass(T)::out) is det.

    % Make two elements of the equivalence class equivalent.
    % It is ok if they already are.
    %
:- func ensure_equivalence(eqvclass(T), T, T) = eqvclass(T).
:- pred ensure_equivalence(T::in, T::in,
    eqvclass(T)::in, eqvclass(T)::out) is det.

:- func ensure_corresponding_equivalences(list(T), list(T),
    eqvclass(T)) = eqvclass(T).
:- pred ensure_corresponding_equivalences(list(T)::in, list(T)::in,
    eqvclass(T)::in, eqvclass(T)::out) is det.
```

```
    % Make two elements of the equivalence class equivalent.
    % It is an error if they are already equivalent.
    %
:- func new_equivalence(eqvclass(T), T, T) = eqvclass(T).
:- pred new_equivalence(T::in, T::in, eqvclass(T)::in, eqvclass(T)::out)
    is det.

    % Test if two elements are equivalent.
    %
:- pred same_eqvclass(eqvclass(T)::in, T::in, T::in) is semidet.

    % Test if a list of elements are equivalent.
    %
:- pred same_eqvclass_list(eqvclass(T)::in, list(T)::in) is semidet.

    % Return the set of the partitions of the equivalence class.
    %
:- func partition_set(eqvclass(T)) = set(set(T)).
:- pred partition_set(eqvclass(T)::in, set(set(T))::out) is det.

    % Return a list of the partitions of the equivalence class.
    %
:- func partition_list(eqvclass(T)) = list(set(T)).
:- pred partition_list(eqvclass(T)::in, list(set(T))::out) is det.

    % Create an equivalence class from a partition set.
    % It is an error if the sets are not disjoint.
    %
:- func partition_set_to_eqvclass(set(set(T))) = eqvclass(T).
:- pred partition_set_to_eqvclass(set(set(T))::in, eqvclass(T)::out) is det.

    % Create an equivalence class from a list of partitions.
    % It is an error if the sets are not disjoint.
    %
:- func partition_list_to_eqvclass(list(set(T))) = eqvclass(T).
:- pred partition_list_to_eqvclass(list(set(T))::in,
    eqvclass(T)::out) is det.

    % Return the set of elements equivalent to the given element.
    % This set will of course include the given element.
    %
:- func get_equivalent_elements(eqvclass(T), T) = set(T).

    % Return the smallest element equivalent to the given element.
    % This may or may not be the given element.
    %
```

```
:- func get_minimum_element(eqvclass(T), T) = T.

    % Remove the given element and all other elements equivalent to it
    % from the given equivalence class.
    %
:- func remove_equivalent_elements(eqvclass(T), T) = eqvclass(T).
:- pred remove_equivalent_elements(T::in,
    eqvclass(T)::in, eqvclass(T)::out) is det.

    % Given a function, divide each partition in the original equivalence class
    % so that two elements of the original partition end up in the same
    % partition in the new equivalence class if and only if the function maps
    % them to the same value.
    %
:- func divide_equivalence_classes(func(T) = U, eqvclass(T)) = eqvclass(T).

%-----------------------------------------------------%
%-----------------------------------------------------%
```

26 exception

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 1997-2008, 2010-2011 The University of Melbourne.
% Copyright (C) 2014-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: exception.m.
% Main author: fjh.
% Stability: medium.
%
% This module defines the Mercury interface for exception handling.
%
% Note that throwing an exception across the C interface won't work.
% That is, if a Mercury procedure that is exported to C using
% 'pragma foreign_export' throws an exception which is not caught within that
% procedure, then you will get undefined behaviour.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module exception.
:- interface.
```

```
:- import_module io.
:- import_module list.
:- import_module maybe.
:- import_module store.
:- import_module univ.

%-----------------------------------------------------%

    % Exceptions of this type are used by many parts of the Mercury
    % implementation to indicate an internal error.
    %
:- type software_error
    --->    software_error(string).

    % A domain error exception, which indicates that the inputs
    % to a predicate or function were outside the domain of that
    % predicate or function. The string indicates where the error occurred.
    %
:- type domain_error
    --->    domain_error(string).

%-----------------------------------------------------%

    % throw(Exception):
    %
    % Throw the specified exception.
    %
:- func throw(T) = _ is erroneous.
:- pred throw(T::in) is erroneous.

    % rethrow(ExceptionResult):
    %
    % Rethrows the specified exception result (which should be
    % of the form `exception(_)', not `succeeded(_)' or `failed'.).
    %
:- pred rethrow(exception_result(T)).
:- mode rethrow(in(bound(exception(ground)))) is erroneous.

:- func rethrow(exception_result(T)) = _.
:- mode rethrow(in(bound(exception(ground)))) = out is erroneous.

% The following type and inst are used by try/3 and try/5.

:- type exception_result(T)
    --->    succeeded(T)
    ;       failed
```

```
    ;       exception(univ).

:- inst cannot_fail for exception_result/1
    --->    succeeded(ground)
    ;       exception(ground).

    % try(Goal, Result):
    %
    % Operational semantics:
    %
    %   Call Goal(R).
    %   If Goal(R) fails, succeed with Result = failed.
    %   If Goal(R) succeeds, succeed with Result = succeeded(R).
    %   If Goal(R) throws an exception E, succeed with Result = exception(E).
    %
    % Declarative semantics:
    %
    %   try(Goal, Result) <=>
    %       ( Goal(R), Result = succeeded(R)
    %       ; not Goal(_), Result = failed
    %       ; Result = exception(_)
    %       ).
    %
:- pred try(pred(T),                exception_result(T)).
:- mode try(pred(out) is det,       out(cannot_fail)) is cc_multi.
:- mode try(pred(out) is semidet,   out)              is cc_multi.
:- mode try(pred(out) is cc_multi,  out(cannot_fail)) is cc_multi.
:- mode try(pred(out) is cc_nondet, out)              is cc_multi.

    % try_io(Goal, Result, IO_0, IO):
    %
    % Operational semantics:
    %
    %   Call Goal(R, IO_0, IO_1).
    %   If it succeeds, succeed with Result = succeeded(R) and IO = IO_1.
    %   If it throws an exception E, succeed with Result = exception(E)
    %   and with the final IO state being whatever state resulted from
    %   the partial computation from IO_0.
    %
    % Declarative semantics:
    %
    %   try_io(Goal, Result, IO_0, IO) <=>
    %       ( Goal(R, IO_0, IO), Result = succeeded(R)
    %       ; Result = exception(_)
    %       ).
    %
:- pred try_io(pred(T, io, io), exception_result(T), io, io).
```

```
:- mode try_io(pred(out, di, uo) is det,
    out(cannot_fail), di, uo) is cc_multi.
:- mode try_io(pred(out, di, uo) is cc_multi,
    out(cannot_fail), di, uo) is cc_multi.

    % try_store(Goal, Result, Store_0, Store):
    %
    % Just like try_io, but for stores rather than io.states.
    %
:- pred try_store(pred(T, store(S), store(S)),
    exception_result(T), store(S), store(S)).
:- mode try_store(pred(out, di, uo) is det,
    out(cannot_fail), di, uo) is cc_multi.
:- mode try_store(pred(out, di, uo) is cc_multi,
    out(cannot_fail), di, uo) is cc_multi.

    % try_all(Goal, MaybeException, Solutions):
    %
    % Operational semantics:
    %
    %   Try to find all solutions to Goal(S), using backtracking.
    %   Collect the solutions found in Solutions, until the goal either
    %   throws an exception or fails. If it throws an exception E,
    %   then set MaybeException = yes(E), otherwise set MaybeException = no.
    %
    % Declaratively it is equivalent to:
    %
    %   all [S] (list.member(S, Solutions) => Goal(S)),
    %   (
    %       MaybeException = yes(_)
    %   ;
    %       MaybeException = no,
    %       all [S] (Goal(S) => list.member(S, Solutions))
    %   ).
    %
:- pred try_all(pred(T), maybe(univ), list(T)).
:- mode try_all(pred(out) is det,     out, out(nil_or_singleton_list))
    is cc_multi.
:- mode try_all(pred(out) is semidet, out, out(nil_or_singleton_list))
    is cc_multi.
:- mode try_all(pred(out) is multi,   out, out) is cc_multi.
:- mode try_all(pred(out) is nondet,  out, out) is cc_multi.

:- inst [] for list/1
    --->    [].
:- inst nil_or_singleton_list for list/1
    --->    []
```

```
    ;       [ground].

    % incremental_try_all(Goal, AccumulatorPred, Acc0, Acc):
    %
    % Declaratively it is equivalent to:
    %
    %   try_all(Goal, MaybeException, Solutions),
    %   list.map(wrap_success, Solutions, Results),
    %   list.foldl(AccumulatorPred, Results, Acc0, Acc1),
    %   (
    %       MaybeException = no,
    %       Acc = Acc1
    %   ;
    %       MaybeException = yes(Exception),
    %       AccumulatorPred(exception(Exception), Acc1, Acc)
    %   )
    %
    % where (wrap_success(S, R) <=> R = succeeded(S)).
    %
    % Operationally, however, incremental_try_all/5 will call
    % AccumulatorPred for each solution as it is obtained, rather than
    % first building a list of the solutions.
    %
:- pred incremental_try_all(pred(T), pred(exception_result(T), A, A), A, A).
:- mode incremental_try_all(pred(out) is nondet,
    pred(in, di, uo) is det, di, uo) is cc_multi.
:- mode incremental_try_all(pred(out) is nondet,
    pred(in, in, out) is det, in, out) is cc_multi.

    % finally(P, PRes, Cleanup, CleanupRes, !IO).
    %
    % Call P and ensure that Cleanup is called afterwards,
    % no matter whether P succeeds or throws an exception.
    % PRes is bound to the output of P.
    % CleanupRes is bound to the output of Cleanup.
    % A exception thrown by P will be rethrown after Cleanup
    % is called, unless Cleanup throws an exception.
    % This predicate performs the same function as the 'finally'
    % clause ('try {...} finally {...}') in languages such as Java.
    %
:- pred finally(pred(T, io, io), T, pred(io.res, io, io), io.res, io, io).
:- mode finally(pred(out, di, uo) is det, out,
    pred(out, di, uo) is det, out, di, uo) is det.
:- mode finally(pred(out, di, uo) is cc_multi, out,
    pred(out, di, uo) is cc_multi, out, di, uo) is cc_multi.

    % throw_if_near_stack_limits checks if the program is near
```

```
    % the limits of the Mercury stacks, and throws an exception
    % (near_stack_limits) if this is the case.
    %
    % This predicate works only in low level C grades; in other grades,
    % it never throws an exception.
    %
    % The predicate is impure instead of semipure because its effect depends
    % not only on the execution of other impure predicates, but all calls.
    %
:- type near_stack_limits
    --->    near_stack_limits.

:- impure pred throw_if_near_stack_limits is det.

%-----------------------------------------------------%
%-----------------------------------------------------%
```

27 fat_sparse_bitset

```
%-----------------------------------------------------%
% vim: ts=4 sw=4 et ft=mercury
%-----------------------------------------------------%
% Copyright (C) 2011-2012 The University of Melbourne.
% Copyright (C) 2014, 2016-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: fat_sparse_bitset.m.
% Author: zs.
% Stability: medium.
%
% This is a variant of the sparse_bitset module using fat lists.
%
%-----------------------------------------------------%

:- module fat_sparse_bitset.
:- interface.

:- import_module enum.
:- import_module list.
:- import_module term.

:- use_module set.

%-----------------------------------------------------%
```

```
:- type fat_sparse_bitset(T). % <= enum(T).

%-----------------------------------------------------%
%
% Initial creation of sets.
%

    % Return an empty set.
    %
:- func init = fat_sparse_bitset(T).
:- pred init(fat_sparse_bitset(T)::out) is det.

    % Note: set.m contains the reverse mode of this predicate, but it is
    % difficult to implement both modes using the representation in this
    % module.
    %
:- pred singleton_set(fat_sparse_bitset(T)::out, T::in) is det <= enum(T).

    % 'make_singleton_set(Elem)' returns a set containing just the single
    % element 'Elem'.
    %
:- func make_singleton_set(T) = fat_sparse_bitset(T) <= enum(T).

%-----------------------------------------------------%
%
% Emptiness and singleton-ness tests.
%

:- pred empty(fat_sparse_bitset(T)).
:- mode empty(in) is semidet.
:- mode empty(out) is det.
:- pragma obsolete(empty/1, [init/0, is_empty/1]).

:- pred is_empty(fat_sparse_bitset(T)::in) is semidet.

:- pred is_non_empty(fat_sparse_bitset(T)::in) is semidet.

    % Is the given set a singleton, and if yes, what is the element?
    %
:- pred is_singleton(fat_sparse_bitset(T)::in, T::out) is semidet <= enum(T).

%-----------------------------------------------------%
%
% Membership tests.
%
```

```
    % 'member(X, Set)' is true iff 'X' is a member of 'Set'.
    % Takes O(rep_size(Set)) time.
    %
:- pred member(T, fat_sparse_bitset(T)) <= enum(T).
:- mode member(in, in) is semidet.
:- mode member(out, in) is nondet.

    % 'contains(Set, X)' is true iff 'X' is a member of 'Set'.
    % Takes O(rep_size(Set)) time.
    %
:- pred contains(fat_sparse_bitset(T)::in, T::in) is semidet <= enum(T).

%-----------------------------------------------%
%
% Insertions and deletions.
%

    % 'insert(Set, X)' returns the union of 'Set' and the set containing
    % only 'X'. Takes O(rep_size(Set)) time and space.
    %
:- func insert(fat_sparse_bitset(T), T) = fat_sparse_bitset(T) <= enum(T).
:- pred insert(T::in, fat_sparse_bitset(T)::in, fat_sparse_bitset(T)::out)
    is det <= enum(T).

    % 'insert_new(X, Set0, Set)' returns the union of 'Set0' and the set
    % containing only 'X' if 'Set0' does not already contain 'X'; if it does,
    % it fails. Takes O(rep_size(Set)) time and space.
    %
:- pred insert_new(T::in,
    fat_sparse_bitset(T)::in, fat_sparse_bitset(T)::out) is semidet <= enum(T).

    % 'insert_list(Set, X)' returns the union of 'Set' and the set containing
    % only the members of 'X'. Same as 'union(Set, list_to_set(X))', but may be
    % more efficient.
    %
:- func insert_list(fat_sparse_bitset(T), list(T)) = fat_sparse_bitset(T)
    <= enum(T).
:- pred insert_list(list(T)::in,
    fat_sparse_bitset(T)::in, fat_sparse_bitset(T)::out) is det <= enum(T).

%-----------------------------------------------%

    % 'delete(Set, X)' returns the difference of 'Set' and the set containing
    % only 'X'. Takes O(rep_size(Set)) time and space.
    %
:- func delete(fat_sparse_bitset(T), T) = fat_sparse_bitset(T) <= enum(T).
:- pred delete(T::in, fat_sparse_bitset(T)::in, fat_sparse_bitset(T)::out)
```

```
    is det <= enum(T).

    % 'delete_list(Set, X)' returns the difference of 'Set' and the set
    % containing only the members of 'X'. Same as
    % 'difference(Set, list_to_set(X))', but may be more efficient.
    %
:- func delete_list(fat_sparse_bitset(T), list(T)) = fat_sparse_bitset(T)
    <= enum(T).
:- pred delete_list(list(T)::in,
    fat_sparse_bitset(T)::in, fat_sparse_bitset(T)::out) is det <= enum(T).

    % 'remove(X, Set0, Set)' returns in 'Set' the difference of 'Set0'
    % and the set containing only 'X', failing if 'Set0' does not contain 'X'.
    % Takes O(rep_size(Set)) time and space.
    %
:- pred remove(T::in, fat_sparse_bitset(T)::in, fat_sparse_bitset(T)::out)
    is semidet <= enum(T).

    % 'remove_list(X, Set0, Set)' returns in 'Set' the difference of 'Set0'
    % and the set containing all the elements of 'X', failing if any element
    % of 'X' is not in 'Set0'. Same as 'subset(list_to_set(X), Set0),
    % difference(Set0, list_to_set(X), Set)', but may be more efficient.
    %
:- pred remove_list(list(T)::in,
    fat_sparse_bitset(T)::in, fat_sparse_bitset(T)::out) is semidet <= enum(T).

    % 'remove_leq(Set, X)' returns 'Set' with all elements less than or equal
    % to 'X' removed. In other words, it returns the set containing all the
    % elements of 'Set' which are greater than 'X'.
    %
:- func remove_leq(fat_sparse_bitset(T), T) = fat_sparse_bitset(T) <= enum(T).
:- pred remove_leq(T::in, fat_sparse_bitset(T)::in, fat_sparse_bitset(T)::out)
    is det <= enum(T).

    % 'remove_gt(Set, X)' returns 'Set' with all elements greater than 'X'
    % removed. In other words, it returns the set containing all the elements
    % of 'Set' which are less than or equal to 'X'.
    %
:- func remove_gt(fat_sparse_bitset(T), T) = fat_sparse_bitset(T) <= enum(T).
:- pred remove_gt(T::in, fat_sparse_bitset(T)::in, fat_sparse_bitset(T)::out)
    is det <= enum(T).

    % 'remove_least(Set0, X, Set)' is true iff 'X' is the least element in
    % 'Set0', and 'Set' is the set which contains all the elements of 'Set0'
    % except 'X'. Takes O(1) time and space.
    %
:- pred remove_least(T::out,
```

```
    fat_sparse_bitset(T)::in, fat_sparse_bitset(T)::out) is semidet <= enum(T).

%-----------------------------------------------------%
%
% Comparisons between sets.
%

    % 'equal(SetA, SetB' is true iff 'SetA' and 'SetB' contain the same
    % elements. Takes O(min(rep_size(SetA), rep_size(SetB))) time.
    %
:- pred equal(fat_sparse_bitset(T)::in, fat_sparse_bitset(T)::in) is semidet.

    % 'subset(Subset, Set)' is true iff 'Subset' is a subset of 'Set'.
    % Same as 'intersect(Set, Subset, Subset)', but may be more efficient.
    %
:- pred subset(fat_sparse_bitset(T)::in, fat_sparse_bitset(T)::in) is semidet.

    % 'superset(Superset, Set)' is true iff 'Superset' is a superset of 'Set'.
    % Same as 'intersect(Superset, Set, Set)', but may be more efficient.
    %
:- pred superset(fat_sparse_bitset(T)::in, fat_sparse_bitset(T)::in)
    is semidet.

%-----------------------------------------------------%
%
% Operations on two or more sets.
%

    % 'union(SetA, SetB)' returns the union of 'SetA' and 'SetB'. The
    % efficiency of the union operation is not sensitive to the argument
    % ordering. Takes O(rep_size(SetA) + rep_size(SetB)) time and space.
    %
:- func union(fat_sparse_bitset(T), fat_sparse_bitset(T)) =
    fat_sparse_bitset(T).
:- pred union(fat_sparse_bitset(T)::in, fat_sparse_bitset(T)::in,
    fat_sparse_bitset(T)::out) is det.

    % 'union_list(Sets, Set)' returns the union of all the sets in Sets.
    %
:- func union_list(list(fat_sparse_bitset(T))) = fat_sparse_bitset(T).
:- pred union_list(list(fat_sparse_bitset(T))::in,
    fat_sparse_bitset(T)::out) is det.

    % 'intersect(SetA, SetB)' returns the intersection of 'SetA' and 'SetB'.
    % The efficiency of the intersection operation is not sensitive to the
    % argument ordering. Takes O(rep_size(SetA) + rep_size(SetB)) time and
    % O(min(rep_size(SetA)), rep_size(SetB)) space.
```

```
    %
:- func intersect(fat_sparse_bitset(T), fat_sparse_bitset(T)) =
    fat_sparse_bitset(T).
:- pred intersect(fat_sparse_bitset(T)::in, fat_sparse_bitset(T)::in,
    fat_sparse_bitset(T)::out) is det.

    % 'intersect_list(Sets, Set)' returns the intersection of all the sets
    % in Sets.
    %
:- func intersect_list(list(fat_sparse_bitset(T))) = fat_sparse_bitset(T).
:- pred intersect_list(list(fat_sparse_bitset(T))::in,
    fat_sparse_bitset(T)::out) is det.

    % 'difference(SetA, SetB)' returns the set containing all the elements
    % of 'SetA' except those that occur in 'SetB'. Takes
    % O(rep_size(SetA) + rep_size(SetB)) time and O(rep_size(SetA)) space.
    %
:- func difference(fat_sparse_bitset(T), fat_sparse_bitset(T)) =
    fat_sparse_bitset(T).
:- pred difference(fat_sparse_bitset(T)::in, fat_sparse_bitset(T)::in,
    fat_sparse_bitset(T)::out) is det.

%-----------------------------------------------%
%
% Operations that divide a set into two parts.
%

    % divide(Pred, Set, InPart, OutPart):
    % InPart consists of those elements of Set for which Pred succeeds;
    % OutPart consists of those elements of Set for which Pred fails.
    %
:- pred divide(pred(T)::in(pred(in) is semidet), fat_sparse_bitset(T)::in,
    fat_sparse_bitset(T)::out, fat_sparse_bitset(T)::out) is det <= enum(T).

    % divide_by_set(DivideBySet, Set, InPart, OutPart):
    % InPart consists of those elements of Set which are also in DivideBySet;
    % OutPart consists of those elements of Set which are not in DivideBySet.
    %
:- pred divide_by_set(fat_sparse_bitset(T)::in, fat_sparse_bitset(T)::in,
    fat_sparse_bitset(T)::out, fat_sparse_bitset(T)::out) is det <= enum(T).

%-----------------------------------------------%
%
% Converting lists to sets.
%

    % 'list_to_set(List)' returns a set containing only the members of 'List'.
```

```
    % In the worst case this will take O(length(List)^2) time and space.
    % If the elements of the list are closely grouped, it will be closer
    % to O(length(List)).
    %
:- func list_to_set(list(T)) = fat_sparse_bitset(T) <= enum(T).
:- pred list_to_set(list(T)::in, fat_sparse_bitset(T)::out) is det <= enum(T).

    % 'sorted_list_to_set(List)' returns a set containing only the members
    % of 'List'. 'List' must be sorted. Takes O(length(List)) time and space.
    %
:- func sorted_list_to_set(list(T)) = fat_sparse_bitset(T) <= enum(T).
:- pred sorted_list_to_set(list(T)::in, fat_sparse_bitset(T)::out)
    is det <= enum(T).

%-----------------------------------------------------%
%
% Converting sets to lists.
%

    % 'to_sorted_list(Set)' returns a list containing all the members of 'Set',
    % in sorted order. Takes O(card(Set)) time and space.
    %
:- func to_sorted_list(fat_sparse_bitset(T)) = list(T) <= enum(T).
:- pred to_sorted_list(fat_sparse_bitset(T)::in, list(T)::out)
    is det <= enum(T).

%-----------------------------------------------------%
%
% Converting between different kinds of sets.
%

    % 'from_set(Set)' returns a bitset containing only the members of 'Set'.
    % Takes O(card(Set)) time and space.
    %
:- func from_set(set.set(T)) = fat_sparse_bitset(T) <= enum(T).

    % 'to_sorted_list(Set)' returns a set.set containing all the members
    % of 'Set', in sorted order. Takes O(card(Set)) time and space.
    %
:- func to_set(fat_sparse_bitset(T)) = set.set(T) <= enum(T).

%-----------------------------------------------------%
%
% Counting.
%

    % 'count(Set)' returns the number of elements in 'Set'.
```

```
    % Takes O(card(Set)) time.
    %
:- func count(fat_sparse_bitset(T)) = int <= enum(T).

%-----------------------------------------------------%
%
% Standard higher order functions on collections.
%

    % all_true(Pred, Set) succeeds iff Pred(Element) succeeds
    % for all the elements of Set.
    %
:- pred all_true(pred(T)::in(pred(in) is semidet), fat_sparse_bitset(T)::in)
    is semidet <= enum(T).

    % 'filter(Pred, Set) = TrueSet' returns the elements of Set for which
    % Pred succeeds.
    %
:- func filter(pred(T), fat_sparse_bitset(T)) = fat_sparse_bitset(T)
    <= enum(T).
:- mode filter(pred(in) is semidet, in) = out is det.

    % 'filter(Pred, Set, TrueSet, FalseSet)' returns the elements of Set
    % for which Pred succeeds, and those for which it fails.
    %
:- pred filter(pred(T), fat_sparse_bitset(T),
    fat_sparse_bitset(T), fat_sparse_bitset(T)) <= enum(T).
:- mode filter(pred(in) is semidet, in, out, out) is det.

    % 'foldl(Func, Set, Start)' calls Func with each element of 'Set'
    % (in sorted order) and an accumulator (with the initial value of 'Start'),
    % and returns the final value. Takes O(card(Set)) time.
    %
:- func foldl(func(T, U) = U, fat_sparse_bitset(T), U) = U <= enum(T).

:- pred foldl(pred(T, U, U), fat_sparse_bitset(T), U, U) <= enum(T).
:- mode foldl(pred(in, in, out) is det, in, in, out) is det.
:- mode foldl(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode foldl(pred(in, di, uo) is det, in, di, uo) is det.
:- mode foldl(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode foldl(pred(in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode foldl(pred(in, di, uo) is semidet, in, di, uo) is semidet.
:- mode foldl(pred(in, in, out) is nondet, in, in, out) is nondet.
:- mode foldl(pred(in, in, out) is cc_multi, in, in, out) is cc_multi.
:- mode foldl(pred(in, di, uo) is cc_multi, in, di, uo) is cc_multi.

:- pred foldl2(pred(T, U, U, V, V), fat_sparse_bitset(T), U, U, V, V)
```

```
    <= enum(T).
:- mode foldl2(pred(in, in, out, in, out) is det, in, in, out, in, out) is det.
:- mode foldl2(pred(in, in, out, mdi, muo) is det, in, in, out, mdi, muo)
    is det.
:- mode foldl2(pred(in, in, out, di, uo) is det, in, in, out, di, uo) is det.
:- mode foldl2(pred(in, di, uo, di, uo) is det, in, di, uo, di, uo) is det.
:- mode foldl2(pred(in, in, out, in, out) is semidet, in, in, out, in, out)
    is semidet.
:- mode foldl2(pred(in, in, out, mdi, muo) is semidet, in, in, out, mdi, muo)
    is semidet.
:- mode foldl2(pred(in, in, out, di, uo) is semidet, in, in, out, di, uo)
    is semidet.
:- mode foldl2(pred(in, in, out, in, out) is nondet, in, in, out, in, out)
    is nondet.
:- mode foldl2(pred(in, in, out, in, out) is cc_multi, in, in, out, in, out)
    is cc_multi.
:- mode foldl2(pred(in, in, out, di, uo) is cc_multi, in, in, out, di, uo)
    is cc_multi.
:- mode foldl2(pred(in, di, uo, di, uo) is cc_multi, in, di, uo, di, uo)
    is cc_multi.

    % 'foldr(Func, Set, Start)' calls Func with each element of 'Set'
    % (in reverse sorted order) and an accumulator (with the initial value
    % of 'Start'), and returns the final value. Takes O(card(Set)) time.
    %
:- func foldr(func(T, U) = U, fat_sparse_bitset(T), U) = U <= enum(T).

:- pred foldr(pred(T, U, U), fat_sparse_bitset(T), U, U) <= enum(T).
:- mode foldr(pred(in, in, out) is det, in, in, out) is det.
:- mode foldr(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode foldr(pred(in, di, uo) is det, in, di, uo) is det.
:- mode foldr(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode foldr(pred(in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode foldr(pred(in, di, uo) is semidet, in, di, uo) is semidet.
:- mode foldr(pred(in, in, out) is nondet, in, in, out) is nondet.
:- mode foldr(pred(in, in, out) is cc_multi, in, in, out) is cc_multi.
:- mode foldr(pred(in, di, uo) is cc_multi, in, di, uo) is cc_multi.

:- pred foldr2(pred(T, U, U, V, V), fat_sparse_bitset(T), U, U, V, V)
    <= enum(T).
:- mode foldr2(pred(in, in, out, in, out) is det, in, in, out, in, out) is det.
:- mode foldr2(pred(in, in, out, mdi, muo) is det, in, in, out, mdi, muo)
    is det.
:- mode foldr2(pred(in, in, out, di, uo) is det, in, in, out, di, uo) is det.
:- mode foldr2(pred(in, di, uo, di, uo) is det, in, di, uo, di, uo) is det.
:- mode foldr2(pred(in, in, out, in, out) is semidet, in, in, out, in, out)
    is semidet.
```

```
:- mode foldr2(pred(in, in, out, mdi, muo) is semidet, in, in, out, mdi, muo)
    is semidet.
:- mode foldr2(pred(in, in, out, di, uo) is semidet, in, in, out, di, uo)
    is semidet.
:- mode foldr2(pred(in, in, out, in, out) is nondet, in, in, out, in, out)
    is nondet.
:- mode foldr2(pred(in, di, uo, di, uo) is cc_multi, in, di, uo, di, uo)
    is cc_multi.
:- mode foldr2(pred(in, in, out, di, uo) is cc_multi, in, in, out, di, uo)
    is cc_multi.
:- mode foldr2(pred(in, in, out, in, out) is cc_multi, in, in, out, in, out)
    is cc_multi.

%---------------------------------------------------%
%---------------------------------------------------%
```

28 float

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 1994-1998,2001-2008,2010, 2012 The University of Melbourne.
% Copyright (C) 2013-2016, 2018-2020 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: float.m.
% Main author: fjh.
% Stability: medium.
%
% Floating point support.
%
% Floats are double precision, except in .spf grades where they
% are single precision.
%
% Note that implementations which support IEEE floating point
% should ensure that in cases where the only valid answer is a "NaN"
% (the IEEE float representation for "not a number"), the det
% functions here will halt with a runtime error (or throw an exception)
% rather than returning a NaN. Quiet (non-signalling) NaNs have a
% semantics which is not valid in Mercury, since they don't obey the
% axiom "all [X] X = X".
%
% XXX Unfortunately the current Mercury implementation does not
% do that on all platforms, since neither ANSI C nor POSIX provide
```

```
% any portable way of ensuring that floating point operations
% whose result is not representable will raise a signal rather
% than returning a NaN. (Maybe C9X will help...?)
% The behaviour is correct on Linux and Digital Unix,
% but not on Solaris, for example.
%
% IEEE floating point also specifies that some functions should
% return different results for +0.0 and -0.0, but that +0.0 and -0.0
% should compare equal. This semantics is not valid in Mercury,
% since it doesn't obey the axiom `all [F, X, Y] X = Y => F(X) = F(Y)'.
% Again, the resolution is that in Mercury, functions which would
% return different results for +0.0 and -0.0 should instead halt
% execution with a run-time error (or throw an exception).
%
% XXX Here too the current Mercury implementation does not
% implement the intended semantics correctly on all platforms.
%
% XXX On machines such as x86 which support extra precision
% for intermediate results, the results may depend on the
% level of optimization, in particular inlining and evaluation
% of constant expressions.
% For example, the goal `1.0/9.0 = std_util.id(1.0)/9.0' may fail.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module float.
:- interface.

:- import_module pretty_printer.

%-----------------------------------------------------%
%
% Arithmetic functions.
%

    % Addition.
    %
:- func (float::in) + (float::in) = (float::uo) is det.

    % Subtraction.
    %
:- func (float::in) - (float::in) = (float::uo) is det.

    % Multiplication.
    %
:- func (float::in) * (float::in) = (float::uo) is det.
```

```
    % Division.
    % Throws a 'domain_error' exception if the right operand is zero.
    % See the comments at the top of math.m to find out how to disable
    % this check.
    %
:- func (float::in) / (float::in) = (float::uo) is det.

    % unchecked_quotient(X, Y) is the same as X / Y, but the behaviour
    % is undefined if the right operand is zero.
    %
:- func unchecked_quotient(float::in, float::in) = (float::uo) is det.

    % Unary plus.
    %
:- func + (float::in) = (float::uo) is det.

    % Unary minus.
    %
:- func - (float::in) = (float::uo) is det.

%-----------------------------------------------------%
%
% Comparison predicates.
%

    % Less than.
    %
:- pred (float::in) < (float::in) is semidet.

    % Less than or equal.
    %
:- pred (float::in) =< (float::in) is semidet.

    % Greater than or equal.
    %
:- pred (float::in) >= (float::in) is semidet.

    % Greater than.
    %
:- pred (float::in) > (float::in) is semidet.

%-----------------------------------------------------%
%
% Conversion from integer types.
%
```

```
    % Convert an int into float.
    %
    % The behaviour when the int exceeds the range of what can be exactly
    % represented by a float is undefined.
    %
:- func float(int) = float.

    % Convert a signed 8-bit integer into a float.
    % Always succeeds as all signed 8-bit integers have an exact
    % floating-point representation.
    %
:- func from_int8(int8) = float.

    % Convert a signed 16-bit integer into a float.
    % Always succeeds as all signed 16-bit integers have an exact
    % floating-point representation.
    %
:- func from_int16(int16) = float.

    % Convert a signed 32-bit integer into a float.
    % The behaviour when the integer exceeds the range of what can be
    % exactly represented by a float is undefined.
    %
:- func cast_from_int32(int32) = float.

    % Convert a signed 64-bit integer into a float.
    % The behaviour when the integer exceeds the range of what can be
    % exactly represented by a float is undefined.
    %
:- func cast_from_int64(int64) = float.

    % Convert an unsigned 8-bit integer into a float.
    % Always succeeds as all unsigned 8-bit integers have an exact
    % floating-point representation.
    %
:- func from_uint8(uint8) =  float.

    % Convert an unsigned 16-bit integer into a float.
    % Always succeeds as all unsigned 16-bit integers have an exact
    % floating-point representation.
    %
:- func from_uint16(uint16) = float.

    % Convert an unsigned 32-bit integer into a float.
    % The behaviour when the integer exceeds the range of what can be
    % exactly represented by a float is undefined.
    %
```

```
:- func cast_from_uint32(uint32) = float.

    % Convert an unsigned 64-bit integer into a float.
    % The behaviour when the integer exceeds the range of what can be
    % exactly represented by a float is undefined.
    %
:- func cast_from_uint64(uint64) = float.

%-----------------------------------------------------%
%
% Conversion to int.
%

    % ceiling_to_int(X) returns the smallest integer not less than X.
    %
:- func ceiling_to_int(float) = int.

    % floor_to_int(X) returns the largest integer not greater than X.
    %
:- func floor_to_int(float) = int.

    % round_to_int(X) returns the integer closest to X.
    % If X has a fractional value of 0.5, it is rounded up.
    %
:- func round_to_int(float) = int.

    % truncate_to_int(X) returns the integer closest to X such that
    % |truncate_to_int(X)| =< |X|.
    %
:- func truncate_to_int(float) = int.

%-----------------------------------------------------%
%
% Miscellaneous functions.
%

    % Absolute value.
    %
:- func abs(float) = float.

    % Maximum.
    %
:- func max(float, float) = float.

    % Minimum.
    %
:- func min(float, float) = float.
```

```
    % pow(Base, Exponent) returns Base raised to the power Exponent.
    % Fewer domain restrictions than math.pow: works for negative Base,
    % and pow(B, 0) = 1.0 for all B, even B=0.0.
    % Only pow(0, <negative>) throws a 'domain_error' exception.
    %
:- func pow(float, int) = float.

    % Compute a non-negative integer hash value for a float.
    %
:- func hash(float) = int.
:- pred hash(float::in, int::out) is det.

%-----------------------------------------------------%
%
% Classification.
%

    % True iff the argument is of infinite magnitude.
    %
:- pred is_infinite(float::in) is semidet.

    % Synonym for the above.
    %
:- pred is_inf(float::in) is semidet.

    % True iff the argument is not-a-number (NaN).
    %
:- pred is_nan(float::in) is semidet.

    % True iff the argument is of infinite magnitude or not-a-number (NaN).
    %
:- pred is_nan_or_infinite(float::in) is semidet.

    % Synonym for the above.
    %
:- pred is_nan_or_inf(float::in) is semidet.

    % True iff the argument is not of infinite magnitude and is not a
    % not-a-number (NaN) value.
    %
:- pred is_finite(float::in) is semidet.

    % True iff the argument is of zero magnitude.
    %
:- pred is_zero(float::in) is semidet.
```

```
%-----------------------------------------------%
%
% System constants.
%

    % Maximum finite floating-point number.
    %
    % max = (1 - radix ** mantissa_digits) * radix ** max_exponent
    %
:- func max = float.

    % Minimum normalised positive floating-point number.
    %
    % min = radix ** (min_exponent - 1)
    %
:- func min = float.

    % Positive infinity.
    %
:- func infinity = float.

    % Smallest number x such that 1.0 + x \= 1.0.
    % This represents the largest relative spacing of two consecutive floating
    % point numbers.
    %
    % epsilon = radix ** (1 - mantissa_digits)
    %
:- func epsilon = float.

    % Radix of the floating-point representation.
    % In the literature, this is sometimes referred to as ‘b’.
    %
:- func radix = int.

    % The number of base-radix digits in the mantissa.
    % In the literature, this is sometimes referred to as ‘p’ or ‘t’.
    %
:- func mantissa_digits = int.

    % Minimum negative integer such that:
    %   radix ** (min_exponent - 1)
    % is a normalised floating-point number. In the literature,
    % this is sometimes referred to as ‘e_min’.
    %
:- func min_exponent = int.

    % Maximum integer such that:
```

```
    %   radix ** (max_exponent - 1)
    % is a normalised floating-point number.
    % In the literature, this is sometimes referred to as ‘e_max’.
    %
:- func max_exponent = int.

%-----------------------------------------------------%

    % Convert a float to a pretty_printer.doc for formatting.
    %
:- func float_to_doc(float) = doc.

%-----------------------------------------------------%
%-----------------------------------------------------%
```

29 gc

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 1999,2001-2007 The University of Melbourne.
% Copyright (C) 2014-2015, 2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: gc.m.
% Author: fjh.
% Stability: medium.
%
% This module defines some procedures for controlling the actions of the
% garbage collector.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module gc.
:- interface.

:- import_module io.

%-----------------------------------------------------%

    % Force a garbage collection.
    %
:- pred garbage_collect(io.state::di, io.state::uo) is det.
```

```
    % Force a garbage collection.
    % Note that this version is not really impure, but it needs to be
    % declared impure to ensure that the compiler won't try to
    % optimize it away.
    %
:- impure pred garbage_collect is det.

%---------------------------------------------------%
%---------------------------------------------------%
```

30 getopt

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 1994-1999,2001-2007, 2011 The University of Melbourne.
% Copyright (C) 2014-2015, 2017-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: getopt.m.
% Authors: fjh, zs.
% Stability: medium.
%
% This module exports the predicate process_options/4, which can be
% used to parse command-line options.
%
% This version allows both short (single-character) options and GNU-style long
% options. It also has the GNU extension of recognizing options anywhere in
% the command-line, not just at the start.
%
% To use this module:
%
% - You must provide an 'option' type which is an enumeration of
%   all your different options.
% - You must provide predicates 'short_option(Char, Option)' and
%   'long_option(String, Option)' which convert the short and/or long names
%   for the option to this enumeration type.
%   (An option can have as many names as you like, long or short.)
% - You must provide a predicate 'option_default(Option, OptionData)'
%   which specifies both the type and the default value for every option.
%
% You may optionally provide a predicate 'special_handler(Option, SpecialData,
% OptionTable, MaybeOptionTable)' for handling special option types.
```

```
% (See below.)
%
% We support the following "simple" option types:
%
%   - bool
%   - int
%   - maybe_int (which have a value of ‘no’ or ‘yes(int)’)
%   - string
%   - maybe_string (which have a value of ‘no’ or ‘yes(string)’)
%
% We also support one "accumulating" option type:
%
%   - accumulating (which accumulates a list of strings)
%
% And the following "special" option types:
%
%   - special
%   - bool_special
%   - int_special
%   - string_special
%   - maybe_string_special
%
% A further special option, file_special, is supported only by the getopt_io
% module, because it requires process_options to take a pair of I/O state
% arguments.
%
% For the "simple" option types, if there are multiple occurrences of the same
% option on the command-line, then the last (right-most) occurrence will take
% precedence. For "accumulating" options, multiple occurrences will be
% appended together into a list.
%
% The "special" option types are handled by a special option handler (see
% ‘special_handler’ below), which may perform arbitrary modifications to the
% option_table. For example, an option which is not yet implemented could be
% handled by a special handler which produces an error report, or an option
% which is a synonym for a set of more "primitive" options could be han-
dled by
% a special handler which sets those "primitive" options.
%
% It is an error to use a "special" option for which there is no handler, or
% for which the handler fails.
%
% Boolean (i.e. bool or bool_special), maybe_int, maybe_string
% and accumulating options can be negated. Negating an accumulating
% option empties the accumulated list of strings.
% Single-character options can be negated by following them
% with another ‘-’, e.g. ‘-x-’ will negate the ‘-x’ option.
```

```
% Long options can be negated by preceding them with `--no-',
% e.g. `--no-foo' will negate the `--foo' option.
%
% Note that arguments following an option may be separated from the op-
tion by
% either whitespace or an equals, `=', character, e.g. `--foo 3' and `-
-foo=3'
% both specify the option `--foo' with the integer argument `3'.
%
% If the argument `--' is encountered on the command-line then option
% processing will immediately terminate, without processing any remaining
% options.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module getopt.
:- interface.

:- import_module bool.
:- import_module char.
:- import_module list.
:- import_module map.
:- import_module maybe.
:- import_module set.

% process_options(OptionOps, Args, NonOptionArgs, Result)
% process_options(OptionOps, Args, OptionArgs, NonOptionArgs, Result)
%
%   Scans through 'Args' looking for options, places all the option
%   arguments in `OptionArgs', places all the non-option arguments in
%   'NonOptionArgs', and records the options in the `OptionTable'.
%   `OptionTable' is a map from a user-defined option type to option_data.
%   If an invalid option is encountered, we return `error(Message)'
%   otherwise we return `ok(OptionTable)' in 'Result'.
%
%   The argument `OptionOps' is a structure holding three or four
%   predicates used to categorize a set of options. Their
%   interfaces should be like these:
%
% :- pred short_option(char::in, option::out) is semidet.
%   True if the character names a valid single-character option.
%
% :- pred long_option(string::in, option::out) is semidet.
%   True if the string names a valid long option.
%
% :- pred option_default(option::out, option_data::out) is multi.
```

```
%   Nondeterministically returns all the options with their
%   corresponding types and default values.
%
% :- pred special_handler(option::in, special_data::in,
%   option_table::in, maybe_option_table(_)::out) is semidet.
%   This predicate is invoked whenever getopt finds an option
%   (long or short) designated as special, with special_data holding
%   the argument of the option (if any). The predicate can change the
%   option table in arbitrary ways in the course of handling the option,
%   or it can return an error message.
%   The canonical examples of special options are -O options in compilers,
%   which set many other options at once.

:- pred process_options(option_ops(OptionType)::in(option_ops),
    list(string)::in, list(string)::out,
    maybe_option_table(OptionType)::out) is det.

:- pred process_options(option_ops(OptionType)::in(option_ops),
    list(string)::in, list(string)::out, list(string)::out,
    maybe_option_table(OptionType)::out) is det.

% process_options_track(OptionOps, Args, OptionArgs,
%       NonOptionArgs, OptionTable0, Result, OptionsSet)

:- pred process_options_track(
    option_ops_track(OptionType)::in(option_ops_track),
    list(string)::in, list(string)::out, list(string)::out,
    option_table(OptionType)::in, maybe_option_table(OptionType)::out,
    set(OptionType)::out) is det.

% Variants of the above that return structured errors.
% These behave as the above versions except that any error values returned are
% members of the option_error/1 type rather than strings.

:- pred process_options_se(option_ops(OptionType)::in(option_ops),
    list(string)::in, list(string)::out,
    maybe_option_table_se(OptionType)::out) is det.

:- pred process_options_se(option_ops(OptionType)::in(option_ops),
    list(string)::in, list(string)::out, list(string)::out,
    maybe_option_table_se(OptionType)::out) is det.

:- pred process_options_track_se(
    option_ops_track(OptionType)::in(option_ops_track),
    list(string)::in, list(string)::out, list(string)::out,
    option_table(OptionType)::in, maybe_option_table_se(OptionType)::out,
    set(OptionType)::out) is det.
```

```
:- pred init_option_table(
    pred(OptionType, option_data)::in(pred(out, out) is nondet),
    option_table(OptionType)::out) is det.

:- pred init_option_table_multi(
    pred(OptionType, option_data)::in(pred(out, out) is multi),
    option_table(OptionType)::out) is det.

:- type option_ops(OptionType)
    --->    option_ops(
                pred(char, OptionType),         % short_option
                pred(string, OptionType),       % long_option
                pred(OptionType, option_data)   % option_default
            )
    ;       option_ops(
                pred(char, OptionType),         % short_option
                pred(string, OptionType),       % long_option
                pred(OptionType, option_data),  % option_default
                pred(OptionType, special_data,  % special option handler
                    option_table(OptionType),
                    maybe_option_table(OptionType))
            )
    ;       option_ops_multi(
                pred(char, OptionType),         % short_option
                pred(string, OptionType),       % long_option
                pred(OptionType, option_data)   % option_default
            )
    ;       option_ops_multi(
                pred(char, OptionType),         % short_option
                pred(string, OptionType),       % long_option
                pred(OptionType, option_data),  % option_default
                pred(OptionType, special_data,  % special option handler
                    option_table(OptionType),
                    maybe_option_table(OptionType))
            ).

:- inst option_ops for option_ops/1
    --->    option_ops(
                pred(in, out) is semidet,               % short_option
                pred(in, out) is semidet,               % long_option
                pred(out, out) is nondet                % option_default
            )
    ;       option_ops_multi(
                pred(in, out) is semidet,               % short_option
                pred(in, out) is semidet,               % long_option
                pred(out, out) is multi                 % option_default
```

```
            )
    ;       option_ops(
                pred(in, out) is semidet,               % short_option
                pred(in, out) is semidet,               % long_option
                pred(out, out) is nondet,               % option_default
                pred(in, in, in, out) is semidet        % special handler
            )
    ;       option_ops_multi(
                pred(in, out) is semidet,               % short_option
                pred(in, out) is semidet,               % long_option
                pred(out, out) is multi,                % option_default
                pred(in, in, in, out) is semidet        % special handler
            ).

:- type option_ops_track(OptionType)
    --->    option_ops_track(
                pred(char, OptionType),         % short_option
                pred(string, OptionType),       % long_option
                pred(OptionType, special_data,  % special option handler
                    option_table(OptionType),
                    maybe_option_table(OptionType),
                    set(OptionType))
            ).

:- inst option_ops_track for option_ops_track/1
    --->    option_ops_track(
                pred(in, out) is semidet,               % short_option
                pred(in, out) is semidet,               % long_option
                pred(in, in, in, out, out) is semidet   % special handler
            ).

:- type option_data
    --->    bool(bool)
    ;       int(int)
    ;       string(string)
    ;       maybe_int(maybe(int))
    ;       maybe_string(maybe(string))
    ;       accumulating(list(string))
    ;       special
    ;       bool_special
    ;       int_special
    ;       string_special
    ;       maybe_string_special.

:- type special_data
    --->    none
    ;       bool(bool)
```

```
        ;       int(int)
        ;       string(string)
        ;       maybe_string(maybe(string)).

:- type option_table(OptionType) == map(OptionType, option_data).

:- type maybe_option_table(OptionType)
        --->    ok(option_table(OptionType))
        ;       error(string).

:- type maybe_option_table_se(OptionType)
        --->    ok(option_table(OptionType))
        ;       error(option_error(OptionType)).

:- type option_error(OptionType)
        --->    unrecognized_option(string)
                % An option that is not recognized appeared on the command line.
                % The argument gives the option as it appeared on the command line.

        ;       option_error(OptionType, string, option_error_reason).
                % An error occurred with a specific option. The first two
                % arguments identify the option enumeration value and the string
                % that appeared on the command line for that option respectively.
                % The third argument describes the nature of the error with that
                % option.

:- type option_error_reason
        --->    unknown_type
                % No type for this option has been specified in the
                % 'option_default'/2 predicate.

        ;       requires_argument
                % The option requires an argument but it occurred on the command
                % line without one.

        ;       does_not_allow_argument(string)
                % The option does not allow an argument but it was provided with
                % one on the command line.
                % The argument gives the contents of the argument position on the
                % command line.

        ;       cannot_negate
                % The option cannot be negated but its negated form appeared on the
                % command line.

        ;       special_handler_failed
                % The special option handler predicate for the option failed.
```

```
        ;       special_handler_missing
                % A special option handler predicate was not provided
                % for the option.

        ;       special_handler_error(string)
                % The special option handler predicate for the option returned an
                % error.
                % The argument is a string describing the error.

        ;       requires_numeric_argument(string).
                % The option requires a numeric argument but it occurred on the
                % command line with a non-numeric argument.
                % The argument gives the contents of the argument position on the
                % command line.

:- func option_error_to_string(option_error(OptionType)) = string.

    % The following three predicates search the option table for
    % an option of the specified type; if it is not found, they
    % report an error by calling error/1.

:- pred lookup_bool_option(option_table(Option)::in, Option::in,
    bool::out) is det.
:- func lookup_bool_option(option_table(Option), Option) = bool.

:- pred lookup_int_option(option_table(Option)::in, Option::in,
    int::out) is det.
:- func lookup_int_option(option_table(Option), Option) = int.

:- pred lookup_string_option(option_table(Option)::in, Option::in,
    string::out) is det.
:- func lookup_string_option(option_table(Option), Option) = string.

:- pred lookup_maybe_int_option(option_table(Option)::in, Option::in,
    maybe(int)::out) is det.
:- func lookup_maybe_int_option(option_table(Option), Option) =
    maybe(int).

:- pred lookup_maybe_string_option(option_table(Option)::in,
    Option::in, maybe(string)::out) is det.
:- func lookup_maybe_string_option(option_table(Option), Option) =
    maybe(string).

:- pred lookup_accumulating_option(option_table(Option)::in,
    Option::in, list(string)::out) is det.
:- func lookup_accumulating_option(option_table(Option), Option) =
```

```
    list(string).

%-----------------------------------------------------%
%-----------------------------------------------------%
```

31 getopt_io

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 2005-2007, 2011 The University of Melbourne.
% Copyright (C) 2014-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: getopt_io.m
% Authors: fjh, zs
% Stability: medium
%
% This module exports the predicate process_options/6, which can be
% used to parse command-line options.
%
% This version allows both short (single-character) options and GNU-style long
% options. It also has the GNU extension of recognizing options anywhere in
% the command-line, not just at the start.
%
% To use this module:
%
% - You must provide an 'option' type which is an enumeration of
%   all your different options.
% - You must provide predicates 'short_option(Char, Option)' and
%   'long_option(String, Option)' which convert the short and/or long names
%   for the option to this enumeration type.
%   (An option can have as many names as you like, long or short.)
% - You must provide a predicate 'option_default(Option, OptionData)'
%   which specifies both the type and the default value for every option.
%
% You may optionally provide a predicate 'special_handler(Option, SpecialData,
% OptionTable, MaybeOptionTable)' for handling special option types.
% (See below.)
%
% We support the following "simple" option types:
%
%   - bool
%   - int
```

```
%   - maybe_int (which have a value of 'no' or 'yes(int)')
%   - string
%   - maybe_string (which have a value of 'no' or 'yes(string)')
%
% We also support one "accumulating" option type:
%
%   - accumulating (which accumulates a list of strings)
%
% And the following "special" option types:
%
%   - special
%   - bool_special
%   - int_special
%   - string_special
%   - maybe_string_special
%   - file_special
%
% For the "simple" option types, if there are multiple occurrences of the same
% option on the command-line, then the last (right-most) occurrence will take
% precedence. For "accumulating" options, multiple occurrences will be
% appended together into a list.
%
% With the exception of file_special, the "special" option types are handled
% by a special option handler (see 'special_handler' below), which may perform
% arbitrary modifications to the option_table. For example, an option which
% is not yet implemented could be handled by a special handler which produces
% an error report, or an option which is a synonym for a set of more
% "primitive" options could be handled by a special handler which sets those
% "primitive" options.
%
% It is an error to use a "special" option for which there is no handler, or
% for which the handler fails.
%
% Boolean (i.e. bool or bool_special), maybe_int, maybe_string
% and accumulating options can be negated. Negating an accumulating
% option empties the accumulated list of strings.
% Single-character options can be negated by following them
% with another '-', e.g. '-x-' will negate the '-x' option.
% Long options can be negated by preceding them with '--no-',
% e.g. '--no-foo' will negate the '--foo' option.
%
% The file_special option type requires no handler, and is implemented
% entirely by this module. It always takes a single argument, a file name.
% Its handling always consists of reading the named file, converting its
% contents into a sequence of words separated by white space, and interpreting
% those words as options in the usual manner.
%
```

```
% Note that arguments following an option may be separated from the op-
tion by
% either whitespace or an equals, '=', character, e.g. '--foo 3' and '-
-foo=3'
% both specify the option '--foo' with the integer argument '3'.
%
% If the argument '--' is encountered on the command-line then option
% processing will immediately terminate, without processing any remaining
% options.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module getopt_io.
:- interface.

:- import_module bool.
:- import_module char.
:- import_module io.
:- import_module list.
:- import_module map.
:- import_module maybe.
:- import_module set.

% process_options(OptionOps, Args, NonOptionArgs, Result)
% process_options(OptionOps, Args, OptionArgs, NonOptionArgs, Result)
%
%   Scans through 'Args' looking for options, places all the option
%   arguments in 'OptionArgs', places all the non-option arguments in
%   'NonOptionArgs', and records the options in the 'OptionTable'.
%   'OptionTable' is a map from a user-defined option type to option_data.
%   If an invalid option is encountered, we return 'error(Message)'
%   otherwise we return 'ok(OptionTable)' in 'Result'.
%
%   The argument 'OptionOps' is a structure holding three or four
%   predicates used to categorize a set of options. Their
%   interfaces should be like these:
%
% :- pred short_option(char::in, option::out) is semidet.
%   True if the character names a valid single-character option.
%
% :- pred long_option(string::in, option::out) is semidet.
%   True if the string names a valid long option.
%
% :- pred option_default(option::out, option_data::out) is multi.
%   Nondeterministically returns all the options with their
%   corresponding types and default values.
```

```
%
% :- pred special_handler(option::in, special_data::in,
%   option_table::in, maybe_option_table(_)::out) is semidet.
%   This predicate is invoked whenever getopt finds an option
%   (long or short) designated as special, with special_data holding
%   the argument of the option (if any). The predicate can change the
%   option table in arbitrary ways in the course of handling the option,
%   or it can return an error message.
%   The canonical examples of special options are -O options in compilers,
%   which set many other options at once.

:- pred process_options(option_ops(OptionType)::in(option_ops),
    list(string)::in, list(string)::out, maybe_option_table(OptionType)::out,
    io::di, io::uo) is det.

:- pred process_options(option_ops(OptionType)::in(option_ops),
    list(string)::in, list(string)::out, list(string)::out,
    maybe_option_table(OptionType)::out, io::di, io::uo) is det.

% process_options_track(OptionOps, Args, OptionArgs,
%       NonOptionArgs, OptionTable0, Result, OptionsSet)

:- pred process_options_track(
    option_ops_track(OptionType)::in(option_ops_track),
    list(string)::in, list(string)::out, list(string)::out,
    option_table(OptionType)::in, maybe_option_table(OptionType)::out,
    set(OptionType)::out, io::di, io::uo) is det.

% Variants of the above that return structured errors.
% These behave as the above versions except that any error values returned are
% members of the option_error/1 type rather than strings.

:- pred process_options_se(option_ops(OptionType)::in(option_ops),
    list(string)::in, list(string)::out,
    maybe_option_table_se(OptionType)::out, io::di, io::uo) is det.

:- pred process_options_se(option_ops(OptionType)::in(option_ops),
    list(string)::in, list(string)::out, list(string)::out,
    maybe_option_table_se(OptionType)::out, io::di, io::uo) is det.

:- pred process_options_track_se(
    option_ops_track(OptionType)::in(option_ops_track),
    list(string)::in, list(string)::out, list(string)::out,
    option_table(OptionType)::in, maybe_option_table_se(OptionType)::out,
    set(OptionType)::out, io::di, io::uo) is det.

:- pred init_option_table(
```

```
    pred(OptionType, option_data)::in(pred(out, out) is nondet),
    option_table(OptionType)::out) is det.

:- pred init_option_table_multi(
    pred(OptionType, option_data)::in(pred(out, out) is multi),
    option_table(OptionType)::out) is det.

:- type option_ops(OptionType)
    --->    option_ops(
                pred(char, OptionType),         % short_option
                pred(string, OptionType),       % long_option
                pred(OptionType, option_data)   % option_default
            )
    ;       option_ops(
                pred(char, OptionType),         % short_option
                pred(string, OptionType),       % long_option
                pred(OptionType, option_data),  % option_default
                pred(OptionType, special_data,  % special option handler
                    option_table(OptionType),
                    maybe_option_table(OptionType))
            )
    ;       option_ops_multi(
                pred(char, OptionType),         % short_option
                pred(string, OptionType),       % long_option
                pred(OptionType, option_data)   % option_default
            )
    ;       option_ops_multi(
                pred(char, OptionType),         % short_option
                pred(string, OptionType),       % long_option
                pred(OptionType, option_data),  % option_default
                pred(OptionType, special_data,  % special option handler
                    option_table(OptionType),
                    maybe_option_table(OptionType))
            ).

:- inst option_ops for option_ops/1
    --->    option_ops(
                pred(in, out) is semidet,               % short_option
                pred(in, out) is semidet,               % long_option
                pred(out, out) is nondet                % option_default
            )
    ;       option_ops_multi(
                pred(in, out) is semidet,               % short_option
                pred(in, out) is semidet,               % long_option
                pred(out, out) is multi                 % option_default
            )
    ;       option_ops(
```

```
                pred(in, out) is semidet,               % short_option
                pred(in, out) is semidet,               % long_option
                pred(out, out) is nondet,               % option_default
                pred(in, in, in, out) is semidet        % special handler
            )
    ;       option_ops_multi(
                pred(in, out) is semidet,               % short_option
                pred(in, out) is semidet,               % long_option
                pred(out, out) is multi,                % option_default
                pred(in, in, in, out) is semidet        % special handler
            ).

:- type option_ops_track(OptionType)
    --->    option_ops_track(
                pred(char, OptionType),         % short_option
                pred(string, OptionType),       % long_option
                pred(OptionType, special_data,  % special option handler
                    option_table(OptionType),
                    maybe_option_table(OptionType),
                    set(OptionType))
            ).

:- inst option_ops_track for option_ops_track/1
    --->    option_ops_track(
                pred(in, out) is semidet,               % short_option
                pred(in, out) is semidet,               % long_option
                pred(in, in, in, out, out) is semidet   % special handler
            ).

:- type option_data
    --->    bool(bool)
    ;       int(int)
    ;       string(string)
    ;       maybe_int(maybe(int))
    ;       maybe_string(maybe(string))
    ;       accumulating(list(string))
    ;       special
    ;       bool_special
    ;       int_special
    ;       string_special
    ;       maybe_string_special
    ;       file_special.

:- type special_data
    --->    none
    ;       bool(bool)
    ;       int(int)
```

```
    ;       string(string)
    ;       maybe_string(maybe(string)).

:- type option_table(OptionType) == map(OptionType, option_data).

:- type maybe_option_table(OptionType)
    --->    ok(option_table(OptionType))
    ;       error(string).

:- type maybe_option_table_se(OptionType)
    --->    ok(option_table(OptionType))
    ;       error(option_error(OptionType)).

:- type option_error(OptionType)
    --->    unrecognized_option(string)
            % An option that is not recognized appeared on the command line.
            % The argument gives the option as it appeared on the command line.

    ;       option_error(OptionType, string, option_error_reason).
            % An error occurred with a specific option. The first two
            % arguments identify the option enumeration value and the string
            % that appeared on the command line for that option respectively.
            % The third argument describes the nature of the error with that
            % option.

:- type option_error_reason
    --->    unknown_type
            % No type for this option has been specified in the
            % 'option_default'/2 predicate.

    ;       requires_argument
            % The option requires an argument but it occurred on the command
            % line without one.

    ;       does_not_allow_argument(string)
            % The option does not allow an argument but it was provided with
            % one on the command line.
            % The argument gives the contents of the argument position on the
            % command line.

    ;       cannot_negate
            % The option cannot be negated but its negated form appeared on the
            % command line.

    ;       special_handler_failed
            % The special option handler predicate for the option failed.
```

```
        ;       special_handler_missing
                % A special option handler predicate was not provided
                % for the option.

        ;       special_handler_error(string)
                % The special option handler predicate for the option returned an
                % error.
                % The argument is a string describing the error.

        ;       requires_numeric_argument(string)
                % The option requires a numeric argument but it occurred on the
                % command line with a non-numeric argument.
                % The argument gives the contents of the argument position on the
                % command line.

        ;       file_special_cannot_open(string, io.error)
                % The option is a file_special option whose argument is the file
                % named by the first argument.
                % Attempting to open this file resulted in the I/O error given
                % by the second argument.

        ;       file_special_cannot_read(string, io.error)
                % The option is a file_special option whose argument is the file
                % named by the first argument.
                % Attempting to read from this file resulted in the I/O er-
    ror given
                % by the second argument.

        ;       file_special_contains_non_option_args(string).
                % The option is a file_special option whose argument is the file
                % named by the argument. This file contained some non-option
                % arguments.

    :- func option_error_to_string(option_error(OptionType)) = string.

        % The following three predicates search the option table for
        % an option of the specified type; if it is not found, they
        % report an error by calling error/1.

    :- pred lookup_bool_option(option_table(Option)::in, Option::in,
        bool::out) is det.
    :- func lookup_bool_option(option_table(Option), Option) = bool.

    :- pred lookup_int_option(option_table(Option)::in, Option::in,
        int::out) is det.
    :- func lookup_int_option(option_table(Option), Option) = int.
```

```
:- pred lookup_string_option(option_table(Option)::in, Option::in,
    string::out) is det.
:- func lookup_string_option(option_table(Option), Option) = string.

:- pred lookup_maybe_int_option(option_table(Option)::in, Option::in,
    maybe(int)::out) is det.
:- func lookup_maybe_int_option(option_table(Option), Option) =
    maybe(int).

:- pred lookup_maybe_string_option(option_table(Option)::in,
    Option::in, maybe(string)::out) is det.
:- func lookup_maybe_string_option(option_table(Option), Option) =
    maybe(string).

:- pred lookup_accumulating_option(option_table(Option)::in,
    Option::in, list(string)::out) is det.
:- func lookup_accumulating_option(option_table(Option), Option) =
    list(string).

%---------------------------------------------------%
%---------------------------------------------------%
```

32 hash_table

```
%---------------------------------------------------%
% vim: ts=4 sw=4 et ft=mercury
%---------------------------------------------------%
% Copyright (C) 2001, 2003-2006, 2010-2012 The University of Melbourne
% Copyright (C) 2013-2020 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: hash_table.m.
% Main author: rafe, wangp.
% Stability: low.
%
% Hash table implementation.
%
% This implementation requires the user to supply a predicate that
% computes a hash value for any given key.
%
% Default hash functions are provided for ints, strings and generic values.
%
% The number of buckets in the hash table is always a power of 2.
%
```

```
% When the occupancy reaches a level set by the user, we create automatically
% a new hash table with double the number of buckets, insert the contents
% of the old table into it, and use it to replace the old one.
%
% CAVEAT: The warning at the head of array.m about the use of unique objects
% also applies here. Briefly, the problem is that the compiler does not yet
% properly understand unique modes, hence we fake it using non-unique modes.
% This means that care must be taken not to use an old version of a
% destructively updated structure (such as a hash_table) since the
% compiler will not currently detect such errors.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module hash_table.
:- interface.

:- import_module array.
:- import_module assoc_list.
:- import_module char.

%-----------------------------------------------------%

:- type hash_table(K, V).

    % XXX This is all fake until the compiler can handle nested unique modes.
    %
:- inst hash_table for hash_table/2
    == bound(ht(ground, ground, hash_pred, array)).
:- mode hash_table_ui == in(hash_table).
:- mode hash_table_di == di(hash_table).
:- mode hash_table_uo == out(hash_table).

:- type hash_pred(K) == ( pred(K, int) ).
:- inst hash_pred    == ( pred(in, out) is det ).

%-----------------------------------------------------%

    % init(HashPred, N, MaxOccupancy):
    %
    % Constructs a new hash table whose initial size is 2 ^ N, and whose
    % size is doubled whenever MaxOccupancy is achieved. Elements are
    % indexed using HashPred.
    %
    % HashPred must compute a hash for a given key.
    % N must be greater than 0.
    % MaxOccupancy must be in (0.0, 1.0).
```

```
    %
    % XXX Values too close to the limits may cause bad things to happen.
    %
:- func init(hash_pred(K), int, float) = hash_table(K, V).
:- mode init(in(hash_pred), in, in) = hash_table_uo is det.

    % init_default(HashFn) constructs a hash table with default size and
    % occupancy arguments.
    %
:- func init_default(hash_pred(K)) = hash_table(K, V).
:- mode init_default(in(hash_pred)) = hash_table_uo is det.

%-----------------------------------------------------%

    % Retrieve the hash_pred associated with a hash table.
    %
:- func hash_pred(hash_table(K, V)) = hash_pred(K).
:- mode hash_pred(hash_table_ui) = out(hash_pred) is det.

    % Returns the number of buckets in a hash table.
    %
:- func num_buckets(hash_table(K, V)) = int.
:- mode num_buckets(hash_table_ui) = out is det.
% :- mode num_buckets(in) = out is det.

    % Returns the number of occupants in a hash table.
    %
:- func num_occupants(hash_table(K, V)) = int.
:- mode num_occupants(hash_table_ui) = out is det.
% :- mode num_occupants(in) = out is det.

%-----------------------------------------------------%

    % Copy the hash table.
    %
    % This is not a deep copy, it copies only enough of the structure to
    % create a new unique table.
    %
:- func copy(hash_table(K, V)) = hash_table(K, V).
:- mode copy(hash_table_ui) = hash_table_uo is det.

    % Insert key-value binding into a hash table; if one is already there,
    % then overwrite the previous value.
    %
:- func set(hash_table(K, V), K, V) = hash_table(K, V).
:- mode set(hash_table_di, in, in) = hash_table_uo is det.
```

```
:- pred set(K::in, V::in,
    hash_table(K, V)::hash_table_di, hash_table(K, V)::hash_table_uo) is det.

    % Field update for hash tables.
    % HT ^ elem(K) := V is equivalent to set(HT, K, V).
    %
:- func 'elem :='(K, hash_table(K, V), V) = hash_table(K, V).
:- mode 'elem :='(in, hash_table_di, in) = hash_table_uo is det.

    % Insert a key-value binding into a hash table. Throw an exception
    % if a binding for the key is already present.
    %
:- func det_insert(hash_table(K, V), K, V) = hash_table(K, V).
:- mode det_insert(hash_table_di, in, in) = hash_table_uo is det.

:- pred det_insert(K::in, V::in,
    hash_table(K, V)::hash_table_di, hash_table(K, V)::hash_table_uo) is det.

    % Change a key-value binding in a hash table. Throw an exception
    % if a binding for the key does not already exist.
    %
:- func det_update(hash_table(K, V), K, V) = hash_table(K, V).
:- mode det_update(hash_table_di, in, in) = hash_table_uo is det.

:- pred det_update(K::in, V::in,
    hash_table(K, V)::hash_table_di, hash_table(K, V)::hash_table_uo) is det.

    % Delete the entry for the given key, leaving the hash table
    % unchanged if there is no such entry.
    %
:- func delete(hash_table(K, V), K) = hash_table(K, V).
:- mode delete(hash_table_di, in) = hash_table_uo is det.

:- pred delete(K::in,
    hash_table(K, V)::hash_table_di, hash_table(K, V)::hash_table_uo) is det.

%-----------------------------------------------%

    % Lookup the value associated with the given key.
    % Fail if there is no entry for the key.
    %
:- func search(hash_table(K, V), K) = V.
:- mode search(hash_table_ui, in) = out is semidet.
% :- mode search(in, in, out) is semidet.

:- pred search(hash_table(K, V), K, V).
:- mode search(hash_table_ui, in, out) is semidet.
```

```
% :- mode search(in, in, out) is semidet.

    % Lookup the value associated with the given key.
    % Throw an exception if there is no entry for the key.
    %
:- func lookup(hash_table(K, V), K) = V.
:- mode lookup(hash_table_ui, in) = out is det.
% :- mode lookup(in, in) = out is det.

    % Field access for hash tables.
    % HT ^ elem(K) is equivalent to lookup(HT, K).
    %
:- func elem(K, hash_table(K, V)) = V.
:- mode elem(in, hash_table_ui) = out is det.
% :- mode elem(in, in) = out is det.

%-----------------------------------------------%

    % Convert a hash table into an association list.
    %
:- func to_assoc_list(hash_table(K, V)) = assoc_list(K, V).
:- mode to_assoc_list(hash_table_ui) = out is det.
% :- mode to_assoc_list(in) = out is det.

    % from_assoc_list(HashPred, N, MaxOccupancy, AssocList) = Table:
    %
    % Convert an association list into a hash table. The first three
    % parameters are the same as for init/3 above.
    %
:- func from_assoc_list(hash_pred(K), int, float, assoc_list(K, V)) =
    hash_table(K, V).
:- mode from_assoc_list(in(hash_pred), in, in, in) = hash_table_uo is det.

    % A simpler version of from_assoc_list/4, the values for N and
    % MaxOccupancy are configured with defaults such as in init_default/1
    %
:- func from_assoc_list(hash_pred(K)::in(hash_pred), assoc_list(K, V)::in) =
    (hash_table(K, V)::hash_table_uo) is det.

    % Fold a function over the key-value bindings in a hash table.
    %
:- func fold(func(K, V, T) = T, hash_table(K, V), T) = T.
:- mode fold(func(in, in, in) = out is det, hash_table_ui, in) = out is det.
:- mode fold(func(in, in, di) = uo is det, hash_table_ui, di) = uo is det.

    % Fold a predicate over the key-value bindings in a hash table.
    %
```

```
:- pred fold(pred(K, V, A, A), hash_table(K, V), A, A).
:- mode fold(in(pred(in, in, in, out) is det), hash_table_ui,
    in, out) is det.
:- mode fold(in(pred(in, in, mdi, muo) is det), hash_table_ui,
    mdi, muo) is det.
:- mode fold(in(pred(in, in, di, uo) is det), hash_table_ui,
    di, uo) is det.
:- mode fold(in(pred(in, in, in, out) is semidet), hash_table_ui,
    in, out) is semidet.
:- mode fold(in(pred(in, in, mdi, muo) is semidet), hash_table_ui,
    mdi, muo) is semidet.
:- mode fold(in(pred(in, in, di, uo) is semidet), hash_table_ui,
    di, uo) is semidet.

:- pred fold2(pred(K, V, A, A, B, B), hash_table(K, V), A, A, B, B).
:- mode fold2(in(pred(in, in, in, out, in, out) is det), hash_table_ui,
    in, out, in, out) is det.
:- mode fold2(in(pred(in, in, in, out, mdi, muo) is det), hash_table_ui,
    in, out, mdi, muo) is det.
:- mode fold2(in(pred(in, in, in, out, di, uo) is det), hash_table_ui,
    in, out, di, uo) is det.
:- mode fold2(in(pred(in, in, in, out, in, out) is semidet), hash_table_ui,
    in, out, in, out) is semidet.
:- mode fold2(in(pred(in, in, in, out, mdi, muo) is semidet), hash_table_ui,
    in, out, mdi, muo) is semidet.
:- mode fold2(in(pred(in, in, in, out, di, uo) is semidet), hash_table_ui,
    in, out, di, uo) is semidet.

:- pred fold3(pred(K, V, A, A, B, B, C, C), hash_table(K, V), A, A, B, B,
    C, C).
:- mode fold3(in(pred(in, in, in, out, in, out, in, out) is det),
    hash_table_ui, in, out, in, out, in, out) is det.
:- mode fold3(in(pred(in, in, in, out, in, out, mdi, muo) is det),
    hash_table_ui, in, out, in, out, mdi, muo) is det.
:- mode fold3(in(pred(in, in, in, out, in, out, di, uo) is det),
    hash_table_ui, in, out, in, out, di, uo) is det.
:- mode fold3(in(pred(in, in, in, out, in, out, in, out) is semidet),
    hash_table_ui, in, out, in, out, in, out) is semidet.
:- mode fold3(in(pred(in, in, in, out, in, out, mdi, muo) is semidet),
    hash_table_ui, in, out, in, out, mdi, muo) is semidet.
:- mode fold3(in(pred(in, in, in, out, in, out, di, uo) is semidet),
    hash_table_ui, in, out, in, out, di, uo) is semidet.

%-----------------------------------------------%

    % Default hash_preds for ints and strings and everything (buwahahaha!)
    %
```

```
:- pragma obsolete(int_hash/2, [int.hash/2]).
:- pred int_hash(int::in, int::out) is det.
:- pragma obsolete(uint_hash/2, [uint.hash/2]).
:- pred uint_hash(uint::in, int::out) is det.
:- pragma obsolete(float_hash/2, [float.hash/2]).
:- pred float_hash(float::in, int::out) is det.
:- pragma obsolete(char_hash/2, [char.hash/2]).
:- pred char_hash(char::in, int::out) is det.
:- pragma obsolete(string_hash/2, [string.hash/2]).
:- pred string_hash(string::in, int::out) is det.
:- pragma obsolete(generic_hash/2).
:- pred generic_hash(T::in, int::out) is det.

%-----------------------------------------------------%
%-----------------------------------------------------%
```

33 injection

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 2005-2006, 2010-2011 The University of Melbourne.
% Copyright (C) 2014-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: injection.m.
% Author: mark.
% Stability: low.
%
% This module provides the 'injection' ADT.  An injection is like a 'map'
% (see map.m) but it allows efficient reverse lookups, similarly to 'bimap'.
% This time efficiency comes at the expense of using twice as much space
% or more.  The difference between an injection and a bimap is that there
% can be values in the range of the injection that are not returned for any
% key, but for which a reverse lookup will still return a valid key.
%
% The invariants on this data structure, which are enforced by this module,
% are as follows:
%
% 1) For any key K, if a forward lookup succeeds with value V then a reverse
% lookup of value V will succeed with key K.
%
% 2) For any value V, if a reverse lookup succeeds with key K then a forward
% lookup of key K will succeed with some value (not necessarily V).
```

```
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module injection.
:- interface.

:- import_module assoc_list.
:- import_module list.
:- import_module map.

%-----------------------------------------------------%

:- type injection(K, V).

%-----------------------------------------------------%

    % Initialize an empty injection.
    %
:- func init = injection(K, V).
:- pred init(injection(K, V)::out) is det.

    % Initialize an injection with the given key-value pair.
    %
:- func singleton(K, V) = injection(K, V).

    % Check whether an injection is empty.
    %
:- pred is_empty(injection(K, V)::in) is semidet.

    % Search the injection for the value corresponding to a given key.
    %
:- func forward_search(injection(K, V), K) = V is semidet.
:- pred forward_search(injection(K, V)::in, K::in, V::out)
    is semidet.

    % Search the injection for the key corresponding to a given value.
    %
:- func reverse_search(injection(K, V), V) = K is semidet.
:- pred reverse_search(injection(K, V)::in, K::out, V::in)
    is semidet.

    % Combined forward/reverse search.
    % (Declaratively equivalent to reverse_search.)
    %
:- pred search(injection(K, V), K, V).
:- mode search(in, in, out) is cc_nondet.
```

```
:- mode search(in, out, in) is semidet.

    % Look up the value for a given key, but throw an exception if it
    % is not present.
    %
:- func lookup(injection(K, V), K) = V.
:- pred lookup(injection(K, V)::in, K::in, V::out) is det.

    % Look up the key for a given value, but throw an exception if it
    % is not present.
    %
:- func reverse_lookup(injection(K, V), V) = K.
:- pred reverse_lookup(injection(K, V)::in, K::out, V::in) is det.

    % Return the list of all keys in the injection.
    %
:- func keys(injection(K, V)) = list(K).
:- pred keys(injection(K, V)::in, list(K)::out) is det.

    % Return the list of all values in the injection.
    %
:- func values(injection(K, V)) = list(V).
:- pred values(injection(K, V)::in, list(V)::out) is det.

    % Succeeds if the injection contains the given key.
    %
:- pred contains_key(injection(K, V)::in, K::in) is semidet.

    % Succeeds if the injection contains the given value.
    %
:- pred contains_value(injection(K, V)::in, V::in) is semidet.

    % Insert a new key-value pair into the injection.  Fails if either
    % the key or value already exists.
    %
:- func insert(injection(K, V), K, V) = injection(K, V) is semidet.
:- pred insert(injection(K, V)::in, K::in, V::in,
    injection(K, V)::out) is semidet.

    % As above but throws an exception if the key or the value already
    % exists.
    %
:- func det_insert(injection(K, V), K, V) = injection(K, V).
:- pred det_insert(injection(K, V)::in, K::in, V::in,
    injection(K, V)::out) is det.

    % Update the value associated with a given key.  Fails if the key
```

```
    % does not already exist, or if the value is already associated
    % with a key.
    %
:- func update(injection(K, V), K, V) = injection(K, V) is semidet.
:- pred update(injection(K, V)::in, K::in, V::in,
    injection(K, V)::out) is semidet.

    % As above, but throws an exception if the key does not already exist,
    % or if the value is already associated with a key.
    %
:- func det_update(injection(K, V), K, V) = injection(K, V).
:- pred det_update(injection(K, V)::in, K::in, V::in,
    injection(K, V)::out) is det.

    % Sets the value associated with a given key, regardless of whether
    % the key exists already or not.  Fails if the value is already
    % associated with a key that is different from the given key.
    %
:- func set(injection(K, V), K, V) = injection(K, V) is semidet.
:- pred set(injection(K, V)::in, K::in, V::in,
    injection(K, V)::out) is semidet.

    % As above, but throws an exception if the value is already associated
    % with a key that is different from the given key.
    %
:- func det_set(injection(K, V), K, V) = injection(K, V).
:- pred det_set(injection(K, V)::in, K::in, V::in,
    injection(K, V)::out) is det.

    % Insert key-value pairs from an assoc_list into the given injection.
    % Fails if any of the individual inserts would fail.
    %
:- func insert_from_assoc_list(assoc_list(K, V), injection(K, V)) =
    injection(K, V) is semidet.
:- pred insert_from_assoc_list(assoc_list(K, V)::in,
    injection(K, V)::in, injection(K, V)::out) is semidet.

    % As above, but throws an exception if any of the individual
    % inserts would fail.
    %
:- func det_insert_from_assoc_list(assoc_list(K, V),
    injection(K, V)) = injection(K, V).
:- pred det_insert_from_assoc_list(assoc_list(K, V)::in,
    injection(K, V)::in, injection(K, V)::out) is det.

    % Set key-value pairs from an assoc_list into the given injection.
    % Fails of any of the individual sets would fail.
```

```
    %
:- func set_from_assoc_list(assoc_list(K, V), injection(K, V)) =
    injection(K, V) is semidet.
:- pred set_from_assoc_list(assoc_list(K, V)::in,
    injection(K, V)::in, injection(K, V)::out) is semidet.

    % As above, but throws an exception if any of the individual sets
    % would fail.
    %
:- func det_set_from_assoc_list(assoc_list(K, V), injection(K, V)) =
    injection(K, V).
:- pred det_set_from_assoc_list(assoc_list(K, V)::in,
    injection(K, V)::in, injection(K, V)::out) is det.

    % Insert key-value pairs from corresponding lists into the given
    % injection.  Fails if any of the individual inserts would fail.
    % Throws an exception if the lists are not of equal length.
    %
:- func insert_from_corresponding_lists(list(K), list(V),
    injection(K, V)) = injection(K, V) is semidet.
:- pred insert_from_corresponding_lists(list(K)::in, list(V)::in,
    injection(K, V)::in, injection(K, V)::out) is semidet.

    % As above, but throws an exception if any of the individual
    % inserts would fail.
    %
:- func det_insert_from_corresponding_lists(list(K), list(V),
    injection(K, V)) = injection(K, V).
:- pred det_insert_from_corresponding_lists(list(K)::in, list(V)::in,
    injection(K, V)::in, injection(K, V)::out) is det.

    % Set key-value pairs from corresponding lists into the given
    % injection.  Fails of any of the individual sets would fail.
    % Throws an exception if the lists are not of equal length.
    %
:- func set_from_corresponding_lists(list(K), list(V),
    injection(K, V)) = injection(K, V) is semidet.
:- pred set_from_corresponding_lists(list(K)::in, list(V)::in,
    injection(K, V)::in, injection(K, V)::out) is semidet.

    % As above, but throws an exception if any of the individual sets
    % would fail.
    %
:- func det_set_from_corresponding_lists(list(K), list(V),
    injection(K, V)) = injection(K, V).
:- pred det_set_from_corresponding_lists(list(K)::in, list(V)::in,
    injection(K, V)::in, injection(K, V)::out) is det.
```

```
    % Delete a key from an injection.  Also deletes any values that
    % correspond to that key.  If the key is not present, leave the
    % injection unchanged.
    %
:- func delete_key(injection(K, V), K) = injection(K, V).
:- pred delete_key(K::in, injection(K, V)::in, injection(K, V)::out) is det.

    % Delete a value from an injection.  Throws an exception if there is
    % a key that maps to this value.  If the value is not present, leave
    % the injection unchanged.
    %
:- func delete_value(injection(K, V), V) = injection(K, V).
:- pred delete_value(V::in, injection(K, V)::in, injection(K, V)::out) is det.

    % Apply delete_key to a list of keys.
    %
:- func delete_keys(injection(K, V), list(K)) = injection(K, V).
:- pred delete_keys(list(K)::in,
    injection(K, V)::in, injection(K, V)::out) is det.

    % Apply delete_value to a list of values.
    %
:- func delete_values(injection(K, V), list(V)) = injection(K, V).
:- pred delete_values(list(V)::in,
    injection(K, V)::in, injection(K, V)::out) is det.

    % Merge the contents of the two injections.  Both sets of keys must
    % be disjoint, and both sets of values must be disjoint.
    %
:- func merge(injection(K, V), injection(K, V)) = injection(K, V).
:- pred merge(injection(K, V)::in, injection(K, V)::in, injection(K, V)::out)
    is det.

    % Merge the contents of the two injections.  For keys that occur in
    % both injections, map them to the value in the second argument.
    % Both sets of values must be disjoint.
    %
:- func overlay(injection(K, V), injection(K, V)) = injection(K, V).
:- pred overlay(injection(K, V)::in, injection(K, V)::in, injection(K, V)::out)
    is det.

    % Apply an injection to a list of keys.
    % Throws an exception if any of the keys are not present.
    %
:- func apply_forward_map_to_list(injection(K, V), list(K)) = list(V).
:- pred apply_forward_map_to_list(injection(K, V)::in, list(K)::in,
```

```
    list(V)::out) is det.

    % Apply the inverse of an injection to a list of values.
    % Throws an exception if any of the values are not present.
    %
:- func apply_reverse_map_to_list(injection(K, V), list(V)) = list(K).
:- pred apply_reverse_map_to_list(injection(K, V)::in, list(V)::in,
    list(K)::out) is det.

    % Apply a transformation to all the keys in the injection.  If two
    % distinct keys become equal under this transformation then the
    % value associated with the greater of these two keys is used in the
    % result.
    %
:- func map_keys(func(V, K) = L, injection(K, V)) = injection(L, V).
:- pred map_keys(pred(V, K, L)::in(pred(in, in, out) is det),
    injection(K, V)::in, injection(L, V)::out) is det.

    % Same as map_keys, but deletes any keys for which the
    % transformation fails.
    %
:- pred filter_map_keys(pred(V, K, L)::in(pred(in, in, out) is semidet),
    injection(K, V)::in, injection(L, V)::out) is det.

    % Apply a transformation to all the values in the injection.  If two
    % distinct values become equal under this transformation then the
    % reverse search of these two values in the original map must lead
    % to the same key.  If it doesn't, then throw an exception.
    %
:- func map_values(func(K, V) = W, injection(K, V)) = injection(K, W).
:- pred map_values(pred(K, V, W)::in(pred(in, in, out) is det),
    injection(K, V)::in, injection(K, W)::out) is det.

    % Extract the forward map from an injection.
    %
:- func forward_map(injection(K, V)) = map(K, V).
:- pred forward_map(injection(K, V)::in, map(K, V)::out) is det.

    % Extract the reverse map from an injection.
    %
:- func reverse_map(injection(K, V)) = map(V, K).
:- pred reverse_map(injection(K, V)::in, map(V, K)::out) is det.

%---------------------------------------------------%
%---------------------------------------------------%
```

34 int

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 1994-2012 The University of Melbourne.
% Copyright (C) 2013-2018, 2020 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: int.m.
% Main authors: conway, fjh.
% Stability: medium.
%
% Predicates and functions for dealing with machine-size integer numbers.
%
% The behaviour of a computation for which overflow occurs is undefined.
% (In the current implementation, the predicates and functions in this
% module do not check for overflow, and the results you get are those
% delivered by the C compiler.  However, future implementations
% might check for overflow.)
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module int.
:- interface.

:- import_module array.
:- import_module enum.
:- import_module pretty_printer.

%-----------------------------------------------------%

:- instance enum(int).

%-----------------------------------------------------%

    % Less than.
    %
:- pred (int::in) < (int::in) is semidet.

    % Greater than.
    %
:- pred (int::in) > (int::in) is semidet.

    % Less than or equal.
```

```
    %
:- pred (int::in) =< (int::in) is semidet.

    % Greater than or equal.
    %
:- pred (int::in) >= (int::in) is semidet.

%-----------------------------------------------------%

    % abs(X) returns the absolute value of X.
    % Throws an exception if X = int.min_int.
    %
:- func abs(int) = int.
:- pred abs(int::in, int::out) is det.

    % unchecked_abs(X) returns the absolute value of X, except that the result
    % is undefined if X = int.min_int.
    %
:- func unchecked_abs(int) = int.

    % nabs(X) returns the negative absolute value of X.
    % Unlike abs/1 this function is defined for X = int.min_int.
    %
:- func nabs(int) = int.

%-----------------------------------------------------%

    % Maximum.
    %
:- func max(int, int) = int.
:- pred max(int::in, int::in, int::out) is det.

    % Minimum.
    %
:- func min(int, int) = int.
:- pred min(int::in, int::in, int::out) is det.

%-----------------------------------------------------%

    % Unary plus.
    %
:- func + (int::in) = (int::uo) is det.

    % Unary minus.
    %
:- func - (int::in) = (int::uo) is det.
```

```
    % Addition.
    %
:- func int + int = int.
:- mode in  + in  = uo  is det.
:- mode uo  + in  = in  is det.
:- mode in  + uo  = in  is det.

:- func plus(int, int) = int.

    % Subtraction.
    %
:- func int - int = int.
:- mode in  - in  = uo  is det.
:- mode uo  - in  = in  is det.
:- mode in  - uo  = in  is det.

:- func minus(int, int) = int.

    % Multiplication.
    %
:- func (int::in) * (int::in) = (int::uo) is det.
:- func times(int, int) = int.

    % Flooring integer division.
    % Truncates towards minus infinity, e.g. (-10) div 3 = (-4).
    %
    % Throws a 'domain_error' exception if the right operand is zero.
    % See the comments at the top of math.m to find out how to disable
    % domain checks.
    %
:- func div(int::in, int::in) = (int::uo) is det.

    % Truncating integer division.
    % Truncates towards zero, e.g. (-10) // 3 = (-3).
    % 'div' has nicer mathematical properties for negative operands,
    % but '//' is typically more efficient.
    %
    % Throws a 'domain_error' exception if the right operand is zero.
    % See the comments at the top of math.m to find out how to disable
    % domain checks.
    %
:- func (int::in) // (int::in) = (int::uo) is det.

    % (/)/2 is a synonym for (//)/2 to bring Mercury into line with
    % the common convention for naming integer division.
    %
:- func (int::in) / (int::in) = (int::uo) is det.
```

```
    % unchecked_quotient(X, Y) is the same as X // Y, but the behaviour
    % is undefined if the right operand is zero.
    %
:- func unchecked_quotient(int::in, int::in) = (int::uo) is det.

    % Modulus.
    % X mod Y = X - (X div Y) * Y
    %
:- func (int::in) mod (int::in) = (int::uo) is det.

    % Remainder.
    % X rem Y = X - (X // Y) * Y
    % 'mod' has nicer mathematical properties for negative X,
    % but 'rem' is typically more efficient.
    %
    % Throws a 'domain_error' exception if the right operand is zero.
    % See the comments at the top of math.m to find out how to disable
    % domain checks.
    %
:- func (int::in) rem (int::in) = (int::uo) is det.

    % unchecked_rem(X, Y) is the same as X rem Y, but the behaviour
    % is undefined if the right operand is zero.
    %
:- func unchecked_rem(int::in, int::in) = (int::uo) is det.

    % even(X) is equivalent to (X mod 2 = 0).
    %
:- pred even(int::in) is semidet.

    % odd(X) is equivalent to (not even(X)), i.e. (X mod 2 = 1).
    %
:- pred odd(int::in) is semidet.

    % Exponentiation.
    % pow(X, Y, Z): Z is X raised to the Yth power.
    % Throws a 'domain_error' exception if Y is negative.
    %
:- func pow(int, int) = int.
:- pred pow(int::in, int::in, int::out) is det.

    % Base 2 logarithm.
    % log2(X) = N is the least integer such that 2 to the power N
    % is greater than or equal to X.
    % Throws a 'domain_error' exception if X is not positive.
    %
```

```
:- func log2(int) = int.
:- pred log2(int::in, int::out) is det.

%-----------------------------------------------%

    % Left shift.
    % X << Y returns X "left shifted" by Y bits.
    % The bit positions vacated by the shift are filled by zeros.
    % Throws an exception if Y is not in [0, bits_per_int).
    %
:- func (int::in) << (int::in) = (int::uo) is det.

    % legacy_left_shift(X, Y) returns X "left shifted" by Y bits.
    % To be precise, if Y is negative, the result is X div (2^(-Y)), otherwise
    % the result is X * (2^Y).
    %
    % NOTE: this function is deprecated and may be removed in a future release.
    %
:- pragma obsolete(legacy_left_shift/2).
:- func legacy_left_shift(int::in, int::in) = (int::uo) is det.

    % unchecked_left_shift(X, Y) is the same as X << Y
    % except that the behaviour is undefined if Y is negative,
    % or greater than or equal to the result of `bits_per_int/1'.
    % It will typically be implemented more efficiently than X << Y.
    %
:- func unchecked_left_shift(int::in, int::in) = (int::uo) is det.

    % Right shift.
    % X >> Y returns X "right shifted" by Y bits.
    % The bit positions vacated by the shift are filled by the sign bit.
    % Throws an exception if Y is not in [0, bits_per_int).
    %
:- func (int::in) >> (int::in) = (int::uo) is det.

    % legacy_right_shift(X, Y) returns X "arithmetic right shifted" by Y bits.
    % To be precise, if Y is negative, the result is X * (2^(-Y)), otherwise
    % the result is X div (2^Y).
    %
:- pragma obsolete(legacy_right_shift/2).
:- func legacy_right_shift(int::in, int::in) = (int::uo) is det.

    % unchecked_right_shift(X, Y) is the same as X >> Y
    % except that the behaviour is undefined if Y is negative,
    % or greater than or equal to the result of `bits_per_int/1'.
    % It will typically be implemented more efficiently than X >> Y.
    %
```

```
:- func unchecked_right_shift(int::in, int::in) = (int::uo) is det.

%-----------------------------------------------%

    % Bitwise complement.
    %
:- func \ (int::in) = (int::uo) is det.

    % Bitwise and.
    %
:- func (int::in) /\ (int::in) = (int::uo) is det.

    % Bitwise or.
    %
:- func (int::in) \/ (int::in) = (int::uo) is det.

    % Bitwise exclusive or (xor).
    %
:- func xor(int, int) = int.
:- mode xor(in, in) = uo is det.
:- mode xor(in, uo) = in is det.
:- mode xor(uo, in) = in is det.

%-----------------------------------------------%

    % is/2, for backwards compatibility with Prolog.
    %
:- pred is(T, T) is det.
:- mode is(uo, di) is det.
:- mode is(out, in) is det.
:- pragma obsolete(is/2).

%-----------------------------------------------%

    % max_int is the maximum value of an int on this machine.
    %
:- func max_int = int.
:- pred max_int(int::out) is det.

    % min_int is the minimum value of an int on this machine.
    %
:- func min_int = int.
:- pred min_int(int::out) is det.

    % bits_per_int is the number of bits in an int on this machine.
    %
:- func bits_per_int = int.
```

```
:- pred bits_per_int(int::out) is det.

%-----------------------------------------------%

    % fold_up(F, Low, High, Acc) <=> list.foldl(F, Low .. High, Acc)
    %
    % NOTE: fold_up/4 is undefined if High = max_int.
    %
:- func fold_up(func(int, T) = T, int, int, T) = T.

    % fold_up(F, Low, High, !Acc) <=> list.foldl(F, Low .. High, !Acc)
    %
    % NOTE: fold_up/5 is undefined if High = max_int.
    %
:- pred fold_up(pred(int, T, T), int, int, T, T).
:- mode fold_up(pred(in, in, out) is det, in, in, in, out) is det.
:- mode fold_up(pred(in, mdi, muo) is det, in, in, mdi, muo) is det.
:- mode fold_up(pred(in, di, uo) is det, in, in, di, uo) is det.
:- mode fold_up(pred(in, array_di, array_uo) is det, in, in,
    array_di, array_uo) is det.
:- mode fold_up(pred(in, in, out) is semidet, in, in, in, out)
    is semidet.
:- mode fold_up(pred(in, mdi, muo) is semidet, in, in, mdi, muo)
    is semidet.
:- mode fold_up(pred(in, di, uo) is semidet, in, in, di, uo)
    is semidet.
:- mode fold_up(pred(in, in, out) is nondet, in, in, in, out)
    is nondet.
:- mode fold_up(pred(in, mdi, muo) is nondet, in, in, mdi, muo)
    is nondet.
:- mode fold_up(pred(in, di, uo) is cc_multi, in, in, di, uo)
    is cc_multi.
:- mode fold_up(pred(in, in, out) is cc_multi, in, in, in, out)
    is cc_multi.

    % fold_up2(F, Low, High, !Acc1, Acc2) <=>
    %   list.foldl2(F, Low .. High, !Acc1, !Acc2)
    %
    % NOTE: fold_up2/7 is undefined if High = max_int.
    %
:- pred fold_up2(pred(int, T, T, U, U), int, int, T, T, U, U).
:- mode fold_up2(pred(in, in, out, in, out) is det, in, in, in, out,
    in, out) is det.
:- mode fold_up2(pred(in, in, out, mdi, muo) is det, in, in, in, out,
    mdi, muo) is det.
:- mode fold_up2(pred(in, in, out, di, uo) is det, in, in, in, out,
    di, uo) is det.
```

```
:- mode fold_up2(pred(in, di, uo, di, uo) is det, in, in, di, uo,
    di, uo) is det.
:- mode fold_up2(pred(in, in, out, array_di, array_uo) is det, in, in,
    in, out, array_di, array_uo) is det.
:- mode fold_up2(pred(in, in, out, in, out) is semidet, in, in,
    in, out, in, out) is semidet.
:- mode fold_up2(pred(in, in, out, mdi, muo) is semidet, in, in,
    in, out, mdi, muo) is semidet.
:- mode fold_up2(pred(in, in, out, di, uo) is semidet, in, in,
    in, out, di, uo) is semidet.
:- mode fold_up2(pred(in, in, out, in, out) is nondet, in, in,
    in, out, in, out) is nondet.
:- mode fold_up2(pred(in, in, out, mdi, muo) is nondet, in, in,
    in, out, mdi, muo) is nondet.

    % fold_up3(F, Low, High, !Acc1, Acc2, !Acc3) <=>
    %   list.foldl3(F, Low .. High, !Acc1, !Acc2, !Acc3)
    %
    % NOTE: fold_up3/9 is undefined if High = max_int.
    %
:- pred fold_up3(pred(int, T, T, U, U, V, V), int, int, T, T, U, U, V, V).
:- mode fold_up3(pred(in, in, out, in, out, in, out) is det,
    in, in, in, out, in, out, in, out) is det.
:- mode fold_up3(pred(in, in, out, in, out, mdi, muo) is det,
    in, in, in, out, in, out, mdi, muo) is det.
:- mode fold_up3(pred(in, in, out, in, out, di, uo) is det,
    in, in, in, out, in, out, di, uo) is det.
:- mode fold_up3(pred(in, in, out, di, uo, di, uo) is det,
    in, in, in, out, di, uo, di, uo) is det.
:- mode fold_up3(pred(in, in, out, in, out, array_di, array_uo) is det,
    in, in, in, out, in, out, array_di, array_uo) is det.
:- mode fold_up3(pred(in, in, out, in, out, in, out) is semidet,
    in, in, in, out, in, out, in, out) is semidet.
:- mode fold_up3(pred(in, in, out, in, out, mdi, muo) is semidet,
    in, in, in, out, in, out, mdi, muo) is semidet.
:- mode fold_up3(pred(in, in, out, in, out, di, uo) is semidet,
    in, in, in, out, in, out, di, uo) is semidet.
:- mode fold_up3(pred(in, in, out, in, out, in, out) is nondet,
    in, in, in, out, in, out, in, out) is nondet.
:- mode fold_up3(pred(in, in, out, in, out, mdi, muo) is nondet,
    in, in, in, out, in, out, mdi, muo) is nondet.

    % fold_down(F, Low, High, Acc) <=> list.foldr(F, Low .. High, Acc)
    %
    % NOTE: fold_down/4 is undefined if Low = min_int.
    %
:- func fold_down(func(int, T) = T, int, int, T) = T.
```

```
    % fold_down(F, Low, High, !Acc) <=> list.foldr(F, Low .. High, !Acc)
    %
    % NOTE: fold_down/5 is undefined if Low min_int.
    %
:- pred fold_down(pred(int, T, T), int, int, T, T).
:- mode fold_down(pred(in, in, out) is det, in, in, in, out) is det.
:- mode fold_down(pred(in, mdi, muo) is det, in, in, mdi, muo) is det.
:- mode fold_down(pred(in, di, uo) is det, in, in, di, uo) is det.
:- mode fold_down(pred(in, array_di, array_uo) is det, in, in,
    array_di, array_uo) is det.
:- mode fold_down(pred(in, in, out) is semidet, in, in, in, out)
    is semidet.
:- mode fold_down(pred(in, mdi, muo) is semidet, in, in, mdi, muo)
    is semidet.
:- mode fold_down(pred(in, di, uo) is semidet, in, in, di, uo)
    is semidet.
:- mode fold_down(pred(in, in, out) is nondet, in, in, in, out)
    is nondet.
:- mode fold_down(pred(in, mdi, muo) is nondet, in, in, mdi, muo)
    is nondet.
:- mode fold_down(pred(in, in, out) is cc_multi, in, in, in, out)
    is cc_multi.
:- mode fold_down(pred(in, di, uo) is cc_multi, in, in, di, uo)
    is cc_multi.

    % fold_down2(F, Low, High, !Acc1, !Acc2) <=>
    %   list.foldr2(F, Low .. High, !Acc1, Acc2).
    %
    % NOTE: fold_down2/7 is undefined if Low = min_int.
    %
:- pred fold_down2(pred(int, T, T, U, U), int, int, T, T, U, U).
:- mode fold_down2(pred(in, in, out, in, out) is det, in, in, in, out,
    in, out) is det.
:- mode fold_down2(pred(in, in, out, mdi, muo) is det, in, in, in, out,
    mdi, muo) is det.
:- mode fold_down2(pred(in, in, out, di, uo) is det, in, in, in, out,
    di, uo) is det.
:- mode fold_down2(pred(in, di, uo, di, uo) is det, in, in, di, uo,
    di, uo) is det.
:- mode fold_down2(pred(in, in, out, array_di, array_uo) is det, in, in,
    in, out, array_di, array_uo) is det.
:- mode fold_down2(pred(in, in, out, in, out) is semidet, in, in,
    in, out, in, out) is semidet.
:- mode fold_down2(pred(in, in, out, di, uo) is semidet, in, in,
    in, out, di, uo) is semidet.
:- mode fold_down2(pred(in, in, out, in, out) is nondet, in, in,
```

```
    in, out, in, out) is nondet.
:- mode fold_down2(pred(in, in, out, mdi, muo) is nondet, in, in,
    in, out, mdi, muo) is nondet.

    % fold_up3(F, Low, High, !Acc1, Acc2, !Acc3) <=>
    %   list.foldr3(F, Low .. High, !Acc1, !Acc2, !Acc3)
    %
    % NOTE: fold_down3/9 is undefined if Low = min_int.
    %
:- pred fold_down3(pred(int, T, T, U, U, V, V), int, int, T, T, U, U, V, V).
:- mode fold_down3(pred(in, in, out, in, out, in, out) is det,
    in, in, in, out, in, out, in, out) is det.
:- mode fold_down3(pred(in, in, out, in, out, mdi, muo) is det,
    in, in, in, out, in, out, mdi, muo) is det.
:- mode fold_down3(pred(in, in, out, in, out, di, uo) is det,
    in, in, in, out, in, out, di, uo) is det.
:- mode fold_down3(pred(in, in, out, di, uo, di, uo) is det,
    in, in, in, out, di, uo, di, uo) is det.
:- mode fold_down3(pred(in, in, out, in, out, array_di, array_uo) is det,
    in, in, in, out, in, out, array_di, array_uo) is det.
:- mode fold_down3(pred(in, in, out, in, out, in, out) is semidet,
    in, in, in, out, in, out, in, out) is semidet.
:- mode fold_down3(pred(in, in, out, in, out, mdi, muo) is semidet,
    in, in, in, out, in, out, mdi, muo) is semidet.
:- mode fold_down3(pred(in, in, out, in, out, di, uo) is semidet,
    in, in, in, out, in, out, di, uo) is semidet.
:- mode fold_down3(pred(in, in, out, in, out, in, out) is nondet,
    in, in, in, out, in, out, in, out) is nondet.
:- mode fold_down3(pred(in, in, out, in, out, mdi, muo) is nondet,
    in, in, in, out, in, out, mdi, muo) is nondet.

%---------------------------------------------------%

    % nondet_int_in_range(Low, High, I):
    %
    % On successive successes, set I to every integer from Low to High.
    %
:- pred nondet_int_in_range(int::in, int::in, int::out) is nondet.

    % all_true_in_range(P, Low, High):
    % True iff P is true for every integer in Low to High.
    %
    % NOTE: all_true_in_range/3 is undefined if High = max_int.
    %
:- pred all_true_in_range(pred(int)::in(pred(in) is semidet),
    int::in, int::in) is semidet.
```

```
%-----------------------------------------------------%

    % Convert an int to a pretty_printer.doc for formatting.
    %
:- func int_to_doc(int) = pretty_printer.doc.

%-----------------------------------------------------%
%
% Computing hashes of ints.
%

    % Compute a hash value for an int.
    %
:- func hash(int) = int.
:- pred hash(int::in, int::out) is det.

%-----------------------------------------------------%
%-----------------------------------------------------%
```

35 int8

```
%-----------------------------------------------------%
% vim: ts=4 sw=4 et ft=mercury
%-----------------------------------------------------%
% Copyright (C) 2017-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: int8.m
% Main author: juliensf
% Stability: low.
%
% Predicates and functions for dealing with signed 8-bit integer numbers.
%
%-----------------------------------------------------%

:- module int8.
:- interface.

:- import_module pretty_printer.

%-----------------------------------------------------%
%
% Conversion from int.
%
```

```
    % from_int(I, I8):
    %
    % Convert an int to an int8.
    % Fails if I is not in [-(2^7), 2^7 - 1].
    %
:- pred from_int(int::in, int8::out) is semidet.

    % det_from_int(I) = I8:
    %
    % Convert an int to an int8.
    % Throws an exception if I is not in [-(2^7), 2^7 - 1].
    %
:- func det_from_int(int) = int8.

    % cast_from_int(I) = I8:
    %
    % Convert an int to an int8.
    % Always succeeds, but will yield a result that is mathematically equal
    % to I only if I is in [-(2^7), 2^7 - 1].
    %
:- func cast_from_int(int) = int8.

%-----------------------------------------------%
%
% Conversion to int.
%

    % to_int(I8) = I:
    %
    % Convert an int8 to an int. Since an int can be only 32 or 64 bits,
    % this is guaranteed to yield a result that is mathematically equal
    % to the original.
    %
:- func to_int(int8) = int.

    % cast_to_int(I8) = I:
    %
    % Convert an int8 to an int. Since an int can be only 32 or 64 bits,
    % this is guaranteed to yield a result that is mathematically equal
    % to the original.
    %
:- func cast_to_int(int8) = int.

%-----------------------------------------------%
%
% Change of signedness.
```

```
%

    % cast_from_uint8(U8) = I8:
    %
    % Convert a uint8 to an int8. This will yield a result that is
    % mathematically equal to U8 only if U8 is in [0, 2^7 - 1].
    %
:- func cast_from_uint8(uint8) = int8.

%-----------------------------------------------------%
%
% Comparisons and related operations.
%

    % Less than.
    %
:- pred (int8::in) < (int8::in) is semidet.

    % Greater than.
    %
:- pred (int8::in) > (int8::in) is semidet.

    % Less than or equal.
    %
:- pred (int8::in) =< (int8::in) is semidet.

    % Greater than or equal.
    %
:- pred (int8::in) >= (int8::in) is semidet.

    % Maximum.
    %
:- func max(int8, int8) = int8.

    % Minimum.
    %
:- func min(int8, int8) = int8.

%-----------------------------------------------------%
%
% Absolute values.
%

    % abs(X) returns the absolute value of X.
    % Throws an exception if X = int8.min_int8.
    %
:- func abs(int8) = int8.
```

```
    % unchecked_abs(X) returns the absolute value of X, except that the result
    % is undefined if X = int8.min_int8.
    %
:- func unchecked_abs(int8) = int8.

    % nabs(X) returns the negative of the absolute value of X.
    % Unlike abs/1 this function is defined for X = int8.min_int8.
    %
:- func nabs(int8) = int8.

%-----------------------------------------------------%
%
% Arithmetic operations.
%

    % Unary plus.
    %
:- func + (int8::in) = (int8::uo) is det.

    % Unary minus.
    %
:- func - (int8::in) = (int8::uo) is det.

    % Addition.
    %
:- func int8 + int8 = int8.
:- mode in + in = uo is det.
:- mode uo + in = in is det.
:- mode in + uo = in is det.

:- func plus(int8, int8) = int8.

    % Subtraction.
    %
:- func int8 - int8 = int8.
:- mode in - in = uo is det.
:- mode uo - in = in is det.
:- mode in - uo = in is det.

:- func minus(int8, int8) = int8.

    % Multiplication.
    %
:- func (int8::in) * (int8::in) = (int8::uo) is det.
:- func times(int8, int8) = int8.
```

```
    % Flooring integer division.
    % Truncates towards minus infinity, e.g. (-10_i8) div 3_i8 = (-4_i8).
    %
    % Throws a 'domain_error' exception if the right operand is zero.
    %
:- func (int8::in) div (int8::in) = (int8::uo) is det.

    % Truncating integer division.
    % Truncates towards zero, e.g. (-10_i8) // 3_i8 = (-3_i8).
    % 'div' has nicer mathematical properties for negative operands,
    % but '//' is typically more efficient.
    %
    % Throws a 'domain_error' exception if the right operand is zero.
    %
:- func (int8::in) // (int8::in) = (int8::uo) is det.

    % (/)/2 is a synonym for (//)/2.
    %
:- func (int8::in) / (int8::in) = (int8::uo) is det.

    % unchecked_quotient(X, Y) is the same as X // Y, but the behaviour
    % is undefined if the right operand is zero.
    %
:- func unchecked_quotient(int8::in, int8::in) = (int8::uo) is det.

    % Modulus.
    % X mod Y = X - (X div Y) * Y
    %
    % Throws a 'domain_error' exception if the right operand is zero.
    %
:- func (int8::in) mod (int8::in) = (int8::uo) is det.

    % Remainder.
    % X rem Y = X - (X // Y) * Y.
    %
    % Throws a 'domain_error/' exception if the right operand is zero.
    %
:- func (int8::in) rem (int8::in) = (int8::uo) is det.

    % unchecked_rem(X, Y) is the same as X rem Y, but the behaviour is
    % undefined if the right operand is zero.
    %
:- func unchecked_rem(int8::in, int8::in) = (int8::uo) is det.

    % even(X) is equivalent to (X mod 2i8 = 0i8).
    %
:- pred even(int8::in) is semidet.
```

```
    % odd(X) is equivalent to (not even(X)), i.e. (X mod 2i8 = 1i8).
    %
:- pred odd(int8::in) is semidet.

%-----------------------------------------------------%
%
% Shift operations.
%

    % Left shift.
    % X << Y returns X "left shifted" by Y bits.
    % The bit positions vacated by the shift are filled by zeros.
    % Throws an exception if Y is not in [0, 8).
    %
:- func (int8::in) << (int::in) = (int8::uo) is det.

    % unchecked_left_shift(X, Y) is the same as X << Y except that the
    % behaviour is undefined if Y is not in [0, 8).
    % It will typically be implemented more efficiently than X << Y.
    %
:- func unchecked_left_shift(int8::in, int::in) = (int8::uo) is det.

    % Right shift.
    % X >> Y returns X "right shifted" by Y bits.
    % The bit positions vacated by the shift are filled by the sign bit.
    % Throws an exception if Y is not in [0, 8).
    %
:- func (int8::in) >> (int::in) = (int8::uo) is det.

    % unchecked_right_shift(X, Y) is the same as X >> Y except that the
    % behaviour is undefined if Y is not in [0, 8).
    % It will typically be implemented more efficiently than X >> Y.
    %
:- func unchecked_right_shift(int8::in, int::in) = (int8::uo) is det.

%-----------------------------------------------------%
%
% Logical operations.
%

    % Bitwise and.
    %
:- func (int8::in) /\ (int8::in) = (int8::uo) is det.

    % Bitwise or.
    %
```

```
:- func (int8::in) \/ (int8::in) = (int8::uo) is det.

    % Bitwise exclusive or (xor).
    %
:- func xor(int8, int8) = int8.
:- mode xor(in, in) = uo is det.
:- mode xor(in, uo) = in is det.
:- mode xor(uo, in) = in is det.

    % Bitwise complement.
    %
:- func \ (int8::in) = (int8::uo) is det.

%-----------------------------------------------------%
%
% Operations on bits and bytes.
%

    % num_zeros(I) = N:
    %
    % N is the number of zeros in the binary representation of I.
    %
:- func num_zeros(int8) = int.

    % num_ones(I) = N:
    % N is the number of ones in the binary representation of I.
    %
:- func num_ones(int8) = int.

    % num_leading_zeros(I) = N:
    %
    % N is the number of leading zeros in the binary representation of I,
    % starting at the most significant bit position.
    % Note that num_leading_zeros(0i8) = 8.
    %
:- func num_leading_zeros(int8) = int.

    % num_trailing_zeros(I) = N:
    %
    % N is the number of trailing zeros in the binary representation of I,
    % starting at the least significant bit position.
    % Note that num_trailing_zeros(0i8) = 8.
    %
:- func num_trailing_zeros(int8) = int.

    % reverse_bits(A) = B:
    %
```

```
    % B is the is value that results from reversing the bits in the binary
    % representation of A.
    %
:- func reverse_bits(int8) = int8.

%---------------------------------------------------%
%
% Limits.

:- func min_int8 = int8.

:- func max_int8 = int8.

%---------------------------------------------------%
%
% Prettyprinting.
%

    % Convert an int8 to a pretty_printer.doc for formatting.
    %
:- func int8_to_doc(int8) = pretty_printer.doc.

%---------------------------------------------------%
%---------------------------------------------------%
```

36 int16

```
%---------------------------------------------------%
% vim: ts=4 sw=4 et ft=mercury
%---------------------------------------------------%
% Copyright (C) 2017-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: int16.m
% Main author: juliensf
% Stability: low.
%
% Predicates and functions for dealing with signed 16-bit integer numbers.
%
%---------------------------------------------------%

:- module int16.
:- interface.
```

```
:- import_module pretty_printer.

%-----------------------------------------------------%
%
% Conversion from int.
%

    % from_int(I, I16):
    %
    % Convert an int to an int16.
    % Fails if I is not in [-(2^15), 2^15 - 1].
    %
:- pred from_int(int::in, int16::out) is semidet.

    % det_from_int(I) = I16:
    %
    % Convert an int to an int16.
    % Throws an exception if I is not in [-(2^15), 2^15 - 1].
    %
:- func det_from_int(int) = int16.

    % cast_from_int(I) = I16:
    %
    % Convert an int to an int16.
    % Always succeeds, but will yield a result that is mathematically equal
    % to I only if I is in [-(2^15), 2^15 - 1].
    %
:- func cast_from_int(int) = int16.

%-----------------------------------------------------%
%
% Conversion to int.
%

    % to_int(I16) = I:
    %
    % Convert an int16 to an int. Since an int can be only 32 or 64 bits,
    % this is guaranteed to yield a result that is mathematically equal
    % to the original.
    %
:- func to_int(int16) = int.

    % cast_to_int(I16) = I:
    %
    % Convert an int16 to an int. Since an int can be only 32 or 64 bits,
    % this is guaranteed to yield a result that is mathematically equal
    % to the original.
```

```
    %
:- func cast_to_int(int16) = int.

%---------------------------------------------------%
%
% Change of signedness.
%

    % cast_from_uint16(U16) = I16:
    %
    % Convert a uint16 to an int16. This will yield a result that is
    % mathematically equal to U16 only if U16 is in [0, 2^15 - 1].
    %
:- func cast_from_uint16(uint16) = int16.

%---------------------------------------------------%
%
% Conversion from byte sequence.
%

    % from_bytes_le(LSB, MSB) = I16:
    %
    % I16 is the int16 whose least and most significant bytes are given by the
    % uint8s LSB and MSB respectively.
    %
:- func from_bytes_le(uint8, uint8) = int16.

    % from_bytes_be(MSB, LSB) = I16:
    %
    % I16 is the int16 whose least and most significant bytes are given by the
    % uint8s LSB and MSB respectively.
    %
:- func from_bytes_be(uint8, uint8) = int16.

%---------------------------------------------------%
%
% Comparisons and related operations.
%

    % Less than.
    %
:- pred (int16::in) < (int16::in) is semidet.

    % Greater than.
    %
:- pred (int16::in) > (int16::in) is semidet.
```

```
    % Less than or equal.
    %
:- pred (int16::in) =< (int16::in) is semidet.

    % Greater than or equal.
    %
:- pred (int16::in) >= (int16::in) is semidet.

    % Maximum.
    %
:- func max(int16, int16) = int16.

    % Minimum.
    %
:- func min(int16, int16) = int16.

%-----------------------------------------------------%
%
% Absolute values.
%

    % abs(X) returns the absolute value of X.
    % Throws an exception if X = int16.min_int16.
    %
:- func abs(int16) = int16.

    % unchecked_abs(X) returns the absolute value of X, except that the result
    % is undefined if X = int16.min_int16.
    %
:- func unchecked_abs(int16) = int16.

    % nabs(X) returns the negative of the absolute value of X.
    % Unlike abs/1 this function is defined for X = int16.min_int16.
    %
:- func nabs(int16) = int16.

%-----------------------------------------------------%
%
% Arithmetic operations.
%

    % Unary plus.
    %
:- func + (int16::in) = (int16::uo) is det.

    % Unary minus.
    %
```

```
:- func - (int16::in) = (int16::uo) is det.

    % Addition.
    %
:- func int16 + int16 = int16.
:- mode in + in = uo is det.
:- mode uo + in = in is det.
:- mode in + uo = in is det.

:- func plus(int16, int16) = int16.

    % Subtraction.
    %
:- func int16 - int16 = int16.
:- mode in - in = uo is det.
:- mode uo - in = in is det.
:- mode in - uo = in is det.

:- func minus(int16, int16) = int16.

    % Multiplication.
    %
:- func (int16::in) * (int16::in) = (int16::uo) is det.
:- func times(int16, int16) = int16.

    % Flooring integer division.
    % Truncates towards minus infinity, e.g. (-10_i16) div 3_i16 = (-4_i16).
    %
    % Throws a 'domain_error' exception if the right operand is zero.
    %
:- func (int16::in) div (int16::in) = (int16::uo) is det.

    % Truncating integer division.
    % Truncates towards zero, e.g. (-10_i16) // 3_i16 = (-3_i16).
    % 'div' has nicer mathematical properties for negative operands,
    % but '//' is typically more efficient.
    %
    % Throws a 'domain_error' exception if the right operand is zero.
    %
:- func (int16::in) // (int16::in) = (int16::uo) is det.

    % (/)/2 is a synonym for (//)/2.
    %
:- func (int16::in) / (int16::in) = (int16::uo) is det.

    % unchecked_quotient(X, Y) is the same as X // Y, but the behaviour
    % is undefined if the right operand is zero.
```

```
    %
:- func unchecked_quotient(int16::in, int16::in) = (int16::uo) is det.

    % Modulus.
    % X mod Y = X - (X div Y) * Y
    %
    % Throws a ‘domain_error’ exception if the right operand is zero.
    %
:- func (int16::in) mod (int16::in) = (int16::uo) is det.

    % Remainder.
    % X rem Y = X - (X // Y) * Y.
    %
    % Throws a ‘domain_error/‘ exception if the right operand is zero.
    %
:- func (int16::in) rem (int16::in) = (int16::uo) is det.

    % unchecked_rem(X, Y) is the same as X rem Y, but the behaviour is
    % undefined if the right operand is zero.
    %
:- func unchecked_rem(int16::in, int16::in) = (int16::uo) is det.

    % even(X) is equivalent to (X mod 2i16 = 0i16).
    %
:- pred even(int16::in) is semidet.

    % odd(X) is equivalent to (not even(X)), i.e. (X mod 2i16 = 1i16).
    %
:- pred odd(int16::in) is semidet.

%-----------------------------------------------%
%
% Shift operations.
%

    % Left shift.
    % X << Y returns X "left shifted" by Y bits.
    % The bit positions vacated by the shift are filled by zeros.
    % Throws an exception if Y is not in [0, 16).
    %
:- func (int16::in) << (int::in) = (int16::uo) is det.

    % unchecked_left_shift(X, Y) is the same as X << Y except that the
    % behaviour is undefined if Y is not in [0, 16).
    % It will typically be implemented more efficiently than X << Y.
    %
:- func unchecked_left_shift(int16::in, int::in) = (int16::uo) is det.
```

```
    % Right shift.
    % X >> Y returns X "right shifted" by Y bits.
    % The bit positions vacated by the shift are filled by the sign bit.
    % Throws an exception if Y is not in [0, 16).
    %
:- func (int16::in) >> (int::in) = (int16::uo) is det.

    % unchecked_right_shift(X, Y) is the same as X >> Y except that the
    % behaviour is undefined if Y is not in [0, 16).
    % It will typically be implemented more efficiently than X >> Y.
    %
:- func unchecked_right_shift(int16::in, int::in) = (int16::uo) is det.

%-----------------------------------------------%
%
% Logical operations.
%

    % Bitwise and.
    %
:- func (int16::in) /\ (int16::in) = (int16::uo) is det.

    % Bitwise or.
    %
:- func (int16::in) \/ (int16::in) = (int16::uo) is det.

    % Bitwise exclusive or (xor).
    %
:- func xor(int16, int16) = int16.
:- mode xor(in, in) = uo is det.
:- mode xor(in, uo) = in is det.
:- mode xor(uo, in) = in is det.

    % Bitwise complement.
    %
:- func \ (int16::in) = (int16::uo) is det.

%-----------------------------------------------%
%
% Operations on bits and bytes.
%

    % num_zeros(I) = N:
    %
    % N is the number of zeros in the binary representation of I.
    %
```

```
:- func num_zeros(int16) = int.

    % num_ones(I) = N:
    %
    % N is the number of ones in the binary representation of I.
    %
:- func num_ones(int16) = int.

    % num_leading_zeros(I) = N:
    %
    % N is the number of leading zeros in the binary representation of I,
    % starting at the most significant bit position.
    % Note that num_leading_zeros(0i16) = 16.
    %
:- func num_leading_zeros(int16) = int.

    % num_trailing_zeros(I) = N:
    %
    % N is the number of trailing zeros in the binary representation of I,
    % starting at the least significant bit position.
    % Note that num_trailing_zeros(0i16) = 16.
    %
:- func num_trailing_zeros(int16) = int.

    % reverse_bytes(A) = B:
    %
    % B is the value that results from reversing the bytes in the binary
    % representation of A.
    %
:- func reverse_bytes(int16) = int16.

    % reverse_bits(A) = B:
    %
    % B is the is value that results from reversing the bits in the binary
    % representation of A.
    %
:- func reverse_bits(int16) = int16.

%-----------------------------------------------%
%
% Limits.
%

:- func min_int16 = int16.

:- func max_int16 = int16.
```

```
%---------------------------------------------------%
%
% Prettyprinting.
%

    % Convert an int16 to a pretty_printer.doc for formatting.
    %
:- func int16_to_doc(int16) = pretty_printer.doc.

%---------------------------------------------------%
%---------------------------------------------------%
```

37 int32

```
%---------------------------------------------------%
% vim: ts=4 sw=4 et ft=mercury
%---------------------------------------------------%
% Copyright (C) 2017-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: int32.m
% Main author: juliensf
% Stability: low.
%
% Predicates and functions for dealing with signed 32-bit integer numbers.
%
%---------------------------------------------------%

:- module int32.
:- interface.

:- import_module pretty_printer.

%---------------------------------------------------%
%
% Conversion from int.
%

    % from_int(I, I32):
    %
    % Convert an int to an int32.
    % Fails if I is not in [-(2^31), 2^31 - 1].
    %
:- pred from_int(int::in, int32::out) is semidet.
```

```
    % det_from_int(I) = I32:
    %
    % Convert an int to an int32.
    % Throws an exception if I is not in [-(2^31), 2^31 - 1].
    %
:- func det_from_int(int) = int32.

    % cast_from_int(I) = I32:
    %
    % Convert an int to an int32.
    % Always succeeds, but will yield a result that is mathematically equal
    % to I only if I is in [-(2^31), 2^31 - 1].
    %
:- func cast_from_int(int) = int32.

%---------------------------------------------------%
%
% Conversion to int.
%

    % to_int(I32) = I:
    %
    % Convert an int32 to an int. Since an int can be only 32 or 64 bits,
    % this is guaranteed to yield a result that is mathematically equal
    % to the original.
    %
:- func to_int(int32) = int.

    % cast_to_int(I32) = I:
    %
    % Convert an int32 to an int. Since an int can be only 32 or 64 bits,
    % this is guaranteed to yield a result that is mathematically equal
    % to the original.
    %
:- func cast_to_int(int32) = int.

%---------------------------------------------------%
%
% Change of signedness.
%

    % cast_from_uint32(U32) = I32:
    %
    % Convert a uint32 to an int32. This will yield a result that is
    % mathematically equal to U32 only if U32 is in [0, 2^31 - 1].
    %
```

```
:- func cast_from_uint32(uint32) = int32.

%-----------------------------------------------------%
%
% Conversion from byte sequence.
%

    % from_bytes_le(Byte0, Byte1, Byte2, Byte3) = I32:
    %
    % I32 is the int32 whose bytes are given in little-endian order by the
    % arguments from left-to-right (i.e. Byte0 is the least significant byte
    % and Byte3 is the most significant byte).
    %
:- func from_bytes_le(uint8, uint8, uint8, uint8) = int32.

    % from_bytes_be(Byte0, Byte1, Byte2, Byte3) = I32:
    %
    % I32 is the int32 whose bytes are given in big-endian order by the
    % arguments in left-to-right order (i.e. Byte0 is the most significant
    % byte and Byte3 is the least significant byte).
    %
:- func from_bytes_be(uint8, uint8, uint8, uint8) = int32.

%-----------------------------------------------------%
%
% Comparisons and related operations.
%

    % Less than.
    %
:- pred (int32::in) < (int32::in) is semidet.

    % Greater than.
    %
:- pred (int32::in) > (int32::in) is semidet.

    % Less than or equal.
    %
:- pred (int32::in) =< (int32::in) is semidet.

    % Greater than or equal.
    %
:- pred (int32::in) >= (int32::in) is semidet.

    % Maximum.
    %
:- func max(int32, int32) = int32.
```

```
    % Minimum.
    %
:- func min(int32, int32) = int32.

%-----------------------------------------------------%
%
% Absolute values.
%

    % abs(X) returns the absolute value of X.
    % Throws an exception if X = int32.min_int32.
    %
:- func abs(int32) = int32.

    % unchecked_abs(X) returns the absolute value of X, except that the result
    % is undefined if X = int32.min_int32.
    %
:- func unchecked_abs(int32) = int32.

    % nabs(X) returns the negative of the absolute value of X.
    % Unlike abs/1 this function is defined for X = int32.min_int32.
    %
:- func nabs(int32) = int32.

%-----------------------------------------------------%
%
% Arithmetic operations.
%

    % Unary plus.
    %
:- func + (int32::in) = (int32::uo) is det.

    % Unary minus.
    %
:- func - (int32::in) = (int32::uo) is det.

    % Addition.
    %
:- func int32 + int32 = int32.
:- mode in + in = uo is det.
:- mode uo + in = in is det.
:- mode in + uo = in is det.

:- func plus(int32, int32) = int32.
```

```
    % Subtraction.
    %
:- func int32 - int32 = int32.
:- mode in - in = uo is det.
:- mode uo - in = in is det.
:- mode in - uo = in is det.

:- func minus(int32, int32) = int32.

    % Multiplication.
    %
:- func (int32::in) * (int32::in) = (int32::uo) is det.
:- func times(int32, int32) = int32.

    % Flooring integer division.
    % Truncates towards minus infinity, e.g. (-10_i32) div 3_i32 = (-4_i32).
    %
    % Throws a 'domain_error' exception if the right operand is zero.
    %
:- func (int32::in) div (int32::in) = (int32::uo) is det.

    % Truncating integer division.
    % Truncates towards zero, e.g. (-10_i32) // 3_i32 = (-3_i32).
    % 'div' has nicer mathematical properties for negative operands,
    % but '//' is typically more efficient.
    %
    % Throws a 'domain_error' exception if the right operand is zero.
    %
:- func (int32::in) // (int32::in) = (int32::uo) is det.

    % (/)/2 is a synonym for (//)/2.
    %
:- func (int32::in) / (int32::in) = (int32::uo) is det.

    % unchecked_quotient(X, Y) is the same as X // Y, but the behaviour
    % is undefined if the right operand is zero.
    %
:- func unchecked_quotient(int32::in, int32::in) = (int32::uo) is det.

    % Modulus.
    % X mod Y = X - (X div Y) * Y
    %
    % Throws a 'domain_error' exception if the right operand is zero.
    %
:- func (int32::in) mod (int32::in) = (int32::uo) is det.

    % Remainder.
```

```
    % X rem Y = X - (X // Y) * Y.
    %
    % Throws a `domain_error/` exception if the right operand is zero.
    %
:- func (int32::in) rem (int32::in) = (int32::uo) is det.

    % unchecked_rem(X, Y) is the same as X rem Y, but the behaviour is
    % undefined if the right operand is zero.
    %
:- func unchecked_rem(int32::in, int32::in) = (int32::uo) is det.

    % even(X) is equivalent to (X mod 2 = 0).
    %
:- pred even(int32::in) is semidet.

    % odd(X) is equivalent to (not even(X)), i.e. (X mod 2 = 1).
    %
:- pred odd(int32::in) is semidet.

%-----------------------------------------------------%
%
% Shift operations.
%

    % Left shift.
    % X << Y returns X "left shifted" by Y bits.
    % The bit positions vacated by the shift are filled by zeros.
    % Throws an exception if Y is not in [0, 32).
    %
:- func (int32::in) << (int::in) = (int32::uo) is det.

    % unchecked_left_shift(X, Y) is the same as X << Y except that the
    % behaviour is undefined if Y is not in [0, 32).
    % It will typically be implemented more efficiently than X << Y.
    %
:- func unchecked_left_shift(int32::in, int::in) = (int32::uo) is det.

    % Right shift.
    % X >> Y returns X "right shifted" by Y bits.
    % The bit positions vacated by the shift are filled by the sign bit.
    % Throws an exception if Y is not in [0, 32).
    %
:- func (int32::in) >> (int::in) = (int32::uo) is det.

    % unchecked_right_shift(X, Y) is the same as X >> Y except that the
    % behaviour is undefined if Y is not in [0, bits_per_int32).
    % It will typically be implemented more efficiently than X >> Y.
```

```
    %
:- func unchecked_right_shift(int32::in, int::in) = (int32::uo) is det.

%-----------------------------------------------------%
%
% Logical operations.
%

    % Bitwise and.
    %
:- func (int32::in) /\ (int32::in) = (int32::uo) is det.

    % Bitwise or.
    %
:- func (int32::in) \/ (int32::in) = (int32::uo) is det.

    % Bitwise exclusive or (xor).
    %
:- func xor(int32, int32) = int32.
:- mode xor(in, in) = uo is det.
:- mode xor(in, uo) = in is det.
:- mode xor(uo, in) = in is det.

    % Bitwise complement.
    %
:- func \ (int32::in) = (int32::uo) is det.

%-----------------------------------------------------%
%
% Operations on bits and bytes.
%

    % num_zeros(I) = N:
    %
    % N is the number of zeros in the binary representation of I.
    %
:- func num_zeros(int32) = int.

    % num_ones(I) = N:
    %
    % N is the number of ones in the binary representation of I.
    %
:- func num_ones(int32) = int.

    % num_leading_zeros(I) = N:
    %
    % N is the number of leading zeros in the binary representation of I,
```

```
    % starting at the most significant bit position.
    % Note that num_leading_zeros(0i32) = 32.
    %
:- func num_leading_zeros(int32) = int.

    % num_trailing_zeros(I) = N:
    %
    % N is the number of trailing zeros in the binary representation of I,
    % starting at the least significant bit position.
    % Note that num_trailing_zeros(0i32) = 32.
    %
:- func num_trailing_zeros(int32) = int.

    % reverse_bytes(A) = B:
    %
    % B is the value that results from reversing the bytes in the binary
    % representation of A.
    %
:- func reverse_bytes(int32) = int32.

    % reverse_bits(A) = B:
    %
    % B is the is value that results from reversing the bits in the binary
    % representation of A.
    %
:- func reverse_bits(int32) = int32.

%---------------------------------------------------%
%
% Limits.
%

:- func min_int32 = int32.

:- func max_int32 = int32.

%---------------------------------------------------%
%
% Prettyprinting.
%

    % Convert an int32 to a pretty_printer.doc for formatting.
    %
:- func int32_to_doc(int32) = pretty_printer.doc.

%---------------------------------------------------%
%---------------------------------------------------%
```

38 int64

```
%-----------------------------------------------------%
% vim: ts=4 sw=4 et ft=mercury
%-----------------------------------------------------%
% Copyright (C) 2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: int64.m
% Main author: juliensf
% Stability: low.
%
% Predicates and functions for dealing with signed 64-bit integer numbers.
%
%-----------------------------------------------------%

:- module int64.
:- interface.

:- import_module pretty_printer.

%-----------------------------------------------------%
%
% Conversion from int.
%

    % from_int(I) = I64:
    %
    % Convert an int to an int64.
    %
    % Since an int can be only 32 or 64 bits, this is guaranteed to yield
    % a result that is mathematically equal to the original.
    %
:- func from_int(int) = int64.

    % cast_from_int(I) = I64:
    %
    % Convert an int to an int64.
    %
    % While a cast from int to intN for N =< 32 may yield a result
    % that is not mathematically equal to the original (because
    % the original integer may be too big to be representable),
    % casting an int to int64 *will* yield a result that is mathematically
```

```
    % equal to the original. It is therefore a synonym for the from_int
    % function. It is provided only for uniformity, to allow an int
    % to be cast to intN for *all* of int8, int16, int32 and int64.
    %
:- func cast_from_int(int) = int64.

%-----------------------------------------------------%
%
% Conversion to int.
%

    % to_int(I64, I):
    %
    % Convert an int64 into an int.
    % Fails if I64 is not in [int.min_int, int.max_int].
    %
:- pred to_int(int64::in, int::out) is semidet.

    % det_to_int(I64) = I:
    %
    % Convert an int64 into an int.
    % Throws an exception if I64 is not in [int.min_int, int.max_int].
    %
:- func det_to_int(int64) = int.

    % cast_to_int(I64) = I:
    %
    % Convert an int64 to an int.
    % Always succeeds. If ints are 64 bits, I will always be
    % mathematically equal to I64. However, if ints are 32 bits,
    % then I will be mathematically equal to I64 only if
    % I64 is in [-(2^31), 2^31 - 1].
    %
:- func cast_to_int(int64) = int.

%-----------------------------------------------------%
%
% Change of signedness.
%

    % cast_from_uint64(U64) = I64:
    %
    % Convert a uint64 to an int64. This will yield a result that is
    % mathematically equal to U64 only if U64 is in [0, 2^63 - 1].
    %
:- func cast_from_uint64(uint64) = int64.
```

```
%---------------------------------------------------%
%
% Conversion from byte sequence.
%

    % from_bytes_le(Byte0, Byte1, ..., Byte7) = I64:
    %
    % I64 is the int64 whose bytes are given in little-endian order by the
    % arguments from left-to-right (i.e. Byte0 is the least significant byte
    % and Byte7 is the most significant byte).
    %
:- func from_bytes_le(uint8, uint8, uint8, uint8, uint8, uint8, uint8, uint8)
    = int64.

    % from_bytes_be(Byte0, Byte1, ..., Byte7) = I64:
    %
    % I64 is the int64 whose bytes are given in big-endian order by the
    % arguments in left-to-right order (i.e. Byte0 is the most significant
    % byte and Byte7 is the least significant byte).
    %
:- func from_bytes_be(uint8, uint8, uint8, uint8, uint8, uint8, uint8, uint8)
    = int64.

%---------------------------------------------------%
%
% Comparisons and related operations.
%

    % Less than.
    %
:- pred (int64::in) < (int64::in) is semidet.

    % Greater than.
    %
:- pred (int64::in) > (int64::in) is semidet.

    % Less than or equal.
    %
:- pred (int64::in) =< (int64::in) is semidet.

    % Greater than or equal.
    %
:- pred (int64::in) >= (int64::in) is semidet.

    % Maximum.
    %
:- func max(int64, int64) = int64.
```

```
    % Minimum.
    %
:- func min(int64, int64) = int64.

%-----------------------------------------------------%
%
% Absolute values.
%

    % abs(X) returns the absolute value of X.
    % Throws an exception if X = int64.min_int64.
    %
:- func abs(int64) = int64.

    % unchecked_abs(X) returns the absolute value of X, except that the result
    % is undefined if X = int64.min_int64.
    %
:- func unchecked_abs(int64) = int64.

    % nabs(X) returns the negative of the absolute value of X.
    % Unlike abs/1 this function is defined for X = int64.min_int64.
    %
:- func nabs(int64) = int64.

%-----------------------------------------------------%
%
% Arithmetic operations.
%

    % Unary plus.
    %
:- func + (int64::in) = (int64::uo) is det.

    % Unary minus.
    %
:- func - (int64::in) = (int64::uo) is det.

    % Addition.
    %
:- func int64 + int64 = int64.
:- mode in + in = uo is det.
:- mode uo + in = in is det.
:- mode in + uo = in is det.

:- func plus(int64, int64) = int64.
```

```
    % Subtraction.
    %
:- func int64 - int64 = int64.
:- mode in - in = uo is det.
:- mode uo - in = in is det.
:- mode in - uo = in is det.

:- func minus(int64, int64) = int64.

    % Multiplication.
    %
:- func (int64::in) * (int64::in) = (int64::uo) is det.
:- func times(int64, int64) = int64.

    % Flooring integer division.
    % Truncates towards minus infinity, e.g. -10_i64 div 3_i64 = -4_i64.
    %
    % Throws a 'domain_error' exception if the right operand is zero.
    %
:- func (int64::in) div (int64::in) = (int64::uo) is det.

    % Truncating integer division.
    % Truncates towards zero, e.g. -10_i64 // 3_i64 = -3_i64.
    % 'div' has nicer mathematical properties for negative operands,
    % but '//' is typically more efficient.
    %
    % Throws a 'domain_error' exception if the right operand is zero.
    %
:- func (int64::in) // (int64::in) = (int64::uo) is det.

    % (/)/2 is a synonym for (//)/2.
    %
:- func (int64::in) / (int64::in) = (int64::uo) is det.

    % unchecked_quotient(X, Y) is the same as X // Y, but the behaviour
    % is undefined if the right operand is zero.
    %
:- func unchecked_quotient(int64::in, int64::in) = (int64::uo) is det.

    % Modulus.
    % X mod Y = X - (X div Y) * Y
    %
    % Throws a 'domain_error' exception if the right operand is zero.
    %
:- func (int64::in) mod (int64::in) = (int64::uo) is det.

    % Remainder.
```

```
    % X rem Y = X - (X // Y) * Y.
    %
    % Throws a `domain_error/` exception if the right operand is zero.
    %
:- func (int64::in) rem (int64::in) = (int64::uo) is det.

    % unchecked_rem(X, Y) is the same as X rem Y, but the behaviour is
    % undefined if the right operand is zero.
    %
:- func unchecked_rem(int64::in, int64::in) = (int64::uo) is det.

    % even(X) is equivalent to (X mod 2 = 0).
    %
:- pred even(int64::in) is semidet.

    % odd(X) is equivalent to (not even(X)), i.e. (X mod 2 = 1).
    %
:- pred odd(int64::in) is semidet.

%-----------------------------------------------------%
%
% Shift operations.
%

    % Left shift.
    % X << Y returns X "left shifted" by Y bits.
    % The bit positions vacated by the shift are filled by zeros.
    % Throws an exception if Y is not in [0, 64).
    %
:- func (int64::in) << (int::in) = (int64::uo) is det.

    % unchecked_left_shift(X, Y) is the same as X << Y except that
    % the behaviour is undefined if Y is not in [0, 64).
    % It will typically be implemented more efficiently than X << Y.
    %
:- func unchecked_left_shift(int64::in, int::in) = (int64::uo) is det.

    % Right shift.
    % X >> Y returns X "right shifted" by Y bits.
    % The bit positions vacated by the shift are filled by the sign bit.
    % Throws an exception if Y is not in [0, 64).
    %
:- func (int64::in) >> (int::in) = (int64::uo) is det.

    % unchecked_right_shift(X, Y) is the same as X >> Y except that
    % the behaviour is undefined if Y is not in [0, 64).
    % It will typically be implemented more efficiently than X >> Y.
```

```
    %
:- func unchecked_right_shift(int64::in, int::in) = (int64::uo) is det.

%-----------------------------------------------------%
%
% Logical operations.
%

    % Bitwise and.
    %
:- func (int64::in) /\ (int64::in) = (int64::uo) is det.

    % Bitwise or.
    %
:- func (int64::in) \/ (int64::in) = (int64::uo) is det.

    % Bitwise exclusive or (xor).
    %
:- func xor(int64, int64) = int64.
:- mode xor(in, in) = uo is det.
:- mode xor(in, uo) = in is det.
:- mode xor(uo, in) = in is det.

    % Bitwise complement.
    %
:- func \ (int64::in) = (int64::uo) is det.

%-----------------------------------------------------%
%
% Operations on bits and bytes.
%

    % num_zeros(I) = N:
    %
    % N is the number of zeros in the binary representation of I.
    %
:- func num_zeros(int64) = int.

    % num_ones(I) = N:
    %
    % N is the number of ones in the binary representation of I.
    %
:- func num_ones(int64) = int.

    % num_leading_zeros(I) = N:
    %
    % N is the number of leading zeros in the binary representation of I,
```

```
    % starting at the most significant bit position.
    % Note that num_leading_zeros(0i64) = 64.
    %
:- func num_leading_zeros(int64) = int.

    % num_trailing_zeros(I) = N:
    %
    % N is the number of trailing zeros in the binary representation of I,
    % starting at the least significant bit position.
    % Note that num_trailing_zeros(0i64) = 64.
    %
:- func num_trailing_zeros(int64) = int.

    % reverse_bytes(A) = B:
    %
    % B is the value that results from reversing the bytes
    % in the binary representation of A.
    %
:- func reverse_bytes(int64) = int64.

    % reverse_bits(A) = B:
    %
    % B is the is value that results from reversing the bits
    % in the binary representation of A.
    %
:- func reverse_bits(int64) = int64.

%---------------------------------------------------%
%
% Limits.
%

:- func min_int64 = int64.

:- func max_int64 = int64.

%---------------------------------------------------%
%
% Prettyprinting.
%

    % Convert an int64 to a pretty_printer.doc for formatting.
    %
:- func int64_to_doc(int64) = pretty_printer.doc.

%---------------------------------------------------%
%---------------------------------------------------%
```

39 integer

```
%---------------------------------------------------%
% vim: ts=4 sw=4 et ft=mercury
%---------------------------------------------------%
% Copyright (C) 1997-2000, 2003-2007, 2011-2012 The University of Melbourne.
% Copyright (C) 2014-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: integer.m.
% Main authors: aet, Dan Hazel <odin@svrc.uq.edu.au>.
% Stability: high.
%
% This modules defines an arbitrary precision integer type (named "integer")
% and basic arithmetic operations on it.
%
% The builtin Mercury type "int" is implemented as machine integers,
% which on virtually all modern machines will be 32 or 64 bits in size.
% If you need to manipulate integers that may not fit into this many bits,
% you will want to use "integer"s instead of "int"s.
%
% NOTE: All the operators we define on "integers" behave the same as the
% corresponding operators on "int"s. This includes the operators related
% to division: /, //, rem, div, and mod.
%
%---------------------------------------------------%
%---------------------------------------------------%

:- module integer.
:- interface.

:- type integer.

%---------------------------------------------------%
%
% Constants.
%

    % Equivalent to integer(-1).
    %
:- func negative_one = integer.

    % Equivalent to integer(0).
```

```
    %
:- func zero = integer.

    % Equivalent to integer(1).
    %
:- func one = integer.

    % Equivalent to integer(2).
    %
:- func two = integer.

    % Equivalent to integer(8).
    %
:- func eight = integer.

    % Equivalent to integer(10).
    %
:- func ten = integer.

    % Equivalent to integer(16).
    %
:- func sixteen = integer.

%-----------------------------------------------%

    % X < Y: Succeed if and only if X is less than Y.
    %
:- pred '<'(integer::in, integer::in) is semidet.

    % X > Y: Succeed if and only if X is greater than Y.
    %
:- pred '>'(integer::in, integer::in) is semidet.

    % X =< Y: Succeed if and only if X is less than or equal to Y.
    %
:- pred '=<'(integer::in, integer::in) is semidet.

    % X >= Y: Succeed if and only if X is greater than or equal to Y.
    %
:- pred '>='(integer::in, integer::in) is semidet.

    % Absolute value.
    %
:- func abs(integer) = integer.

    % True if the argument is equal to integer.zero.
    %
```

```
:- pred is_zero(integer::in) is semidet.

%-----------------------------------------------%

    % Unary plus.
    %
:- func '+'(integer) = integer.

    % Unary minus.
    %
:- func '-'(integer) = integer.

    % Addition.
    %
:- func integer + integer = integer.

    % Subtraction.
    %
:- func integer - integer = integer.

    % Multiplication.
    %
:- func integer * integer = integer.

    % Flooring integer division.
    % Behaves as int.div.
    %
:- func integer div integer = integer.

    % Truncating integer division.
    % Behaves as int.(//).
    %
:- func integer // integer = integer.

    % Modulus.
    % Behaves as int.mod.
    %
:- func integer mod integer = integer.

    % Remainder.
    % Behaves as int.rem.
    %
:- func integer rem integer = integer.

    % divide_with_rem(X, Y, Q, R) where Q = X // Y and R = X rem Y
    % where both answers are calculated at the same time.
    %
```

```
:- pred divide_with_rem(integer::in, integer::in,
    integer::out, integer::out) is det.

    % Exponentiation.
    % pow(X, Y) = Z: Z is X raised to the Yth power.
    % Throws a 'domain_error' exception if Y is negative.
    %
:- func pow(integer, integer) = integer.

%-----------------------------------------------------%

    % Left shift.
    % Behaves as int.(<<).
    %
:- func integer << int = integer.

    % Right shift.
    % Behaves as int.(>>).
    %
:- func integer >> int = integer.

%-----------------------------------------------------%

    % Bitwise complement.
    %
:- func \ integer = integer.

    % Bitwise and.
    %
:- func integer /\ integer = integer.

    % Bitwise or.
    %
:- func integer \/ integer = integer.

    % Bitwise exclusive or (xor).
    %
:- func integer `xor` integer = integer.

%-----------------------------------------------------%

    % Convert an integer to an int.
    % Fails if the integer is not in the range [min_int, max_int].
    %
:- pred to_int(integer::in, int::out) is semidet.

    % As above, but throws an exception rather than failing.
```

```
    %
:- func det_to_int(integer) = int.

:- func int(integer) = int.
:- pragma obsolete(int/1).

%-----------------------------------------------%

    % Convert an integer to a uint.
    % Fails if the integer is not in the range [0, max_uint].
    %
:- pred to_uint(integer::in, uint::out) is semidet.

    % As above, but throws an exception rather than failing.
    %
:- func det_to_uint(integer) = uint.

    % Convert an integer to an int8.
    % Fails if the integer is not in the range [-128, 127].
    %
:- pred to_int8(integer::in, int8::out) is semidet.

    % As above, but throws an exception rather than failing.
    %
:- func det_to_int8(integer) = int8.

    % Convert an integer to a uint8.
    % Fails if the integer is not in the range [0, 255].
    %
:- pred to_uint8(integer::in, uint8::out) is semidet.

    % As above, but throws an exception rather than failing.
    %
:- func det_to_uint8(integer) = uint8.

    % Convert an integer to an int16.
    % Fails if the integer is not in the range [-32768, 32767].
    %
:- pred to_int16(integer::in, int16::out) is semidet.

    % As above, but throws an exception rather than failing.
    %
:- func det_to_int16(integer) = int16.

    % Convert an integer to a uint16.
    % Fails if the integer is not in the range [0, 65535].
    %
```

```
:- pred to_uint16(integer::in, uint16::out) is semidet.

    % As above, but throws an exception rather than failing.
    %
:- func det_to_uint16(integer) = uint16.

    % Convert an integer to an int32.
    % Fails if the integer is not in the range [-2147483648, 2147483647].
    %
:- pred to_int32(integer::in, int32::out) is semidet.

    % As above, but throws an exception rather than failing.
    %
:- func det_to_int32(integer) = int32.

    % Convert an integer to a uint32.
    % Fails if the integer is not in range [0, 4294967295].
    %
:- pred to_uint32(integer::in, uint32::out) is semidet.

    % As above, but throws an exception rather than failing.
    %
:- func det_to_uint32(integer) = uint32.

    % Convert an integer to an int64.
    % Fails if the integer is not in the range
    % [-9223372036854775808, 9223372036854775807].
    %
:- pred to_int64(integer::in, int64::out) is semidet.

    % As above, but throws an exception rather than failing.
    %
:- func det_to_int64(integer) = int64.

    % Convert an integer to a uint64.
    % Fails if the integer is not in range [0, 18446744073709551615].
    %
:- pred to_uint64(integer::in, uint64::out) is semidet.

    % As above, but throws an exception rather than failing.
    %
:- func det_to_uint64(integer) = uint64.

%-----------------------------------------------%

    % Convert an integer to a float.
    %
```

```
:- func float(integer) = float.

%-----------------------------------------------------%

    % Convert an integer to a string (in base 10).
    %
:- func to_string(integer) = string.

    % to_base_string(Integer, Base) = String:
    %
    % Convert an integer to a string in a given Base.
    %
    % Base must be between 2 and 36, both inclusive; if it is not,
    % the predicate will throw an exception.
    %
:- func to_base_string(integer, int) = string.

%-----------------------------------------------------%

    % Convert an int to integer.
    %
:- func integer(int) = integer.

    % Convert a uint to an integer.
    %
:- func from_uint(uint) = integer.

    % Convert an int8 to an integer.
    %
:- func from_int8(int8) = integer.

    % Convert a uint8 to an integer.
    %
:- func from_uint8(uint8) = integer.

    % Convert an int16 to an integer.
    %
:- func from_int16(int16) = integer.

    % Convert a uint16 to an integer.
    %
:- func from_uint16(uint16) = integer.

    % Convert an int32 to an integer.
    %
:- func from_int32(int32) = integer.
```

```
    % Convert a uint32 to an integer.
    %
:- func from_uint32(uint32) = integer.

    % Convert an int64 to an integer.
    %
:- func from_int64(int64) = integer.

    % Convert a uint64 to an integer.
    %
:- func from_uint64(uint64) = integer.

    % Convert a string to an integer. The string must contain only digits
    % [0-9], optionally preceded by a plus or minus sign. If the string does
    % not match this syntax, then the predicate fails.
    %
:- func from_string(string::in) = (integer::out) is semidet.
:- pragma obsolete(from_string/1).
:- pred from_string(string::in, integer::out) is semidet.

    % As above, but throws an exception rather than failing.
    %
:- func det_from_string(string) = integer.

    % Convert a string in the specified base (2-36) to an integer.
    % The string must contain one or more digits in the specified base,
    % optionally preceded by a plus or minus sign. For bases > 10, digits
    % 10 to 35 are represented by the letters A-Z or a-z. If the string
    % does not match this syntax, then the predicate fails.
    %
:- func from_base_string(int, string) = integer is semidet.
:- pragma obsolete(from_base_string/2).
:- pred from_base_string(int::in, string::in, integer::out) is semidet.

    % As above, but throws an exception rather than failing.
    %
:- func det_from_base_string(int, string) = integer.

%---------------------------------------------------%
%---------------------------------------------------%
```

40 io

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
```

```
%-----------------------------------------------------%
% Copyright (C) 1993-2012 The University of Melbourne.
% Copyright (C) 2013-2020 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: io.m.
% Main author: fjh.
% Stability: medium to high.
%
% This file encapsulates all the file I/O.
%
% We implement a purely logical I/O system using non-logical I/O primi-
tives of
% the underlying system. We ensure referential transparency by passing around
% a ‘‘state-of-the-world’’ argument using unique modes. The compiler will check
% that the state of the world argument is properly single-threaded, and will
% also ensure that the program doesn’t attempt to backtrack over any I/O.
%
% Attempting any operation on a stream which has already been closed results
% in undefined behaviour.
%
% In multithreaded programs, each thread in the program has its own set of
% "current" input and output streams. At the time it is created, a child
% thread inherits the current streams from its parent. Predicates that
% change which stream is current affect only the calling thread.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module io.
:- interface.

:- import_module array.
:- import_module bitmap.
:- import_module bool.
:- import_module char.
:- import_module deconstruct.
:- import_module list.
:- import_module map.
:- import_module maybe.
:- import_module stream.
:- import_module string.
:- import_module time.
:- import_module univ.

%-----------------------------------------------------%
```

```
%
% Exported types.
%

    % The state of the universe.
    %
:- type io.state.

    % An alternative, more concise name for ‘io.state’.
    %
:- type io == io.state.

    % Opaque handles for text I/O streams.
    %
:- type input_stream.
:- type output_stream.

    % Alternative names for the above.
    %
:- type text_input_stream == input_stream.
:- type text_output_stream == output_stream.

    % Opaque handles for binary I/O streams.
    %
:- type binary_input_stream.
:- type binary_output_stream.

    % A unique identifier for an I/O stream.
    %
:- type stream_id.

    % Various types used for the result from the access predicates.
    %
:- type res
    --->    ok
    ;       error(io.error).

:- type res(T)
    --->    ok(T)
    ;       error(io.error).

    % maybe_partial_res is used where it is possible to return a partial result
    % when an error occurs.
    %
:- type maybe_partial_res(T)
    --->    ok(T)
    ;       error(T, io.error).
```

```
:- type maybe_partial_res_2(T1, T2)
    --->    ok2(T1, T2)
    ;       error2(T1, T2, io.error).

:- inst maybe_partial_res(T) for maybe_partial_res/1
    --->    ok(T)
    ;       error(T, ground).

:- type result
    --->    ok
    ;       eof
    ;       error(io.error).

:- type result(T)
    --->    ok(T)
    ;       eof
    ;       error(io.error).

    % maybe_incomplete_result is returned when reading multibyte values from a
    % binary stream. 'incomplete(Bytes)' is returned when at least one byte of
    % a value has already been read but there are insufficient bytes
    % remaining the stream to complete the value. In that case, 'Bytes' will
    % contain the bytes that have already been read from the stream, in the
    % order in which they were read.
    %
:- type maybe_incomplete_result(T)
    --->    ok(T)
    ;       eof
    ;       incomplete(list(uint8))
    ;       error(io.error).

:- type read_result(T)
    --->    ok(T)
    ;       eof
    ;       error(string, int). % error message, line number

:- type io.error.   % Use error_message to decode it.

    % whence denotes the base for a seek operation.
    %   set - seek relative to the start of the file
    %   cur - seek relative to the current position in the file
    %   end - seek relative to the end of the file.
    %
:- type whence
    --->    set
    ;       cur
```

```
    ;       end.

%---------------------------------------------------%
%
% Opening and closing streams, both text and binary.
%

    % Attempts to open a text file for input.
    % Result is either 'ok(Stream)' or 'error(ErrorCode)'.
    %
:- pred open_input(string::in, io.res(io.text_input_stream)::out,
    io::di, io::uo) is det.

    % Attempts to open a binary file for input.
    % Result is either 'ok(Stream)' or 'error(ErrorCode)'.
    %
:- pred open_binary_input(string::in,
    io.res(io.binary_input_stream)::out, io::di, io::uo) is det.

%---------------------------------------------------%

    % Attempts to open a text file for output.
    % Result is either 'ok(Stream)' or 'error(ErrorCode)'.
    %
:- pred open_output(string::in, io.res(io.text_output_stream)::out,
    io::di, io::uo) is det.

    % Attempts to open a file for binary output.
    % Result is either 'ok(Stream)' or 'error(ErrorCode)'.
    %
:- pred open_binary_output(string::in,
    io.res(io.binary_output_stream)::out, io::di, io::uo) is det.

%---------------------------------------------------%

    % Attempts to open a text file for appending.
    % Result is either 'ok(Stream)' or 'error(ErrorCode)'.
    %
:- pred open_append(string::in, io.res(io.text_output_stream)::out,
    io::di, io::uo) is det.

    % Attempts to open a file for binary appending.
    % Result is either 'ok(Stream)' or 'error(ErrorCode)'.
    %
:- pred open_binary_append(string::in,
    io.res(io.binary_output_stream)::out, io::di, io::uo) is det.
```

```
%---------------------------------------------------%

    % Closes an open text input stream.
    % Throw an io.error exception if an I/O error occurs.
    %
:- pred close_input(io.text_input_stream::in, io::di, io::uo) is det.

    % Closes an open binary input stream. This will throw an io.error
    % exception if an I/O error occurs.
    %
:- pred close_binary_input(io.binary_input_stream::in,
    io::di, io::uo) is det.

%---------------------------------------------------%

    % Closes an open text output stream.
    % This will throw an io.error exception if an I/O error occurs.
    %
:- pred close_output(io.text_output_stream::in, io::di, io::uo) is det.

    % Closes an open binary output stream.
    % This will throw an io.error exception if an I/O error occurs.
    %
:- pred close_binary_output(io.binary_output_stream::in,
    io::di, io::uo) is det.

%---------------------------------------------------%
%
% Switching streams.
%

    % set_input_stream(NewStream, OldStream, !IO):
    % Changes the current input stream to NewStream.
    % Returns the previous input stream as OldStream.
    %
:- pred set_input_stream(io.text_input_stream::in,
    io.text_input_stream::out, io::di, io::uo) is det.

    % Changes the current input stream to the stream specified.
    % Returns the previous stream.
    %
:- pred set_binary_input_stream(io.binary_input_stream::in,
    io.binary_input_stream::out, io::di, io::uo) is det.

%---------------------------------------------------%

    % set_output_stream(NewStream, OldStream, !IO):
```

```
    % Changes the current output stream to NewStream.
    % Returns the previous output stream as OldStream.
    %
:- pred set_output_stream(io.text_output_stream::in,
    io.text_output_stream::out, io::di, io::uo) is det.

    % Changes the current binary output stream to the stream specified.
    % Returns the previous stream.
    %
:- pred set_binary_output_stream(io.binary_output_stream::in,
    io.binary_output_stream::out, io::di, io::uo) is det.

%---------------------------------------------------%
%
% Prolog style predicates for opening and switching streams.
%

    % see(FileName, Result, !IO):
    % Attempts to open the named file for input, and if successful,
    % sets the current input stream to the newly opened stream.
    % Result is either 'ok' or 'error(ErrorCode)'.
    %
:- pred see(string::in, io.res::out, io::di, io::uo) is det.

    % Attempts to open a file for binary input, and if successful sets
    % the current binary input stream to the newly opened stream.
    % Result is either 'ok' or 'error(ErrorCode)'.
    %
:- pred see_binary(string::in, io.res::out, io::di, io::uo) is det.

%---------------------------------------------------%

    % seen(!IO):
    % Closes the current input stream.
    % The current input stream reverts to standard input.
    % This will throw an io.error exception if an I/O error occurs.
    %
:- pred seen(io::di, io::uo) is det.

    % Closes the current input stream. The current input stream reverts
    % to standard input. This will throw an io.error exception if
    % an I/O error occurs.
    %
:- pred seen_binary(io::di, io::uo) is det.

%---------------------------------------------------%
```

```
    % tell(FileName, Result, !IO):
    % Attempts to open the named file for output, and if successful,
    % sets the current output stream to the newly opened stream.
    % Result is either 'ok' or 'error(ErrCode)'.
    %
:- pred tell(string::in, io.res::out, io::di, io::uo) is det.

    % Attempts to open a file for binary output, and if successful sets
    % the current binary output stream to the newly opened stream.
    % As per Prolog tell/1. Result is either 'ok' or 'error(ErrCode)'.
    %
:- pred tell_binary(string::in, io.res::out, io::di, io::uo) is det.

%---------------------------------------------------%

    % told(!IO):
    % Closes the current output stream.
    % The current output stream reverts to standard output.
    % This will throw an io.error exception if an I/O error occurs.
    %
:- pred told(io::di, io::uo) is det.

    % Closes the current binary output stream. The default binary output
    % stream reverts to standard output. As per Prolog told/0. This will
    % throw an io.error exception if an I/O error occurs.
    %
:- pred told_binary(io::di, io::uo) is det.

%---------------------------------------------------%
%
% Seeking on binary streams.
%

    % Seek to an offset relative to Whence (documented above)
    % on a specified binary input stream. Attempting to seek on a pipe
    % or tty results in implementation dependent behaviour.
    %
    % A successful seek undoes any effects of putback_byte on the stream.
    %
:- pred seek_binary_input(io.binary_input_stream::in, io.whence::in,
    int::in, io::di, io::uo) is det.

    % As above, but the offset is always a 64-bit value.
    %
:- pred seek_binary_input64(io.binary_input_stream::in, io.whence::in,
    int64::in, io::di, io::uo) is det.
```

```
%---------------------------------------------------%

    % Seek to an offset relative to Whence (documented above)
    % on a specified binary output stream. Attempting to seek on a pipe
    % or tty results in implementation dependent behaviour.
    %
:- pred seek_binary_output(io.binary_output_stream::in, io.whence::in,
    int::in, io::di, io::uo) is det.

    % As above, but the offset is always a 64-bit value.
    %
:- pred seek_binary_output64(io.binary_output_stream::in, io.whence::in,
    int64::in, io::di, io::uo) is det.

%---------------------------------------------------%

    % Returns the offset (in bytes) into the specified binary input stream.
    % Throws an exception if the offset is outside the range that can be
    % represented by the int type. To avoid this possibility, you can use the
    % 64-bit offset version of this predicate below.
    %
:- pred binary_input_stream_offset(io.binary_input_stream::in, int::out,
    io::di, io::uo) is det.

    % As above, but the offset is always a 64-bit value.
    %
:- pred binary_input_stream_offset64(io.binary_input_stream::in, int64::out,
    io::di, io::uo) is det.

%---------------------------------------------------%

    % Returns the offset (in bytes) into the specified binary output stream.
    % Throws an exception if the offset is outside the range that can be
    % represented by the int type. To avoid this possibility, you can use the
    % 64-bit offset version of this predicate below.
    %
:- pred binary_output_stream_offset(io.binary_output_stream::in, int::out,
    io::di, io::uo) is det.

    % As above, but the offset is always a 64-bit value.
    %
:- pred binary_output_stream_offset64(io.binary_output_stream::in, int64::out,
    io::di, io::uo) is det.

%---------------------------------------------------%
%
% Standard stream id predicates.
```

```
%

    % Retrieves the standard input stream.
    %
:- func stdin_stream = io.text_input_stream.

    % Retrieves the standard input stream.
    % Does not modify the I/O state.
    %
:- pred stdin_stream(io.text_input_stream::out, io::di, io::uo) is det.

    % Retrieves the standard binary input stream.
    % Does not modify the I/O state.
    %
:- pred stdin_binary_stream(io.binary_input_stream::out,
    io::di, io::uo) is det.

%---------------------------------------------------%

    % Retrieves the standard output stream.
    %
:- func stdout_stream = io.text_output_stream.

    % Retrieves the standard output stream.
    % Does not modify the I/O state.
    %
:- pred stdout_stream(io.text_output_stream::out, io::di, io::uo) is det.

    % Retrieves the standard binary output stream.
    % Does not modify the I/O state.
    %
:- pred stdout_binary_stream(io.binary_output_stream::out,
    io::di, io::uo) is det.

%---------------------------------------------------%

    % Retrieves the standard error stream.
    %
:- func stderr_stream = io.text_output_stream.

    % Retrieves the standard error stream.
    % Does not modify the I/O state.
    %
:- pred stderr_stream(io.text_output_stream::out, io::di, io::uo) is det.

%---------------------------------------------------%
%
```

```
% Current stream id predicates.
%

    % Retrieves the current input stream.
    % Does not modify the I/O state.
    %
:- pred input_stream(io.text_input_stream::out, io::di, io::uo) is det.

    % Retrieves the current binary input stream.
    % Does not modify the I/O state.
    %
:- pred binary_input_stream(io.binary_input_stream::out,
    io::di, io::uo) is det.

%---------------------------------------------------%

    % Retrieves the current output stream.
    % Does not modify the I/O state.
    %
:- pred output_stream(io.text_output_stream::out, io::di, io::uo) is det.

    % Retrieves the current binary output stream.
    % Does not modify the I/O state.
    %
:- pred binary_output_stream(io.binary_output_stream::out,
    io::di, io::uo) is det.

%---------------------------------------------------%
%
% Getting and setting stream properties.
%

    % Retrieves the human-readable name associated with the current input
    % stream or the specified output stream. For file streams, this is
    % the filename. For stdin, this is the string "<standard input>".
    %
:- pred input_stream_name(string::out, io::di, io::uo) is det.
:- pred input_stream_name(io.text_input_stream::in, string::out,
    io::di, io::uo) is det.

    % Retrieves the human-readable name associated with the current binary
    % input stream or the specified binary input stream. For file streams,
    % this is the filename.
    %
:- pred binary_input_stream_name(string::out, io::di, io::uo) is det.
:- pred binary_input_stream_name(io.binary_input_stream::in, string::out,
    io::di, io::uo) is det.
```

```
    % Retrieves the human-readable name associated with the current
    % output stream or the specified output stream.
    % For file streams, this is the filename.
    % For stdout this is the string "<standard output>".
    % For stderr this is the string "<standard error>".
    %
:- pred output_stream_name(string::out, io::di, io::uo) is det.
:- pred output_stream_name(io.text_output_stream::in, string::out,
    io::di, io::uo) is det.

    % Retrieves the human-readable name associated with the current
    % binary output stream or the specified binary output stream.
    % For file streams, this is the filename.
    %
:- pred binary_output_stream_name(string::out, io::di, io::uo) is det.
:- pred binary_output_stream_name(io.binary_output_stream::in,
    string::out, io::di, io::uo) is det.

%---------------------------------------------------%

    % Return the line number of the current input stream or the specified
    % input stream. Lines are normally numbered starting at 1, but this
    % can be overridden by calling set_line_number.
    %
:- pred get_line_number(int::out, io::di, io::uo) is det.
:- pred get_line_number(io.text_input_stream::in, int::out, io::di, io::uo)
    is det.

    % Set the line number of the current input stream or the specified
    % input stream.
    %
:- pred set_line_number(int::in, io::di, io::uo) is det.
:- pred set_line_number(io.text_input_stream::in, int::in, io::di, io::uo)
    is det.

    % Return the line number of the current output stream or the
    % specified output stream. Lines are normally numbered starting at 1,
    % but this can be overridden by calling set_output_line_number.
    %
:- pred get_output_line_number(int::out, io::di, io::uo) is det.
:- pred get_output_line_number(io.text_output_stream::in, int::out,
    io::di, io::uo) is det.

    % Set the line number of the current output stream.
    %
:- pred set_output_line_number(int::in, io::di, io::uo) is det.
```

```
:- pred set_output_line_number(io.text_output_stream::in, int::in,
    io::di, io::uo) is det.

%---------------------------------------------------%
%
% Reading values of primitive types.
%

    % Read a character (code point) from the current input stream
    % or from the specified stream.
    %
:- pred read_char(io.result(char)::out, io::di, io::uo) is det.
:- pred read_char(io.text_input_stream::in, io.result(char)::out,
    io::di, io::uo) is det.

    % Reads a character (code point) from the specified stream.
    % This interface avoids memory allocation when there is no error.
    %
:- pred read_char_unboxed(io.text_input_stream::in, io.result::out, char::out,
    io::di, io::uo) is det.

    % Un-read a character (code point) from the current input stream
    % or from the specified stream.
    % You can put back as many characters as you like.
    % You can even put back something that you didn't actually read.
    %
    % On some systems and backends, only one byte of pushback is guaranteed.
    % 'putback_char' will throw an io.error exception if the pushback buffer
    % is full.
    %
:- pred putback_char(char::in, io::di, io::uo) is det.
:- pred putback_char(io.text_input_stream::in, char::in, io::di, io::uo)
    is det.

% Note that there are no read equivalents of write_int, write_intN,
% write_uint, write_uintN, or write_float. Mercury programs that want to read
% numbers must first read in strings, and try to convert the appropriate
% parts of those strings to numbers. This allows them to handle any errors
% in that conversion process in whatever way they like. Since there are many
% possible ways to handle conversion failures, it is not very likely that
% a programmer's chosen method would agree with the one used by a
% system-supplied predicate for reading in e.g. floats, if this module had one.

%---------------------------------------------------%

    % Reads a single 8-bit byte from the current binary input stream
    % or from the specified binary input stream.
```

```
    %
:- pred read_byte(io.result(int)::out, io::di, io::uo) is det.
:- pred read_byte(io.binary_input_stream::in, io.result(int)::out,
    io::di, io::uo) is det.

    % Reads a single signed 8-bit integer from the current binary input
    % stream or from the specified binary input stream.
    %
:- pred read_binary_int8(io.result(int8)::out, io::di, io::uo) is det.
:- pred read_binary_int8(io.binary_input_stream::in, io.result(int8)::out,
    io::di, io::uo) is det.

    % Reads a single unsigned 8-bit integer from the current binary input
    % stream or from the specified binary input stream.
    %
:- pred read_binary_uint8(io.result(uint8)::out, io::di, io::uo) is det.
:- pred read_binary_uint8(io.binary_input_stream::in, io.result(uint8)::out,
    io::di, io::uo) is det.

%-----------------------------------------------%

    % Un-reads a byte from the current binary input stream or from the
    % specified stream. The byte is taken from the bottom 8 bits of the
    % specified int.
    %
    % You can put back as many bytes as you like.
    % You can even put back something that you did not actually read.
    %
    % On some systems and backends, only one byte of pushback is guaranteed.
    % 'putback_byte' will throw an io.error exception if the pushback buffer
    % is full.
    %
    % Pushing back a byte decrements the file position by one, except when
    % the file position is already zero, in which case the new file position
    % is unspecified.
    %
:- pred putback_byte(int::in, io::di, io::uo) is det.
:- pred putback_byte(io.binary_input_stream::in, int::in, io::di, io::uo)
    is det.

    % Like putback_byte, but where the byte value un-read is the 8 bits of the
    % int8 reinterpreted as a uint8.
    %
:- pred putback_int8(int8::in, io::di, io::uo) is det.
:- pred putback_int8(io.binary_input_stream::in, int8::in, io::di, io::uo)
    is det.
```

```
    % Like putback_byte, but where the byte value un-read is the 8 bits of the
    % uint8.
    %
:- pred putback_uint8(uint8::in, io::di, io::uo) is det.
:- pred putback_uint8(io.binary_input_stream::in, uint8::in, io::di, io::uo)
    is det.

%---------------------------------------------------%

    % The following predicates read multibyte integer values, either from
    % the current binary input stream, or from the specified binary
    % input stream.
    %
    % The names of these predicates have the form:
    %
    %    read_binary_<TYPE><SUFFIX>
    %
    % where <TYPE> is the name of one of the Mercury multibyte fixed size
    % integer types. The optional <SUFFIX> specifies the order in which
    % the bytes that make up the multibyte integer occur in input stream.
    % The suffix may be one of:
    %
    % "_le":    the bytes are in little endian byte order.
    % "_be":    the bytes are in big endian byte order.
    % none:     the bytes are in the byte order of the underlying platform.

:- pred read_binary_int16(maybe_incomplete_result(int16)::out,
    io::di, io::uo) is det.
:- pred read_binary_int16(io.binary_input_stream::in,
    maybe_incomplete_result(int16)::out, io::di, io::uo) is det.
:- pred read_binary_int16_le(maybe_incomplete_result(int16)::out,
    io::di, io::uo) is det.
:- pred read_binary_int16_le(io.binary_input_stream::in,
    maybe_incomplete_result(int16)::out, io::di, io::uo) is det.
:- pred read_binary_int16_be(maybe_incomplete_result(int16)::out,
    io::di, io::uo) is det.
:- pred read_binary_int16_be(io.binary_input_stream::in,
    maybe_incomplete_result(int16)::out, io::di, io::uo) is det.
:- pred read_binary_uint16(maybe_incomplete_result(uint16)::out,
    io::di, io::uo) is det.
:- pred read_binary_uint16(io.binary_input_stream::in,
    maybe_incomplete_result(uint16)::out, io::di, io::uo) is det.
:- pred read_binary_uint16_le(maybe_incomplete_result(uint16)::out,
    io::di, io::uo) is det.
:- pred read_binary_uint16_le(io.binary_input_stream::in,
    maybe_incomplete_result(uint16)::out, io::di, io::uo) is det.
:- pred read_binary_uint16_be(maybe_incomplete_result(uint16)::out,
```

```
    io::di, io::uo) is det.
:- pred read_binary_uint16_be(io.binary_input_stream::in,
    maybe_incomplete_result(uint16)::out, io::di, io::uo) is det.

:- pred read_binary_int32(maybe_incomplete_result(int32)::out,
    io::di, io::uo) is det.
:- pred read_binary_int32(io.binary_input_stream::in,
    maybe_incomplete_result(int32)::out, io::di, io::uo) is det.
:- pred read_binary_int32_le(maybe_incomplete_result(int32)::out,
    io::di, io::uo) is det.
:- pred read_binary_int32_le(io.binary_input_stream::in,
    maybe_incomplete_result(int32)::out, io::di, io::uo) is det.
:- pred read_binary_int32_be(maybe_incomplete_result(int32)::out,
    io::di, io::uo) is det.
:- pred read_binary_int32_be(io.binary_input_stream::in,
    maybe_incomplete_result(int32)::out, io::di, io::uo) is det.
:- pred read_binary_uint32(maybe_incomplete_result(uint32)::out,
    io::di, io::uo) is det.
:- pred read_binary_uint32(io.binary_input_stream::in,
    maybe_incomplete_result(uint32)::out, io::di, io::uo) is det.
:- pred read_binary_uint32_le(maybe_incomplete_result(uint32)::out,
    io::di, io::uo) is det.
:- pred read_binary_uint32_le(io.binary_input_stream::in,
    maybe_incomplete_result(uint32)::out, io::di, io::uo) is det.
:- pred read_binary_uint32_be(maybe_incomplete_result(uint32)::out,
    io::di, io::uo) is det.
:- pred read_binary_uint32_be(io.binary_input_stream::in,
    maybe_incomplete_result(uint32)::out, io::di, io::uo) is det.

:- pred read_binary_int64(maybe_incomplete_result(int64)::out,
    io::di, io::uo) is det.
:- pred read_binary_int64(io.binary_input_stream::in,
    maybe_incomplete_result(int64)::out, io::di, io::uo) is det.
:- pred read_binary_int64_le(maybe_incomplete_result(int64)::out,
    io::di, io::uo) is det.
:- pred read_binary_int64_le(io.binary_input_stream::in,
    maybe_incomplete_result(int64)::out, io::di, io::uo) is det.
:- pred read_binary_int64_be(maybe_incomplete_result(int64)::out,
    io::di, io::uo) is det.
:- pred read_binary_int64_be(io.binary_input_stream::in,
    maybe_incomplete_result(int64)::out, io::di, io::uo) is det.
:- pred read_binary_uint64(maybe_incomplete_result(uint64)::out,
    io::di, io::uo) is det.
:- pred read_binary_uint64(io.binary_input_stream::in,
    maybe_incomplete_result(uint64)::out, io::di, io::uo) is det.
:- pred read_binary_uint64_le(maybe_incomplete_result(uint64)::out,
    io::di, io::uo) is det.
```

```
:- pred read_binary_uint64_le(io.binary_input_stream::in,
    maybe_incomplete_result(uint64)::out, io::di, io::uo) is det.
:- pred read_binary_uint64_be(maybe_incomplete_result(uint64)::out,
    io::di, io::uo) is det.
:- pred read_binary_uint64_be(io.binary_input_stream::in,
    maybe_incomplete_result(uint64)::out, io::di, io::uo) is det.

%---------------------------------------------------%
%
% Writing values of primitive types.
%

    % Writes a character to the current output stream
    % or to the specified output stream.
    %
:- pred write_char(char::in, io::di, io::uo) is det.
:- pred write_char(io.text_output_stream::in, char::in, io::di, io::uo)
    is det.

    % Writes a signed or unsigned integer to the current output stream
    % or to the specified output stream.
    %
:- pred write_int(int::in, io::di, io::uo) is det.
:- pred write_int(io.text_output_stream::in, int::in, io::di, io::uo) is det.
:- pred write_uint(uint::in, io::di, io::uo) is det.
:- pred write_uint(io.text_output_stream::in, uint::in, io::di, io::uo) is det.

    % Write a signed or unsigned 8-bit integer to the current output stream
    % or to the specified output stream.
    %
:- pred write_int8(int8::in, io::di, io::uo) is det.
:- pred write_int8(io.text_output_stream::in, int8::in, io::di, io::uo) is det.
:- pred write_uint8(uint8::in, io::di, io::uo) is det.
:- pred write_uint8(io.text_output_stream::in, uint8::in, io::di, io::uo)
    is det.

    % Write a signed or unsigned 16-bit integer to the current output stream
    % or to the specified output stream.
    %
:- pred write_int16(int16::in, io::di, io::uo) is det.
:- pred write_int16(io.text_output_stream::in, int16::in, io::di, io::uo)
    is det.
:- pred write_uint16(uint16::in, io::di, io::uo) is det.
:- pred write_uint16(io.text_output_stream::in, uint16::in, io::di, io::uo)
    is det.

    % Write a signed or unsigned 32-bit integer to the current output stream
```

```
    % or to the specified output stream.
    %
:- pred write_int32(int32::in, io::di, io::uo) is det.
:- pred write_int32(io.text_output_stream::in, int32::in, io::di, io::uo)
    is det.
:- pred write_uint32(uint32::in, io::di, io::uo) is det.
:- pred write_uint32(io.text_output_stream::in, uint32::in, io::di, io::uo)
    is det.

    % Write a signed or unsigned 64-bit integer to the current output stream
    % or to the specified output stream.
    %
:- pred write_int64(int64::in, io::di, io::uo) is det.
:- pred write_int64(io.text_output_stream::in, int64::in, io::di, io::uo)
    is det.
:- pred write_uint64(uint64::in, io::di, io::uo) is det.
:- pred write_uint64(io.text_output_stream::in, uint64::in, io::di, io::uo)
    is det.

    % Writes a floating point number to the current output stream
    % or to the specified output stream.
    %
:- pred write_float(float::in, io::di, io::uo) is det.
:- pred write_float(io.text_output_stream::in, float::in, io::di, io::uo)
    is det.

    % Writes a string to the current output stream or to the
    % specified output stream.
    %
:- pred write_string(string::in, io::di, io::uo) is det.
:- pred write_string(io.text_output_stream::in, string::in, io::di, io::uo)
    is det.

    % Writes a newline character to the current output stream
    % or to the specified stream.
    %
:- pred nl(io::di, io::uo) is det.
:- pred nl(io.text_output_stream::in, io::di, io::uo) is det.

%---------------------------------------------------%

    % Writes a single byte to the current binary output stream
    % or to the specified binary output stream. The byte is taken from
    % the bottom 8 bits of the specified int.
    %
:- pred write_byte(int::in, io::di, io::uo) is det.
:- pred write_byte(io.binary_output_stream::in, int::in, io::di, io::uo)
```

```
    is det.

    % Writes a signed or unsigned 8-bit integer to the current binary
    % output stream or to the specified binary output stream.
    %
:- pred write_binary_int8(int8::in, io::di, io::uo) is det.
:- pred write_binary_int8(io.binary_output_stream::in, int8::in,
    io::di, io::uo) is det.
:- pred write_binary_uint8(uint8::in, io::di, io::uo) is det.
:- pred write_binary_uint8(io.binary_output_stream::in, uint8::in,
    io::di, io::uo) is det.

%---------------------------------------------------%

    % The following predicates write multibyte integer values, either to the
    % current binary output stream, or to the specified binary output stream.
    %
    % These names of these predicates have the form:
    %
    %    write_binary_<TYPE><SUFFIX>
    %
    % where <TYPE> is the name of one of the Mercury multibyte fixed size
    % integer types. The optional <SUFFIX> specifies the order in which
    % the bytes that make up the multibyte integer are written to the stream.
    % The suffix may be one of:
    %
    % "_le":    the bytes are in little endian byte order.
    % "_be":    the bytes are in big endian byte order.
    % none:     the bytes are in the byte order of the underlying platform.

:- pred write_binary_int16(int16::in, io::di, io::uo) is det.
:- pred write_binary_int16(io.binary_output_stream::in, int16::in,
    io::di, io::uo) is det.
:- pred write_binary_uint16(uint16::in, io::di, io::uo) is det.
:- pred write_binary_uint16(io.binary_output_stream::in, uint16::in,
    io::di, io::uo) is det.
:- pred write_binary_int16_le(int16::in, io::di, io::uo) is det.
:- pred write_binary_int16_le(io.binary_output_stream::in, int16::in,
    io::di, io::uo) is det.
:- pred write_binary_uint16_le(uint16::in, io::di, io::uo) is det.
:- pred write_binary_uint16_le(io.binary_output_stream::in, uint16::in,
    io::di, io::uo) is det.
:- pred write_binary_int16_be(int16::in, io::di, io::uo) is det.
:- pred write_binary_int16_be(io.binary_output_stream::in, int16::in,
    io::di, io::uo) is det.
:- pred write_binary_uint16_be(uint16::in, io::di, io::uo) is det.
:- pred write_binary_uint16_be(io.binary_output_stream::in, uint16::in,
```

```
    io::di, io::uo) is det.

:- pred write_binary_int32(int32::in, io::di, io::uo) is det.
:- pred write_binary_int32(io.binary_output_stream::in, int32::in,
    io::di, io::uo) is det.
:- pred write_binary_uint32(uint32::in, io::di, io::uo) is det.
:- pred write_binary_uint32(io.binary_output_stream::in, uint32::in,
    io::di, io::uo) is det.
:- pred write_binary_int32_le(int32::in, io::di, io::uo) is det.
:- pred write_binary_int32_le(io.binary_output_stream::in, int32::in,
    io::di, io::uo) is det.
:- pred write_binary_uint32_le(uint32::in, io::di, io::uo) is det.
:- pred write_binary_uint32_le(io.binary_output_stream::in, uint32::in,
    io::di, io::uo) is det.
:- pred write_binary_int32_be(int32::in, io::di, io::uo) is det.
:- pred write_binary_int32_be(io.binary_output_stream::in, int32::in,
    io::di, io::uo) is det.
:- pred write_binary_uint32_be(uint32::in, io::di, io::uo) is det.
:- pred write_binary_uint32_be(io.binary_output_stream::in, uint32::in,
    io::di, io::uo) is det.

:- pred write_binary_int64(int64::in, io::di, io::uo) is det.
:- pred write_binary_int64(io.binary_output_stream::in, int64::in,
    io::di, io::uo) is det.
:- pred write_binary_uint64(uint64::in, io::di, io::uo) is det.
:- pred write_binary_uint64(io.binary_output_stream::in, uint64::in,
    io::di, io::uo) is det.
:- pred write_binary_int64_le(int64::in, io::di, io::uo) is det.
:- pred write_binary_int64_le(io.binary_output_stream::in, int64::in,
    io::di, io::uo) is det.
:- pred write_binary_uint64_le(uint64::in, io::di, io::uo) is det.
:- pred write_binary_uint64_le(io.binary_output_stream::in, uint64::in,
    io::di, io::uo) is det.
:- pred write_binary_int64_be(int64::in, io::di, io::uo) is det.
:- pred write_binary_int64_be(io.binary_output_stream::in, int64::in,
    io::di, io::uo) is det.
:- pred write_binary_uint64_be(uint64::in, io::di, io::uo) is det.
:- pred write_binary_uint64_be(io.binary_output_stream::in, uint64::in,
    io::di, io::uo) is det.

%---------------------------------------------------%
%
% Text input predicates.
%

    % Read a whitespace delimited word from the current input stream
    % or from the specified stream.
```

```
    %
    % See ‘char.is_whitespace’ for the definition of whitespace characters
    % used by this predicate.
    %
:- pred read_word(io.result(list(char))::out, io::di, io::uo) is det.
:- pred read_word(io.text_input_stream::in, io.result(list(char))::out,
    io::di, io::uo) is det.

    % Read a line from the current input stream or from the specified
    % stream, returning the result as a list of characters (code points).
    %
    % See the documentation for ‘string.line’ for the definition of a line.
    %
:- pred read_line(io.result(list(char))::out, io::di, io::uo) is det.
:- pred read_line(io.text_input_stream::in, io.result(list(char))::out,
    io::di, io::uo) is det.

    % Read a line from the current input stream or from the specified
    % stream, returning the result as a string.
    %
    % See the documentation for ‘string.line’ for the definition of a line.
    %
    % WARNING: the returned string is NOT guaranteed to be valid UTF-8
    % or UTF-16.
    %
:- pred read_line_as_string(io.result(string)::out, io::di, io::uo) is det.
:- pred read_line_as_string(io.text_input_stream::in, io.result(string)::out,
    io::di, io::uo) is det.

    % Discards all the whitespace characters satisfying ‘char.is_whitespace’
    % from the current stream or from the specified stream.
    %
:- pred ignore_whitespace(io.result::out, io::di, io::uo) is det.
:- pred ignore_whitespace(io.text_input_stream::in, io.result::out,
    io::di, io::uo) is det.

%---------------------------------------------------%
%
% Bitmap input and output predicates.
%

    % Fill a bitmap from the current binary input stream
    % or from the specified binary input stream.
    % Return the number of bytes read. On end-of-file, the number of
    % bytes read will be less than the size of the bitmap, and
    % the result will be ‘ok’.
    % Throws an exception if the bitmap has a partial final byte.
```

```
    %
:- pred read_bitmap(bitmap::bitmap_di, bitmap::bitmap_uo,
    int::out, io.res::out, io::di, io::uo) is det.
:- pred read_bitmap(io.binary_input_stream::in,
    bitmap::bitmap_di, bitmap::bitmap_uo,
    int::out, io.res::out, io::di, io::uo) is det.

    % read_bitmap(StartByte, NumBytes, !Bitmap, BytesRead, Result, !IO)
    %
    % Read NumBytes bytes into a bitmap starting at StartByte from the
    % current binary input stream, or from the specified binary input stream.
    % Return the number of bytes read. On end-of-file, the number of
    % bytes read will be less than NumBytes, and the result will be 'ok'.
    %
:- pred read_bitmap(byte_index::in, num_bytes::in,
    bitmap::bitmap_di, bitmap::bitmap_uo, num_bytes::out, io.res::out,
    io::di, io::uo) is det.
:- pred read_bitmap(io.binary_input_stream::in, byte_index::in, num_bytes::in,
    bitmap::bitmap_di, bitmap::bitmap_uo, num_bytes::out, io.res::out,
    io::di, io::uo) is det.

%-----------------------------------------------%

    % Write a bitmap to the current binary output stream
    % or to the specified binary output stream. The bitmap must not contain
    % a partial final byte.
    %
:- pred write_bitmap(bitmap, io, io).
%:- mode write_bitmap(bitmap_ui, di, uo) is det.
:- mode write_bitmap(in, di, uo) is det.
:- pred write_bitmap(io.binary_output_stream, bitmap, io, io).
%:- mode write_bitmap(in, bitmap_ui, di, uo) is det.
:- mode write_bitmap(in, in, di, uo) is det.

    % write_bitmap(BM, StartByte, NumBytes, !IO):
    % write_bitmap(Stream, BM, StartByte, NumBytes, !IO):
    %
    % Write part of a bitmap to the current binary output stream
    % or to the specified binary output stream.
    %
:- pred write_bitmap(bitmap, int, int, io, io).
%:- mode write_bitmap(bitmap_ui, in, in, di, uo) is det.
:- mode write_bitmap(in, in, in, di, uo) is det.
:- pred write_bitmap(io.binary_output_stream, bitmap, int, int, io, io).
%:- mode write_bitmap(in, bitmap_ui, in, in, di, uo) is det.
:- mode write_bitmap(in, in, in, in, di, uo) is det.
```

```
%---------------------------------------------------%
%
% Reading values of arbitrary types.
%

    % Read a ground term of any type, written using standard Mercury syntax,
    % from the current stream or from the specified input stream.
    % The type of the term read is determined by the context from which
    % 'io.read' is called.
    %
    % This predicate reads the input stream until reaching one of
    % an end-of-term token, end-of-file, or I/O error.
    %
    % - If it finds no non-whitespace characters before the end-of-file,
    %   then it returns 'eof'.
    %
    % - If it finds a sequence of tokens ending with an end-of-term token,
    %   which is a '.' followed by whitespace, then it leaves the trailing
    %   whitespace in the input stream, and decides what to do based on
    %   the contents of the token sequence before the end-of-term token.
    %
    %   - If the tokens form a syntactically correct ground term of the
    %     expected type, then it returns 'ok(Term)'.
    %
    %   - If tokens do not form a syntactically correct term, or if the term
    %     they form is not ground, or if the term is not a valid term of the
    %     expected type, then it returns 'error(Message, LineNumber)'.
    %
    % - If it encounters an I/O error, then it also returns
    %   'error(Message, LineNumber)'.
    %
    % See 'char.is_whitespace' for the definition of whitespace characters
    % used by this predicate.
    %
:- pred read(io.read_result(T)::out, io::di, io::uo) is det.
:- pred read(io.text_input_stream::in, io.read_result(T)::out,
    io::di, io::uo) is det.

    % The type 'posn' represents a position within a string.
    %
:- type posn
    --->    posn(
                % The first two fields are used only for computing
                % term contexts, for use e.g. in error messages.
                %
                % Line numbers start at 1; offsets start at zero.
                % So the usual posn at the start of a file is posn(1, 0, 0).
```

```
                posn_current_line_number        :: int,
                posn_offset_of_start_of_line    :: int,
                posn_current_offset             :: int
            ).

    % read_from_string(FileName, String, MaxPos, Result, Posn0, Posn):
    %
    % Does the same job as read/4, but reads from a string, not from a stream.
    %
    % FileName is the name of the source (for use in error messages).
    % String is the string to be parsed.
    % Posn0 is the position to start parsing from.
    % Posn is the position one past where the term read in ends.
    % MaxPos is the offset in the string which should be considered the
    % end-of-stream -- this is the upper bound for Posn.
    % (In the usual case, MaxPos is just the length of the String.)
    % WARNING: if MaxPos > length of String, then the behaviour is UNDEFINED.
    %
:- pred read_from_string(string::in, string::in, int::in, read_result(T)::out,
    posn::in, posn::out) is det.

    % Reads a binary representation of a term of type T from the current
    % binary input stream or from the specified binary input stream.
    %
    % Note: if you attempt to read a binary representation written by
    % a different program, or a different version of the same program,
    % then the results are not guaranteed to be meaningful. Another caveat
    % is that higher-order types cannot be read. (If you try, you will get
    % a runtime error.)
    %
    % XXX Note also that in the current implementation, read_binary
    % will not work on the Java back-end.
    %
:- pred read_binary(io.result(T)::out, io::di, io::uo) is det.
:- pred read_binary(io.binary_input_stream::in, io.result(T)::out,
    io::di, io::uo) is det.

%---------------------------------------------------%
%
% Writing values of arbitrary types.
%

% These will all throw an io.error exception if an I/O error occurs.

    % print/3 writes its argument to the standard output stream.
    % print/4 writes its second argument to the output stream specified in
    % its first argument. In all cases, the argument to output can be of any
```

```
    % type. It is output in a format that is intended to be human readable.
    %
    % If the argument is just a single string or character, it will be printed
    % out exactly as is (unquoted). If the argument is of type integer (i.e.
    % an arbitrary precision integer), then its decimal representation will be
    % printed. If the argument is of type univ, then the value stored in the
    % the univ will be printed out, but not the type. If the argument is of
    % type date_time, it will be printed out in the same form as the string
    % returned by the function date_to_string/1. If the argument is of type
    % duration, it will be printed out in the same form as the string
    % returned by the function duration_to_string/1.
    %
    % print/5 is the same as print/4 except that it allows the caller to
    % specify how non-canonical types should be handled. print/3 and
    % print/4 implicitly specify 'canonicalize' as the method for handling
    % non-canonical types. This means that for higher-order types, or types
    % with user-defined equality axioms, or types defined using the foreign
    % language interface (i.e. pragma foreign_type), the text output will only
    % describe the type that is being printed, not the value.
    %
    % print_cc/3 is the same as print/3 except that it specifies
    % 'include_details_cc' rather than 'canonicalize'. This means that it will
    % print the details of non-canonical types. However, it has determinism
    % 'cc_multi'.
    %
    % Note that even if 'include_details_cc' is specified, some implementations
    % may not be able to print all the details for higher-order types or types
    % defined using the foreign language interface.
    %
:- pred print(T::in, io::di, io::uo) is det.
:- pred print(io.text_output_stream::in, T::in, io::di, io::uo) is det.

:- pred print(io.text_output_stream, deconstruct.noncanon_handling, T, io, io).
:- mode print(in, in(do_not_allow), in, di, uo) is det.
:- mode print(in, in(canonicalize), in, di, uo) is det.
:- mode print(in, in(include_details_cc), in, di, uo) is cc_multi.
:- mode print(in, in, in, di, uo) is cc_multi.

:- pred print_cc(T::in, io::di, io::uo) is cc_multi.

    % print_line calls print and then writes a newline character.
    %
:- pred print_line(T::in, io::di, io::uo) is det.
:- pred print_line(io.text_output_stream::in, T::in, io::di, io::uo) is det.

:- pred print_line(io.text_output_stream, deconstruct.noncanon_handling,
    T, io, io).
```

```
:- mode print_line(in, in(do_not_allow), in, di, uo) is det.
:- mode print_line(in, in(canonicalize), in, di, uo) is det.
:- mode print_line(in, in(include_details_cc), in, di, uo) is cc_multi.
:- mode print_line(in, in, in, di, uo) is cc_multi.

:- pred print_line_cc(T::in, io::di, io::uo) is cc_multi.

    % write/3 writes its argument to the current output stream.
    % write/4 writes its second argument to the output stream specified
    % in its first argument. In all cases, the argument to output may be
    % of any type. The argument is written in a format that is intended to
    % be valid Mercury syntax whenever possible.
    %
    % Strings and characters are always printed out in quotes, using backslash
    % escapes if necessary and backslash or octal escapes for all characters
    % for which char.is_control/1 is true. For higher-order types, or for types
    % defined using the foreign language interface (pragma foreign_type), the
    % text output will only describe the type that is being printed, not the
    % value, and the result may not be parsable by 'read'. For the types
    % containing existential quantifiers, the type 'type_desc' and closure
    % types, the result may not be parsable by 'read', either. But in all other
    % cases the format used is standard Mercury syntax, and if you append a
    % period and newline (".\n"), then the results can be read in again using
    % 'read'.
    %
    % write/5 is the same as write/4 except that it allows the caller
    % to specify how non-canonical types should be handled. write_cc/3
    % is the same as write/3 except that it specifies 'include_details_cc'
    % rather than 'canonicalize'.
    %
:- pred write(T::in, io::di, io::uo) is det.
:- pred write(io.text_output_stream::in, T::in, io::di, io::uo) is det.

:- pred write(io.text_output_stream, deconstruct.noncanon_handling, T, io, io).
:- mode write(in, in(do_not_allow), in, di, uo) is det.
:- mode write(in, in(canonicalize), in, di, uo) is det.
:- mode write(in, in(include_details_cc), in, di, uo) is cc_multi.
:- mode write(in, in, in, di, uo) is cc_multi.

:- pred write_cc(T::in, io::di, io::uo) is cc_multi.
:- pred write_cc(io.text_output_stream::in, T::in, io::di, io::uo) is cc_multi.

    % write_line calls write and then writes a newline character.
    %
:- pred write_line(T::in, io::di, io::uo) is det.
:- pred write_line(io.text_output_stream::in, T::in, io::di, io::uo) is det.
```

```
:- pred write_line(io.text_output_stream, deconstruct.noncanon_handling, T,
    io, io).
:- mode write_line(in, in(do_not_allow), in, di, uo) is det.
:- mode write_line(in, in(canonicalize), in, di, uo) is det.
:- mode write_line(in, in(include_details_cc), in, di, uo) is cc_multi.
:- mode write_line(in, in, in, di, uo) is cc_multi.

:- pred write_line_cc(T::in, io::di, io::uo) is cc_multi.

    % Writes a binary representation of a term to the current binary output
    % stream or to the specified stream, in a format suitable for read-
ing in
    % again with read_binary.
    %
:- pred write_binary(T::in, io::di, io::uo) is det.
:- pred write_binary(io.binary_output_stream::in, T::in, io::di, io::uo)
    is det.

%---------------------------------------------------%
%
% Formatted output.
%

    % Formats the specified arguments according to the format string,
    % using string.format, and then writes the result to the current
    % output stream or to the specified output stream.
    % (See the documentation of string.format for details.)
    %
:- pred format(string::in, list(poly_type)::in, io::di, io::uo) is det.
:- pred format(io.text_output_stream::in, string::in, list(poly_type)::in,
    io::di, io::uo) is det.

%---------------------------------------------------%
%
% Writing out several values.
%

    % Writes a list of strings to the current output stream
    % or to the specified output stream.
    %
:- pred write_strings(list(string)::in, io::di, io::uo) is det.
:- pred write_strings(io.text_output_stream::in, list(string)::in,
    io::di, io::uo) is det.

    % Writes the specified arguments to the current output stream
    % or to the specified output stream.
    %
```

```
:- pred write_many(list(poly_type)::in, io::di, io::uo) is det.
:- pred write_many(io.text_output_stream::in, list(poly_type)::in,
    io::di, io::uo) is det.

    % write_list(List, Separator, OutputPred, !IO):
    % write_list(Stream, List, Separator, OutputPred, !IO):
    %
    % Applies OutputPred to each element of List, printing Separator
    % (to the current output stream or to Stream) between each element.
    %
:- pred write_list(list(T), string, pred(T, io, io), io, io).
:- mode write_list(in, in, pred(in, di, uo) is det, di, uo) is det.
:- mode write_list(in, in, pred(in, di, uo) is cc_multi, di, uo)
    is cc_multi.

    % write_list(Stream, List, Separator, OutputPred, !IO):
    % Sets the current output stream to Stream, then applies OutputPred to
    % each element of List, printing Separator between each element.
    % The original output stream is restored whether returning normally
    % or if an exception is thrown.
    %
:- pred write_list(io.text_output_stream, list(T), string,
    pred(T, io, io), io, io).
:- mode write_list(in, in, in, pred(in, di, uo) is det, di, uo) is det.
:- mode write_list(in, in, in, pred(in, di, uo) is cc_multi, di, uo)
    is cc_multi.

    % write_array(Array, Separator, OutputPred, !IO):
    % Applies OutputPred to each element of Array, printing Separator
    % to the current output stream between each element.
    %
:- pred write_array(array(T), string, pred(T, io, io), io, io).
:- mode write_array(in, in, pred(in, di, uo) is det, di, uo) is det.
%:- mode write_array(array_ui, in, pred(in, di, uo) is det, di uo) is det.
:- mode write_array(in, in, pred(in, di, uo) is cc_multi, di, uo) is cc_multi.
%:- mode write_array(array_ui, in, pred(in, di, uo) is cc_multi, di uo)
% is cc_multi.

    % write_array(Stream, Array, Separator, OutputPred, !IO):
    % Sets the current output stream to Stream, then applies OutputPred to
    % each element of Array, printing Separator between each element.
    % The original output stream is restored whether returning normally
    % or if an exception is thrown.
    %
:- pred write_array(io.text_output_stream, array(T), string, pred(T, io, io),
    io, io).
:- mode write_array(in, in, in, pred(in, di, uo) is det, di, uo) is det.
```

```
%:- mode write_array(in, array_ui, in, pred(in, di, uo) is det, di uo) is det.
:- mode write_array(in, in, in, pred(in, di, uo) is cc_multi, di, uo)
    is cc_multi.
%:- mode write_array(in, array_ui, in, pred(in, di, uo) is cc_multi, di uo)
% is cc_multi.

%---------------------------------------------------%
%
% Flushing output to the operating system.
%

    % Flush the output buffer of the current output stream
    % or to the specified output stream.
    %
:- pred flush_output(io::di, io::uo) is det.
:- pred flush_output(io.text_output_stream::in, io::di, io::uo) is det.

    % Flush the output buffer of the current binary output stream.
    % or of the specified binary output stream.
    %
:- pred flush_binary_output(io::di, io::uo) is det.
:- pred flush_binary_output(io.binary_output_stream::in,
    io::di, io::uo) is det.

%---------------------------------------------------%
%
% Whole file input predicates.
%

    % Read all the characters (code points) from the current input stream
    % or from the specified stream, until eof or error.
    %
:- pred read_file(io.maybe_partial_res(list(char))::out,
    io::di, io::uo) is det.
:- pred read_file(io.text_input_stream::in,
    io.maybe_partial_res(list(char))::out, io::di, io::uo) is det.

    % Read all the characters (code points) from the current input stream
    % or from the specified stream, until eof or error. Returns the result
    % as a string rather than as a list of char.
    %
    % Returns an error if the file contains a null character, because
    % null characters are not allowed in Mercury strings.
    %
    % WARNING: the returned string is NOT guaranteed to be valid UTF-8
    % or UTF-16.
    %
```

```
:- pred read_file_as_string(io.maybe_partial_res(string)::out,
    io::di, io::uo) is det.
:- pred read_file_as_string(io.text_input_stream::in,
    io.maybe_partial_res(string)::out, io::di, io::uo) is det.

    % The same as read_file_as_string, but returns not only a string,
    % but also the number of code units in that string.
    %
    % WARNING: the returned string is NOT guaranteed to be valid UTF-8
    % or UTF-16.
    %
:- pred read_file_as_string_and_num_code_units(
    io.maybe_partial_res_2(string, int)::out, io::di, io::uo) is det.
:- pred read_file_as_string_and_num_code_units(io.text_input_stream::in,
    io.maybe_partial_res_2(string, int)::out, io::di, io::uo) is det.

    % Reads all the bytes until eof or error from the current binary input
    % stream or from the specified binary input stream.
    %
:- pred read_binary_file(
    io.result(list(int))::out, io::di, io::uo) is det.
:- pred read_binary_file(io.binary_input_stream::in,
    io.result(list(int))::out, io::di, io::uo) is det.

    % Reads all the bytes until eof or error from the current binary input
    % stream or from the specified binary input stream into a bitmap.
    %
:- pred read_binary_file_as_bitmap(
    io.res(bitmap)::out, io::di, io::uo) is det.
:- pred read_binary_file_as_bitmap(io.binary_input_stream::in,
    io.res(bitmap)::out, io::di, io::uo) is det.

%---------------------------------------------------%
%
% Processing the contents of a whole file.
%

    % Applies the given closure to each character (code point) read from
    % the input stream in turn, until eof or error.
    %
:- pred input_stream_foldl(pred(char, T, T), T, io.maybe_partial_res(T),
    io, io).
:- mode input_stream_foldl((pred(in, in, out) is det), in, out,
    di, uo) is det.
:- mode input_stream_foldl((pred(in, in, out) is cc_multi), in, out,
    di, uo) is cc_multi.
```

```
    % Applies the given closure to each character (code point) read from the
    % input stream in turn, until eof or error.
    %
:- pred input_stream_foldl(io.text_input_stream, pred(char, T, T),
    T, io.maybe_partial_res(T), io, io).
:- mode input_stream_foldl(in, in(pred(in, in, out) is det),
    in, out, di, uo) is det.
:- mode input_stream_foldl(in, in(pred(in, in, out) is cc_multi),
    in, out, di, uo) is cc_multi.

    % Applies the given closure to each character (code point) read from
    % the input stream in turn, until eof or error.
    %
:- pred input_stream_foldl_io(pred(char, io, io), io.res, io, io).
:- mode input_stream_foldl_io((pred(in, di, uo) is det), out, di, uo)
    is det.
:- mode input_stream_foldl_io((pred(in, di, uo) is cc_multi), out, di, uo)
    is cc_multi.

    % Applies the given closure to each character (code point) read from the
    % input stream in turn, until eof or error.
    %
:- pred input_stream_foldl_io(io.text_input_stream, pred(char, io, io),
    io.res, io, io).
:- mode input_stream_foldl_io(in, in(pred(in, di, uo) is det),
    out, di, uo) is det.
:- mode input_stream_foldl_io(in, in(pred(in, di, uo) is cc_multi),
    out, di, uo) is cc_multi.

    % Applies the given closure to each character (code point) read from
    % the input stream in turn, until eof or error.
    %
:- pred input_stream_foldl2_io(pred(char, T, T, io, io),
    T, io.maybe_partial_res(T), io, io).
:- mode input_stream_foldl2_io((pred(in, in, out, di, uo) is det),
    in, out, di, uo) is det.
:- mode input_stream_foldl2_io((pred(in, in, out, di, uo) is cc_multi),
    in, out, di, uo) is cc_multi.

    % Applies the given closure to each character (code point) read from the
    % input stream in turn, until eof or error.
    %
:- pred input_stream_foldl2_io(io.text_input_stream,
    pred(char, T, T, io, io),
    T, maybe_partial_res(T), io, io).
:- mode input_stream_foldl2_io(in,
    in(pred(in, in, out, di, uo) is det),
```

```
    in, out, di, uo) is det.
:- mode input_stream_foldl2_io(in,
    in(pred(in, in, out, di, uo) is cc_multi),
    in, out, di, uo) is cc_multi.

    % Applies the given closure to each character (code point) read from the
    % input stream in turn, until eof or error, or the closure returns 'no' as
    % its second argument.
    %
:- pred input_stream_foldl2_io_maybe_stop(
    pred(char, bool, T, T, io, io),
    T, io.maybe_partial_res(T), io, io).
:- mode input_stream_foldl2_io_maybe_stop(
    (pred(in, out, in, out, di, uo) is det),
    in, out, di, uo) is det.
:- mode input_stream_foldl2_io_maybe_stop(
    (pred(in, out, in, out, di, uo) is cc_multi),
    in, out, di, uo) is cc_multi.

    % Applies the given closure to each character (code point) read from the
    % input stream in turn, until eof or error, or the closure returns 'no' as
    % its second argument.
    %
:- pred input_stream_foldl2_io_maybe_stop(io.text_input_stream,
    pred(char, bool, T, T, io, io),
    T, maybe_partial_res(T), io, io).
:- mode input_stream_foldl2_io_maybe_stop(in,
    (pred(in, out, in, out, di, uo) is det),
    in, out, di, uo) is det.
:- mode input_stream_foldl2_io_maybe_stop(in,
    (pred(in, out, in, out, di, uo) is cc_multi),
    in, out, di, uo) is cc_multi.

%---------------------------------------------------%

    % Applies the given closure to each byte read from the current binary
    % input stream in turn, until eof or error.
    %
:- pred binary_input_stream_foldl(pred(int, T, T),
    T, maybe_partial_res(T), io, io).
:- mode binary_input_stream_foldl((pred(in, in, out) is det),
    in, out, di, uo) is det.
:- mode binary_input_stream_foldl((pred(in, in, out) is cc_multi),
    in, out, di, uo) is cc_multi.

    % Applies the given closure to each byte read from the given binary
    % input stream in turn, until eof or error.
```

```
    %
:- pred binary_input_stream_foldl(io.binary_input_stream,
    pred(int, T, T), T, maybe_partial_res(T), io, io).
:- mode binary_input_stream_foldl(in, in(pred(in, in, out) is det),
    in, out, di, uo) is det.
:- mode binary_input_stream_foldl(in, in(pred(in, in, out) is cc_multi),
    in, out, di, uo) is cc_multi.

    % Applies the given closure to each byte read from the current binary
    % input stream in turn, until eof or error.
    %
:- pred binary_input_stream_foldl_io(pred(int, io, io),
    io.res, io, io).
:- mode binary_input_stream_foldl_io((pred(in, di, uo) is det),
    out, di, uo) is det.
:- mode binary_input_stream_foldl_io((pred(in, di, uo) is cc_multi),
    out, di, uo) is cc_multi.

    % Applies the given closure to each byte read from the given binary
    % input stream in turn, until eof or error.
    %
:- pred binary_input_stream_foldl_io(io.binary_input_stream,
    pred(int, io, io), io.res, io, io).
:- mode binary_input_stream_foldl_io(in, in(pred(in, di, uo) is det),
    out, di, uo) is det.
:- mode binary_input_stream_foldl_io(in, in(pred(in, di, uo) is cc_multi),
    out, di, uo) is cc_multi.

    % Applies the given closure to each byte read from the current binary
    % input stream in turn, until eof or error.
    %
:- pred binary_input_stream_foldl2_io(
    pred(int, T, T, io, io), T, maybe_partial_res(T), io, io).
:- mode binary_input_stream_foldl2_io(
    in(pred(in, in, out, di, uo) is det), in, out, di, uo) is det.
:- mode binary_input_stream_foldl2_io(
    in(pred(in, in, out, di, uo) is cc_multi), in, out, di, uo) is cc_multi.

    % Applies the given closure to each byte read from the given binary
    % input stream in turn, until eof or error.
    %
:- pred binary_input_stream_foldl2_io(io.binary_input_stream,
    pred(int, T, T, io, io), T, maybe_partial_res(T), io, io).
:- mode binary_input_stream_foldl2_io(in,
    (pred(in, in, out, di, uo) is det), in, out, di, uo) is det.
:- mode binary_input_stream_foldl2_io(in,
    (pred(in, in, out, di, uo) is cc_multi), in, out, di, uo) is cc_multi.
```

```
    % Applies the given closure to each byte read from the current binary
    % input stream in turn, until eof or error, or the closure returns ‘no’
    % as its second argument.
    %
:- pred binary_input_stream_foldl2_io_maybe_stop(
    pred(int, bool, T, T, io, io), T, maybe_partial_res(T), io, io).
:- mode binary_input_stream_foldl2_io_maybe_stop(
    (pred(in, out, in, out, di, uo) is det), in, out, di, uo) is det.
:- mode binary_input_stream_foldl2_io_maybe_stop(
    (pred(in, out, in, out, di, uo) is cc_multi), in, out, di, uo) is cc_multi.

    % Applies the given closure to each byte read from the given binary input
    % stream in turn, until eof or error, or the closure returns ‘no’ as its
    % second argument.
    %
:- pred binary_input_stream_foldl2_io_maybe_stop(io.binary_input_stream,
    pred(int, bool, T, T, io, io), T, maybe_partial_res(T), io, io).
:- mode binary_input_stream_foldl2_io_maybe_stop(in,
    (pred(in, out, in, out, di, uo) is det), in, out, di, uo) is det.
:- mode binary_input_stream_foldl2_io_maybe_stop(in,
    (pred(in, out, in, out, di, uo) is cc_multi), in, out, di, uo) is cc_multi.

%---------------------------------------------------%
%
% File handling predicates.
%

    % remove_file(FileName, Result, !IO) attempts to remove the file
    % ‘FileName’, binding Result to ok/0 if it succeeds, or error/1 if it
    % fails. If ‘FileName’ names a file that is currently open, the behaviour
    % is implementation-dependent.
    %
:- pred remove_file(string::in, io.res::out, io::di, io::uo) is det.

    % remove_file_recursively(FileName, Result, !IO) attempts to remove
    % the file ‘FileName’, binding Result to ok/0 if it succeeds, or error/1
    % if it fails. If ‘FileName’ names a file that is currently open, the
    % behaviour is implementation-dependent.
    %
    % Unlike ‘remove_file’, this predicate will attempt to remove non-empty
    % directories (recursively). If it fails, some of the directory elements
    % may already have been removed.
    %
:- pred remove_file_recursively(string::in, io.res::out, io::di, io::uo)
    is det.
```

```
    % rename_file(OldFileName, NewFileName, Result, !IO).
    %
    % Attempts to rename the file `OldFileName' as `NewFileName', binding
    % Result to ok/0 if it succeeds, or error/1 if it fails. If `OldFileName'
    % names a file that is currently open, the behaviour is
    % implementation-dependent. If `NewFileName' names a file that already
    % exists the behaviour is also implementation-dependent; on some systems,
    % the file previously named `NewFileName' will be deleted and replaced
    % with the file previously named `OldFileName'.
    %
:- pred rename_file(string::in, string::in, io.res::out, io::di, io::uo)
    is det.

%---------------------------------------------------%

    % Succeeds if this platform can read and create symbolic links.
    %
:- pred have_symlinks is semidet.

    % make_symlink(FileName, LinkFileName, Result, !IO).
    %
    % Attempts to make `LinkFileName' be a symbolic link to `FileName'.
    % If `FileName' is a relative path, it is interpreted relative
    % to the directory containing `LinkFileName'.
    %
:- pred make_symlink(string::in, string::in, io.res::out, io::di, io::uo)
    is det.

    % read_symlink(FileName, Result, !IO) returns `ok(LinkTarget)'
    % if `FileName' is a symbolic link pointing to `LinkTarget', and
    % `error(Error)' otherwise. If `LinkTarget' is a relative path,
    % it should be interpreted relative the directory containing `FileName',
    % not the current directory.
    %
:- pred read_symlink(string::in, io.res(string)::out, io::di, io::uo) is det.

%---------------------------------------------------%

:- type access_type
    --->    read
    ;       write
    ;       execute.

    % check_file_accessibility(FileName, AccessTypes, Result):
    %
    % Check whether the current process can perform the operations given
    % in `AccessTypes' on `FileName'.
```

```
    % XXX When using the .NET CLI, this predicate will sometimes report
    % that a directory is writable when in fact it is not.
    % XXX On the Erlang backend, or on Windows with some compilers, 'execute'
    % access is not checked.
    %
:- pred check_file_accessibility(string::in, list(access_type)::in,
    io.res::out, io::di, io::uo) is det.

:- type file_type
    --->    regular_file
    ;       directory
    ;       symbolic_link
    ;       named_pipe
    ;       socket
    ;       character_device
    ;       block_device
    ;       message_queue
    ;       semaphore
    ;       shared_memory
    ;       unknown.

    % file_type(FollowSymLinks, FileName, TypeResult)
    % finds the type of the given file.
    %
:- pred file_type(bool::in, string::in, io.res(file_type)::out,
    io::di, io::uo) is det.

    % file_modification_time(FileName, TimeResult)
    % finds the last modification time of the given file.
    %
:- pred file_modification_time(string::in, io.res(time_t)::out,
    io::di, io::uo) is det.

%---------------------------------------------------%
%
% Predicates for handling temporary files.
%

    % make_temp_file(Result, !IO) creates an empty file whose name is different
    % to the name of any existing file. If successful Result returns the name
    % of the file. It is the responsibility of the caller to delete the file
    % when it is no longer required.
    %
    % The file is placed in the directory returned by get_temp_directory/3.
    %
    % On the Erlang and Java backends, this does not attempt to create the file
    % with restrictive permissions (600 on Unix-like systems) and therefore
```

```
    % should not be used when security is required.
    %
:- pred make_temp_file(io.res(string)::out, io::di, io::uo) is det.

    % Like make_temp_file/3 except it throws an io.error exception if the
    % temporary file could not be created.
    %
:- pragma obsolete(make_temp/3).
:- pred make_temp(string::out, io::di, io::uo) is det.

    % make_temp_file(Dir, Prefix, Suffix, Result, !IO) creates an empty file
    % whose name is different to the name of any existing file. The file will
    % reside in the directory specified by Dir and will have a prefix us-
ing up
    % to the first 5 code units of Prefix. If successful, Result returns the
    % name of the file. It is the responsibility of the caller to delete the
    % file when it is no longer required.
    %
    % The reason for truncating Prefix is historical; in future the behaviour
    % may be changed. Note that the truncation is performed without re-
gard for
    % code point boundaries. It is recommended to use only (printable) ASCII
    % characters in the prefix string.
    %
    % The C backend has the following limitations:
    %   - Suffix may be ignored.
    %
    % The C# backend has the following limitations:
    %   - Dir is ignored.
    %   - Prefix is ignored.
    %   - Suffix is ignored.
    %
    % On the Erlang backend Suffix is ignored.
    %
    % On the Erlang and Java backends, this does not attempt to create the file
    % with restrictive permissions (600 on Unix-like systems) and therefore
    % should not be used when security is required.
    %
:- pred make_temp_file(string::in, string::in, string::in, io.res(string)::out,
    io::di, io::uo) is det.

    % Same as make_temp_file except it does not take a suffix argument and
    % throws an io.error exception if the temporary file could not be created.
    %
:- pragma obsolete(make_temp/5).
:- pred make_temp(string::in, string::in, string::out, io::di, io::uo) is det.
```

```
    % make_temp_directory(Result, !IO) creates an empty directory whose name
    % is different from the name of any existing directory.
    %
    % On the Java backend this is insecure as the file permissions are not set.
    %
    % This is unimplemented on the Erlang backend.
    %
:- pred make_temp_directory(io.res(string)::out, io::di, io::uo) is det.

    % make_temp_directory(Dir, Prefix, Suffix, Result, !IO) creates an empty
    % directory whose name is different from the name of any existing
    % directory. The new directory will reside in the existing directory
    % specified by Dir and will have a prefix using up to the first 5
    % characters of Prefix and a Suffix. Result returns the name of the
    % new directory. It is the responsibility of the program to delete the
    % directory when it is no longer needed.
    %
    % The C backend has the following limitations:
    %   - Suffix is ignored.
    %
    % The C# backend has the following limitations:
    %   - Prefix is ignored.
    %   - Suffix is ignored.
    %
    % On the Java backend this is insecure as the file permissions are not set.
    %
    % This is unimplemented on the Erlang backend.
    %
:- pred make_temp_directory(string::in, string::in, string::in,
    io.res(string)::out, io::di, io::uo) is det.

    % Test if the make_temp_directory predicates are available. This is false
    % for the Erlang backends and either C backend without support for
    % mkdtemp(3).
    %
:- pred have_make_temp_directory is semidet.

    % get_temp_directory(DirName, !IO)
    %
    % DirName is the name of a directory where applications should put
    % temporary files.
    %
    % This is implementation-dependent. For current Mercury implementations,
    % it is determined as follows:
    % 1. For the non-Java back-ends:
    %    - On Microsoft Windows systems, the file will reside in
    %      the current directory if the TMP environment variable
```

```
    %       is not set, or in the directory specified by TMP if it is set.
    %     - On Unix systems, the file will reside in /tmp if the TMPDIR
    %       environment variable is not set, or in the directory specified
    %       by TMPDIR if it is set.
    % 2. For the Java back-end, the system-dependent default
    %    temporary-file directory will be used, specified by the Java
    %    system property java.io.tmpdir. On UNIX systems the default
    %    value of this property is typically "/tmp" or "/var/tmp";
    %    on Microsoft Windows systems it is typically "c:\\temp".
    %
:- pred get_temp_directory(string::out, io::di, io::uo) is det.

%---------------------------------------------------%
%
% Global state predicates.
%

    % progname(DefaultProgname, Progname).
    %
    % Returns the name that the program was invoked with, if available,
    % or DefaultProgname if the name is not available.
    % Does not modify the I/O state.
    %
:- pred progname(string::in, string::out, io::di, io::uo) is det.

    % progname_base(DefaultProgname, Progname).
    %
    % Like 'progname', except that it strips off any path name
    % preceding the program name. Useful for error messages.
    %
:- pred progname_base(string::in, string::out, io::di, io::uo) is det.

    % Returns the arguments that the program was invoked with,
    % if available, otherwise an empty list. Does not modify the I/O state.
    %
:- pred command_line_arguments(list(string)::out, io::di, io::uo) is det.

%---------------------------------------------------%

    % The I/O state contains an integer used to record the program's exit
    % status. When the program finishes, it will return this exit status
    % to the operating system. The following predicates can be used to get
    % and set the exit status.
    %
:- pred get_exit_status(int::out, io::di, io::uo) is det.
:- pred set_exit_status(int::in, io::di, io::uo) is det.
```

```
%---------------------------------------------------%

    % The following predicates provide an interface to the environment list.
    % Do not attempt to put spaces or '=' signs in the names of environment
    % variables, or bad things may result!
    %
    % First argument is the name of the environment variable. Returns
    % yes(Value) if the variable was set (Value will be set to the value
    % of the variable) and no if the variable was not set.
    %
:- pred get_environment_var(string::in, maybe(string)::out,
    io::di, io::uo) is det.

    % First argument is the name of the environment variable, second argument
    % is the value to be assigned to that variable. Res is 'ok' on suc-
cess or
    % 'error(ErrorCode)' if the system runs out of environment space or if
    % the environment cannot be modified.
    %
    % Note that the environment cannot be modified on Java.
    %
:- pred set_environment_var(string::in, string::in, io.res::out,
    io::di, io::uo) is det.

    % Same as set_environment_var/5, but throws an exception if an error
    % occurs.
    %
:- pred set_environment_var(string::in, string::in, io::di, io::uo) is det.

    % Test if the set_environment_var/{4,5} predicates are available.
    % This is false for Java backends.
    %
:- pred have_set_environment_var is semidet.

%---------------------------------------------------%
%
% System access predicates.
%

    % Invokes the operating system shell with the specified Command.
    % Result is either `ok(ExitStatus)', if it was possible to invoke
    % the command, or `error(ErrorCode)' if not. The ExitStatus will be 0
    % if the command completed successfully or the return value of the system
    % call. If a signal kills the system call, then Result will be an error
    % indicating which signal occurred.
    %
:- pred call_system(string::in, io.res(int)::out, io::di, io::uo) is det.
```

```
:- type system_result
    --->    exited(int)
    ;       signalled(int).

    % call_system_return_signal(Command, Result, !IO):
    %
    % Invokes the operating system shell with the specified Command.
    % Result is either `ok(ExitStatus)' if it was possible to invoke
    % the command or `error(Error)' if the command could not be executed.
    % If the command could be executed then ExitStatus is either
    % `exited(ExitCode)' if the command ran to completion or
    % `signalled(SignalNum)' if the command was killed by a signal.
    % If the command ran to completion then ExitCode will be 0 if the command
    % ran successfully and the return value of the command otherwise.
    %
:- pred call_system_return_signal(string::in,
    io.res(system_result)::out, io::di, io::uo) is det.

%---------------------------------------------------%
%
% Managing the globals structure that Mercury attaches to the I/O state.
%

    % The I/O state includes a `globals' field which is not used by the
    % standard library, but can be used by the application. The globals field
    % is of type `univ' so that the application can store any data it wants
    % there. The following predicates can be used to access this global state.
    %
    % Does not modify the I/O state.
    %
:- pred get_globals(univ::out, io::di, io::uo) is det.
:- pred set_globals(univ::in, io::di, io::uo) is det.

    % update_globals(UpdatePred, !IO).
    % Update the `globals' field in the I/O state based upon its current value.
    % This is equivalent to the following:
    %
    %   get_globals(Globals0, !IO),
    %   UpdatePred(Globals0, Globals),
    %   set_globals(Globals, !IO)
    %
    % In parallel grades calls to update_globals/3 are atomic.
    % If `UpdatePred' throws an exception then the `globals' field is
    % left unchanged.
    %
:- pred update_globals(pred(univ, univ)::in(pred(in, out) is det),
```

```
    io::di, io::uo) is det.

%---------------------------------------------------%
%
% Predicates that report statistics about the current program execution.
%

    % Write memory/time usage statistics to stderr.
    %
:- pred report_stats(io::di, io::uo) is det.

    % Write statistics to stderr; what statistics will be written
    % is controlled by the first argument, which acts a selector.
    % What selector values cause what statistics to be printed
    % is implementation defined.
    %
    % The Melbourne implementation supports the following selectors:
    %
    % "standard"
    %   Writes memory/time usage statistics.
    %
    % "full_memory_stats"
    %   Writes complete memory usage statistics, including information
    %   about all procedures and types. Requires compilation with memory
    %   profiling enabled.
    %
    % "tabling"
    %   Writes statistics about the internals of the tabling system.
    %   Requires the runtime to have been compiled with the macro
    %   MR_TABLE_STATISTICS defined.
    %
:- pred report_stats(string::in, io::di, io::uo) is det.

%---------------------------------------------------%
%
% Interpreting I/O error messages.
%

    % Construct an error code including the specified error message.
    %
:- func make_io_error(string) = io.error.

    % Look up the error message corresponding to a particular error code.
    %
:- func error_message(io.error) = string.
:- pred error_message(io.error::in, string::out) is det.
```

```
%-----------------------------------------------%
%
% Instances of the stream typeclasses.
%

:- instance stream.error(io.error).

:- instance stream.stream(text_output_stream, io).
:- instance stream.output(text_output_stream, io).
:- instance stream.writer(text_output_stream, char,   io).
:- instance stream.writer(text_output_stream, float,  io).
:- instance stream.writer(text_output_stream, int,    io).
:- instance stream.writer(text_output_stream, int8,   io).
:- instance stream.writer(text_output_stream, int16,  io).
:- instance stream.writer(text_output_stream, int32,  io).
:- instance stream.writer(text_output_stream, uint,   io).
:- instance stream.writer(text_output_stream, uint8,  io).
:- instance stream.writer(text_output_stream, uint16, io).
:- instance stream.writer(text_output_stream, uint8,  io).
:- instance stream.writer(text_output_stream, string, io).
:- instance stream.writer(text_output_stream, univ,   io).
:- instance stream.line_oriented(text_output_stream, io).

:- instance stream.stream(text_input_stream, io).
:- instance stream.input(text_input_stream, io).
:- instance stream.reader(text_input_stream, char, io, io.error).
:- instance stream.unboxed_reader(text_input_stream, char, io, io.error).
:- instance stream.reader(text_input_stream, line, io, io.error).
:- instance stream.reader(text_input_stream, text_file, io, io.error).

:- instance stream.line_oriented(text_input_stream, io).
:- instance stream.putback(text_input_stream, char, io, io.error).

:- instance stream.stream(binary_output_stream, io).
:- instance stream.output(binary_output_stream, io).
:- instance stream.writer(binary_output_stream, byte, io).
:- instance stream.writer(binary_output_stream, int8, io).
:- instance stream.writer(binary_output_stream, uint8, io).
:- instance stream.writer(binary_output_stream, bitmap.slice, io).
:- instance stream.seekable(binary_output_stream, io).

:- instance stream.stream(binary_input_stream,  io).
:- instance stream.input(binary_input_stream, io).
:- instance stream.reader(binary_input_stream, int, io, io.error).
:- instance stream.reader(binary_input_stream, int8, io, io.error).
:- instance stream.reader(binary_input_stream, uint8, io, io.error).
:- instance stream.bulk_reader(binary_input_stream, int, bitmap, io, io.error).
```

```
:- instance stream.putback(binary_input_stream, int, io, io.error).
:- instance stream.putback(binary_input_stream, int8, io, io.error).
:- instance stream.putback(binary_input_stream, uint8, io, io.error).
:- instance stream.seekable(binary_input_stream, io).

%---------------------------------------------------%
%---------------------------------------------------%
```

41 kv_list

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 2020 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: kv_list.m.
% Main author: zs.
% Stability: medium to high.
%
% This file defines the type kv_list(K, V), which represents lists of
% key-value pairs, and provides a range of operations on such lists.
%
% The kv_list module resembles the assoc_list module quite closely.
% The data type it defines stores the same information, and the set
% of operations they provide is the same, modulo the fact that the
% operations that convert between the two representations are here.
% The difference is that kv_list uses one memory cell, not two,
% to represent one key-value pair, which should mean that it requires
% fewer memory allocations. On the other hand, values of type assoc_list
% may be operated on as plain lists, while this cannot be done on values
% of type kv_list.
%
%---------------------------------------------------%
%---------------------------------------------------%

:- module kv_list.
:- interface.

:- import_module assoc_list.
:- import_module list.

%---------------------------------------------------%
```

```
:- type kv_list(K, V)
    --->    kv_nil
    ;       kv_cons(K, V, kv_list(K, V)).

%-----------------------------------------------------%
%
% Creating kv_lists from lists of keys and values.
%

    % Zip together a list of keys and a list of values.
    % Throw an exception if they are of different lengths.
    %
:- func from_corresponding_lists(list(K), list(V)) = kv_list(K, V).
:- pred from_corresponding_lists(list(K)::in, list(V)::in,
    kv_list(K, V)::out) is det.

%-----------------------------------------------------%
%
% Conversion to and from assoc_lists.
%

:- func assoc_list_to_kv_list(assoc_list(K, V)) = kv_list(K, V).
:- func kv_list_to_assoc_list(kv_list(K, V)) = assoc_list(K, V).

%-----------------------------------------------------%
%
% Operations on lists of keys and/or values.
%

    % Swap the two sides of the pairs in each member of the list.
    %
:- func reverse_members(kv_list(K, V)) = kv_list(V, K).
:- pred reverse_members(kv_list(K, V)::in, kv_list(V, K)::out) is det.

    % Return the first member of each pair.
    %
:- func keys(kv_list(K, V)) = list(K).
:- pred keys(kv_list(K, V)::in, list(K)::out) is det.

    % Return the second member of each pair.
    %
:- func values(kv_list(K, V)) = list(V).
:- pred values(kv_list(K, V)::in, list(V)::out) is det.

    % Return two lists containing respectively the first and the second member
    % of each pair in the kv_list.
    %
```

```
:- pred keys_and_values(kv_list(K, V)::in, list(K)::out, list(V)::out) is det.

%-----------------------------------------------%
%
% Searching kv_lists.
%

    % Find the first element of the kv_list list that matches
    % the given key, and return the associated value.
    % Fail if there is no matching key.
    %
:- pred search(kv_list(K, V)::in, K::in, V::out) is semidet.

    % Find the first element of the kv_list list that matches
    % the given key, and return the associated value.
    % Throw an exception if there is no matching key.
    %
:- pred lookup(kv_list(K, V)::in, K::in, V::out) is det.

    % A field access version of search.
    %
:- func kv_list(K, V) ^ elem(K) = V is semidet.

    % A field access version of lookup.
    %
:- func kv_list(K, V) ^ det_elem(K) = V is det.

%-----------------------------------------------%
%
% Updating elements in kv_lists.
%

    % Find the first element of the kv_list list that matches
    % the given key, and update the associated value.
    % Fail if there is no matching key.
    %
:- pred update(K::in, V::in, kv_list(K, V)::in, kv_list(K, V)::out)
    is semidet.

%-----------------------------------------------%
%
% Removing elements from kv_lists.
%

    % Find the first element of the association list that matches the given
    % key. Return the associated value, and the original list with the selected
    % element removed.
```

```
    %
:- pred remove(kv_list(K, V)::in, K::in, V::out, kv_list(K, V)::out)
    is semidet.

    % As above, but with an argument ordering that is more conducive to
    % the use of state variable notation.
    %
:- pred svremove(K::in, V::out, kv_list(K, V)::in, kv_list(K, V)::out)
    is semidet.

%-----------------------------------------------------%
%
% Mapping keys or values.
%

:- func map_keys_only(func(K) = L, kv_list(K, V)) = kv_list(L, V).
:- pred map_keys_only(pred(K, L), kv_list(K, V), kv_list(L, V)).
:- mode map_keys_only(pred(in, out) is det, in, out) is det.

:- func map_values_only(func(V) = W, kv_list(K, V)) = kv_list(K, W).
:- pred map_values_only(pred(V, W), kv_list(K, V), kv_list(K, W)).
:- mode map_values_only(pred(in, out) is det, in, out) is det.

:- func map_values(func(K, V) = W, kv_list(K, V)) = kv_list(K, W).
:- pred map_values(pred(K, V, W), kv_list(K, V), kv_list(K, W)).
:- mode map_values(pred(in, in, out) is det, in, out) is det.

%-----------------------------------------------------%
%
% Filtering elements in kv_lists.
%

    % filter(Pred, List, TrueList) takes a closure with one input argument,
    % and for each key-value pair in List, calls the closure on the key K.
    % The key-value pair is included in TrueList iff Pred(K) is true.
    %
:- func filter(pred(K)::in(pred(in) is semidet),
    kv_list(K, V)::in) = (kv_list(K, V)::out) is det.
:- pred filter(pred(K)::in(pred(in) is semidet),
    kv_list(K, V)::in, kv_list(K, V)::out) is det.

    % negated_filter(Pred, List, FalseList) takes a closure with one
    % input argument, and for each key-value pair in List, calls the closure
    % on the key K. The key-value pair is included in TrueList iff Pred(K)
    % is false.
    %
:- func negated_filter(pred(K)::in(pred(in) is semidet),
```

```
    kv_list(K, V)::in) = (kv_list(K, V)::out) is det.
:- pred negated_filter(pred(K)::in(pred(in) is semidet),
    kv_list(K, V)::in, kv_list(K, V)::out) is det.

    % filter(Pred, List, TrueList, FalseList) takes a closure with
    % one input argument, and for each key-value pair in List,
    % calls the closure on the key K. If Pred(K) is true, the key-value pair
    % is included in TrueList; otherwise, it is included in FalseList.
    %
:- pred filter(pred(K)::in(pred(in) is semidet),
    kv_list(K, V)::in, kv_list(K, V)::out, kv_list(K, V)::out) is det.

%-----------------------------------------------%
%
% Merging kv_lists.
%

    % merge(L1, L2, L):
    %
    % L is the result of merging the elements of L1 and L2, in ascending order.
    % L1 and L2 must be sorted on the keys.
    %
:- func merge(kv_list(K, V), kv_list(K, V)) = kv_list(K, V).
:- pred merge(kv_list(K, V)::in, kv_list(K, V)::in,
    kv_list(K, V)::out) is det.

%-----------------------------------------------%
%
% Folding over kv_lists.
%

    % foldl(Func, List, Start) = End calls Func
    % with each key-value in List, working left-to-right,
    % and an accumulator whose initial value is Start,
    % and returns the final value in End.
    %
:- func foldl(func(K, V, A) = A, kv_list(K, V), A) = A.

    % foldl(Pred, List, Start End) calls Pred
    % with each key-value pair in List, working left-to-right,
    % and an accumulator whose initial value is Start,
    % and returns the final value in End.
    %
:- pred foldl(pred(K, V, A, A), kv_list(K, V), A, A).
:- mode foldl(pred(in, in, in, out) is det, in, in, out) is det.
:- mode foldl(pred(in, in, mdi, muo) is det, in, mdi, muo) is det.
:- mode foldl(pred(in, in, di, uo) is det, in, di, uo) is det.
```

```
:- mode foldl(pred(in, in, in, out) is semidet, in, in, out) is semidet.
:- mode foldl(pred(in, in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode foldl(pred(in, in, di, uo) is semidet, in, di, uo) is semidet.
:- mode foldl(pred(in, in, in, out) is nondet, in, in, out) is nondet.

    % foldl_keys(Func List, Start) = End calls Func
    % with each key in List, working left-to-right, and an accumulator
    % whose initial value is Start, and returns the final value in End.
    %
:- func foldl_keys(func(K, A) = A, kv_list(K, V), A) = A.

    % foldl_keys(Pred, List, Start End) calls Pred
    % with each key in List, working left-to-right, and an accumulator
    % whose initial value is Start, and returns the final value in End.
    %
:- pred foldl_keys(pred(K, A, A), kv_list(K, V), A, A).
:- mode foldl_keys(pred(in, in, out) is det, in, in, out) is det.
:- mode foldl_keys(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode foldl_keys(pred(in, di, uo) is det, in, di, uo) is det.
:- mode foldl_keys(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode foldl_keys(pred(in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode foldl_keys(pred(in, di, uo) is semidet, in, di, uo) is semidet.
:- mode foldl_keys(pred(in, in, out) is multi, in, in, out) is multi.
:- mode foldl_keys(pred(in, in, out) is nondet, in, in, out) is nondet.

    % foldl_values(Func List, Start) = End calls Func
    % with each value in List, working left-to-right, and an accumulator
    % whose initial value is Start, and returns the final value in End.
    %
:- func foldl_values(func(V, A) = A, kv_list(K, V), A) = A.

    % foldl_values(Pred, List, Start End) calls Pred
    % with each value in List, working left-to-right, and an accumulator
    % whose initial value is Start, and returns the final value in End.
    %
:- pred foldl_values(pred(V, A, A), kv_list(K, V), A, A).
:- mode foldl_values(pred(in, in, out) is det, in,
    in, out) is det.
:- mode foldl_values(pred(in, mdi, muo) is det, in,
    mdi, muo) is det.
:- mode foldl_values(pred(in, di, uo) is det, in,
    di, uo) is det.
:- mode foldl_values(pred(in, in, out) is semidet, in,
    in, out) is semidet.
:- mode foldl_values(pred(in, mdi, muo) is semidet, in,
    mdi, muo) is semidet.
:- mode foldl_values(pred(in, di, uo) is semidet, in,
```

```
    di, uo) is semidet.
:- mode foldl_values(pred(in, in, out) is multi, in,
    in, out) is multi.
:- mode foldl_values(pred(in, in, out) is nondet, in,
    in, out) is nondet.

    % As foldl, but with two accumulators.
    %
:- pred foldl2(pred(K, V, A, A, B, B), kv_list(K, V), A, A, B, B).
:- mode foldl2(pred(in, in, in, out, in, out) is det, in, in, out,
    in, out) is det.
:- mode foldl2(pred(in, in, in, out, mdi, muo) is det, in, in, out,
    mdi, muo) is det.
:- mode foldl2(pred(in, in, in, out, di, uo) is det, in, in, out,
    di, uo) is det.
:- mode foldl2(pred(in, in, in, out, in, out) is semidet, in, in, out,
    in, out) is semidet.
:- mode foldl2(pred(in, in, in, out, mdi, muo) is semidet, in,in, out,
    mdi, muo) is semidet.
:- mode foldl2(pred(in, in, in, out, di, uo) is semidet, in, in, out,
    di, uo) is semidet.
:- mode foldl2(pred(in, in, in, out, in, out) is nondet, in, in, out,
    in, out) is nondet.

    % As foldl_values, but with two accumulators.
    %
:- pred foldl2_values(pred(V, A, A, B, B), kv_list(K, V),
    A, A, B, B).
:- mode foldl2_values(pred(in, in, out, in, out) is det, in,
    in, out, in, out) is det.
:- mode foldl2_values(pred(in, in, out, mdi, muo) is det, in,
    in, out, mdi, muo) is det.
:- mode foldl2_values(pred(in, in, out, di, uo) is det, in,
    in, out, di, uo) is det.
:- mode foldl2_values(pred(in, in, out, in, out) is semidet, in,
    in, out, in, out) is semidet.
:- mode foldl2_values(pred(in, in, out, mdi, muo) is semidet, in,
    in, out, mdi, muo) is semidet.
:- mode foldl2_values(pred(in, in, out, di, uo) is semidet, in,
    in, out, di, uo) is semidet.
:- mode foldl2_values(pred(in, in, out, in, out) is multi, in,
    in, out, in, out) is multi.
:- mode foldl2_values(pred(in, in, out, in, out) is nondet, in,
    in, out, in, out) is nondet.

    % As foldl, but with three accumulators.
    %
```

```
:- pred foldl3(pred(K, V, A, A, B, B, C, C), kv_list(K, V),
    A, A, B, B, C, C).
:- mode foldl3(pred(in, in, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out) is det.
:- mode foldl3(pred(in, in, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, mdi, muo) is det.
:- mode foldl3(pred(in, in, in, out, in, out, di, uo) is det, in,
    in, out, in, out, di, uo) is det.
:- mode foldl3(pred(in, in, in, out, in, out, in, out) is semidet, in,
    in, out, in, out, in, out) is semidet.
:- mode foldl3(pred(in, in, in, out, in, out, mdi, muo) is semidet, in,
        in, out, in, out, mdi, muo) is semidet.
:- mode foldl3(pred(in, in, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, di, uo) is semidet.
:- mode foldl3(pred(in, in, in, out, in, out, in, out) is nondet, in,
    in, out, in, out, in, out) is nondet.

    % As foldl_values, but with three accumulators.
    %
:- pred foldl3_values(pred(V, A, A, B, B, C, C), kv_list(K, V),
    A, A, B, B, C, C).
:- mode foldl3_values(pred(in, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out) is det.
:- mode foldl3_values(pred(in, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, mdi, muo) is det.
:- mode foldl3_values(pred(in, in, out, in, out, di, uo) is det,
    in, in, out, in, out, di, uo) is det.
:- mode foldl3_values(pred(in, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out) is semidet.
:- mode foldl3_values(pred(in, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, mdi, muo) is semidet.
:- mode foldl3_values(pred(in, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, di, uo) is semidet.
:- mode foldl3_values(pred(in, in, out, in, out, in, out) is multi,
    in, in, out, in, out, in, out) is multi.
:- mode foldl3_values(pred(in, in, out, in, out, in, out) is nondet,
    in, in, out, in, out, in, out) is nondet.

%---------------------------------------------------%
%---------------------------------------------------%
```

42 lazy

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
```

```
%---------------------------------------------------%
% Copyright (C) 1999, 2006, 2009-2010 The University of Melbourne.
% Copyright (C) 2013-2016, 2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: lazy.m.
% Main authors: fjh, pbone.
% Stability: medium.
%
% Provides support for optional explicit lazy evaluation.
%
% This module provides the data type `lazy(T)' and the functions `val',
% `delay', and `force', which can be used to emulate lazy evaluation.
%
% A field within a data structure can be made lazy by wrapping it within a lazy
% type. Or a lazy data structure can be implemented, for example:
%
% :- type lazy_list(T)
%     --->    lazy_list(
%                 lazy(list_cell(T))
%             ).
%
% :- type list_cell(T)
%     --->    cons(T, lazy_list(T))
%     ;       nil.
%
% Note that this makes every list cell lazy, whereas:
%
%   lazy(list(T))
%
% uses only one thunk for the entire list. And:
%
%   list(lazy(T))
%
% uses one thunk for every element, but the list's structure is not lazy.
%
%---------------------------------------------------%

:- module lazy.
:- interface.

    % A `lazy(T)' is a value of type `T' which will only be evaluated on
    % demand.
    %
:- type lazy(T).
```

```
    % Convert a value from type `T' to `lazy(T)'.
    %
:- func val(T) = lazy(T).

    % Construct a lazily-evaluated `lazy(T)' from a closure.
    %
:- func delay((func) = T) = lazy(T).

    % Force the evaluation of a `lazy(T)', and return the result as type `T'.
    % Note that if the type `T' may itself contain subterms of type `lazy(T)',
    % as is the case when `T' is a recursive type, those subterms will not be
    % evaluated -- `force/1' only forces evaluation of the `lazy/1' term at
    % the top level.
    %
    % A second call to `force' will not re-evaluate the lazy expression, it
    % will simply return `T'.
    %
:- func force(lazy(T)) = T.

    % Get the value of a lazy expression if it has already been made available
    % with `force/1'. This is useful as it can provide information without
    % incurring (much) cost.
    %
:- impure pred read_if_val(lazy(T)::in, T::out) is semidet.

    % Test lazy values for equality.
    %
:- pred equal_values(lazy(T)::in, lazy(T)::in) is semidet.

:- pred compare_values(comparison_result::uo, lazy(T)::in, lazy(T)::in) is det.

%-----------------------------------------------------%
%
% The declarative semantics of the above constructs are given by the
% following equations:
%
%   val(X) = delay((func) = X).
%
%   force(delay(F)) = apply(F).
%
% The operational semantics satisfy the following:
%
% - `val/1' and `delay/1' both take O(1) time and use O(1) additional space.
%   In particular, `delay/1' does not evaluate its argument using `apply/1'.
%
% - When `force/1' is first called for a given term, it uses `apply/1' to
%   evaluate the term, and then saves the result computed by destructively
```

```
%   modifying its argument; subsequent calls to ‘force/1’ on the same term
%   will return the same result.  So the time to evaluate ‘force(X)’, where
%   ‘X = delay(F)’, is O(the time to evaluate ‘apply(F)’) for the first call,
%   and O(1) time for subsequent calls.
%
% - Equality on values of type ‘lazy(T)’ is implemented by calling ‘force/1’
%   on both arguments and comparing the results.  So if ‘X’ and ‘Y’ have type
%   ‘lazy(T)’, and both ‘X’ and ‘Y’ are ground, then the time to evaluate
%   ‘X = Y’ is O(the time to evaluate ‘X1 = force(X)’ + the time to evaluate
%   ‘Y1 = force(Y)’ + the time to unify ‘X1’ and ‘Y1’).
%
%-----------------------------------------------------%
%-----------------------------------------------------%
```

43 lexer

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 1993-2000, 2003-2008, 2011-2012 The University of Melbourne.
% Copyright (C) 2014-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: lexer.m.
% Main author: fjh.
% Stability: high.
%
% Lexical analysis. This module defines the representation of tokens
% and exports predicates for reading in tokens from an input stream.
%
% See ISO Prolog 6.4. Also see the comments at the top of parser.m.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module lexer.
:- interface.

:- import_module char.
:- import_module io.
:- import_module integer.

%-----------------------------------------------------%
```

```
:- type token
    --->    name(string)
    ;       variable(string)
    ;       integer(integer_base, integer, signedness, integer_size)
    ;       float(float)
    ;       string(string)      % "...."
    ;       implementation_defined(string) % $name
    ;       open                % '('
    ;       open_ct             % '(' without any preceding whitespace
    ;       close               % ')'
    ;       open_list           % '['
    ;       close_list          % ']'
    ;       open_curly          % '{'
    ;       close_curly         % '}'
    ;       ht_sep              % '|'
    ;       comma               % ','
    ;       end                 % '.'
    ;       junk(char)          % junk character in the input stream
    ;       error(string)       % some other invalid token
    ;       io_error(io.error)  % error reading from the input stream
    ;       eof                 % end-of-file

    ;       integer_dot(integer).
            % The lexer will never return integer_dot. This token is used
            % internally in the lexer, to keep the grammar LL(1) so that
            % only one character of pushback is needed. But the lexer will
            % convert integer_dot/1 tokens to integer/1 tokens before
            % returning them.

:- type integer_base
    --->    base_2
    ;       base_8
    ;       base_10
    ;       base_16.

:- type signedness
    --->    signed
    ;       unsigned.

:- type integer_size
    --->    size_word
    ;       size_8_bit
    ;       size_16_bit
    ;       size_32_bit
    ;       size_64_bit.

    % For every token, we record the line number of the line on
```

```
    % which the token occurred.
    %
:- type token_context == int.   % line number

    % This "fat list" representation is more efficient than a list of pairs.
    %
:- type token_list
    --->    token_cons(token, token_context, token_list)
    ;       token_nil.

    % A line_context and a line_posn together contain exactly the same
    % fields as a posn, with the same semantics. The difference is that
    % stepping past a single character requires no memory allocation
    % whatsoever *unless* that character is a newline.
    %
    % XXX We should consider making both fields of line_context into uint32s,
    % to allow them to fit into a single 64 bit word. Simplicity would then
    % require line_posn's argumeny being a uint32 as well.

:- type line_context
    --->    line_context(
                line_context_current_line_number        :: int,
                line_context_offset_of_start_of_line    :: int
            ).

:- type line_posn
    --->    line_posn(
                line_posn_current_offset_in_file        :: int
            ).

    % Read a list of tokens either from the current input stream
    % or from the specified input stream.
    % Keep reading until we encounter either an 'end' token
    % (i.e. a full stop followed by whitespace) or the end-of-file.
    %
    % See 'char.is_whitespace' for the definition of whitespace characters
    % used by this predicate.
    %
:- pred get_token_list(token_list::out, io::di, io::uo) is det.
:- pred get_token_list(io.text_input_stream::in, token_list::out,
    io::di, io::uo) is det.

    % The type 'offset' represents a (zero-based) offset into a string.
    %
:- type offset == int.

    % string_get_token_list_max(String, MaxOffset, Tokens,
```

```
    %   InitialPos, FinalPos):
    %
    % Scan a list of tokens from a string, starting at the current offset
    % specified by InitialPos. Keep scanning until either we encounter either
    % an 'end' token (i.e. a full stop followed by whitespace) or until we
    % reach MaxOffset. (MaxOffset must be =< the length of the string.)
    % Return the tokens scanned in Tokens, and return the position one
    % character past the end of the last token in FinalPos.
    %
    % See 'char.is_whitespace' for the definition of whitespace characters
    % used by this predicate.
    %
:- pred string_get_token_list_max(string::in, offset::in, token_list::out,
    posn::in, posn::out) is det.

:- pred linestr_get_token_list_max(string::in, offset::in, token_list::out,
    line_context::in, line_context::out, line_posn::in, line_posn::out) is det.

    % string_get_token_list(String, Tokens, InitialPos, FinalPos):
    %
    % calls string_get_token_list_max above with MaxPos = length of String.
    %
:- pred string_get_token_list(string::in, token_list::out,
    posn::in, posn::out) is det.

    % Convert a token to a human-readable string describing the token.
    %
:- pred token_to_string(token::in, string::out) is det.

%---------------------------------------------------%
%---------------------------------------------------%
```

44 library

```
%---------------------------------------------------%
% vim: ts=4 sw=4 et ft=mercury
%---------------------------------------------------%
% Copyright (C) 1993-2007, 2009-2014 The University of Melbourne.
% Copyright (C) 2013-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% This module imports all the modules in the Mercury library.
%
% It is used as a way for the Makefiles to know which library interface
% files, objects, etc., need to be installed.
```

```
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module library.
:- interface.

    % version(VersionString, FullarchString)
    %
:- pred library.version(string::out, string::out) is det.
```

45 list

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 1993-2012 The University of Melbourne.
% Copyright (C) 2013-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: list.m.
% Authors: fjh, conway, trd, zs, philip, warwick, ...
% Stability: medium to high.
%
% This module defines the list type, and various utility predicates that
% operate on lists.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module list.
:- interface.

:- import_module pretty_printer.

%-----------------------------------------------------%

    % The definition of the type `list(T)':
    % A list is either an empty list, denoted `[]',
    % or an element `Head' of type `T' followed by a tail `Tail'
    % of type `list(T)', denoted `[Head | Tail]'.
    %
:- type list(T)
    --->    []
```

```
    ;       [T | list(T)].

%---------------------------------------------------%

    % These instantiation states and modes can be used for instantiation
    % state subtyping.
    %
    % They could also be used for partial instantiation but partial
    % instantiation does not work completely, for information see the
    % LIMITATIONS file distributed with Mercury.
    %
:- inst list_skel(I) for list/1
    --->    []
    ;       [I | list_skel(I)].
:- inst list(I) == list_skel(I).

:- inst empty_list for list/1
    --->    [].
:- inst non_empty_list for list/1
    --->    [ground | ground].

%---------------------------------------------------%

:- pred is_empty(list(T)::in) is semidet.

:- pred is_not_empty(list(T)::in) is semidet.

%---------------------------------------------------%

:- func head(list(T)) = T is semidet.

    % det_head(List) returns the first element of List,
    % calling error/1 if List is empty.
    %
:- func det_head(list(T)) = T.

:- func tail(list(T)) = list(T) is semidet.

    % det_tail(List) returns the tail of List,
    % calling error/1 if List is empty.
    %
:- func det_tail(list(T)) = list(T).

    % cons(X, Y) = Z <=> Z = [X | Y].
    %
:- func cons(T, list(T)) = list(T).
:- pred cons(T::in, list(T)::in, list(T)::out) is det.
```

```
%---------------------------------------------------%

    % Standard append predicate:
    % append(Start, End, List) is true iff
    % 'List' is the result of concatenating 'Start' and 'End'.
    %
:- pred append(list(T), list(T), list(T)).
:- mode append(di, di, uo) is det.
:- mode append(in, in, out) is det.
:- mode append(in, in, in) is semidet.    % implied
:- mode append(in, out, in) is semidet.
:- mode append(out, out, in) is multi.
% The following mode is semidet in the sense that it doesn't
% succeed more than once - but it does create a choice-point,
% which means it's inefficient and that the compiler can't deduce
% that it is semidet. Use remove_suffix instead.
% :- mode append(out, in, in) is semidet.

:- func append(list(T), list(T)) = list(T).

    % L1 ++ L2 = L :- append(L1, L2, L).
    %
:- func list(T) ++ list(T) = list(T).

    % remove_suffix(List, Suffix, Prefix):
    %
    % The same as append(Prefix, Suffix, List) except that
    % this is semidet whereas append(out, in, in) is nondet.
    %
:- pred remove_suffix(list(T)::in, list(T)::in, list(T)::out) is semidet.

%---------------------------------------------------%

    % associativity of append
:- promise all [A, B, C, ABC]
    (
        ( some [AB] (list.append(A, B, AB), list.append(AB, C, ABC)) )
    <=>
        ( some [BC] (list.append(B, C, BC), list.append(A, BC, ABC)) )
    ).
    % construction equivalence law.
:- promise all [L, H, T] ( append([H], T, L) <=> L = [H | T] ).

%---------------------------------------------------%

    % length(List, Length):
```

```
    %
    % True iff 'Length' is the length of 'List', i.e. if 'List' contains
    % 'Length' elements.
    %
:- pred length(list(_T), int).
:- mode length(in, out) is det.

:- func length(list(T)) = int.

    % same_length(ListA, ListB):
    %
    % True iff 'ListA' and 'ListB' have the same length,
    % i.e. iff they both contain the same number of elements.
    %
:- pred same_length(list(T1), list(T2)).
:- mode same_length(in, in) is semidet.

    % As above, but for three lists.
    %
:- pred same_length3(list(T1)::in, list(T2)::in, list(T3)::in)
    is semidet.

%-----------------------------------------------%

    % member(Elem, List):
    %
    % True iff 'List' contains 'Elem'.
    %
:- pred member(T, list(T)).
:- mode member(in, in) is semidet.
:- mode member(out, in) is nondet.

    % member(Elem, List, SubList):
    %
    % True iff 'List' contains 'Elem', and 'SubList' is a suffix of 'List'
    % beginning with 'Elem'.
    % Same as 'SubList = [Elem | _], append(_, SubList, List)'.
    %
:- pred member(T::out, list(T)::in, list(T)::out) is nondet.

    % member_index0(Elem, List, Index).
    %
    % True iff 'List' contains 'Elem' at the zero-based index 'Index'.
    %
:- pred member_index0(T, list(T), int).
:- mode member_index0(in, in, in) is semidet.
:- mode member_index0(in, in, out) is nondet.
```

```
:- mode member_index0(out, in, out) is nondet.

    % member_indexes0(Elem, List, Indexes).
    %
    % True iff 'List' contains 'Elem' at the zero-based indexes 'Indexes'.
    % 'Indexes' will be sorted.
    %
:- pred member_indexes0(T::in, list(T)::in, list(int)::out) is det.

    % contains(List, Elem) iff member(Elem, List).
    % Sometimes you need the arguments in this order, because you want to
    % construct a closure with only the list.
    %
:- pred contains(list(T)::in, T::in) is semidet.

%-----------------------------------------------%

    % index*(List, Position, Elem):
    %
    % These predicates select an element in a list from it's position.
    % The 'index0' preds consider the first element to be element
    % number zero, whereas the 'index1' preds consider the first element
    % to be element number one. The 'det_' preds call error/1 if the index
    % is out of range, whereas the semidet preds fail if the index is out of
    % range.
    %
:- pred index0(list(T)::in, int::in, T::out) is semidet.
:- pred index1(list(T)::in, int::in, T::out) is semidet.

:- func det_index0(list(T), int) = T.
:- pred det_index0(list(T)::in, int::in, T::out) is det.
:- func det_index1(list(T), int) = T.
:- pred det_index1(list(T)::in, int::in, T::out) is det.

    % nth_member_search(List, Elem, Position):
    %
    % Elem is the Position'th member of List.
    % (Position numbers start from 1.)
    %
:- pred nth_member_search(list(T)::in, T::in, int::out) is semidet.

    % A deterministic version of nth_member_search, which throws an exception
    % instead of failing if the element is not found in the list.
    %
:- pred nth_member_lookup(list(T)::in, T::in, int::out) is det.

    % index*_of_first_occurrence(List, Elem, Position):
```

```
    %
    % Computes the least value of Position such that
    % list_index*(List, Position, Elem). The 'det_' funcs call error/1
    % if Elem is not a member of List.
    %
:- pred index0_of_first_occurrence(list(T)::in, T::in, int::out) is semidet.
:- pred index1_of_first_occurrence(list(T)::in, T::in, int::out) is semidet.
:- func det_index0_of_first_occurrence(list(T), T) = int.
:- func det_index1_of_first_occurrence(list(T), T) = int.

%-----------------------------------------------------%

    % reverse(List, Reverse):
    %
    % 'Reverse' is a list containing the same elements as 'List'
    % but in reverse order.
    %
:- pred reverse(list(T), list(T)).
:- mode reverse(in, out) is det.
:- mode reverse(out, in) is det.

:- func reverse(list(T)) = list(T).

    % reverse_prepend(Xs, Ys, Zs):
    %
    % Same as 'Zs = list.reverse(Xs) ++ Ys' but more efficient.
    %
:- pred reverse_prepend(list(T)::in, list(T)::in, list(T)::out) is det.
:- func reverse_prepend(list(T), list(T)) = list(T).

%-----------------------------------------------------%

    % insert(Elem, List0, List):
    %
    % 'List' is the result of inserting 'Elem' somewhere in 'List0'.
    % Same as 'delete(List, Elem, List0)'.
    %
:- pred insert(T, list(T), list(T)).
:- mode insert(in, in, in) is semidet.
:- mode insert(in, out, in) is nondet.
:- mode insert(out, out, in) is nondet.
:- mode insert(in, in, out) is multi.

    % delete(List, Elem, Remainder):
    %
    % True iff 'Elem' occurs in 'List', and 'Remainder' is the result of
    % deleting one occurrence of 'Elem' from 'List'.
```

```
    %
:- pred delete(list(T), T, list(T)).
:- mode delete(in, in, in) is semidet.
:- mode delete(in, in, out) is nondet.
:- mode delete(in, out, out) is nondet.
:- mode delete(out, in, in) is multi.

    % delete_first(List0, Elem, List) is true iff Elem occurs in List0
    % and List is List0 with the first occurrence of Elem removed.
    %
:- pred delete_first(list(T)::in, T::in, list(T)::out) is semidet.

    % delete_all(List0, Elem) = List is true iff List is List0 with
    % all occurrences of Elem removed.
    %
:- func delete_all(list(T), T) = list(T).
:- pred delete_all(list(T), T, list(T)).
:- mode delete_all(di, in, uo) is det.
:- mode delete_all(in, in, out) is det.

    % delete_elems(List0, Elems) = List is true iff List is List0 with
    % all occurrences of all elements of Elems removed.
    %
:- func delete_elems(list(T), list(T)) = list(T).
:- pred delete_elems(list(T)::in, list(T)::in, list(T)::out) is det.

    % sublist(SubList, FullList) is true if one can obtain SubList
    % by starting with FullList and deleting some of its elements.
    %
:- pred sublist(list(T)::in, list(T)::in) is semidet.

%-----------------------------------------------%

    % replace(List0, D, R, List) is true iff List is List0
    % with an occurrence of D replaced with R.
    %
:- pred replace(list(T), T, T, list(T)).
:- mode replace(in, in, in, in) is semidet.
:- mode replace(in, in, in, out) is nondet.

    % replace_first(List0, D, R, List) is true iff List is List0
    % with the first occurrence of D replaced with R.
    %
:- pred replace_first(list(T)::in, T::in, T::in, list(T)::out) is semidet.

    % replace_all(List0, D, R) = List is true iff List is List0
    % with all occurrences of D replaced with R.
```

```
    %
:- func replace_all(list(T), T, T) = list(T).
:- pred replace_all(list(T)::in, T::in, T::in, list(T)::out) is det.

    % replace_nth(List0, N, R, List) is true iff List is List0
    % with N'th element replaced with R.
    % Fails if N < 1 or if length of List0 < N.
    % (Position numbers start from 1.)
    %
:- pred replace_nth(list(T)::in, int::in, T::in, list(T)::out) is semidet.

    % det_replace_nth(List0, N, R) = List is true iff List is List0
    % with N'th element replaced with R.
    % Throws an exception if N < 1 or if length of List0 < N.
    % (Position numbers start from 1.)
    %
:- func det_replace_nth(list(T), int, T) = list(T).
:- pred det_replace_nth(list(T)::in, int::in, T::in, list(T)::out) is det.

%---------------------------------------------------%

    % Lo `..` Hi = [Lo, Lo + 1, ..., Hi] if Lo =< Hi, and [] otherwise.
    %
:- func int `..` int = list(int).

%---------------------------------------------------%

    % series(X, OK, Succ) = [X0, X1, ..., Xn]
    %
    % where X0 = X and successive elements Xj, Xk are computed as
    % Xk = Succ(Xj). The series terminates as soon as an element Xi is
    % generated such that OK(Xi) fails; Xi is not included in the output.
    %
:- func series(T, pred(T), func(T) = T) = list(T).
:- mode series(in, pred(in) is semidet, func(in) = out is det) = out is det.

%---------------------------------------------------%

    % remove_dups(L0) = L:
    %
    % L is the result of deleting the second and subsequent occurrences
    % of every element that occurs twice in L0.
    %
:- func remove_dups(list(T)) = list(T).
:- pred remove_dups(list(T)::in, list(T)::out) is det.

    % remove_adjacent_dups(L0) = L:
```

```
    %
    % L is the result of replacing every sequence of duplicate elements in L0
    % with a single such element.
    %
:- func remove_adjacent_dups(list(T)) = list(T).
:- pred remove_adjacent_dups(list(T)::in, list(T)::out) is det.

    % remove_adjacent_dups(P, L0, L) is true iff L is the result
    % of replacing every sequence of elements in L0 which are equivalent
    % with respect to the ordering, with the first occurrence in L0 of
    % such an element.
    %
:- pred remove_adjacent_dups(comparison_pred(X)::in(comparison_pred),
    list(X)::in, list(X)::out) is det.

%-----------------------------------------------------%

    % merge(L1, L2) = L:
    %
    % L is the result of merging the elements of L1 and L2, in ascending order.
    % L1 and L2 must be sorted.
    %
:- func merge(list(T), list(T)) = list(T).
:- pred merge(list(T)::in, list(T)::in, list(T)::out) is det.

    % merge(Compare, As, Bs) = Sorted is true iff, assuming As and
    % Bs are sorted with respect to the ordering defined by Compare,
    % Sorted is a list containing the elements of As and Bs which is
    % also sorted. For elements which are equivalent in the ordering,
    % if they come from the same list then they appear in the same
    % sequence in Sorted as they do in that list, otherwise the elements
    % from As appear before the elements from Bs.
    %
:- func merge(comparison_func(X), list(X), list(X)) = list(X).
:- pred merge(comparison_pred(X)::in(comparison_pred),
    list(X)::in, list(X)::in, list(X)::out) is det.

    % merge_and_remove_dups(L1, L2) = L:
    %
    % L is the result of merging the elements of L1 and L2, in ascending order,
    % and eliminating any duplicates. L1 and L2 must be sorted and must each
    % not contain any duplicates.
    %
:- func merge_and_remove_dups(list(T), list(T)) = list(T).
:- pred merge_and_remove_dups(list(T)::in, list(T)::in, list(T)::out) is det.

    % merge_and_remove_dups(Compare, As, Bs) = Sorted is true iff,
```

```
    % assuming As and Bs are sorted with respect to the ordering defined
    % by Compare and neither contains any duplicates, Sorted is a list
    % containing the elements of As and Bs which is also sorted and
    % contains no duplicates. If an element from As is duplicated in
    % Bs (that is, they are equivalent in the ordering), then the element
    % from As is the one that appears in Sorted.
    %
:- func merge_and_remove_dups(comparison_func(X), list(X), list(X))
    = list(X).
:- pred merge_and_remove_dups(comparison_pred(X)::in(comparison_pred),
    list(X)::in, list(X)::in, list(X)::out) is det.

%-----------------------------------------------------%

    % sort(List) = SortedList:
    %
    % SortedList is List sorted.
    %
:- func sort(list(T)) = list(T).
:- pred sort(list(T)::in, list(T)::out) is det.

    % sort_and_remove_dups(List) = SortedList:
    %
    % SortedList is List sorted with the second and subsequent occurrence of
    % any duplicates removed.
    %
:- func sort_and_remove_dups(list(T)) = list(T).
:- pred sort_and_remove_dups(list(T)::in, list(T)::out) is det.

%-----------------------------------------------------%

    % sort(Compare, Unsorted) = Sorted is true iff Sorted is a
    % list containing the same elements as Unsorted, where Sorted is
    % sorted with respect to the ordering defined by Compare,
    % and the elements that are equivalent in this ordering appear
    % in the same sequence in Sorted as they do in Unsorted
    % (that is, the sort is stable).
    %
:- func sort(comparison_func(X), list(X)) = list(X).
:- pred sort(comparison_pred(X)::in(comparison_pred), list(X)::in,
    list(X)::out) is det.

    % sort_and_remove_dups(Compare, Unsorted, Sorted) is true iff
    % Sorted is a list containing the same elements as Unsorted, where
    % Sorted is sorted with respect to the ordering defined by the
    % predicate term Compare, except that if two elements in Unsorted
    % are equivalent with respect to this ordering only the one which
```

```
    % occurs first will be in Sorted.
    %
:- pred sort_and_remove_dups(comparison_pred(X)::in(comparison_pred),
    list(X)::in, list(X)::out) is det.

%-----------------------------------------------%

    % split_list(N, List, Start, End):
    %
    % splits 'List' into a prefix 'Start' of length 'N', and a remainder
    % 'End'. Fails if 'N' is not in '0 .. length(List)'.
    % See also: take, drop and split_upto.
    %
:- pred split_list(int::in, list(T)::in, list(T)::out, list(T)::out)
    is semidet.

    % det_split_list(N, List, Start, End):
    %
    % A deterministic version of split_list, which throws an exception
    % instead of failing if 'N' is not in 0 .. length(List).
    %
:- pred det_split_list(int::in, list(T)::in, list(T)::out, list(T)::out)
    is det.

    % split_upto(N, List, Start, End):
    %
    % splits 'List' into a prefix 'Start' of length 'min(N, length(List))',
    % and a remainder 'End'. Throws an exception if 'N' < 0.
    % See also: split_list, take, drop.
    %
:- pred split_upto(int::in, list(T)::in, list(T)::out, list(T)::out) is det.

%-----------------------------------------------%

    % last(List, Last) is true if Last is the last element of List.
    %
:- pred last(list(T)::in, T::out) is semidet.

    % A deterministic version of last, which throws an exception instead of
    % failing if the input list is empty.
    %
:- func det_last(list(T)) = T.
:- pred det_last(list(T)::in, T::out) is det.

    % split_last(List, AllButLast, Last) is true if Last is the
    % last element of List and AllButLast is the list of elements before it.
    %
```

```
:- pred split_last(list(T)::in, list(T)::out, T::out) is semidet.

    % A deterministic version of split_last, which throws an exception
    % instead of failing if the input list is empty.
    %
:- pred det_split_last(list(T)::in, list(T)::out, T::out) is det.

%---------------------------------------------------%

    % take(N, List, Start):
    %
    % 'Start' is the first 'Len' elements of 'List'.
    % Fails if 'N' is not in '0 .. length(List)'.
    %
:- pred take(int::in, list(T)::in, list(T)::out) is semidet.

    % det_take(Len, List, Start):
    %
    % As above, but throw an exception instead of failing.
    %
:- pred det_take(int::in, list(T)::in, list(T)::out) is det.

    % take_upto(Len, List) = Start:
    %
    % 'Start' is the first 'Len' elements of 'List'. If 'List' has less than
    % 'Len' elements, return the entire list. Throws an exception if 'N' < 0.
    %
:- func take_upto(int, list(T)) = list(T).
:- pred take_upto(int::in, list(T)::in, list(T)::out) is det.

%---------------------------------------------------%

    % drop(N, List, End):
    %
    % 'End' is the remainder of 'List' after removing the first 'N' elements.
    % Fails if 'N' is not in '0 .. length(List)'.
    % See also: split_list.
    %
:- pred drop(int::in, list(T)::in, list(T)::out) is semidet.

    % det_drop(N, List, End):
    %
    % 'End' is the remainder of 'List' after removing the first 'N' elements.
    % Throws an exception if 'N' is not in '0 .. length(List)'.
    % See also: split_list.
    %
:- pred det_drop(int::in, list(T)::in, list(T)::out) is det.
```

```
%---------------------------------------------------%

    % take_while(Pred, List, Start, End)
    %
    % List = Start ++ End. Start is the longest prefix of List where Pred
    % succeeds for every element in Start. End is the remainder of the list.
    %
:- pred take_while(pred(T)::in(pred(in) is semidet), list(T)::in,
    list(T)::out, list(T)::out) is det.

    % takewhile/4 is the old name for take_while/4.
    %
:- pragma obsolete(takewhile/4).
:- pred takewhile(pred(T)::in(pred(in) is semidet), list(T)::in,
    list(T)::out, list(T)::out) is det.

    % take_while(Pred, List) = Start :-
    %     take_while(Pred, List, Start, _End)
    %
    % Start is the longest prefix of List where Pred succeeds for every element
    % in Start.
    %
:- func take_while(pred(T), list(T)) = list(T).
:- mode take_while(pred(in) is semidet, in) = out is det.
:- pred take_while(pred(T)::in(pred(in) is semidet), list(T)::in,
    list(T)::out) is det.

%---------------------------------------------------%

    % drop_while(Pred, List) = End :-
    %     take_while(Pred, List, _Start, End).
    %
    % End is the remainder of List after removing all the consecutive
    % elements from the start of List for which Pred succeeds.
    %
:- func drop_while(pred(T), list(T)) = list(T).
:- mode drop_while(pred(in) is semidet, in) = out is det.
:- pred drop_while(pred(T)::in(pred(in) is semidet), list(T)::in,
    list(T)::out) is det.

%---------------------------------------------------%

    % duplicate(Count, Elem) = List is true iff List is a list
    % containing Count duplicate copies of Elem.
    %
:- func duplicate(int, T) = list(T).
```

```
:- pred duplicate(int::in, T::in, list(T)::out) is det.

    % all_same(List) is true if all elements of the list are the same.
    %
:- pred all_same(list(T)::in) is semidet.

%---------------------------------------------------%

    % condense(ListOfLists) = List:
    %
    % ‘List’ is the result of concatenating all the elements of ‘ListOfLists’.
    %
:- func condense(list(list(T))) = list(T).
:- pred condense(list(list(T))::in, list(T)::out) is det.

    % chunk(List, ChunkSize) = Chunks:
    %
    % Takes a list ‘List’ and breaks it into a list of lists ‘Chunks’,
    % such that the length of each list in ‘Chunks’ is at most ‘ChunkSize.
    % (More precisely, the length of each list in ‘Chunks’ other than the
    % last one is exactly ‘ChunkSize’, and the length of the last list in
    % ‘Chunks’ is between one and ‘ChunkSize’.)
    %
:- func chunk(list(T), int) = list(list(T)).
:- pred chunk(list(T)::in, int::in, list(list(T))::out) is det.

%---------------------------------------------------%

    % zip(ListA, ListB) = List:
    %
    % List is the result of alternating the elements of ListA and ListB,
    % starting with the first element of ListA (followed by the first element
    % of ListB, then the second element of listA, then the second element
    % of ListB, etc.). When there are no more elements remaining in one of
    % the lists, the remainder of the nonempty list is appended.
    %
:- func zip(list(T), list(T)) = list(T).
:- pred zip(list(T)::in, list(T)::in, list(T)::out) is det.

%---------------------------------------------------%

    % perm(List0, List):
    %
    % True iff ‘List’ is a permutation of ‘List0’.
    %
:- pred perm(list(T)::in, list(T)::out) is multi.
```

```
%---------------------------------------------------%

    % Convert a list to a pretty_printer.doc for formatting.
    %
:- func list_to_doc(list(T)) = pretty_printer.doc.

%---------------------------------------------------%
%
% The following group of predicates use higher-order terms to simplify
% various list processing tasks. They implement pretty much standard
% sorts of operations provided by standard libraries for functional languages.
%
%---------------------------------------------------%

    % find_first_match(Pred, List, FirstMatch) takes a closure with one
    % input argument. It returns the first element X of the list (if any)
    % for which Pred(X) is true.
    %
:- pred find_first_match(pred(X)::in(pred(in) is semidet), list(X)::in,
    X::out) is semidet.

    % any_true(Pred, List):
    % Succeeds iff Pred succeeds for at least one element of List.
    % Same as `not all_false(Pred, List)'.
    %
:- pred any_true(pred(X)::in(pred(in) is semidet), list(X)::in) is semidet.

    % any_false(Pred, List):
    % Succeeds iff Pred fails for at least one element of List.
    % Same as `not all_true(Pred, List)'.
    %
:- pred any_false(pred(X)::in(pred(in) is semidet), list(X)::in) is semidet.

    % all_true(Pred, List) takes a closure with one input argument.
    % If Pred succeeds for every member of List, all_true succeeds.
    % If Pred fails for any member of List, all_true fails.
    %
:- pred all_true(pred(X)::in(pred(in) is semidet), list(X)::in) is semidet.

    % all_false(Pred, List) takes a closure with one input argument.
    % If Pred fails for every member of List, all_false succeeds.
    % If Pred succeeds for any member of List, all_false fails.
    %
:- pred all_false(pred(X)::in(pred(in) is semidet), list(X)::in) is semidet.

    % all_true_corresponding(Pred, ListA, ListB):
    % Succeeds if Pred succeeds for every corresponding pair of elements from
```

```
    % ListA and ListB. Fails if Pred fails for any pair of corresponding
    % elements.
    %
    % An exception is raised if the list arguments differ in length.
    %
:- pred all_true_corresponding(pred(X, Y)::in(pred(in, in) is semidet),
    list(X)::in, list(Y)::in) is semidet.

    % all_false_corresponding(Pred, ListA, ListB):
    % Succeeds if Pred fails for every corresponding pair of elements from
    % ListA and ListB. Fails if Pred succeeds for any pair of corresponding
    % elements.
    %
    % An exception is raised if the list arguments differ in length.
    %
:- pred all_false_corresponding(pred(X, Y)::in(pred(in, in) is semidet),
    list(X)::in, list(Y)::in) is semidet.

%-----------------------------------------------%

    % filter(Pred, List) = TrueList takes a closure with one
    % input argument and for each member X of List, calls the closure.
    % X is included in TrueList iff Pred(X) is true.
    %
:- func filter(pred(X)::in(pred(in) is semidet), list(X)::in)
    = (list(X)::out) is det.
:- pred filter(pred(X)::in(pred(in) is semidet), list(X)::in,
    list(X)::out) is det.

    % filter(Pred, List, TrueList, FalseList) takes a closure with one
    % input argument and for each member X of List, calls the closure.
    % X is included in TrueList iff Pred(X) is true.
    % X is included in FalseList iff Pred(X) is false.
    %
:- pred filter(pred(X)::in(pred(in) is semidet), list(X)::in,
    list(X)::out, list(X)::out) is det.

    % negated_filter(Pred, List) = FalseList takes a closure with one
    % input argument and for each member of List 'X', calls the closure.
    % X is included in FalseList iff Pred(X) is false.
    %
:- func negated_filter(pred(X)::in(pred(in) is semidet), list(X)::in)
    = (list(X)::out) is det.
:- pred negated_filter(pred(X)::in(pred(in) is semidet), list(X)::in,
    list(X)::out) is det.

    % filter_map(Transformer, List, TrueList) takes a semidet function
```

```
    % and calls it with each element of List. If a call succeeds, then
    % its return value is included in TrueList.
    %
:- func filter_map(func(X) = Y, list(X)) = list(Y).
:- mode filter_map(func(in) = out is semidet, in) = out is det.

    % filter_map(Transformer, List, TrueList) takes a predicate
    % with one input argument and one output argument, and calls it
    % with each element of List. If a call succeeds, then
    % its output is included in TrueList.
    %
:- pred filter_map(pred(X, Y)::in(pred(in, out) is semidet),
    list(X)::in, list(Y)::out) is det.

    % filter_map(Transformer, List, TrueList, FalseList) takes
    % a predicate with one input argument and one output argument.
    % It is called with each element of List. If a call succeeds,
    % then the output is included in TrueList; otherwise, the failing
    % input is included in FalseList.
    %
:- pred filter_map(pred(X, Y)::in(pred(in, out) is semidet),
    list(X)::in, list(Y)::out, list(X)::out) is det.

    % Same as filter_map/3 except that it only returns the first match:
    %
    %   find_first_map(X, Y, Z) <=> filter_map(X, Y, [Z | _])
    %
:- pred find_first_map(pred(X, Y)::in(pred(in, out) is semidet),
    list(X)::in, Y::out) is semidet.

    % Same as find_first_map, except with two outputs.
    %
:- pred find_first_map2(pred(X, A, B)::in(pred(in, out, out) is semidet),
    list(X)::in, A::out, B::out) is semidet.

    % Same as find_first_map, except with three outputs.
    %
:- pred find_first_map3(
    pred(X, A, B, C)::in(pred(in, out, out, out) is semidet),
    list(X)::in, A::out, B::out, C::out) is semidet.

    % find_index_of_match(Match, List, Index0, Index)
    %
    % Find the index of the first item in List for which Match is true,
    % where the first element in the list has the index Index0.
    % (Index0 is *not* the number of items to skip at the head of List.)
    %
```

```
:- pred find_index_of_match(pred(T), list(T), int, int).
:- mode find_index_of_match(pred(in) is semidet, in, in, out) is semidet.

%-----------------------------------------------%

    % map(T, L) = M:
    % map(T, L, M):
    %
    % Apply the closure T to transform the elements of L
    % into the elements of M.
    %
:- func map(func(X) = Y, list(X)) = list(Y).
:- pred map(pred(X, Y), list(X), list(Y)).
:- mode map(pred(in, out) is det, in, out) is det.
:- mode map(pred(in, out) is cc_multi, in, out) is cc_multi.
:- mode map(pred(in, out) is semidet, in, out) is semidet.
:- mode map(pred(in, out) is multi, in, out) is multi.
:- mode map(pred(in, out) is nondet, in, out) is nondet.
:- mode map(pred(in, in) is semidet, in, in) is semidet.

    % map2(T, L, M1, M2) uses the closure T
    % to transform the elements of L into the elements of M1 and M2.
    %
:- pred map2(pred(A, B, C), list(A), list(B), list(C)).
:- mode map2(pred(in, out, out) is det, in, out, out) is det.
:- mode map2(pred(in, out, out) is cc_multi, in, out, out) is cc_multi.
:- mode map2(pred(in, out, out) is semidet, in, out, out) is semidet.
:- mode map2(pred(in, out, out) is multi, in, out, out) is multi.
:- mode map2(pred(in, out, out) is nondet, in, out, out) is nondet.
:- mode map2(pred(in, in, in) is semidet, in, in, in) is semidet.

    % map3(T, L, M1, M2, M3) uses the closure T
    % to transform the elements of L into the elements of M1, M2 and M3.
    %
:- pred map3(pred(A, B, C, D), list(A), list(B), list(C), list(D)).
:- mode map3(pred(in, out, out, out) is det, in, out, out, out) is det.
:- mode map3(pred(in, out, out, out) is cc_multi, in, out, out, out)
    is cc_multi.
:- mode map3(pred(in, out, out, out) is semidet, in, out, out, out)
    is semidet.
:- mode map3(pred(in, out, out, out) is multi, in, out, out, out)
    is multi.
:- mode map3(pred(in, out, out, out) is nondet, in, out, out, out)
    is nondet.
:- mode map3(pred(in, in, in, in) is semidet, in, in, in, in) is semidet.

    % map4(T, L, M1, M2, M3, M4) uses the closure T
```

```
    % to transform the elements of L into the elements of M1, M2, M3 and M4.
    %
:- pred map4(pred(A, B, C, D, E), list(A), list(B), list(C), list(D),
    list(E)).
:- mode map4(pred(in, out, out, out, out) is det, in, out, out, out, out)
    is det.
:- mode map4(pred(in, out, out, out, out) is cc_multi, in, out, out, out,
    out) is cc_multi.
:- mode map4(pred(in, out, out, out, out) is semidet, in, out, out, out,
    out) is semidet.
:- mode map4(pred(in, out, out, out, out) is multi, in, out, out, out,
    out) is multi.
:- mode map4(pred(in, out, out, out, out) is nondet, in, out, out, out,
    out) is nondet.
:- mode map4(pred(in, in, in, in, in) is semidet, in, in, in, in, in)
    is semidet.

    % map5(T, L, M1, M2, M3, M4, M5) uses the closure T
    % to transform the elements of L into the elements of M1, M2, M3, M4
    % and M5.
    %
:- pred map5(pred(A, B, C, D, E, F), list(A), list(B), list(C), list(D),
    list(E), list(F)).
:- mode map5(pred(in, out, out, out, out, out) is det, in, out, out, out,
    out, out) is det.
:- mode map5(pred(in, out, out, out, out, out) is cc_multi, in, out, out,
    out, out, out) is cc_multi.
:- mode map5(pred(in, out, out, out, out, out) is semidet, in, out, out,
    out, out, out) is semidet.
:- mode map5(pred(in, out, out, out, out, out) is multi, in, out, out,
    out, out, out) is multi.
:- mode map5(pred(in, out, out, out, out, out) is nondet, in, out, out,
    out, out, out) is nondet.
:- mode map5(pred(in, in, in, in, in, in) is semidet, in, in, in, in, in,
    in) is semidet.

    % map6(T, L, M1, M2, M3, M4, M5, M6) uses the closure T
    % to transform the elements of L into the elements of M1, M2, M3, M4,
    % M5 and M6.
    %
:- pred map6(pred(A, B, C, D, E, F, G), list(A), list(B), list(C),
    list(D), list(E), list(F), list(G)).
:- mode map6(pred(in, out, out, out, out, out, out) is det, in, out, out,
    out, out, out, out) is det.
:- mode map6(pred(in, out, out, out, out, out, out) is cc_multi, in, out,
    out, out, out, out, out) is cc_multi.
:- mode map6(pred(in, out, out, out, out, out, out) is semidet, in, out,
```

```
    out, out, out, out, out) is semidet.
:- mode map6(pred(in, out, out, out, out, out, out) is multi, in, out,
    out, out, out, out, out) is multi.
:- mode map6(pred(in, out, out, out, out, out, out) is nondet, in, out,
    out, out, out, out, out) is nondet.
:- mode map6(pred(in, in, in, in, in, in, in) is semidet, in, in, in, in,
    in, in, in) is semidet.

    % map7(T, L, M1, M2, M3, M4, M5, M6, M7) uses the closure T
    % to transform the elements of L into the elements of M1, M2, M3, M4,
    % M5, M6 and M7.
    %
:- pred map7(pred(A, B, C, D, E, F, G, H), list(A), list(B), list(C),
    list(D), list(E), list(F), list(G), list(H)).
:- mode map7(pred(in, out, out, out, out, out, out, out) is det,
    in, out, out, out, out, out, out, out) is det.
:- mode map7(pred(in, out, out, out, out, out, out, out) is cc_multi,
    in, out, out, out, out, out, out, out) is cc_multi.
:- mode map7(pred(in, out, out, out, out, out, out, out) is semidet,
    in, out, out, out, out, out, out, out) is semidet.
:- mode map7(pred(in, out, out, out, out, out, out, out) is multi,
    in, out, out, out, out, out, out, out) is multi.
:- mode map7(pred(in, out, out, out, out, out, out, out) is nondet,
    in, out, out, out, out, out, out, out) is nondet.
:- mode map7(pred(in, in, in, in, in, in, in, in) is semidet,
    in, in, in, in, in, in, in, in) is semidet.

    % map8(T, L, M1, M2, M3, M4, M5, M6, M7) uses the closure T
    % to transform the elements of L into the elements of M1, M2, M3, M4,
    % M5, M6, M7 and M8.
    %
:- pred map8(pred(A, B, C, D, E, F, G, H, I), list(A), list(B), list(C),
    list(D), list(E), list(F), list(G), list(H), list(I)).
:- mode map8(pred(in, out, out, out, out, out, out, out, out) is det,
    in, out, out, out, out, out, out, out, out) is det.
:- mode map8(pred(in, out, out, out, out, out, out, out, out) is cc_multi,
    in, out, out, out, out, out, out, out, out) is cc_multi.
:- mode map8(pred(in, out, out, out, out, out, out, out, out) is semidet,
    in, out, out, out, out, out, out, out, out) is semidet.
:- mode map8(pred(in, out, out, out, out, out, out, out, out) is multi,
    in, out, out, out, out, out, out, out, out) is multi.
:- mode map8(pred(in, out, out, out, out, out, out, out, out) is nondet,
    in, out, out, out, out, out, out, out, out) is nondet.
:- mode map8(pred(in, in, in, in, in, in, in, in, in) is semidet,
    in, in, in, in, in, in, in, in, in) is semidet.

%-----------------------------------------------%
```

```
    % map_corresponding(F, [A1, .. An], [B1, .. Bn]) =
    %   [F(A1, B1), .., F(An, Bn)].
    %
    % An exception is raised if the list arguments differ in length.
    %
:- func map_corresponding(func(A, B) = R, list(A), list(B)) = list(R).
:- pred map_corresponding(pred(A, B, R), list(A), list(B), list(R)).
:- mode map_corresponding(in(pred(in, in, out) is det), in, in, out)
    is det.
:- mode map_corresponding(in(pred(in, in, out) is semidet), in, in, out)
    is semidet.

    % map_corresponding3(F, [A1, .. An], [B1, .. Bn], [C1, .. Cn]) =
    %   [F(A1, B1, C1), .., F(An, Bn, Cn)].
    %
    % An exception is raised if the list arguments differ in length.
    %
:- func map_corresponding3(func(A, B, C) = R, list(A), list(B), list(C))
    = list(R).
:- pred map_corresponding3(pred(A, B, C, R), list(A), list(B), list(C),
    list(R)).
:- mode map_corresponding3(in(pred(in, in, in, out) is det),
    in, in, in, out) is det.
:- mode map_corresponding3(in(pred(in, in, in, out) is semidet),
    in, in, in, out) is semidet.

%---------------------------------------------------%

    % filter_map_corresponding/3 is like map_corresponding/3
    % except the function argument is semidet and the output list
    % consists of only those applications of the function argument that
    % succeeded.
    %
:- func filter_map_corresponding(func(A, B) = R, list(A), list(B))
    = list(R).
:- mode filter_map_corresponding(func(in, in) = out is semidet, in, in)
    = out is det.
:- pred filter_map_corresponding(
    pred(A, B, R)::in(pred(in, in, out) is semidet),
    list(A)::in, list(B)::in, list(R)::out) is det.

    % filter_map_corresponding3/4 is like map_corresponding3/4
    % except the function argument is semidet and the output list
    % consists of only those applications of the function argument that
    % succeeded.
    %
```

```
:- func filter_map_corresponding3(func(A, B, C) = R,
    list(A), list(B), list(C)) = list(R).
:- mode filter_map_corresponding3(func(in, in, in) = out is semidet,
    in, in, in) = out is det.
:- pred filter_map_corresponding3(
    pred(A, B, C, R)::in(pred(in, in, in, out) is semidet),
    list(A)::in, list(B)::in, list(C)::in, list(R)::out) is det.

%-----------------------------------------------%

    % foldl(Func, List, Start) = End calls Func with each element of List
    % (working left-to-right) and an accumulator (with the initial value
    % of Start), and returns the final value in End.
    %
:- func foldl(func(L, A) = A, list(L), A) = A.

    % foldl(Pred, List, Start, End) calls Pred with each element of List
    % (working left-to-right) and an accumulator (with the initial value
    % of Start), and returns the final value in End.
    %
:- pred foldl(pred(L, A, A), list(L), A, A).
:- mode foldl(pred(in, in, out) is det, in, in, out) is det.
:- mode foldl(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode foldl(pred(in, di, uo) is det, in, di, uo) is det.
:- mode foldl(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode foldl(pred(in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode foldl(pred(in, di, uo) is semidet, in, di, uo) is semidet.
:- mode foldl(pred(in, in, out) is multi, in, in, out) is multi.
:- mode foldl(pred(in, in, out) is nondet, in, in, out) is nondet.
:- mode foldl(pred(in, mdi, muo) is nondet, in, mdi, muo) is nondet.
:- mode foldl(pred(in, in, out) is cc_multi, in, in, out) is cc_multi.
:- mode foldl(pred(in, di, uo) is cc_multi, in, di, uo) is cc_multi.

    % foldl2(Pred, List, !Acc1, !Acc2):
    % Does the same job as foldl, but with two accumulators.
    % (Although no more expressive than foldl, this is often
    % a more convenient format, and a little more efficient).
    %
:- pred foldl2(pred(L, A, A, Z, Z), list(L), A, A, Z, Z).
:- mode foldl2(pred(in, in, out, in, out) is det,
    in, in, out, in, out) is det.
:- mode foldl2(pred(in, in, out, mdi, muo) is det,
    in, in, out, mdi, muo) is det.
:- mode foldl2(pred(in, in, out, di, uo) is det,
    in, in, out, di, uo) is det.
:- mode foldl2(pred(in, di, uo, di, uo) is det,
    in, di, uo, di, uo) is det.
```

```
:- mode foldl2(pred(in, in, out, in, out) is semidet,
    in, in, out, in, out) is semidet.
:- mode foldl2(pred(in, in, out, mdi, muo) is semidet,
    in, in, out, mdi, muo) is semidet.
:- mode foldl2(pred(in, in, out, di, uo) is semidet,
    in, in, out, di, uo) is semidet.
:- mode foldl2(pred(in, in, out, in, out) is nondet,
    in, in, out, in, out) is nondet.
:- mode foldl2(pred(in, in, out, mdi, muo) is nondet,
    in, in, out, mdi, muo) is nondet.
:- mode foldl2(pred(in, in, out, in, out) is cc_multi,
    in, in, out, in, out) is cc_multi.
:- mode foldl2(pred(in, in, out, mdi, muo) is cc_multi,
    in, in, out, mdi, muo) is cc_multi.
:- mode foldl2(pred(in, in, out, di, uo) is cc_multi,
    in, in, out, di, uo) is cc_multi.
:- mode foldl2(pred(in, di, uo, di, uo) is cc_multi,
    in, di, uo, di, uo) is cc_multi.

    % foldl3(Pred, List, !Acc1, !Acc2, !Acc3):
    % Does the same job as foldl, but with three accumulators.
    % (Although no more expressive than foldl, this is often
    % a more convenient format, and a little more efficient).
    %
:- pred foldl3(pred(L, A, A, B, B, C, C), list(L),
    A, A, B, B, C, C).
:- mode foldl3(pred(in, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out) is det.
:- mode foldl3(pred(in, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, mdi, muo) is det.
:- mode foldl3(pred(in, in, out, in, out, di, uo) is det,
    in, in, out, in, out, di, uo) is det.
:- mode foldl3(pred(in, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out) is semidet.
:- mode foldl3(pred(in, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, mdi, muo) is semidet.
:- mode foldl3(pred(in, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, di, uo) is semidet.
:- mode foldl3(pred(in, in, out, in, out, in, out) is nondet,
    in, in, out, in, out, in, out) is nondet.
:- mode foldl3(pred(in, in, out, in, out, mdi, muo) is nondet,
    in, in, out, in, out, mdi, muo) is nondet.
:- mode foldl3(pred(in, in, out, in, out, in, out) is cc_multi,
    in, in, out, in, out, in, out) is cc_multi.
:- mode foldl3(pred(in, in, out, in, out, di, uo) is cc_multi,
    in, in, out, in, out, di, uo) is cc_multi.
```

```
    % foldl4(Pred, List, !Acc1, !Acc2, !Acc3, !Acc4):
    % Does the same job as foldl, but with four accumulators.
    % (Although no more expressive than foldl, this is often
    % a more convenient format, and a little more efficient).
    %
:- pred foldl4(pred(L, A, A, B, B, C, C, D, D), list(L),
    A, A, B, B, C, C, D, D).
:- mode foldl4(pred(in, in, out, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out, in, out) is det.
:- mode foldl4(pred(in, in, out, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, in, out, mdi, muo) is det.
:- mode foldl4(pred(in, in, out, in, out, in, out, di, uo) is det,
    in, in, out, in, out, in, out, di, uo) is det.
:- mode foldl4(pred(in, in, out, in, out, in, out, in, out) is cc_multi,
    in, in, out, in, out, in, out, in, out) is cc_multi.
:- mode foldl4(pred(in, in, out, in, out, in, out, di, uo) is cc_multi,
    in, in, out, in, out, in, out, di, uo) is cc_multi.
:- mode foldl4(pred(in, in, out, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out, in, out) is semidet.
:- mode foldl4(pred(in, in, out, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode foldl4(pred(in, in, out, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, in, out, di, uo) is semidet.
:- mode foldl4(pred(in, in, out, in, out, in, out, in, out) is nondet,
    in, in, out, in, out, in, out, in, out) is nondet.
:- mode foldl4(pred(in, in, out, in, out, in, out, mdi, muo) is nondet,
    in, in, out, in, out, in, out, mdi, muo) is nondet.

    % foldl5(Pred, List, !Acc1, !Acc2, !Acc3, !Acc4, !Acc5):
    % Does the same job as foldl, but with five accumulators.
    % (Although no more expressive than foldl, this is often
    % a more convenient format, and a little more efficient).
    %
:- pred foldl5(pred(L, A, A, B, B, C, C, D, D, E, E), list(L),
    A, A, B, B, C, C, D, D, E, E).
:- mode foldl5(pred(in, in, out, in, out, in, out, in, out, in, out)
    is det,
    in, in, out, in, out, in, out, in, out, in, out) is det.
:- mode foldl5(pred(in, in, out, in, out, in, out, in, out, mdi, muo)
    is det,
    in, in, out, in, out, in, out, in, out, mdi, muo) is det.
:- mode foldl5(pred(in, in, out, in, out, in, out, in, out, di, uo)
    is det,
    in, in, out, in, out, in, out, in, out, di, uo) is det.
:- mode foldl5(pred(in, in, out, in, out, in, out, in, out, in, out)
    is semidet,
    in, in, out, in, out, in, out, in, out, in, out) is semidet.
```

```
:- mode foldl5(pred(in, in, out, in, out, in, out, in, out, mdi, muo)
    is semidet,
    in, in, out, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode foldl5(pred(in, in, out, in, out, in, out, in, out, di, uo)
    is semidet,
    in, in, out, in, out, in, out, in, out, di, uo) is semidet.
:- mode foldl5(pred(in, in, out, in, out, in, out, in, out, in, out)
    is nondet,
    in, in, out, in, out, in, out, in, out, in, out) is nondet.
:- mode foldl5(pred(in, in, out, in, out, in, out, in, out, mdi, muo)
    is nondet,
    in, in, out, in, out, in, out, in, out, mdi, muo) is nondet.
:- mode foldl5(pred(in, in, out, in, out, in, out, in, out, in, out)
    is cc_multi,
    in, in, out, in, out, in, out, in, out, in, out) is cc_multi.
:- mode foldl5(pred(in, in, out, in, out, in, out, in, out, di, uo)
    is cc_multi,
    in, in, out, in, out, in, out, in, out, di, uo) is cc_multi.

    % foldl6(Pred, List, !Acc1, !Acc2, !Acc3, !Acc4, !Acc5, !Acc6):
    % Does the same job as foldl, but with six accumulators.
    % (Although no more expressive than foldl, this is often
    % a more convenient format, and a little more efficient).
    %
:- pred foldl6(pred(L, A, A, B, B, C, C, D, D, E, E, F, F), list(L),
    A, A, B, B, C, C, D, D, E, E, F, F).
:- mode foldl6(pred(in, in, out, in, out, in, out, in, out, in, out,
    in, out) is det,
    in, in, out, in, out, in, out, in, out, in, out, in, out) is det.
:- mode foldl6(pred(in, in, out, in, out, in, out, in, out, in, out,
    mdi, muo) is det,
    in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is det.
:- mode foldl6(pred(in, in, out, in, out, in, out, in, out, in, out,
    di, uo) is det,
    in, in, out, in, out, in, out, in, out, in, out, di, uo) is det.
:- mode foldl6(pred(in, in, out, in, out, in, out, in, out, in, out,
    in, out) is cc_multi,
    in, in, out, in, out, in, out, in, out, in, out, in, out) is cc_multi.
:- mode foldl6(pred(in, in, out, in, out, in, out, in, out, in, out,
    di, uo) is cc_multi,
    in, in, out, in, out, in, out, in, out, in, out, di, uo) is cc_multi.
:- mode foldl6(pred(in, in, out, in, out, in, out, in, out, in, out,
    in, out) is semidet,
    in, in, out, in, out, in, out, in, out, in, out, in, out) is semidet.
:- mode foldl6(pred(in, in, out, in, out, in, out, in, out, in, out,
    mdi, muo) is semidet,
    in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is semidet.
```

```
:- mode foldl6(pred(in, in, out, in, out, in, out, in, out, in, out,
    di, uo) is semidet,
    in, in, out, in, out, in, out, in, out, in, out, di, uo) is semidet.
:- mode foldl6(pred(in, in, out, in, out, in, out, in, out, in, out,
    in, out) is nondet,
    in, in, out, in, out, in, out, in, out, in, out, in, out) is nondet.

%---------------------------------------------------%

    % foldr(Func, List, Start) = End calls Func with each element of List
    % (working right-to-left) and an accumulator (with the initial value
    % of Start), and returns the final value in End.
    %
:- func foldr(func(L, A) = A, list(L), A) = A.

    % foldr(Pred, List, Start, End) calls Pred with each element of List
    % (working right-to-left) and an accumulator (with the initial value
    % of Start), and returns the final value in End.
    %
:- pred foldr(pred(L, A, A), list(L), A, A).
:- mode foldr(pred(in, in, out) is det, in, in, out) is det.
:- mode foldr(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode foldr(pred(in, di, uo) is det, in, di, uo) is det.
:- mode foldr(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode foldr(pred(in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode foldr(pred(in, di, uo) is semidet, in, di, uo) is semidet.
:- mode foldr(pred(in, in, out) is multi, in, in, out) is multi.
:- mode foldr(pred(in, in, out) is nondet, in, in, out) is nondet.
:- mode foldr(pred(in, mdi, muo) is nondet, in, mdi, muo) is nondet.
:- mode foldr(pred(in, di, uo) is cc_multi, in, di, uo) is cc_multi.
:- mode foldr(pred(in, in, out) is cc_multi, in, in, out) is cc_multi.

    % foldr2(Pred, List, !Acc1, !Acc2):
    % Does the same job as foldr, but with two accumulators.
    % (Although no more expressive than foldl, this is often
    % a more convenient format, and a little more efficient).
    %
:- pred foldr2(pred(L, A, A, B, B), list(L), A, A, B, B).
:- mode foldr2(pred(in, in, out, in, out) is det, in, in, out,
    in, out) is det.
:- mode foldr2(pred(in, in, out, mdi, muo) is det, in, in, out,
    mdi, muo) is det.
:- mode foldr2(pred(in, in, out, di, uo) is det, in, in, out,
    di, uo) is det.
:- mode foldr2(pred(in, in, out, in, out) is semidet, in, in, out,
    in, out) is semidet.
:- mode foldr2(pred(in, in, out, mdi, muo) is semidet, in, in, out,
```

```
    mdi, muo) is semidet.
:- mode foldr2(pred(in, in, out, di, uo) is semidet, in, in, out,
    di, uo) is semidet.
:- mode foldr2(pred(in, in, out, in, out) is nondet, in, in, out,
    in, out) is nondet.
:- mode foldr2(pred(in, in, out, mdi, muo) is nondet, in, in, out,
    mdi, muo) is nondet.

    % foldr3(Pred, List, !Acc1, !Acc2, !Acc3):
    % Does the same job as foldr, but with two accumulators.
    % (Although no more expressive than foldl, this is often
    % a more convenient format, and a little more efficient).
    %
:- pred foldr3(pred(L, A, A, B, B, C, C), list(L), A, A, B, B, C, C).
:- mode foldr3(pred(in, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out) is det.
:- mode foldr3(pred(in, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, mdi, muo) is det.
:- mode foldr3(pred(in, in, out, in, out, di, uo) is det, in,
    in, out, in, out, di, uo) is det.
:- mode foldr3(pred(in, in, out, in, out, in, out) is semidet, in,
    in, out, in, out, in, out) is semidet.
:- mode foldr3(pred(in, in, out, in, out, mdi, muo) is semidet, in,
    in, out, in, out, mdi, muo) is semidet.
:- mode foldr3(pred(in, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, di, uo) is semidet.
:- mode foldr3(pred(in, in, out, in, out, in, out) is nondet, in,
    in, out, in, out, in, out) is nondet.
:- mode foldr3(pred(in, in, out, in, out, mdi, muo) is nondet, in,
    in, out, in, out, mdi, muo) is nondet.

%---------------------------------------------------%

    % foldl_corresponding(P, As, Bs, !Acc):
    %
    % Does the same job as foldl, but works on two lists in parallel.
    % An exception is raised if the list arguments differ in length.
    %
:- pred foldl_corresponding(pred(A, B, C, C), list(A), list(B), C, C).
:- mode foldl_corresponding(pred(in, in, in, out) is det,
    in, in, in, out) is det.
:- mode foldl_corresponding(pred(in, in, mdi, muo) is det,
    in, in, mdi, muo) is det.
:- mode foldl_corresponding(pred(in, in, di, uo) is det,
    in, in, di, uo) is det.
:- mode foldl_corresponding(pred(in, in, in, out) is semidet,
    in, in, in, out) is semidet.
```

```
:- mode foldl_corresponding(pred(in, in, mdi, muo) is semidet,
    in, in, mdi, muo) is semidet.
:- mode foldl_corresponding(pred(in, in, di, uo) is semidet,
    in, in, di, uo) is semidet.
:- mode foldl_corresponding(pred(in, in, in, out) is nondet,
    in, in, in, out) is nondet.
:- mode foldl_corresponding(pred(in, in, mdi, muo) is nondet,
    in, in, mdi, muo) is nondet.
:- mode foldl_corresponding(pred(in, in, in, out) is cc_multi,
    in, in, in, out) is cc_multi.
:- mode foldl_corresponding(pred(in, in, di, uo) is cc_multi,
    in, in, di, uo) is cc_multi.

:- func foldl_corresponding(func(A, B, C) = C, list(A), list(B), C) = C.

    % foldl2_corresponding(F, As, Bs, !Acc1, !Acc2):
    % Does the same job as foldl_corresponding, but has two
    % accumulators.
    %
:- pred foldl2_corresponding(pred(A, B, C, C, D, D), list(A), list(B),
    C, C, D, D).
:- mode foldl2_corresponding(pred(in, in, in, out, in, out) is det,
    in, in, in, out, in, out) is det.
:- mode foldl2_corresponding(pred(in, in, in, out, mdi, muo) is det,
    in, in, in, out, mdi, muo) is det.
:- mode foldl2_corresponding(pred(in, in, in, out, di, uo) is det,
    in, in, in, out, di, uo) is det.
:- mode foldl2_corresponding(pred(in, in, in, out, in, out) is semidet,
    in, in, in, out, in, out) is semidet.
:- mode foldl2_corresponding(pred(in, in, in, out, mdi, muo) is semidet,
    in, in, in, out, mdi, muo) is semidet.
:- mode foldl2_corresponding(pred(in, in, in, out, di, uo) is semidet,
    in, in, in, out, di, uo) is semidet.
:- mode foldl2_corresponding(pred(in, in, in, out, in, out) is nondet,
    in, in, in, out, in, out) is nondet.
:- mode foldl2_corresponding(pred(in, in, in, out, mdi, muo) is nondet,
    in, in, in, out, mdi, muo) is nondet.
:- mode foldl2_corresponding(pred(in, in, in, out, in, out) is cc_multi,
    in, in, in, out, in, out) is cc_multi.
:- mode foldl2_corresponding(pred(in, in, in, out, di, uo) is cc_multi,
    in, in, in, out, di, uo) is cc_multi.

    % foldl3_corresponding(F, As, Bs, !Acc1, !Acc2, !Acc3):
    % Does the same job as foldl_corresponding, but has three
    % accumulators.
    %
:- pred foldl3_corresponding(pred(A, B, C, C, D, D, E, E),
```

```
    list(A), list(B), C, C, D, D, E, E).
:- mode foldl3_corresponding(
    pred(in, in, in, out, in, out, in, out) is det, in, in, in, out,
    in, out, in, out) is det.
:- mode foldl3_corresponding(
    pred(in, in, in, out, in, out, mdi, muo) is det, in, in, in, out,
    in, out, mdi, muo) is det.
:- mode foldl3_corresponding(
    pred(in, in, in, out, in, out, di, uo) is det, in, in, in, out,
    in, out, di, uo) is det.
:- mode foldl3_corresponding(
    pred(in, in, in, out, in, out, in, out) is semidet, in, in, in, out,
    in, out, in, out) is semidet.
:- mode foldl3_corresponding(
    pred(in, in, in, out, in, out, mdi, muo) is semidet, in, in, in, out,
    in, out, mdi, muo) is semidet.
:- mode foldl3_corresponding(
    pred(in, in, in, out, in, out, di, uo) is semidet, in, in, in, out,
    in, out, di, uo) is semidet.

    % foldl_corresponding3(P, As, Bs, Cs, !Acc):
    % Like foldl_corresponding but folds over three corresponding
    % lists.
    %
:- pred foldl_corresponding3(pred(A, B, C, D, D),
    list(A), list(B), list(C), D, D).
:- mode foldl_corresponding3(pred(in, in, in, in, out) is det,
    in, in, in, in, out) is det.
:- mode foldl_corresponding3(pred(in, in, in, mdi, muo) is det,
    in, in, in, mdi, muo) is det.
:- mode foldl_corresponding3(pred(in, in, in, di, uo) is det,
    in, in, in, di, uo) is det.
:- mode foldl_corresponding3(pred(in, in, in, in, out) is semidet,
    in, in, in, in, out) is semidet.
:- mode foldl_corresponding3(pred(in, in, in, mdi, muo) is semidet,
    in, in, in, mdi, muo) is semidet.
:- mode foldl_corresponding3(pred(in, in, in, di, uo) is semidet,
    in, in, in, di, uo) is semidet.

    % foldl2_corresponding3(P, As, Bs, Cs, !Acc1, !Acc2):
    % like foldl_corresponding3 but with two accumulators.
    %
:- pred foldl2_corresponding3(pred(A, B, C, D, D, E, E),
    list(A), list(B), list(C), D, D, E, E).
:- mode foldl2_corresponding3(pred(in, in, in, in, out, in, out) is det,
    in, in, in, in, out, in, out) is det.
:- mode foldl2_corresponding3(pred(in, in, in, in, out, mdi, muo) is det,
```

```
    in, in, in, in, out, mdi, muo) is det.
:- mode foldl2_corresponding3(pred(in, in, in, in, out, di, uo) is det,
    in, in, in, in, out, di, uo) is det.
:- mode foldl2_corresponding3(
    pred(in, in, in, in, out, in, out) is semidet,
    in, in, in, in, out, in, out) is semidet.
:- mode foldl2_corresponding3(
    pred(in, in, in, in, out, mdi, muo) is semidet,
    in, in, in, in, out, mdi, muo) is semidet.
:- mode foldl2_corresponding3(
    pred(in, in, in, in, out, di, uo) is semidet,
    in, in, in, in, out, di, uo) is semidet.

    % foldl3_corresponding3(P, As, Bs, Cs, !Acc1, !Acc2, !Acc3):
    % like foldl_corresponding3 but with three accumulators.
    %
:- pred foldl3_corresponding3(pred(A, B, C, D, D, E, E, F, F),
    list(A), list(B), list(C), D, D, E, E, F, F).
:- mode foldl3_corresponding3(
    pred(in, in, in, in, out, in, out, in, out) is det,
    in, in, in, in, out, in, out, in, out) is det.
:- mode foldl3_corresponding3(
    pred(in, in, in, in, out, in, out, mdi, muo) is det,
    in, in, in, in, out, in, out, mdi, muo) is det.
:- mode foldl3_corresponding3(
    pred(in, in, in, in, out, in, out, di, uo) is det,
    in, in, in, in, out, in, out, di, uo) is det.
:- mode foldl3_corresponding3(
    pred(in, in, in, in, out, in, out, in, out) is semidet,
    in, in, in, in, out, in, out, in, out) is semidet.
:- mode foldl3_corresponding3(
    pred(in, in, in, in, out, in, out, mdi, muo) is semidet,
    in, in, in, in, out, in, out, mdi, muo) is semidet.
:- mode foldl3_corresponding3(
    pred(in, in, in, in, out, in, out, di, uo) is semidet,
    in, in, in, in, out, in, out, di, uo) is semidet.

    % foldl4_corresponding3(P, As, Bs, Cs, !Acc1, !Acc2, !Acc3, !Acc4):
    % like foldl_corresponding3 but with four accumulators.
    %
:- pred foldl4_corresponding3(pred(A, B, C, D, D, E, E, F, F, G, G),
    list(A), list(B), list(C), D, D, E, E, F, F, G, G).
:- mode foldl4_corresponding3(
    pred(in, in, in, in, out, in, out, in, out, in, out) is det,
    in, in, in, in, out, in, out, in, out, in, out) is det.
:- mode foldl4_corresponding3(
    pred(in, in, in, in, out, in, out, in, out, mdi, muo) is det,
```

```
    in, in, in, in, out, in, out, in, out, mdi, muo) is det.
:- mode foldl4_corresponding3(
    pred(in, in, in, in, out, in, out, in, out, di, uo) is det,
    in, in, in, in, out, in, out, in, out, di, uo) is det.
:- mode foldl4_corresponding3(
    pred(in, in, in, in, out, in, out, in, out, in, out) is semidet,
    in, in, in, in, out, in, out, in, out, in, out) is semidet.
:- mode foldl4_corresponding3(
    pred(in, in, in, in, out, in, out, in, out, mdi, muo) is semidet,
    in, in, in, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode foldl4_corresponding3(
    pred(in, in, in, in, out, in, out, in, out, di, uo) is semidet,
    in, in, in, in, out, in, out, in, out, di, uo) is semidet.

%---------------------------------------------%

    % map_foldl(Pred, InList, OutList, Start, End) calls Pred
    % with an accumulator (with the initial value of Start) on
    % each element of InList (working left-to-right) to transform
    % InList into OutList. The final value of the accumulator is
    % returned in End.
    %
:- pred map_foldl(pred(L, M, A, A), list(L), list(M), A, A).
:- mode map_foldl(pred(in, out, in, out) is det, in, out, in, out)
    is det.
:- mode map_foldl(pred(in, out, mdi, muo) is det, in, out, mdi, muo)
    is det.
:- mode map_foldl(pred(in, out, di, uo) is det, in, out, di, uo)
    is det.
:- mode map_foldl(pred(in, out, in, out) is semidet, in, out, in, out)
    is semidet.
:- mode map_foldl(pred(in, out, mdi, muo) is semidet, in, out, mdi, muo)
    is semidet.
:- mode map_foldl(pred(in, out, di, uo) is semidet, in, out, di, uo)
    is semidet.
:- mode map_foldl(pred(in, in, di, uo) is semidet, in, in, di, uo)
    is semidet.
:- mode map_foldl(pred(in, out, in, out) is nondet, in, out, in, out)
    is nondet.
:- mode map_foldl(pred(in, out, mdi, muo) is nondet, in, out, mdi, muo)
    is nondet.
:- mode map_foldl(pred(in, out, in, out) is cc_multi, in, out, in, out)
    is cc_multi.
:- mode map_foldl(pred(in, out, mdi, muo) is cc_multi, in, out, mdi, muo)
    is cc_multi.
:- mode map_foldl(pred(in, out, di, uo) is cc_multi, in, out, di, uo)
    is cc_multi.
```

```
    % Same as map_foldl, but with two accumulators.
    %
:- pred map_foldl2(pred(L, M, A, A, B, B), list(L), list(M), A, A, B, B).
:- mode map_foldl2(pred(in, out, in, out, in, out) is det,
    in, out, in, out, in, out) is det.
:- mode map_foldl2(pred(in, out, in, out, mdi, muo) is det,
    in, out, in, out, mdi, muo) is det.
:- mode map_foldl2(pred(in, out, in, out, di, uo) is det,
    in, out, in, out, di, uo) is det.
:- mode map_foldl2(pred(in, out, in, out, in, out) is semidet,
    in, out, in, out, in, out) is semidet.
:- mode map_foldl2(pred(in, out, in, out, mdi, muo) is semidet,
    in, out, in, out, mdi, muo) is semidet.
:- mode map_foldl2(pred(in, out, in, out, di, uo) is semidet,
    in, out, in, out, di, uo) is semidet.
:- mode map_foldl2(pred(in, in, in, out, di, uo) is semidet,
    in, in, in, out, di, uo) is semidet.
:- mode map_foldl2(pred(in, out, in, out, in, out) is cc_multi,
    in, out, in, out, in, out) is cc_multi.
:- mode map_foldl2(pred(in, out, in, out, mdi, muo) is cc_multi,
    in, out, in, out, mdi, muo) is cc_multi.
:- mode map_foldl2(pred(in, out, in, out, di, uo) is cc_multi,
    in, out, in, out, di, uo) is cc_multi.
:- mode map_foldl2(pred(in, out, in, out, in, out) is nondet,
    in, out, in, out, in, out) is nondet.

    % Same as map_foldl, but with three accumulators.
    %
:- pred map_foldl3(pred(L, M, A, A, B, B, C, C), list(L), list(M),
    A, A, B, B, C, C).
:- mode map_foldl3(pred(in, out, in, out, in, out, di, uo) is det,
    in, out, in, out, in, out, di, uo) is det.
:- mode map_foldl3(pred(in, out, in, out, in, out, in, out) is det,
    in, out, in, out, in, out, in, out) is det.
:- mode map_foldl3(pred(in, out, in, out, in, out, di, uo) is cc_multi,
    in, out, in, out, in, out, di, uo) is cc_multi.
:- mode map_foldl3(pred(in, out, in, out, in, out, in, out) is cc_multi,
    in, out, in, out, in, out, in, out) is cc_multi.
:- mode map_foldl3(pred(in, out, in, out, in, out, in, out) is semidet,
    in, out, in, out, in, out, in, out) is semidet.
:- mode map_foldl3(pred(in, out, in, out, in, out, in, out) is nondet,
    in, out, in, out, in, out, in, out) is nondet.

    % Same as map_foldl, but with four accumulators.
    %
:- pred map_foldl4(pred(L, M, A, A, B, B, C, C, D, D), list(L), list(M),
```

```
    A, A, B, B, C, C, D, D).
:- mode map_foldl4(pred(in, out, in, out, in, out, in, out, di, uo)
    is det,
    in, out, in, out, in, out, in, out, di, uo) is det.
:- mode map_foldl4(pred(in, out, in, out, in, out, in, out, in, out)
    is det,
    in, out, in, out, in, out, in, out, in, out) is det.
:- mode map_foldl4(pred(in, out, in, out, in, out, in, out, di, uo)
    is cc_multi,
    in, out, in, out, in, out, in, out, di, uo) is cc_multi.
:- mode map_foldl4(pred(in, out, in, out, in, out, in, out, in, out)
    is cc_multi,
    in, out, in, out, in, out, in, out, in, out) is cc_multi.
:- mode map_foldl4(pred(in, out, in, out, in, out, in, out, in, out)
    is semidet,
    in, out, in, out, in, out, in, out, in, out) is semidet.
:- mode map_foldl4(pred(in, out, in, out, in, out, in, out, in, out)
    is nondet,
    in, out, in, out, in, out, in, out, in, out) is nondet.

    % Same as map_foldl, but with five accumulators.
    %
:- pred map_foldl5(pred(L, M, A, A, B, B, C, C, D, D, E, E),
    list(L), list(M), A, A, B, B, C, C, D, D, E, E).
:- mode map_foldl5(pred(in, out, in, out, in, out, in, out, in, out,
    di, uo) is det,
    in, out, in, out, in, out, in, out, in, out, di, uo) is det.
:- mode map_foldl5(pred(in, out, in, out, in, out, in, out, in, out,
    in, out) is det,
    in, out, in, out, in, out, in, out, in, out, in, out) is det.
:- mode map_foldl5(pred(in, out, in, out, in, out, in, out, in, out,
    di, uo) is cc_multi,
    in, out, in, out, in, out, in, out, in, out, di, uo) is cc_multi.
:- mode map_foldl5(pred(in, out, in, out, in, out, in, out, in, out,
    in, out) is cc_multi,
    in, out, in, out, in, out, in, out, in, out, in, out) is cc_multi.
:- mode map_foldl5(pred(in, out, in, out, in, out, in, out, in, out,
    in, out) is semidet,
    in, out, in, out, in, out, in, out, in, out, in, out) is semidet.
:- mode map_foldl5(pred(in, out, in, out, in, out, in, out, in, out,
    in, out) is nondet,
    in, out, in, out, in, out, in, out, in, out, in, out) is nondet.

    % Same as map_foldl, but with six accumulators.
    %
:- pred map_foldl6(pred(L, M, A, A, B, B, C, C, D, D, E, E, F, F),
    list(L), list(M), A, A, B, B, C, C, D, D, E, E, F, F).
```

```
:- mode map_foldl6(pred(in, out, in, out, in, out, in, out, in, out,
    in, out, di, uo) is det,
    in, out, in, out, in, out, in, out, in, out, in, out, di, uo) is det.
:- mode map_foldl6(pred(in, out, in, out, in, out, in, out, in, out,
    in, out, in, out) is det,
    in, out, in, out, in, out, in, out, in, out, in, out, in, out) is det.
:- mode map_foldl6(pred(in, out, in, out, in, out, in, out, in, out,
    in, out, di, uo) is cc_multi,
    in, out, in, out, in, out, in, out, in, out, in, out, di, uo)
    is cc_multi.
:- mode map_foldl6(pred(in, out, in, out, in, out, in, out, in, out,
    in, out, in, out) is cc_multi,
    in, out, in, out, in, out, in, out, in, out, in, out, in, out)
    is cc_multi.
:- mode map_foldl6(pred(in, out, in, out, in, out, in, out, in, out,
    in, out, in, out) is semidet,
    in, out, in, out, in, out, in, out, in, out, in, out, in, out)
    is semidet.
:- mode map_foldl6(pred(in, out, in, out, in, out, in, out, in, out,
    in, out, in, out) is nondet,
    in, out, in, out, in, out, in, out, in, out, in, out, in, out)
    is nondet.

    % Same as map_foldl, but with two mapped outputs.
    %
:- pred map2_foldl(pred(L, M, N, A, A), list(L), list(M), list(N),
    A, A).
:- mode map2_foldl(pred(in, out, out, in, out) is det, in, out, out,
    in, out) is det.
:- mode map2_foldl(pred(in, out, out, mdi, muo) is det, in, out, out,
    mdi, muo) is det.
:- mode map2_foldl(pred(in, out, out, di, uo) is det, in, out, out,
    di, uo) is det.
:- mode map2_foldl(pred(in, out, out, in, out) is semidet, in, out, out,
    in, out) is semidet.
:- mode map2_foldl(pred(in, out, out, mdi, muo) is semidet, in, out, out,
    mdi, muo) is semidet.
:- mode map2_foldl(pred(in, out, out, di, uo) is semidet, in, out, out,
    di, uo) is semidet.
:- mode map2_foldl(pred(in, out, out, in, out) is nondet, in, out, out,
    in, out) is nondet.
:- mode map2_foldl(pred(in, out, out, mdi, muo) is nondet, in, out, out,
    mdi, muo) is nondet.
:- mode map2_foldl(pred(in, out, out, in, out) is cc_multi, in, out, out,
    in, out) is cc_multi.
:- mode map2_foldl(pred(in, out, out, di, uo) is cc_multi, in, out, out,
    di, uo) is cc_multi.
```

```
    % Same as map_foldl, but with two mapped outputs and two
    % accumulators.
    %
:- pred map2_foldl2(pred(L, M, N, A, A, B, B), list(L), list(M), list(N),
    A, A, B, B).
:- mode map2_foldl2(pred(in, out, out, in, out, di, uo) is det,
    in, out, out, in, out, di, uo) is det.
:- mode map2_foldl2(pred(in, out, out, in, out, in, out) is det,
    in, out, out, in, out, in, out) is det.
:- mode map2_foldl2(pred(in, out, out, in, out, di, uo) is cc_multi,
    in, out, out, in, out, di, uo) is cc_multi.
:- mode map2_foldl2(pred(in, out, out, in, out, in, out) is cc_multi,
    in, out, out, in, out, in, out) is cc_multi.
:- mode map2_foldl2(pred(in, out, out, in, out, in, out) is semidet,
    in, out, out, in, out, in, out) is semidet.
:- mode map2_foldl2(pred(in, out, out, in, out, in, out) is nondet,
    in, out, out, in, out, in, out) is nondet.

    % Same as map_foldl, but with two mapped outputs and three
    % accumulators.
    %
:- pred map2_foldl3(pred(L, M, N, A, A, B, B, C, C),
    list(L), list(M), list(N), A, A, B, B, C, C).
:- mode map2_foldl3(
    pred(in, out, out, in, out, in, out, in, out) is det,
    in, out, out, in, out, in, out, in, out) is det.
:- mode map2_foldl3(
    pred(in, out, out, in, out, in, out, di, uo) is det,
    in, out, out, in, out, in, out, di, uo) is det.
:- mode map2_foldl3(
    pred(in, out, out, in, out, in, out, in, out) is cc_multi,
    in, out, out, in, out, in, out, in, out) is cc_multi.
:- mode map2_foldl3(
    pred(in, out, out, in, out, in, out, di, uo) is cc_multi,
    in, out, out, in, out, in, out, di, uo) is cc_multi.
:- mode map2_foldl3(
    pred(in, out, out, in, out, in, out, in, out) is semidet,
    in, out, out, in, out, in, out, in, out) is semidet.
:- mode map2_foldl3(
    pred(in, out, out, in, out, in, out, in, out) is nondet,
    in, out, out, in, out, in, out, in, out) is nondet.

    % Same as map_foldl, but with two mapped outputs and four
    % accumulators.
    %
:- pred map2_foldl4(pred(L, M, N, A, A, B, B, C, C, D, D),
```

```
    list(L), list(M), list(N), A, A, B, B, C, C, D, D).
:- mode map2_foldl4(
    pred(in, out, out, in, out, in, out, in, out, in, out) is det,
    in, out, out, in, out, in, out, in, out, in, out) is det.
:- mode map2_foldl4(
    pred(in, out, out, in, out, in, out, in, out, di, uo) is det,
    in, out, out, in, out, in, out, in, out, di, uo) is det.
:- mode map2_foldl4(
    pred(in, out, out, in, out, in, out, in, out, in, out) is cc_multi,
    in, out, out, in, out, in, out, in, out, in, out) is cc_multi.
:- mode map2_foldl4(
    pred(in, out, out, in, out, in, out, in, out, di, uo) is cc_multi,
    in, out, out, in, out, in, out, in, out, di, uo) is cc_multi.
:- mode map2_foldl4(
    pred(in, out, out, in, out, in, out, in, out, in, out) is semidet,
    in, out, out, in, out, in, out, in, out, in, out) is semidet.
:- mode map2_foldl4(
    pred(in, out, out, in, out, in, out, in, out, in, out) is nondet,
    in, out, out, in, out, in, out, in, out, in, out) is nondet.

    % Same as map_foldl, but with three mapped outputs.
    %
:- pred map3_foldl(pred(L, M, N, O, A, A), list(L), list(M), list(N),
    list(O), A, A).
:- mode map3_foldl(pred(in, out, out, out, in, out) is det, in, out, out,
    out, in, out) is det.
:- mode map3_foldl(pred(in, out, out, out, mdi, muo) is det, in, out, out,
    out, mdi, muo) is det.
:- mode map3_foldl(pred(in, out, out, out, di, uo) is det, in, out, out,
    out, di, uo) is det.
:- mode map3_foldl(pred(in, out, out, out, in, out) is semidet, in, out,
    out, out, in, out) is semidet.
:- mode map3_foldl(pred(in, out, out, out, mdi, muo) is semidet, in, out,
    out, out, mdi, muo) is semidet.
:- mode map3_foldl(pred(in, out, out, out, di, uo) is semidet, in, out,
    out, out, di, uo) is semidet.
:- mode map3_foldl(pred(in, out, out, out, in, out) is nondet, in, out,
    out, out, in, out) is nondet.
:- mode map3_foldl(pred(in, out, out, out, mdi, muo) is nondet, in, out,
    out, out, mdi, muo) is nondet.
:- mode map3_foldl(pred(in, out, out, out, in, out) is cc_multi, in, out,
    out, out, in, out) is cc_multi.
:- mode map3_foldl(pred(in, out, out, out, di, uo) is cc_multi, in, out,
    out, out, di, uo) is cc_multi.

    % Same as map_foldl, but with three mapped outputs and two
    % accumulators.
```

```
    %
:- pred map3_foldl2(pred(L, M, N, O, A, A, B, B), list(L),
    list(M), list(N), list(O), A, A, B, B).
:- mode map3_foldl2(pred(in, out, out, out, in, out, di, uo) is det,
    in, out, out, out, in, out, di, uo) is det.
:- mode map3_foldl2(pred(in, out, out, out, in, out, in, out) is det,
    in, out, out, out, in, out, in, out) is det.
:- mode map3_foldl2(pred(in, out, out, out, in, out, di, uo) is cc_multi,
    in, out, out, out, in, out, di, uo) is cc_multi.
:- mode map3_foldl2(pred(in, out, out, out, in, out, in, out) is cc_multi,
    in, out, out, out, in, out, in, out) is cc_multi.
:- mode map3_foldl2(pred(in, out, out, out, in, out, in, out) is semidet,
    in, out, out, out, in, out, in, out) is semidet.
:- mode map3_foldl2(pred(in, out, out, out, in, out, in, out) is nondet,
    in, out, out, out, in, out, in, out) is nondet.

    % Same as map_foldl, but with four mapped outputs.
    %
:- pred map4_foldl(pred(L, M, N, O, P, A, A), list(L), list(M), list(N),
    list(O), list(P), A, A).
:- mode map4_foldl(pred(in, out, out, out, out, in, out) is det,
    in, out, out, out, out, in, out) is det.
:- mode map4_foldl(pred(in, out, out, out, out, mdi, muo) is det,
    in, out, out, out, out, mdi, muo) is det.
:- mode map4_foldl(pred(in, out, out, out, out, di, uo) is det,
    in, out, out, out, out, di, uo) is det.
:- mode map4_foldl(pred(in, out, out, out, out, in, out) is semidet,
    in, out, out, out, out, in, out) is semidet.
:- mode map4_foldl(pred(in, out, out, out, out, mdi, muo) is semidet,
    in, out, out, out, out, mdi, muo) is semidet.
:- mode map4_foldl(pred(in, out, out, out, out, di, uo) is semidet,
    in, out, out, out, out, di, uo) is semidet.
:- mode map4_foldl(pred(in, out, out, out, out, in, out) is nondet,
    in, out, out, out, out, in, out) is nondet.
:- mode map4_foldl(pred(in, out, out, out, out, mdi, muo) is nondet,
    in, out, out, out, out, mdi, muo) is nondet.
:- mode map4_foldl(pred(in, out, out, out, out, in, out) is cc_multi,
    in, out, out, out, out, in, out) is cc_multi.
:- mode map4_foldl(pred(in, out, out, out, out, di, uo) is cc_multi,
    in, out, out, out, out, di, uo) is cc_multi.

%-----------------------------------------------%

    % map_foldr(Pred, InList, OutList, Start, End) calls Pred
    % with an accumulator (with the initial value of Start) on
    % each element of InList (working right-to-left) to transform
    % InList into OutList. The final value of the accumulator is
```

```
    % returned in End.
    %
:- pred map_foldr(pred(L, M, A, A), list(L), list(M), A, A).
:- mode map_foldr(pred(in, out, in, out) is det, in, out, in, out)
    is det.
:- mode map_foldr(pred(in, out, mdi, muo) is det, in, out, mdi, muo)
    is det.
:- mode map_foldr(pred(in, out, di, uo) is det, in, out, di, uo)
    is det.
:- mode map_foldr(pred(in, out, in, out) is semidet, in, out, in, out)
    is semidet.
:- mode map_foldr(pred(in, out, mdi, muo) is semidet, in, out, mdi, muo)
    is semidet.
:- mode map_foldr(pred(in, out, di, uo) is semidet, in, out, di, uo)
    is semidet.
:- mode map_foldr(pred(in, in, di, uo) is semidet, in, in, di, uo)
    is semidet.

%-----------------------------------------------%

    % map_corresponding_foldl/6 is like map_corresponding except
    % that it has an accumulator threaded through it.
    %
:- pred map_corresponding_foldl(pred(A, B, C, D, D),
    list(A), list(B), list(C), D, D).
:- mode map_corresponding_foldl(pred(in, in, out, in, out) is det,
    in, in, out, in, out) is det.
:- mode map_corresponding_foldl(pred(in, in, out, mdi, muo) is det,
    in, in, out, mdi, muo) is det.
:- mode map_corresponding_foldl(pred(in, in, out, di, uo) is det,
    in, in, out, di, uo) is det.
:- mode map_corresponding_foldl(pred(in, in, out, in, out) is semidet,
    in, in, out, in, out) is semidet.
:- mode map_corresponding_foldl(pred(in, in, out, mdi, muo) is semidet,
    in, in, out, mdi, muo) is semidet.
:- mode map_corresponding_foldl(pred(in, in, out, di, uo) is semidet,
    in, in, out, di, uo) is semidet.

    % Like map_corresponding_foldl/6 except that it has two
    % accumulators.
    %
:- pred map_corresponding_foldl2(pred(A, B, C, D, D, E, E),
    list(A), list(B), list(C), D, D, E, E).
:- mode map_corresponding_foldl2(
    pred(in, in, out, in, out, in, out) is det, in, in, out, in, out,
    in, out) is det.
:- mode map_corresponding_foldl2(
```

```
    pred(in, in, out, in, out, mdi, muo) is det, in, in, out, in, out,
    mdi, muo) is det.
:- mode map_corresponding_foldl2(
    pred(in, in, out, in, out, di, uo) is det, in, in, out, in, out,
    di, uo) is det.
:- mode map_corresponding_foldl2(
    pred(in, in, out, in, out, in, out) is semidet, in, in, out, in, out,
    in, out) is semidet.
:- mode map_corresponding_foldl2(
    pred(in, in, out, in, out, mdi, muo) is semidet, in, in, out, in, out,
    mdi, muo) is semidet.
:- mode map_corresponding_foldl2(
    pred(in, in, out, in, out, di, uo) is semidet, in, in, out, in, out,
    di, uo) is semidet.

    % Like map_corresponding_foldl/6 except that it has three
    % accumulators.
    %
:- pred map_corresponding_foldl3(pred(A, B, C, D, D, E, E, F, F),
    list(A), list(B), list(C), D, D, E, E, F, F).
:- mode map_corresponding_foldl3(
    pred(in, in, out, in, out, in, out, in, out) is det, in, in, out, in, out,
    in, out, in, out) is det.
:- mode map_corresponding_foldl3(
    pred(in, in, out, in, out, in, out, mdi, muo) is det, in, in, out, in, out,
    in, out, mdi, muo) is det.
:- mode map_corresponding_foldl3(
    pred(in, in, out, in, out, in, out, di, uo) is det, in, in, out, in, out,
    in, out, di, uo) is det.
:- mode map_corresponding_foldl3(
    pred(in, in, out, in, out, in, out, in, out) is semidet, in, in, out,
    in, out, in, out, in, out) is semidet.
:- mode map_corresponding_foldl3(
    pred(in, in, out, in, out, in, out, mdi, muo) is semidet, in, in, out,
    in, out, in, out, mdi, muo) is semidet.
:- mode map_corresponding_foldl3(
    pred(in, in, out, in, out, in, out, di, uo) is semidet, in, in, out,
    in, out, in, out, di, uo) is semidet.

    % map_corresponding3_foldl/7 is like map_corresponding3 except
    % that it has an accumulator threaded through it.
    %
:- pred map_corresponding3_foldl(pred(A, B, C, D, E, E),
    list(A), list(B), list(C), list(D), E, E).
:- mode map_corresponding3_foldl(pred(in, in, in, out, in, out) is det,
    in, in, in, out, in, out) is det.
:- mode map_corresponding3_foldl(pred(in, in, in, out, mdi, muo) is det,
```

```
    in, in, in, out, mdi, muo) is det.
:- mode map_corresponding3_foldl(pred(in, in, in, out, di, uo) is det,
    in, in, in, out, di, uo) is det.
:- mode map_corresponding3_foldl(
    pred(in, in, in, out, in, out) is semidet,
    in, in, in, out, in, out) is semidet.
:- mode map_corresponding3_foldl(
    pred(in, in, in, out, mdi, muo) is semidet,
    in, in, in, out, mdi, muo) is semidet.
:- mode map_corresponding3_foldl(
    pred(in, in, in, out, di, uo) is semidet,
    in, in, in, out, di, uo) is semidet.

%---------------------------------------------------%

    % filter_map_foldl(Transformer, List, TrueList, Start, End):
    % Takes a predicate with one input argument, one output argument and an
    % accumulator. It is called with each element of List. If a call succeeds,
    % then the output is included in TrueList and the accumulator is updated.
    %
:- pred filter_map_foldl(pred(X, Y, A, A)::in(pred(in, out, in, out)
    is semidet), list(X)::in, list(Y)::out, A::in, A::out) is det.

%---------------------------------------------------%
%---------------------------------------------------%
```

46 map

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 1993-2012 The University of Melbourne.
% Copyright (C) 2013-2015, 2017-2020 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: map.m.
% Main author: fjh, conway.
% Stability: high.
%
% This file provides the 'map' abstract data type.
%
% A map (also known as a dictionary or an associative array) is a collection
% of (Key, Value) pairs that allows you to look up any Value given its Key.
% Each Key has exactly only one corresponding Value. (If you want the ability
```

```
% to store more than one Value for a given Key, use either multi_map.m
% or one_or_more_map.m.)
%
% The implementation uses balanced 2-3-4 trees, as provided by tree234.m.
% Virtually all the predicates in this file just forward the work
% to the corresponding predicate in tree234.m.
%
% Note: 2-3-4 trees do not have a canonical representation for any given map.
% This means that two maps that represent the same set of key-value pairs
% may have different internal representations, and that therefore they
% may fail to unify and may compare as unequal. The reason for the difference
% in the internal representation is usually that the (Key, Value) pairs were
% inserted into the two maps in different orders, or that the two maps
% have a different history of deletions. If you want to know whether
% two maps contain the same set of (Key, Data) pairs, use the map.equal/2
% predicate below.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module map.
:- interface.

:- import_module assoc_list.
:- import_module list.
:- import_module maybe.
:- import_module set.

%-----------------------------------------------------%

:- type map(_K, _V).

%-----------------------------------------------------%
%
% Initial creation of maps.
%

    % Create an empty map.
    %
:- func init = (map(K, V)::uo) is det.
:- pred init(map(_, _)::uo) is det.

    % Create a map containing only the given key-value pair.
    %
:- func singleton(K, V) = map(K, V).

%-----------------------------------------------------%
```

```
%
% Emptiness tests.
%

    % Check whether a map is empty.
    %
:- pred is_empty(map(_, _)::in) is semidet.

%---------------------------------------------------%
%
% Searching for a key.
%

    % Succeed iff the map contains the given key.
    %
:- pred contains(map(K, _V)::in, K::in) is semidet.

    % Return the value associated with the given key in the map.
    % Fail if the map does not contain that key.
    %
:- func search(map(K, V), K) = V is semidet.
:- pred search(map(K, V)::in, K::in, V::out) is semidet.

    % Return the value associated with the given key in the map.
    % Throw an exception if the map does not contain that key.
    %
:- func lookup(map(K, V), K) = V.
:- pred lookup(map(K, V)::in, K::in, V::out) is det.

    % Search the map for key-value pairs with the given value.
    %
:- pred inverse_search(map(K, V)::in, V::in, K::out) is nondet.

    % Search for a key-value pair using the key. If there is no entry
    % for the given key, returns the pair for the next lower key instead.
    % Fails if there is no key with the given or lower value.
    %
:- pred lower_bound_search(map(K, V)::in, K::in, K::out, V::out)
    is semidet.

    % Search for a key-value pair using the key. If there is no entry
    % for the given key, returns the pair for the next lower key instead.
    % Throws an exception if there is no key with the given or lower value.
    %
:- pred lower_bound_lookup(map(K, V)::in, K::in, K::out, V::out) is det.

    % Search for a key-value pair using the key. If there is no entry
```

```
    % for the given key, returns the pair for the next higher key instead.
    % Fails if there is no key with the given or higher value.
    %
:- pred upper_bound_search(map(K, V)::in, K::in, K::out, V::out)
    is semidet.

    % Search for a key-value pair using the key. If there is no entry
    % for the given key, returns the pair for the next higher key instead.
    % Throws an exception if there is no key with the given or higher value.
    %
:- pred upper_bound_lookup(map(K, V)::in, K::in, K::out, V::out) is det.

%-----------------------------------------------------%
%
% Looking for the minimum and maximum keys.
%

    % Return the largest key in the map, if there is one.
    %
:- func max_key(map(K, V)) = K is semidet.

    % As above, but throw an exception if there is no largest key.
    %
:- func det_max_key(map(K, V)) = K.

    % Return the smallest key in the map, if there is one.
    %
:- func min_key(map(K,V)) = K is semidet.

    % As above, but throw an exception if there is no smallest key.
    %
:- func det_min_key(map(K, V)) = K.

%-----------------------------------------------------%
%
% Insertions and deletions.
%

    % Insert a new key and corresponding value into a map.
    % Fail if the key already exists.
    %
:- func insert(map(K, V), K, V) = map(K, V) is semidet.
:- pred insert(K::in, V::in, map(K, V)::in, map(K, V)::out) is semidet.

    % Insert a new key and corresponding value into a map.
    % Throw an exception if the key already exists.
    %
```

```
:- func det_insert(map(K, V), K, V) = map(K, V).
:- pred det_insert(K::in, V::in, map(K, V)::in, map(K, V)::out) is det.

    % Apply det_insert to key - value pairs from corresponding lists.
    %
:- func det_insert_from_corresponding_lists(map(K, V), list(K), list(V))
    = map(K, V).
:- pred det_insert_from_corresponding_lists(list(K)::in,
    list(V)::in, map(K, V)::in, map(K, V)::out) is det.

    % Apply det_insert to key - value pairs from an assoc_list.
    %
:- func det_insert_from_assoc_list(map(K, V), assoc_list(K, V)) = map(K, V).
:- pred det_insert_from_assoc_list(assoc_list(K, V)::in,
    map(K, V)::in, map(K, V)::out) is det.

%---------------------------------------------------%

    % search_insert(K, V, MaybeOldV, !Map):
    %
    % Search for the key K in the map. If the key is already in the map,
    % with corresponding value OldV, set MaybeOldV to yes(OldV). If it
    % is not in the map, then insert it into the map with value V,
    % and set MaybeOldV to no.
    %
:- pred search_insert(K::in, V::in, maybe(V)::out,
    map(K, V)::in, map(K, V)::out) is det.

%---------------------------------------------------%

    % Update the value corresponding to a given key
    % Fail if the key doesn't already exist.
    %
:- func update(map(K, V), K, V) = map(K, V) is semidet.
:- pred update(K::in, V::in, map(K, V)::in, map(K, V)::out) is semidet.

    % Update the value corresponding to a given key
    % Throw an exception if the key doesn't already exist.
    %
:- func det_update(map(K, V), K, V) = map(K, V).
:- pred det_update(K::in, V::in, map(K, V)::in, map(K, V)::out) is det.

%---------------------------------------------------%

    % If the key is already present update its corresponding value.
    % If the key is not present, insert it with the given value.
    %
```

```
:- func set(map(K, V), K, V) = map(K, V).
:- pred set(K::in, V::in, map(K, V)::in, map(K, V)::out) is det.

    % Apply set to key - value pairs from corresponding lists.
    %
:- func set_from_corresponding_lists(map(K, V), list(K), list(V)) = map(K, V).
:- pred set_from_corresponding_lists(list(K)::in, list(V)::in,
    map(K, V)::in, map(K, V)::out) is det.

    % Apply set to key - value pairs from an assoc_list.
    %
:- func set_from_assoc_list(map(K, V), assoc_list(K, V)) = map(K, V).
:- pred set_from_assoc_list(assoc_list(K, V)::in,
    map(K, V)::in, map(K, V)::out) is det.

%-----------------------------------------------%

    % Delete a key-value pair from a map.
    % If the key is not present, leave the map unchanged.
    %
:- func delete(map(K, V), K) = map(K, V).
:- pred delete(K::in, map(K, V)::in, map(K, V)::out) is det.

    % Apply delete/3 to a list of keys.
    %
:- func delete_list(map(K, V), list(K)) = map(K, V).
:- pred delete_list(list(K)::in, map(K, V)::in, map(K, V)::out) is det.

    % Apply delete/3 to a sorted list of keys. The fact that the list
    % is sorted may make this more efficient. (If the list is not sorted,
    % the predicate or function will either throw an exception or return
    % incorrect output.)
    %
:- func delete_sorted_list(map(K, V), list(K)) = map(K, V).
:- pred delete_sorted_list(list(K)::in, map(K, V)::in, map(K, V)::out) is det.

%-----------------------------------------------%

    % Delete a key-value pair from a map and return the value.
    % Fail if the key is not present.
    %
:- pred remove(K::in, V::out, map(K, V)::in, map(K, V)::out) is semidet.

    % Delete a key-value pair from a map and return the value.
    % Throw an exception if the key is not present.
    %
:- pred det_remove(K::in, V::out, map(K, V)::in, map(K, V)::out) is det.
```

```
    % Remove the smallest item from the map, fail if the map is empty.
    %
:- pred remove_smallest(K::out, V::out, map(K, V)::in, map(K, V)::out)
    is semidet.

%-----------------------------------------------------%
%
% Field selection for maps.
%

    % Map ^ elem(Key) = search(Map, Key).
    %
:- func elem(K, map(K, V)) = V is semidet.

    % Map ^ det_elem(Key) = lookup(Map, Key).
    %
:- func det_elem(K, map(K, V)) = V.

    % Field update for maps.

    % (Map ^ elem(Key) := Value) = set(Map, Key, Value).
    %
:- func 'elem :='(K, map(K, V), V) = map(K, V).

    % (Map ^ det_elem(Key) := Value) = det_update(Map, Key, Value).
    %
:- func 'det_elem :='(K, map(K, V), V) = map(K, V).

%-----------------------------------------------------%
%
% Returning keys and values.
%

    % Return all the keys in the map, and their corresponding values,
    % one key-value pair at a time.
    %
:- pred member(map(K, V)::in, K::out, V::out) is nondet.

    % Given a map, return a list of all the keys in the map.
    %
:- func keys(map(K, _V)) = list(K).
:- pred keys(map(K, _V)::in, list(K)::out) is det.

    % Given a map, return a list of all the keys in the map,
    % in sorted order.
    %
```

```
:- func sorted_keys(map(K, _V)) = list(K).
:- pred sorted_keys(map(K, _V)::in, list(K)::out) is det.

    % Given a map, return a list of all the keys in the map,
    % as a set.
    %
:- func keys_as_set(map(K, _V)) = set(K).
:- pred keys_as_set(map(K, _V)::in, set(K)::out) is det.

    % Given a map, return a list of all the values in the map.
    %
:- func values(map(_K, V)) = list(V).
:- pred values(map(_K, V)::in, list(V)::out) is det.

:- pred keys_and_values(map(K, V)::in, list(K)::out, list(V)::out) is det.

%-----------------------------------------------%
%
% Operations on values.
%

    % Update the value at the given key by applying the supplied
    % transformation to it. Fails if the key is not found. This is faster
    % than first searching for the value and then updating it.
    %
:- pred transform_value(pred(V, V)::in(pred(in, out) is det), K::in,
    map(K, V)::in, map(K, V)::out) is semidet.

    % Same as transform_value/4, but throws an exception if the key is not
    % found.
    %
:- func det_transform_value(func(V) = V, K, map(K, V)) = map(K, V).
:- pred det_transform_value(pred(V, V)::in(pred(in, out) is det), K::in,
    map(K, V)::in, map(K, V)::out) is det.

%-----------------------------------------------%
%
% Converting maps to lists.
%

    % Convert an association list to a map.
    %
:- func from_assoc_list(assoc_list(K, V)) = map(K, V).
:- pred from_assoc_list(assoc_list(K, V)::in, map(K, V)::out) is det.

    % Convert a sorted association list with no duplicated keys to a map.
    %
```

```
:- func from_sorted_assoc_list(assoc_list(K, V)) = map(K, V).
:- pred from_sorted_assoc_list(assoc_list(K, V)::in, map(K, V)::out) is det.

    % Convert a reverse sorted association list with no duplicated keys
    % to a map.
    %
:- func from_rev_sorted_assoc_list(assoc_list(K, V)) = map(K, V).
:- pred from_rev_sorted_assoc_list(assoc_list(K, V)::in, map(K, V)::out)
    is det.

    % Convert a pair of lists (which must be of the same length) to a map.
    %
:- func from_corresponding_lists(list(K), list(V)) = map(K, V).
:- pred from_corresponding_lists(list(K)::in, list(V)::in, map(K, V)::out)
    is det.

%---------------------------------------------------%
%
% Converting lists to maps.
%

    % Convert a map to an association list.
    %
:- func to_assoc_list(map(K, V)) = assoc_list(K, V).
:- pred to_assoc_list(map(K, V)::in, assoc_list(K, V)::out) is det.

    % Convert a map to an association list which is sorted on the keys.
    %
:- func to_sorted_assoc_list(map(K, V)) = assoc_list(K, V).
:- pred to_sorted_assoc_list(map(K, V)::in, assoc_list(K, V)::out) is det.

%---------------------------------------------------%
%
% Reversing a map.
%

    % Consider the original map a set of key-value pairs. This predicate
    % returns a map that maps each value to the set of keys it is paired with
    % in the original map.
    %
:- func reverse_map(map(K, V)) = map(V, set(K)).

%---------------------------------------------------%
%
% Selecting subsets of maps.
%
```

```
    % select takes a map and a set of keys, and returns a map
    % containing the keys in the set and their corresponding values.
    %
:- func select(map(K, V), set(K)) = map(K, V).
:- pred select(map(K, V)::in, set(K)::in, map(K, V)::out) is det.

    % select_sorted_list takes a map and a sorted list of keys without
    % duplicates, and returns a map containing the keys in the list
    % and their corresponding values.
    %
:- func select_sorted_list(map(K, V), list(K)) = map(K, V).
:- pred select_sorted_list(map(K, V)::in, list(K)::in, map(K, V)::out) is det.

    % select_unselect takes a map and a set of keys, and returns two maps:
    % the first containing the keys in the set and their corresponding values,
    % the second containing the keys NOT in the set and their corresponding
    % values.
    %
:- pred select_unselect(map(K, V)::in, set(K)::in,
    map(K, V)::out, map(K, V)::out) is det.

    % select_unselect_sorted_list takes a map and a sorted list of keys
    % without duplicates, and returns two maps:
    % the first containing the keys in the list and their corresponding values,
    % the second containing the keys NOT in the list and their corresponding
    % values.
    %
:- pred select_unselect_sorted_list(map(K, V)::in, list(K)::in,
    map(K, V)::out, map(K, V)::out) is det.

%-----------------------------------------------------%
%
% Selecting subsets of values.
%

    % Given a list of keys, produce a list of their corresponding
    % values in a specified map.
    %
:- func apply_to_list(list(K), map(K, V)) = list(V).
:- pred apply_to_list(list(K)::in, map(K, V)::in, list(V)::out) is det.

%-----------------------------------------------------%
%
% Operations on two or more maps.
%

    % Merge the contents of the two maps.
```

```
    % Throws an exception if both sets of keys are not disjoint.
    %
    % The cost of this predicate is proportional to the number of elements
    % in the second map, so for efficiency, you want to put the bigger map
    % first and the smaller map second.
    %
:- func merge(map(K, V), map(K, V)) = map(K, V).
:- pred merge(map(K, V)::in, map(K, V)::in, map(K, V)::out) is det.

    % For overlay(MapA, MapB, Map), if MapA and MapB both contain the
    % same key, then Map will map that key to the value from MapB.
    % In other words, MapB takes precedence over MapA.
    %
:- func overlay(map(K, V), map(K, V)) = map(K, V).
:- pred overlay(map(K, V)::in, map(K, V)::in, map(K, V)::out) is det.

    % overlay_large_map(MapA, MapB, Map) performs the same task as
    % overlay(MapA, MapB, Map). However, while overlay takes time
    % proportional to the size of MapB, overlay_large_map takes time
    % proportional to the size of MapA. In other words, it preferable when
    % MapB is the larger map.
    %
:- func overlay_large_map(map(K, V), map(K, V)) = map(K, V).
:- pred overlay_large_map(map(K, V)::in, map(K, V)::in, map(K, V)::out)
    is det.

%-----------------------------------------------%

    % Given two maps M1 and M2, create a third map M3 that has only the
    % keys that occur in both M1 and M2. For keys that occur in both M1
    % and M2, compute the corresponding values. If they are the same,
    % include the key/value pair in M3. If they differ, do not include the
    % key in M3.
    %
    % This predicate effectively considers the input maps to be sets of
    % key/value pairs, computes the intersection of those two sets, and
    % returns the map corresponding to the intersection.
    %
    % common_subset is very similar to intersect, but can succeed
    % even with an output map that does not contain an entry for a key
    % value that occurs in both input maps.
    %
:- func common_subset(map(K, V), map(K, V)) = map(K, V).

    % Given two maps M1 and M2, create a third map M3 that has only the
    % keys that occur in both M1 and M2. For keys that occur in both M1
    % and M2, compute the value in the final map by applying the supplied
```

```
    % predicate to the values associated with the key in M1 and M2.
    % Fail if and only if this predicate fails on the values associated
    % with some common key.
    %
:- func intersect(func(V, V) = V, map(K, V), map(K, V)) = map(K, V).

:- pred intersect(pred(V, V, V), map(K, V), map(K, V), map(K, V)).
:- mode intersect(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode intersect(pred(in, in, out) is det, in, in, out) is det.

    % Calls intersect. Throws an exception if intersect fails.
    %
:- func det_intersect((func(V, V) = V)::in(func(in, in) = out is semidet),
    map(K, V)::in, map(K, V)::in) = (map(K, V)::out) is det.
:- pred det_intersect((pred(V, V, V))::in(pred(in, in, out) is semidet),
    map(K, V)::in, map(K, V)::in, map(K, V)::out) is det.

    % intersect_list(Pred, M, Ms, ResultM):
    %
    % Take the non-empty list of maps [M | Ms], and intersect pairs of
    % those maps (using map.intersect above) until there is only one map left.
    % Return this map as ResultM. The order of in which those intersect
    % operations are performed is not defined, so the caller should choose
    % a Pred for which the order does not matter.
    %
:- pred intersect_list(pred(V, V, V), map(K, V), list(map(K, V)), map(K, V)).
:- mode intersect_list(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode intersect_list(pred(in, in, out) is det, in, in, out) is det.

%-----------------------------------------------%

    % Given two maps M1 and M2, create a third map M3 that contains all
    % the keys that occur in either M1 and M2. For keys that occur in both M1
    % and M2, compute the value in the final map by applying the supplied
    % closure to the values associated with the key in M1 and M2.
    % Fail if and only if this closure fails on the values associated
    % with some common key.
    %
:- func union(func(V, V) = V, map(K, V), map(K, V)) = map(K, V).
:- pred union(pred(V, V, V), map(K, V), map(K, V), map(K, V)).
:- mode union(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode union(pred(in, in, out) is det, in, in, out) is det.

    % Calls union. Throws an exception if union fails.
    %
:- func det_union((func(V, V) = V)::in(func(in, in) = out is semidet),
    map(K, V)::in, map(K, V)::in) = (map(K, V)::out) is det.
```

```
:- pred det_union(pred(V, V, V)::in(pred(in, in, out) is semidet),
    map(K, V)::in, map(K, V)::in, map(K, V)::out) is det.

    % union_list(Pred, M, Ms, ResultM):
    %
    % Take the non-empty list of maps [M | Ms], and union pairs of those maps
    % (using union above) until there is only one map left. Return this map
    % as ResultM. The order of in which those union operations are performed
    % is not defined, so the caller should choose a Pred for which the order
    % does not matter.
    %
:- pred union_list(pred(V, V, V), map(K, V), list(map(K, V)), map(K, V)).
:- mode union_list(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode union_list(pred(in, in, out) is det, in, in, out) is det.

%---------------------------------------------------%
%
% Counting.
%

    % Count the number of elements in the map.
    %
:- func count(map(K, V)) = int.
:- pred count(map(K, V)::in, int::out) is det.

%---------------------------------------------------%
%
% Comparisons between maps.
%

    % True if both maps have the same set of key-value pairs, regardless of
    % how the maps were constructed.
    %
    % Unifying maps does not work as one might expect, because the internal
    % structures of two maps that contain the same set of key-value pairs
    % may be different.
    %
:- pred equal(map(K, V)::in, map(K, V)::in) is semidet.

%---------------------------------------------------%
%
% Optimization.
%

    % Declaratively, a no-operation.
    % Operationally, a suggestion that the implementation
    % optimize the representation of the map in the expectation
```

```
    % of a number of lookups but few or no modifications.
    %
    % This operation is here only for "cultural compatibility"
    % with the modules that operation on trees that may be unbalanced.
    % 2-3-4 trees are always guaranteed to be balanced, so they do not need
    % any such optimization.
    %
:- func optimize(map(K, V)) = map(K, V).
:- pred optimize(map(K, V)::in, map(K, V)::out) is det.

%-----------------------------------------------------%
%
% Standard higher order functions on collections.
%


    % Perform an inorder traversal of the map, applying
    % an accumulator predicate for each key-value pair.
    %
:- func foldl(func(K, V, A) = A, map(K, V), A) = A.
:- pred foldl(pred(K, V, A, A), map(K, V), A, A).
:- mode foldl(pred(in, in, in, out) is det, in, in, out) is det.
:- mode foldl(pred(in, in, mdi, muo) is det, in, mdi, muo) is det.
:- mode foldl(pred(in, in, di, uo) is det, in, di, uo) is det.
:- mode foldl(pred(in, in, in, out) is semidet, in, in, out) is semidet.
:- mode foldl(pred(in, in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode foldl(pred(in, in, di, uo) is semidet, in, di, uo) is semidet.
:- mode foldl(pred(in, in, in, out) is cc_multi, in, in, out) is cc_multi.
:- mode foldl(pred(in, in, di, uo) is cc_multi, in, di, uo) is cc_multi.
:- mode foldl(pred(in, in, mdi, muo) is cc_multi, in, mdi, muo) is cc_multi.

    % Perform an inorder traversal of the map, applying an accumulator
    % predicate with two accumulators for each key-value pair.
    % (Although no more expressive than foldl, this is often
    % a more convenient format, and a little more efficient).
    %
:- pred foldl2(pred(K, V, A, A, B, B), map(K, V), A, A, B, B).
:- mode foldl2(pred(in, in, in, out, in, out) is det,
    in, in, out, in, out) is det.
:- mode foldl2(pred(in, in, in, out, mdi, muo) is det,
    in, in, out, mdi, muo) is det.
:- mode foldl2(pred(in, in, in, out, di, uo) is det,
    in, in, out, di, uo) is det.
:- mode foldl2(pred(in, in, di, uo, di, uo) is det,
    in, di, uo, di, uo) is det.
:- mode foldl2(pred(in, in, in, out, in, out) is semidet,
    in, in, out, in, out) is semidet.
```

```
:- mode foldl2(pred(in, in, in, out, mdi, muo) is semidet,
    in, in, out, mdi, muo) is semidet.
:- mode foldl2(pred(in, in, in, out, di, uo) is semidet,
    in, in, out, di, uo) is semidet.
:- mode foldl2(pred(in, in, in, out, in, out) is cc_multi,
    in, in, out, in, out) is cc_multi.
:- mode foldl2(pred(in, in, in, out, mdi, muo) is cc_multi,
    in, in, out, mdi, muo) is cc_multi.
:- mode foldl2(pred(in, in, in, out, di, uo) is cc_multi,
    in, in, out, di, uo) is cc_multi.
:- mode foldl2(pred(in, in, di, uo, di, uo) is cc_multi,
    in, di, uo, di, uo) is cc_multi.

    % Perform an inorder traversal of the map, applying an accumulator
    % predicate with three accumulators for each key-value pair.
    % (Although no more expressive than foldl, this is often
    % a more convenient format, and a little more efficient).
    %
:- pred foldl3(pred(K, V, A, A, B, B, C, C), map(K, V), A, A, B, B, C, C).
:- mode foldl3(pred(in, in, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out) is det.
:- mode foldl3(pred(in, in, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, mdi, muo) is det.
:- mode foldl3(pred(in, in, in, out, in, out, di, uo) is det,
    in, in, out, in, out, di, uo) is det.
:- mode foldl3(pred(in, in, in, out, di, uo, di, uo) is det,
    in, in, out, di, uo, di, uo) is det.
:- mode foldl3(pred(in, in, di, uo, di, uo, di, uo) is det,
    in, di, uo, di, uo, di, uo) is det.
:- mode foldl3(pred(in, in, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out) is semidet.
:- mode foldl3(pred(in, in, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, mdi, muo) is semidet.
:- mode foldl3(pred(in, in, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, di, uo) is semidet.

    % Perform an inorder traversal of the map, applying an accumulator
    % predicate with four accumulators for each key-value pair.
    % (Although no more expressive than foldl, this is often
    % a more convenient format, and a little more efficient).
    %
:- pred foldl4(pred(K, V, A, A, B, B, C, C, D, D), map(K, V),
    A, A, B, B, C, C, D, D).
:- mode foldl4(pred(in, in, in, out, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out, in, out) is det.
:- mode foldl4(pred(in, in, in, out, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, in, out, mdi, muo) is det.
```

```
:- mode foldl4(pred(in, in, in, out, in, out, in, out, di, uo) is det,
    in, in, out, in, out, in, out, di, uo) is det.
:- mode foldl4(pred(in, in, in, out, in, out, di, uo, di, uo) is det,
    in, in, out, in, out, di, uo, di, uo) is det.
:- mode foldl4(pred(in, in, in, out, di, uo, di, uo, di, uo) is det,
    in, in, out, di, uo, di, uo, di, uo) is det.
:- mode foldl4(pred(in, in, di, uo, di, uo, di, uo, di, uo) is det,
    in, di, uo, di, uo, di, uo, di, uo) is det.
:- mode foldl4(pred(in, in, in, out, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out, in, out) is semidet.
:- mode foldl4(pred(in, in, in, out, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode foldl4(pred(in, in, in, out, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, in, out, di, uo) is semidet.

    % Perform an inorder traversal of the map, applying an accumulator
    % predicate with five accumulators for each key-value pair.
    % (Although no more expressive than foldl, this is often
    % a more convenient format, and a little more efficient).
    %
:- pred foldl5(pred(K, V, A, A, B, B, C, C, D, D, E, E), map(K, V),
    A, A, B, B, C, C, D, D, E, E).
:- mode foldl5(pred(in, in, in, out, in, out, in, out, in, out, in, out)
    is det,
    in, in, out, in, out, in, out, in, out, in, out) is det.
:- mode foldl5(pred(in, in, in, out, in, out, in, out, in, out, mdi, muo)
    is det,
    in, in, out, in, out, in, out, in, out, mdi, muo) is det.
:- mode foldl5(pred(in, in, in, out, in, out, in, out, in, out, di, uo)
    is det,
    in, in, out, in, out, in, out, in, out, di, uo) is det.
:- mode foldl5(pred(in, in, in, out, in, out, in, out, in, out, in, out)
    is semidet,
    in, in, out, in, out, in, out, in, out, in, out) is semidet.
:- mode foldl5(pred(in, in,in, out,  in, out, in, out, in, out, mdi, muo)
    is semidet,
    in, in, out, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode foldl5(pred(in, in, in, out, in, out, in, out, in, out, di, uo)
    is semidet,
    in, in, out, in, out, in, out, in, out, di, uo) is semidet.

%-----------------------------------------------%

    % Perform an inorder traversal by key of the map, applying an accumulator
    % predicate for value.
    %
:- pred foldl_values(pred(V, A, A), map(K, V), A, A).
```

```
:- mode foldl_values(pred(in, in, out) is det, in, in, out) is det.
:- mode foldl_values(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode foldl_values(pred(in, di, uo) is det, in, di, uo) is det.
:- mode foldl_values(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode foldl_values(pred(in, mdi, muo) is semidet, in, mdi, muo)
    is semidet.
:- mode foldl_values(pred(in, di, uo) is semidet, in, di, uo) is semidet.
:- mode foldl_values(pred(in, in, out) is cc_multi, in, in, out)
    is cc_multi.
:- mode foldl_values(pred(in, di, uo) is cc_multi, in, di, uo) is cc_multi.
:- mode foldl_values(pred(in, mdi, muo) is cc_multi, in, mdi, muo)
    is cc_multi.

    % As above, but with two accumulators.
    %
:- pred foldl2_values(pred(V, A, A, B, B), map(K, V), A, A, B, B).
:- mode foldl2_values(pred(in, in, out, in, out) is det, in,
    in, out, in, out) is det.
:- mode foldl2_values(pred(in, in, out, mdi, muo) is det, in,
    in, out, mdi, muo) is det.
:- mode foldl2_values(pred(in, in, out, di, uo) is det, in,
    in, out, di, uo) is det.
:- mode foldl2_values(pred(in, in, out, in, out) is semidet, in,
    in, out, in, out) is semidet.
:- mode foldl2_values(pred(in, in, out, mdi, muo) is semidet, in,
    in, out, mdi, muo) is semidet.
:- mode foldl2_values(pred(in, in, out, di, uo) is semidet, in,
    in, out, di, uo) is semidet.
:- mode foldl2_values(pred(in, in, out, in, out) is cc_multi, in,
    in, out, in, out) is cc_multi.
:- mode foldl2_values(pred(in, in, out, mdi, muo) is cc_multi, in,
    in, out, mdi, muo) is cc_multi.
:- mode foldl2_values(pred(in, in, out, di, uo) is cc_multi, in,
    in, out, di, uo) is cc_multi.

    % As above, but with three accumulators.
    %
:- pred foldl3_values(pred(V, A, A, B, B, C, C), map(K, V),
    A, A, B, B, C, C).
:- mode foldl3_values(pred(in, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out) is det.
:- mode foldl3_values(pred(in, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, mdi, muo) is det.
:- mode foldl3_values(pred(in, in, out, in, out, di, uo) is det, in,
    in, out, in, out, di, uo) is det.
:- mode foldl3_values(pred(in, in, out, in, out, in, out) is semidet, in,
    in, out, in, out, in, out) is semidet.
```

```
:- mode foldl3_values(pred(in, in, out, in, out, mdi, muo) is semidet, in,
    in, out, in, out, mdi, muo) is semidet.
:- mode foldl3_values(pred(in, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, di, uo) is semidet.
:- mode foldl3_values(pred(in, in, out, in, out, in, out) is cc_multi, in,
    in, out, in, out, in, out) is cc_multi.
:- mode foldl3_values(pred(in, in, out, in, out, mdi, muo) is cc_multi, in,
    in, out, in, out, mdi, muo) is cc_multi.
:- mode foldl3_values(pred(in, in, out, in, out, di, uo) is cc_multi, in,
    in, out, in, out, di, uo) is cc_multi.

%-----------------------------------------------%

    % As above, but with four accumulators.
    %
:- pred foldl4_values(pred(V, A, A, B, B, C, C, D, D), map(K, V),
    A, A, B, B, C, C, D, D).
:- mode foldl4_values(pred(in, in, out, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out, in, out) is det.
:- mode foldl4_values(pred(in, in, out, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, in, out, mdi, muo) is det.
:- mode foldl4_values(pred(in, in, out, in, out, in, out, di, uo) is det,
    in, in, out, in, out, in, out, di, uo) is det.
:- mode foldl4_values(pred(in, in, out, in, out, di, uo, di, uo) is det,
    in, in, out, in, out, di, uo, di, uo) is det.
:- mode foldl4_values(pred(in, in, out, di, uo, di, uo, di, uo) is det,
    in, in, out, di, uo, di, uo, di, uo) is det.
:- mode foldl4_values(pred(in, di, uo, di, uo, di, uo, di, uo) is det,
    in, di, uo, di, uo, di, uo, di, uo) is det.
:- mode foldl4_values(pred(in, in, out, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out, in, out) is semidet.
:- mode foldl4_values(pred(in, in, out, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode foldl4_values(pred(in, in, out, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, in, out, di, uo) is semidet.

    % As above, but with five accumulators.
    %
:- pred foldl5_values(pred(V, A, A, B, B, C, C, D, D, E, E), map(K, V),
    A, A, B, B, C, C, D, D, E, E).
:- mode foldl5_values(pred(in, in, out, in, out, in, out, in, out, in, out)
    is det,
    in, in, out, in, out, in, out, in, out, in, out) is det.
:- mode foldl5_values(pred(in, in, out, in, out, in, out, in, out, mdi, muo)
    is det,
    in, in, out, in, out, in, out, in, out, mdi, muo) is det.
:- mode foldl5_values(pred(in, in, out, in, out, in, out, in, out, di, uo)
```

```
    is det,
    in, in, out, in, out, in, out, in, out, di, uo) is det.
:- mode foldl5_values(pred(in, in, out, in, out, in, out, in, out, in, out)
    is semidet,
    in, in, out, in, out, in, out, in, out, in, out) is semidet.
:- mode foldl5_values(pred(in,in, out,  in, out, in, out, in, out, mdi, muo)
    is semidet,
    in, in, out, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode foldl5_values(pred(in, in, out, in, out, in, out, in, out, di, uo)
    is semidet,
    in, in, out, in, out, in, out, in, out, di, uo) is semidet.

:- func foldr(func(K, V, A) = A, map(K, V), A) = A.
:- pred foldr(pred(K, V, A, A), map(K, V), A, A).
:- mode foldr(pred(in, in, in, out) is det, in, in, out) is det.
:- mode foldr(pred(in, in, mdi, muo) is det, in, mdi, muo) is det.
:- mode foldr(pred(in, in, di, uo) is det, in, di, uo) is det.
:- mode foldr(pred(in, in, in, out) is semidet, in, in, out) is semidet.
:- mode foldr(pred(in, in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode foldr(pred(in, in, di, uo) is semidet, in, di, uo) is semidet.
:- mode foldr(pred(in, in, in, out) is cc_multi, in, in, out) is cc_multi.
:- mode foldr(pred(in, in, mdi, muo) is cc_multi, in, mdi, muo)
    is cc_multi.
:- mode foldr(pred(in, in, di, uo) is cc_multi, in, di, uo) is cc_multi.

:- pred foldr2(pred(K, V, A, A, B, B), map(K, V), A, A, B, B).
:- mode foldr2(pred(in, in, in, out, in, out) is det,
    in, in, out, in, out) is det.
:- mode foldr2(pred(in, in, in, out, mdi, muo) is det,
    in, in, out, mdi, muo) is det.
:- mode foldr2(pred(in, in, in, out, di, uo) is det,
    in, in, out, di, uo) is det.
:- mode foldr2(pred(in, in, di, uo, di, uo) is det,
    in, di, uo, di, uo) is det.
:- mode foldr2(pred(in, in, in, out, in, out) is semidet,
    in, in, out, in, out) is semidet.
:- mode foldr2(pred(in, in, in, out, mdi, muo) is semidet,
    in, in, out, mdi, muo) is semidet.
:- mode foldr2(pred(in, in, in, out, di, uo) is semidet,
    in, in, out, di, uo) is semidet.

:- pred foldr3(pred(K, V, A, A, B, B, C, C), map(K, V), A, A, B, B, C, C).
:- mode foldr3(pred(in, in, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out) is det.
:- mode foldr3(pred(in, in, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, mdi, muo) is det.
:- mode foldr3(pred(in, in, in, out, in, out, di, uo) is det,
```

```
    in, in, out, in, out, di, uo) is det.
:- mode foldr3(pred(in, in, in, out, di, uo, di, uo) is det,
    in, in, out, di, uo, di, uo) is det.
:- mode foldr3(pred(in, in, di, uo, di, uo, di, uo) is det,
    in, di, uo, di, uo, di, uo) is det.
:- mode foldr3(pred(in, in, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out) is semidet.
:- mode foldr3(pred(in, in, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, mdi, muo) is semidet.
:- mode foldr3(pred(in, in, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, di, uo) is semidet.

:- pred foldr4(pred(K, V, A, A, B, B, C, C, D, D), map(K, V),
    A, A, B, B, C, C, D, D).
:- mode foldr4(pred(in, in, in, out, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out, in, out) is det.
:- mode foldr4(pred(in, in, in, out, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, in, out, mdi, muo) is det.
:- mode foldr4(pred(in, in, in, out, in, out, in, out, di, uo) is det,
    in, in, out, in, out, in, out, di, uo) is det.
:- mode foldr4(pred(in, in, in, out, in, out, di, uo, di, uo) is det,
    in, in, out, in, out, di, uo, di, uo) is det.
:- mode foldr4(pred(in, in, in, out, di, uo, di, uo, di, uo) is det,
    in, in, out, di, uo, di, uo, di, uo) is det.
:- mode foldr4(pred(in, in, di, uo, di, uo, di, uo, di, uo) is det,
    in, di, uo, di, uo, di, uo, di, uo) is det.
:- mode foldr4(pred(in, in, in, out, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out, in, out) is semidet.
:- mode foldr4(pred(in, in, in, out, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode foldr4(pred(in, in, in, out, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, in, out, di, uo) is semidet.

:- pred foldr5(pred(K, V, A, A, B, B, C, C, D, D, E, E), map(K, V),
    A, A, B, B, C, C, D, D, E, E).
:- mode foldr5(pred(in, in, in, out, in, out, in, out, in, out, in, out)
    is det,
    in, in, out, in, out, in, out, in, out, in, out) is det.
:- mode foldr5(pred(in, in, in, out, in, out, in, out, in, out, mdi, muo)
    is det,
    in, in, out, in, out, in, out, in, out, mdi, muo) is det.
:- mode foldr5(pred(in, in, in, out, in, out, in, out, in, out, di, uo)
    is det,
    in, in, out, in, out, in, out, in, out, di, uo) is det.
:- mode foldr5(pred(in, in, in, out, in, out, in, out, in, out, in, out)
    is semidet,
    in, in, out, in, out, in, out, in, out, in, out) is semidet.
```

```
:- mode foldr5(pred(in, in, in, out, in, out, in, out, in, out, mdi, muo)
    is semidet,
    in, in, out, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode foldr5(pred(in, in, in, out, in, out, in, out, in, out, di, uo)
    is semidet,
    in, in, out, in, out, in, out, in, out, di, uo) is semidet.

%-----------------------------------------------%

    % Apply a transformation predicate to all the values in a map.
    %
:- func map_values(func(K, V) = W, map(K, V)) = map(K, W).
:- pred map_values(pred(K, V, W), map(K, V), map(K, W)).
:- mode map_values(pred(in, in, out) is det, in, out) is det.
:- mode map_values(pred(in, in, out) is semidet, in, out) is semidet.

    % Same as map_values, but do not pass the key to the given predicate.
    %
:- func map_values_only(func(V) = W, map(K, V)) = map(K, W).
:- pred map_values_only(pred(V, W), map(K, V), map(K, W)).
:- mode map_values_only(pred(in, out) is det, in, out) is det.
:- mode map_values_only(pred(in, out) is semidet, in, out) is semidet.

    % Apply a transformation predicate to all the values in a map.
    %
:- pred filter_map_values(pred(K, V, W)::in(pred(in, in, out) is semidet),
    map(K, V)::in, map(K, W)::out) is det.

    % Same as map_values, but do not pass the key to the given predicate.
    %
:- pred filter_map_values_only(pred(V, W)::in(pred(in, out) is semidet),
    map(K, V)::in, map(K, W)::out) is det.

%-----------------------------------------------%

    % Perform an inorder traversal by key of the map, applying a transformation
    % predicate to each value while updating an accumulator.
    %
:- pred map_foldl(pred(K, V, W, A, A), map(K, V), map(K, W), A, A).
:- mode map_foldl(pred(in, in, out, in, out) is det, in, out, in, out)
    is det.
:- mode map_foldl(pred(in, in, out, mdi, muo) is det, in, out, mdi, muo)
    is det.
:- mode map_foldl(pred(in, in, out, di, uo) is det, in, out, di, uo)
    is det.
:- mode map_foldl(pred(in, in, out, in, out) is semidet, in, out,
    in, out) is semidet.
```

```
:- mode map_foldl(pred(in, in, out, mdi, muo) is semidet, in, out,
    mdi, muo) is semidet.
:- mode map_foldl(pred(in, in, out, di, uo) is semidet, in, out,
    di, uo) is semidet.

    % As map_foldl, but with two accumulators.
    %
:- pred map_foldl2(pred(K, V, W, A, A, B, B), map(K, V), map(K, W),
    A, A, B, B).
:- mode map_foldl2(pred(in, in, out, in, out, in, out) is det,
    in, out, in, out, in, out) is det.
:- mode map_foldl2(pred(in, in, out, in, out, mdi, muo) is det,
    in, out, in, out, mdi, muo) is det.
:- mode map_foldl2(pred(in, in, out, in, out, di, uo) is det,
    in, out, in, out, di, uo) is det.
:- mode map_foldl2(pred(in, in, out, di, uo, di, uo) is det,
    in, out, di, uo, di, uo) is det.
:- mode map_foldl2(pred(in, in, out, in, out, in, out) is semidet,
    in, out, in, out, in, out) is semidet.
:- mode map_foldl2(pred(in, in, out, in, out, mdi, muo) is semidet,
    in, out, in, out, mdi, muo) is semidet.
:- mode map_foldl2(pred(in, in, out, in, out, di, uo) is semidet,
    in, out, in, out, di, uo) is semidet.

    % As map_foldl, but with three accumulators.
    %
:- pred map_foldl3(pred(K, V, W, A, A, B, B, C, C), map(K, V), map(K, W),
    A, A, B, B, C, C).
:- mode map_foldl3(pred(in, in, out, in, out, in, out, in, out) is det,
    in, out, in, out, in, out, in, out) is det.
:- mode map_foldl3(pred(in, in, out, in, out, in, out, mdi, muo) is det,
    in, out, in, out, in, out, mdi, muo) is det.
:- mode map_foldl3(pred(in, in, out, di, uo, di, uo, di, uo) is det,
    in, out, di, uo, di, uo, di, uo) is det.
:- mode map_foldl3(pred(in, in, out, in, out, in, out, di, uo) is det,
    in, out, in, out, in, out, di, uo) is det.
:- mode map_foldl3(pred(in, in, out, in, out, di, uo, di, uo) is det,
    in, out, in, out, di, uo, di, uo) is det.
:- mode map_foldl3(pred(in, in, out, in, out, in, out, in, out) is semidet,
    in, out, in, out, in, out, in, out) is semidet.
:- mode map_foldl3(pred(in, in, out, in, out, in, out, mdi, muo) is semidet,
    in, out, in, out, in, out, mdi, muo) is semidet.
:- mode map_foldl3(pred(in, in, out, in, out, in, out, di, uo) is semidet,
    in, out, in, out, in, out, di, uo) is semidet.

    % As map_foldl, but with four accumulators.
    %
```

```
:- pred map_foldl4(pred(K, V, W, A, A, B, B, C, C, D, D), map(K, V), map(K, W),
    A, A, B, B, C, C, D, D).
:- mode map_foldl4(pred(in, in, out, in, out, in, out, in, out, in, out)
    is det,
    in, out, in, out, in, out, in, out, in, out) is det.
:- mode map_foldl4(pred(in, in, out, in, out, in, out, in, out, mdi, muo)
    is det,
    in, out, in, out, in, out, in, out, mdi, muo) is det.
:- mode map_foldl4(pred(in, in, out, in, out, di, uo, di, uo, di, uo) is det,
    in, out, in, out, di, uo, di, uo, di, uo) is det.
:- mode map_foldl4(pred(in, in, out, in, out, in, out, in, out, di, uo) is det,
    in, out, in, out, in, out, in, out, di, uo) is det.
:- mode map_foldl4(pred(in, in, out, in, out, in, out, di, uo, di, uo) is det,
    in, out, in, out, in, out, di, uo, di, uo) is det.
:- mode map_foldl4(pred(in, in, out, in, out, in, out, in, out, in, out)
    is semidet,
    in, out, in, out, in, out, in, out, in, out) is semidet.
:- mode map_foldl4(pred(in, in, out, in, out, in, out, in, out, mdi, muo)
    is semidet,
    in, out, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode map_foldl4(pred(in, in, out, in, out, in, out, in, out, di, uo)
    is semidet,
    in, out, in, out, in, out, in, out, di, uo) is semidet.

%---------------------------------------------%

    % As map_foldl, but without passing the key to the predicate.
    %
:- pred map_values_foldl(pred(V, W, A, A), map(K, V), map(K, W), A, A).
:- mode map_values_foldl(pred(in, out, di, uo) is det,
    in, out, di, uo) is det.
:- mode map_values_foldl(pred(in, out, in, out) is det,
    in, out, in, out) is det.
:- mode map_values_foldl(pred(in, out, in, out) is semidet,
    in, out, in, out) is semidet.

    % As map_values_foldl, but with two accumulators.
    %
:- pred map_values_foldl2(pred(V, W, A, A, B, B), map(K, V), map(K, W),
    A, A, B, B).
:- mode map_values_foldl2(pred(in, out, di, uo, di, uo) is det,
    in, out, di, uo, di, uo) is det.
:- mode map_values_foldl2(pred(in, out, in, out, di, uo) is det,
    in, out, in, out, di, uo) is det.
:- mode map_values_foldl2(pred(in, out, in, out, in, out) is det,
    in, out, in, out, in, out) is det.
:- mode map_values_foldl2(pred(in, out, in, out, in, out) is semidet,
```

```
    in, out, in, out, in, out) is semidet.

    % As map_values_foldl, but with three accumulators.
    %
:- pred map_values_foldl3(pred(V, W, A, A, B, B, C, C),
    map(K, V), map(K, W), A, A, B, B, C, C).
:- mode map_values_foldl3(pred(in, out, di, uo, di, uo, di, uo) is det,
    in, out, di, uo, di, uo, di, uo) is det.
:- mode map_values_foldl3(pred(in, out, in, out, di, uo, di, uo) is det,
    in, out, in, out, di, uo, di, uo) is det.
:- mode map_values_foldl3(pred(in, out, in, out, in, out, di, uo) is det,
    in, out, in, out, in, out, di, uo) is det.
:- mode map_values_foldl3(pred(in, out, in, out, in, out, in, out) is det,
    in, out, in, out, in, out, in, out) is det.
:- mode map_values_foldl3(
    pred(in, out, in, out, in, out, in, out) is semidet,
    in, out, in, out, in, out, in, out) is semidet.

%---------------------------------------------------%
%---------------------------------------------------%
```

47 math

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 1995-2007, 2011-2012 The University of Melbourne.
% Copyright (C) 2014, 2016-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: math.m.
% Main author: bromage.
% Stability: high.
%
% Higher mathematical operations. (The basics are in float.m.)
%
% By default, domain errors are currently handled by throwing an exception.
% For better performance, each operation in this module that can throw a domain
% exception also has an unchecked version that omits the domain check.
%
% The unchecked operations are semantically safe, since the target math
% library and/or floating point hardware perform these checks for you.
% The benefit of having the Mercury library perform the checks instead is
% that Mercury will tell you in which function or predicate the error
```

```
% occurred, as well as giving you a stack trace if that is enabled; with
% the unchecked operations you only have the information that the
% floating-point exception signal handler gives you.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module math.
:- interface.

%-----------------------------------------------------%
%
% Mathematical constants
%

    % Pythagoras' number.
    %
:- func pi = float.

    % Base of natural logarithms.
    %
:- func e = float.

%-----------------------------------------------------%
%
% "Next integer" operations
%

    % ceiling(X) = Ceil is true if Ceil is the smallest integer
    % not less than X.
    % If X is of infinite magnitude then Ceil = X.
    %
:- func ceiling(float) = float.

    % floor(X) = Floor is true if Floor is the largest integer
    % not greater than X.
    % If X is of infinite magnitude then Floor = X.
    %
:- func floor(float) = float.

    % round(X) = Round is true if Round is the integer closest to X.
    % If X has a fractional value of 0.5, it is rounded up.
    % If X is of infinite magnitude then Round = X.
    %
:- func round(float) = float.

    % truncate(X) = Trunc is true if Trunc is the integer closest to X
```

```
    % such that |Trunc| =< |X|.
    % If X is of infinite magnitude then Trunc = X.
    %
:- func truncate(float) = float.

%-----------------------------------------------------%
%
% Polynomial roots
%

    % sqrt(X) = Sqrt is true if Sqrt is the positive square root of X.
    %
    % Domain restriction: X >= 0
    %
:- func sqrt(float) = float.
:- func unchecked_sqrt(float) = float.

:- type quadratic_roots
    --->    no_roots
    ;       one_root(float)
    ;       two_roots(float, float).

    % solve_quadratic(A, B, C) = Roots is true if Roots are
    % the solutions to the equation Ax^2 + Bx + C.
    %
    % Domain restriction: A \= 0
    %
:- func solve_quadratic(float, float, float) = quadratic_roots.

%-----------------------------------------------------%
%
% Power/logarithm operations
%

    % pow(X, Y) = Res is true if Res is X raised to the power of Y.
    %
    % Domain restriction: X >= 0 and (X = 0 implies Y > 0)
    %
:- func pow(float, float) = float.
:- func unchecked_pow(float, float) = float.

    % exp(X) = Exp is true if Exp is e raised to the power of X.
    %
:- func exp(float) = float.

    % ln(X) = Log is true if Log is the natural logarithm of X.
    %
```

```
    % Domain restriction: X > 0
    %
:- func ln(float) = float.
:- func unchecked_ln(float) = float.

    % log10(X) = Log is true if Log is the logarithm to base 10 of X.
    %
    % Domain restriction: X > 0
    %
:- func log10(float) = float.
:- func unchecked_log10(float) = float.

    % log2(X) = Log is true if Log is the logarithm to base 2 of X.
    %
    % Domain restriction: X > 0
    %
:- func log2(float) = float.
:- func unchecked_log2(float) = float.

    % log(B, X) = Log is true if Log is the logarithm to base B of X.
    %
    % Domain restriction: X > 0 and B > 0 and B \= 1
    %
:- func log(float, float) = float.
:- func unchecked_log(float, float) = float.

%-----------------------------------------------%
%
% Trigonometric operations
%

    % sin(X) = Sin is true if Sin is the sine of X.
    %
:- func sin(float) = float.

    % cos(X) = Cos is true if Cos is the cosine of X.
    %
:- func cos(float) = float.

    % tan(X) = Tan is true if Tan is the tangent of X.
    %
:- func tan(float) = float.

    % asin(X) = ASin is true if ASin is the inverse sine of X,
    % where ASin is in the range [-pi/2,pi/2].
    %
    % Domain restriction: X must be in the range [-1,1]
```

```
    %
:- func asin(float) = float.
:- func unchecked_asin(float) = float.

    % acos(X) = ACos is true if ACos is the inverse cosine of X,
    % where ACos is in the range [0, pi].
    %
    % Domain restriction: X must be in the range [-1,1]
    %
:- func acos(float) = float.
:- func unchecked_acos(float) = float.

    % atan(X) = ATan is true if ATan is the inverse tangent of X,
    % where ATan is in the range [-pi/2,pi/2].
    %
:- func atan(float) = float.

    % atan2(Y, X) = ATan is true if ATan is the inverse tangent of Y/X,
    % where ATan is in the range [-pi,pi].
    %
:- func atan2(float, float) = float.

%-----------------------------------------------%
%
% Hyperbolic functions
%

    % sinh(X) = Sinh is true if Sinh is the hyperbolic sine of X.
    %
:- func sinh(float) = float.

    % cosh(X) = Cosh is true if Cosh is the hyperbolic cosine of X.
    %
:- func cosh(float) = float.

    % tanh(X) = Tanh is true if Tanh is the hyperbolic tangent of X.
    %
:- func tanh(float) = float.

%-----------------------------------------------%
%
% Fused multiply-add operation.
%

    % Succeeds if this grade and platform provide the fused multiply-add
    % operation.
    %
```

```
:- pred have_fma is semidet.

    % fma(X, Y, Z) = FMA is true if FMA = (X * Y) + Z, rounded as one
    % floating-point operation.
    %
    % This function is (currently) only available on the C backends and only if
    % the target math library supports it.
    % Use have_fma/0 to check whether it is supported.
    %
:- func fma(float, float, float) = float.

%---------------------------------------------------%
%---------------------------------------------------%
```

48 maybe

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 1994-2006, 2010-2011 The University of Melbourne.
% Copyright (C) 2016-2018 The Mercury Team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: maybe.m.
% Main author: fjh.
% Stability: high.
%
% This module defines the "maybe" type.
%
%---------------------------------------------------%
%---------------------------------------------------%

:- module maybe.
:- interface.

:- import_module list.

%---------------------------------------------------%

:- type maybe(T)
    --->    no
    ;       yes(T).

:- inst maybe(I) for maybe/1
```

```
    --->    no
    ;       yes(I).

:- inst maybe_yes(I) for maybe/1
    --->    yes(I).

:- type maybe_error
    --->    ok
    ;       error(string).

:- type maybe_error(T) == maybe_error(T, string).

    % Either a T, or an error E.
:- type maybe_error(T, E)
    --->    ok(T)
    ;       error(E).

:- inst maybe_error(I) for maybe_error/2
    --->    ok(I)
    ;       error(ground).

:- inst maybe_error_ok(I) for maybe_error/2
    --->    ok(I).

:- type maybe_errors(T) == maybe_errors(T, string).

    % Either a T, or one or more errors E.
:- type maybe_errors(T, E)
    --->    ok(T)
    ;       error(E, list(E)).

:- inst maybe_errors_ok(I) for maybe_errors/2
    --->    ok(I).

    % map_maybe(P, yes(Value0), yes(Value)) :- P(Value, Value).
    % map_maybe(_, no, no).
    %
:- pred map_maybe(pred(T, U), maybe(T), maybe(U)).
:- mode map_maybe(pred(in, out) is det, in, out) is det.
:- mode map_maybe(pred(in, out) is semidet, in, out) is semidet.
:- mode map_maybe(pred(in, out) is multi, in, out) is multi.
:- mode map_maybe(pred(in, out) is nondet, in, out) is nondet.

    % map_maybe(_, no) = no.
    % map_maybe(F, yes(Value)) = yes(F(Value)).
    %
:- func map_maybe(func(T) = U, maybe(T)) = maybe(U).
```

```
    % fold_maybe(_, no, Acc) = Acc.
    % fold_maybe(F, yes(Value), Acc0) = F(Value, Acc0).
    %
:- func fold_maybe(func(T, U) = U, maybe(T), U) = U.

    % fold_maybe(_, no, !Acc).
    % fold_maybe(P, yes(Value), !Acc) :- P(Value, !Acc).
    %
:- pred fold_maybe(pred(T, U, U), maybe(T), U, U).
:- mode fold_maybe(pred(in, in, out) is det, in, in, out) is det.
:- mode fold_maybe(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode fold_maybe(pred(in, di, uo) is det, in, di, uo) is det.
:- mode fold_maybe(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode fold_maybe(pred(in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode fold_maybe(pred(in, di, uo) is semidet, in, di, uo) is semidet.

    % As above, but with two accumulators.
    %
:- pred fold2_maybe(pred(T, U, U, V, V), maybe(T), U, U, V, V).
:- mode fold2_maybe(pred(in, in, out, in, out) is det, in, in, out,
    in, out) is det.
:- mode fold2_maybe(pred(in, in, out, mdi, muo) is det, in, in, out,
    mdi, muo) is det.
:- mode fold2_maybe(pred(in, in, out, di, uo) is det, in, in, out,
    di, uo) is det.
:- mode fold2_maybe(pred(in, in, out, in, out) is semidet, in, in, out,
    in, out) is semidet.
:- mode fold2_maybe(pred(in, in, out, mdi, muo) is semidet, in, in, out,
    mdi, muo) is semidet.
:- mode fold2_maybe(pred(in, in, out, di, uo) is semidet, in, in, out,
    di, uo) is semidet.

    % map_fold_maybe(_, no, no, !Acc).
    % map_fold_maybe(P, yes(Value0), yes(Value), !Acc) :-
    %     P(Value, Value, !Acc).
    %
:- pred map_fold_maybe(pred(T, U, Acc, Acc), maybe(T), maybe(U), Acc, Acc).
:- mode map_fold_maybe(pred(in, out, in, out) is det, in, out,
    in, out) is det.
:- mode map_fold_maybe(pred(in, out, mdi, muo) is det, in, out,
    mdi, muo) is det.
:- mode map_fold_maybe(pred(in, out, di, uo) is det, in, out,
    di, uo) is det.
:- mode map_fold_maybe(pred(in, out, in, out) is semidet, in, out,
    in, out) is semidet.
:- mode map_fold_maybe(pred(in, out, mdi, muo) is semidet, in, out,
```

```
    mdi, muo) is semidet.
:- mode map_fold_maybe(pred(in, out, di, uo) is semidet, in, out,
    di, uo) is semidet.

    % As above, but with two accumulators.
    %
:- pred map_fold2_maybe(pred(T, U, Acc1, Acc1, Acc2, Acc2),
    maybe(T), maybe(U), Acc1, Acc1, Acc2, Acc2).
:- mode map_fold2_maybe(pred(in, out, in, out, in, out) is det,
    in, out, in, out, in, out) is det.
:- mode map_fold2_maybe(pred(in, out, in, out, mdi, muo) is det,
    in, out, in, out, mdi, muo) is det.
:- mode map_fold2_maybe(pred(in, out, in, out, di, uo) is det,
    in, out, in, out, di, uo) is det.
:- mode map_fold2_maybe(pred(in, out, in, out, in, out) is semidet,
    in, out, in, out, in, out) is semidet.
:- mode map_fold2_maybe(pred(in, out, in, out, mdi, muo) is semidet,
    in, out, in, out, mdi, muo) is semidet.
:- mode map_fold2_maybe(pred(in, out, in, out, di, uo) is semidet,
    in, out, in, out, di, uo) is semidet.

    % As above, but with three accumulators.
    %
:- pred map_fold3_maybe(pred(T, U, Acc1, Acc1, Acc2, Acc2, Acc3, Acc3),
    maybe(T), maybe(U), Acc1, Acc1, Acc2, Acc2, Acc3, Acc3).
:- mode map_fold3_maybe(pred(in, out, in, out, in, out, in, out) is det,
    in, out, in, out, in, out, in, out) is det.
:- mode map_fold3_maybe(pred(in, out, in, out, in, out, mdi, muo) is det,
    in, out, in, out, in, out, mdi, muo) is det.
:- mode map_fold3_maybe(pred(in, out, in, out, in, out, di, uo) is det,
    in, out, in, out, in, out, di, uo) is det.
:- mode map_fold3_maybe(pred(in, out, in, out, in, out, in, out) is semidet,
    in, out, in, out, in, out, in, out) is semidet.
:- mode map_fold3_maybe(pred(in, out, in, out, in, out, mdi, muo) is semidet,
    in, out, in, out, in, out, mdi, muo) is semidet.
:- mode map_fold3_maybe(pred(in, out, in, out, in, out, di, uo) is semidet,
    in, out, in, out, in, out, di, uo) is semidet.

    % maybe_is_yes(yes(X), X).
    %
    % This is useful as an argument to list.filter_map.
    %
:- pred maybe_is_yes(maybe(T)::in, T::out) is semidet.

    % pred_to_maybe(Pred) = MaybeResult.
    %
    % Make a maybe value from a semidet predicate.
```

```
    %
:- func pred_to_maybe(pred(T)) = maybe(T).
:- mode pred_to_maybe(pred(out) is semidet) = out is det.

:- func func_to_maybe((func) = T) = maybe(T).
:- mode func_to_maybe((func) = out is semidet) = out is det.

    % Return the value from within the maybe or a default value if there is
    % none.
    %
:- func maybe_default(T, maybe(T)) = T.

%---------------------------------------------------%
%---------------------------------------------------%
```

49 multi_map

```
%---------------------------------------------------%
% vim: ts=4 sw=4 et ft=mercury
%---------------------------------------------------%
% Copyright (C) 1995, 1997, 2000, 2002-2006, 2011 The University of Melbourne.
% Copyright (C) 2014-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: multi_map.m.
% Main author: dylan.
% Stability: medium.
%
% This file provides the 'multi_map' ADT.
% A map (also known as a dictionary or an associative array) is a collection
% of (Key, Value) pairs which allows you to look up any Value given the Key.
% A multi_map is similar, but it allows more than one Value for each Key.
%
% This is implemented almost as a special case of map.m.
%
%---------------------------------------------------%
%---------------------------------------------------%

:- module multi_map.
:- interface.

:- import_module assoc_list.
:- import_module list.
:- import_module map.
```

```
:- import_module set.

%-----------------------------------------------------%

:- type multi_map(K, V) == map(K, list(V)).

%-----------------------------------------------------%

    % Return an empty multi_map.
    %
:- func init = multi_map(_K, _V).
:- pred init(multi_map(_K, _V)::uo) is det.

    % Check whether the multi_map is empty.
    %
:- pred is_empty(multi_map(_K, _V)::in) is semidet.

%-----------------------------------------------------%

    % Check whether the multi_map has an entry for the given key.
    %
:- pred contains(multi_map(K, _V)::in, K::in) is semidet.

    % Succeed once for each key-value pair in the multi_map.
    %
:- pred member(multi_map(K, V)::in, K::out, V::out) is nondet.

    % If the multi_map has an entry for the given key, return the
    % list of corresponding values.
    %
:- pred search(multi_map(K, V)::in, K::in, list(V)::out) is semidet.

    % If the multi_map has an entry for the given key,
    % succeed once for each of the corresponding values.
    %
:- pred nondet_search(multi_map(K, V)::in, K::in, V::out) is nondet.

    % If the multi_map has an entry for the given key,
    % succeed once for each of the corresponding values.
    % Otherwise, throw an exception.
    %
:- func lookup(multi_map(K, V), K) = list(V).
:- pred lookup(multi_map(K, V)::in, K::in, list(V)::out) is det.

    % If the multi_map has an entry for the given key,
    % succeed once for each of the corresponding values.
    % Otherwise, throw an exception.
```

```
    %
:- pred nondet_lookup(multi_map(K, V)::in, K::in, V::out) is nondet.

    % If the multi_map has an entry for keys with the given value,
    % succeed once for each of those keys.
    %
    % NOTE: The implementation of this predicate is necessarily inefficient,
    % and so this predicate is intended for non-performance-critical uses only.
    %
:- pred inverse_search(multi_map(K, V)::in, V::in, K::out) is nondet.

%-----------------------------------------------%

    % Add the given key-value pair to the multi_map.
    % Fail if the key already exists.
    %
:- pred insert(K::in, V::in,
    multi_map(K, V)::in, multi_map(K, V)::out) is semidet.

    % Add the given key-value pair to the multi_map.
    % Throw an exception if the key already exists.
    %
:- func det_insert(multi_map(K, V), K, V) = multi_map(K, V).
:- pred det_insert(K::in, V::in,
    multi_map(K, V)::in, multi_map(K, V)::out) is det.

    % Add the given key-value pair to the multi_map.
    % Fail if the key does not already exist.
    %
:- pred update(K::in, V::in,
    multi_map(K, V)::in, multi_map(K, V)::out) is semidet.

    % Add the given key-value pair to the multi_map.
    % Throw an exception if the key does not already exist.
    %
:- func det_update(multi_map(K, V), K, V) = multi_map(K, V).
:- pred det_update(K::in, V::in,
    multi_map(K, V)::in, multi_map(K, V)::out) is det.

    % Replace the list of values corresponding to the given key.
    % Fails if the key does not already exist.
    %
:- pred replace(K::in, list(V)::in,
    multi_map(K, V)::in, multi_map(K, V)::out) is semidet.

    % Replace the list of values corresponding to the given key.
    % Throws an exception if the key does not already exist.
```

```
    %
:- func det_replace(multi_map(K, V), K, list(V)) = multi_map(K, V).
:- pred det_replace(K::in, list(V)::in,
    multi_map(K, V)::in, multi_map(K, V)::out) is det.

    % Add the given key-value pair to the multi_map.
    % (‘add’ is a synonym for ‘set’.)
    %
:- func set(multi_map(K, V), K, V) = multi_map(K, V).
:- pred set(K::in, V::in,
    multi_map(K, V)::in, multi_map(K, V)::out) is det.
:- func add(multi_map(K, V), K, V) = multi_map(K, V).
:- pred add(K::in, V::in,
    multi_map(K, V)::in, multi_map(K, V)::out) is det.

    % Add the given value-key pair to the multi_map.
    %
:- func reverse_set(multi_map(K, V), V, K) = multi_map(K, V).
:- pred reverse_set(V::in, K::in,
    multi_map(K, V)::in, multi_map(K, V)::out) is det.

%---------------------------------------------------%

    % Delete a key and its corresponding values from a multi_map.
    % If the key is not present, leave the multi_map unchanged.
    %
:- func delete(multi_map(K, V), K) = multi_map(K, V).
:- pred delete(K::in,
    multi_map(K, V)::in, multi_map(K, V)::out) is det.

    % Delete the given key-value pair from a multi_map.
    % If the key is not present, leave the multi_map unchanged.
    %
:- func delete(multi_map(K, V), K, V) = multi_map(K, V).
:- pred delete(K::in, V::in,
    multi_map(K, V)::in, multi_map(K, V)::out) is det.

    % Delete a key from a multi_map and return the list of values
    % previously corresponding to it.
    % Fail if the key is not present.
    %
:- pred remove(K::in, list(V)::out,
    multi_map(K, V)::in, multi_map(K, V)::out) is semidet.

    % Delete a key from a multi_map and return the list of values
    % previously corresponding to it.
    % Throw an exception if the key is not present.
```

```
    %
:- pred det_remove(K::in, list(V)::out,
    multi_map(K, V)::in, multi_map(K, V)::out) is det.

    % Remove the smallest key and its corresponding values from the multi_map.
    % Fails if the multi_map is empty.
    %
:- pred remove_smallest(K::out, list(V)::out,
    multi_map(K, V)::in, multi_map(K, V)::out) is semidet.

%---------------------------------------------------%

    % Select takes a multi_map and a set of keys and returns a multi_map
    % containing only the keys in the set, together with their corresponding
    % values.
    %
:- func select(multi_map(K, V), set(K)) = multi_map(K, V).
:- pred select(multi_map(K, V)::in, set(K)::in, multi_map(K, V)::out) is det.

%---------------------------------------------------%

    % merge(MultiMapA, MultiMapB, MultiMap):
    %
    % Merge ‘MultiMapA’ and ‘MultiMapB’ so that
    %
    % - if a key occurs in both ‘MultiMapA’ and ‘MultiMapB’, then the values
    %   corresponding to that key in ‘MultiMap’ will be the concatenation
    %   of the values to that key from ‘MultiMapA’ and ‘MultiMapB’; while
    % - if a key occurs in only one of ‘MultiMapA’ and ‘MultiMapB’, then
    %   the values corresponding to it in that map will be carried over
    %   to ‘MultiMap’.
    %
:- func merge(multi_map(K, V), multi_map(K, V))
    = multi_map(K, V).
:- pred merge(multi_map(K, V)::in, multi_map(K, V)::in,
    multi_map(K, V)::out) is det.

%---------------------------------------------------%

    % Declaratively, a no-operation.
    % Operationally, a suggestion that the implementation optimize
    % the representation of the multi_map, in the expectation that the
    % following operations will consist of searches and lookups
    % but (almost) no updates.
    %
:- func optimize(multi_map(K, V)) = multi_map(K, V).
:- pred optimize(multi_map(K, V)::in, multi_map(K, V)::out) is det.
```

```
%---------------------------------------------------%

    % Convert a multi_map to an association list.
    %
:- func to_flat_assoc_list(multi_map(K, V)) = assoc_list(K, V).
:- pred to_flat_assoc_list(multi_map(K, V)::in,
    assoc_list(K, V)::out) is det.

    % Convert an association list to a multi_map.
    %
:- func from_flat_assoc_list(assoc_list(K, V)) = multi_map(K, V).
:- pred from_flat_assoc_list(assoc_list(K, V)::in,
    multi_map(K, V)::out) is det.

    % Convert a multi_map to an association list, with all the values
    % for each key in one element of the association list.
    %
:- func to_assoc_list(multi_map(K, V)) = assoc_list(K, list(V)).
:- pred to_assoc_list(multi_map(K, V)::in,
    assoc_list(K, list(V))::out) is det.

    % Convert an association list with all the values for each key
    % in one element of the list to a multi_map.
    %
:- func from_assoc_list(assoc_list(K, list(V))) = multi_map(K, V).
:- pred from_assoc_list(assoc_list(K, list(V))::in,
    multi_map(K, V)::out) is det.

    % Convert a sorted association list to a multi_map.
    %
:- func from_sorted_assoc_list(assoc_list(K, list(V)))
    = multi_map(K, V).
:- pred from_sorted_assoc_list(assoc_list(K, list(V))::in,
    multi_map(K, V)::out) is det.

    % Convert the corresponding elements of a list of keys and a
    % list of values (which must be of the same length) to a multi_map.
    % A key may occur more than once in the list of keys.
    % Throw an exception if the two lists are not the same length.
    %
:- func from_corresponding_lists(list(K), list(V))
    = multi_map(K, V).
:- pred from_corresponding_lists(list(K)::in, list(V)::in,
    multi_map(K, V)::out) is det.

    % Convert the corresponding elements of a list of keys and a
```

```
    % *list of lists* of values to a multi_map.
    % A key may *not* occur more than once in the list of keys.
    % Throw an exception if the two lists are not the same length,
    % or if a key does occur more than once in the list of keys.
    %
:- func from_corresponding_list_lists(list(K), list(list(V)))
    = multi_map(K, V).
:- pred from_corresponding_list_lists(list(K)::in, list(list(V))::in,
    multi_map(K, V)::out) is det.

%---------------------------------------------------%

    % Given a list of keys, produce a list of their values in a
    % specified multi_map.
    %
:- func apply_to_list(list(K), multi_map(K, V)) = list(V).
:- pred apply_to_list(list(K)::in, multi_map(K, V)::in, list(V)::out) is det.

%---------------------------------------------------%

    % Given a multi_map, return a list of all the keys in it.
    %
:- func keys(multi_map(K, V)) = list(K).
:- pred keys(multi_map(K, V)::in, list(K)::out) is det.

    % Given a multi_map, return a list of all the keys in it
    % in sorted order.
    %
:- func sorted_keys(multi_map(K, V)) = list(K).
:- pred sorted_keys(multi_map(K, V)::in, list(K)::out) is det.

   % Given a multi_map, return a list of all the keys in it
    % as a set
    %
:- func keys_as_set(multi_map(K, V)) = set(K).
:- pred keys_as_set(multi_map(K, V)::in, set(K)::out) is det.

    % Given a multi_map, return a list of all the values in it.
    %
:- func values(multi_map(K, V)) = list(V).
:- pred values(multi_map(K, V)::in, list(V)::out) is det.

%---------------------------------------------------%

    % Count the number of keys in the multi_map.
    %
:- func count(multi_map(K, V)) = int.
```

```
:- pred count(multi_map(K, V)::in, int::out) is det.

    % Count the number of key-value pairs in the multi_map.
    %
:- func all_count(multi_map(K, V)) = int.
:- pred all_count(multi_map(K, V)::in, int::out) is det.

%---------------------------------------------------%
%---------------------------------------------------%
```

50 one_or_more

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 2020 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: one_or_more.m.
% Stability: high.
%
% This module defines the one_or_more type, values of which represent
% nonempty lists, and various utility predicates that operate on them.
%
%---------------------------------------------------%
%---------------------------------------------------%

:- module one_or_more.
:- interface.

:- import_module list.
:- import_module pretty_printer.

%---------------------------------------------------%

:- type one_or_more(T)
    --->    one_or_more(T, list(T)).
            % The head, which is the one element that must be present,
            % and the tail of the list, which may be empty.

%---------------------------------------------------%

    % Convert from one_or_more to list.
    %
```

```
:- func one_or_more_to_list(one_or_more(T)) = list(T).
:- mode one_or_more_to_list(di) = uo is det.
:- mode one_or_more_to_list(in) = out is det.

    % Convert from list to one_or_more. The first version fails
    % if the list is empty, while the second throws an exception.
    %
:- pred list_to_one_or_more(list(T)::in, one_or_more(T)::out) is semidet.
:- pred det_list_to_one_or_more(list(T)::in, one_or_more(T)::out) is det.

%-----------------------------------------------%

    % Return the head and the tail respectively.
    %
:- func head(one_or_more(T)) = T.
:- func tail(one_or_more(T)) = list(T).

    % Add a new element to the front of a one_or_more.
    %
    % In list terms:
    %
    %   cons(X, Y) = Z <=> Z = [X | Y].
    %
:- func cons(T, one_or_more(T)) = one_or_more(T).
:- pred cons(T::in, one_or_more(T)::in, one_or_more(T)::out) is det.

%-----------------------------------------------%

    % Standard append predicate:
    % append(Start, End, List) is true iff
    % 'List' is the result of concatenating 'Start' and 'End'.
    %
:- pred append(one_or_more(T), one_or_more(T), one_or_more(T)).
:- mode append(di, di, uo) is det.
:- mode append(in, in, out) is det.
:- mode append(in, in, in) is semidet.    % implied

    % L1 ++ L2 = L :- append(L1, L2, L).
    %
:- func one_or_more(T) ++ one_or_more(T) = one_or_more(T).

    % Append a list and a one_or_more, in either order.
    %
:- pred append_one_or_more_list(one_or_more(T), list(T), one_or_more(T)).
:- mode append_one_or_more_list(di, di, uo) is det.
:- mode append_one_or_more_list(in, in, out) is det.
```

```
:- pred append_list_one_or_more(list(T), one_or_more(T), one_or_more(T)).
:- mode append_list_one_or_more(di, di, uo) is det.
:- mode append_list_one_or_more(in, in, out) is det.

%---------------------------------------------------%

    % Assert the associativity of append.
    %
:- promise all [A, B, C, ABC]
    (
        ( some [AB] (
            one_or_more.append(A, B, AB),
            one_or_more.append(AB, C, ABC))
        )
    <=>
        ( some [BC] (
            one_or_more.append(B, C, BC),
            one_or_more.append(A, BC, ABC))
        )
    ).

%---------------------------------------------------%

    % length(List, Length):
    %
    % True iff 'Length' is the length of 'List', i.e. if 'List' contains
    % 'Length' elements.
    %
:- func length(one_or_more(T)) = int.
:- pred length(one_or_more(T)::in, int::out) is det.

    % same_length(ListA, ListB):
    %
    % True iff 'ListA' and 'ListB' have the same length,
    % i.e. iff they both contain the same number of elements.
    %
:- pred same_length(one_or_more(T1)::in, one_or_more(T2)::in) is semidet.

    % As above, but for three lists.
    %
:- pred same_length3(one_or_more(T1)::in, one_or_more(T2)::in,
    one_or_more(T3)::in) is semidet.

%---------------------------------------------------%

    % member(Elem, List):
    %
```

```
    % True iff ‘List’ contains ‘Elem’.
    %
:- pred member(T, one_or_more(T)).
:- mode member(in, in) is semidet.
:- mode member(out, in) is multi.

    % member_index0(Elem, List, Index).
    %
    % True iff ‘List’ contains ‘Elem’ at the zero-based index ‘Index’.
    %
:- pred member_index0(T, one_or_more(T), int).
:- mode member_index0(in, in, in) is semidet.
:- mode member_index0(in, in, out) is nondet.
:- mode member_index0(out, in, out) is multi.

    % member_indexes0(Elem, List, Indexes).
    %
    % True iff ‘List’ contains ‘Elem’ at the zero-based indexes ‘Indexes’.
    % ‘Indexes’ will be sorted.
    %
:- pred member_indexes0(T::in, one_or_more(T)::in, list(int)::out) is det.

    % contains(List, Elem) iff member(Elem, List).
    % Sometimes you need the arguments in this order, because you want to
    % construct a closure with only the list.
    %
:- pred contains(one_or_more(T)::in, T::in) is semidet.

%-----------------------------------------------%

    % index*(List, Position, Elem):
    %
    % These predicates select an element in a list from it’s position.
    % The ‘index0’ preds consider the first element to be element
    % number zero, whereas the ‘index1’ preds consider the first element
    % to be element number one. The ‘det_’ preds call error/1 if the index
    % is out of range, whereas the semidet preds fail if the index is out of
    % range.
    %
:- pred index0(one_or_more(T)::in, int::in, T::out) is semidet.
:- pred index1(one_or_more(T)::in, int::in, T::out) is semidet.

:- func det_index0(one_or_more(T), int) = T.
:- pred det_index0(one_or_more(T)::in, int::in, T::out) is det.
:- func det_index1(one_or_more(T), int) = T.
:- pred det_index1(one_or_more(T)::in, int::in, T::out) is det.
```

```
    % nth_member_search(List, Elem, Position):
    %
    % Elem is the Position'th member of List.
    % (Position numbers start from 1.)
    %
:- pred nth_member_search(one_or_more(T)::in, T::in, int::out) is semidet.

    % A deterministic version of nth_member_search, which throws an exception
    % instead of failing if the element is not found in the list.
    %
:- pred nth_member_lookup(one_or_more(T)::in, T::in, int::out) is det.

    % index*_of_first_occurrence(List, Elem, Position):
    %
    % Computes the least value of Position such that
    % index*(List, Position, Elem). The 'det_' funcs call error/1
    % if Elem is not a member of List.
    %
:- pred index0_of_first_occurrence(one_or_more(T)::in, T::in, int::out)
    is semidet.
:- pred index1_of_first_occurrence(one_or_more(T)::in, T::in, int::out)
    is semidet.
:- func det_index0_of_first_occurrence(one_or_more(T), T) = int.
:- func det_index1_of_first_occurrence(one_or_more(T), T) = int.

%-----------------------------------------------------%

    % reverse(List, Reverse):
    %
    % 'Reverse' is a list containing the same elements as 'List'
    % but in reverse order.
    %
:- func reverse(one_or_more(T)) = one_or_more(T).
:- pred reverse(one_or_more(T), one_or_more(T)).
:- mode reverse(in, out) is det.
:- mode reverse(out, in) is det.

%-----------------------------------------------------%

    % delete(List, Elem, Remainder):
    %
    % True iff 'Elem' occurs in 'List', and 'Remainder' is the result of
    % deleting one occurrence of 'Elem' from 'List'.
    %
:- pred delete(one_or_more(T)::in, T::in, list(T)::out) is nondet.

    % delete_first(List0, Elem, List) is true iff Elem occurs in List0
```

```
    % and List is List0 with the first occurrence of Elem removed.
    %
:- pred delete_first(one_or_more(T)::in, T::in, list(T)::out) is semidet.

    % delete_all(List0, Elem) = List is true iff List is List0 with
    % all occurrences of Elem removed.
    %
:- func delete_all(one_or_more(T), T) = list(T).
:- pred delete_all(one_or_more(T), T, list(T)).
:- mode delete_all(di, in, uo) is det.
:- mode delete_all(in, in, out) is det.

    % delete_elems(List0, Elems) = List is true iff List is List0 with
    % all occurrences of all elements of Elems removed.
    %
:- func delete_elems(one_or_more(T), list(T)) = list(T).
:- pred delete_elems(one_or_more(T)::in, list(T)::in, list(T)::out) is det.

    % sublist(SubList, FullList) is true if one can obtain SubList
    % by starting with FullList and deleting some of its elements.
    %
:- pred sublist(one_or_more(T)::in, one_or_more(T)::in) is semidet.

%-----------------------------------------------%

    % replace(List0, D, R, List) is true iff List is List0
    % with an occurrence of D replaced with R.
    %
:- pred replace(one_or_more(T), T, T, one_or_more(T)).
:- mode replace(in, in, in, in) is semidet.
:- mode replace(in, in, in, out) is nondet.

    % replace_first(List0, D, R, List) is true iff List is List0
    % with the first occurrence of D replaced with R.
    %
:- pred replace_first(one_or_more(T)::in, T::in, T::in, one_or_more(T)::out)
    is semidet.

    % replace_all(List0, D, R) = List is true iff List is List0
    % with all occurrences of D replaced with R.
    %
:- func replace_all(one_or_more(T), T, T) = one_or_more(T).
:- pred replace_all(one_or_more(T)::in, T::in, T::in, one_or_more(T)::out)
    is det.

    % replace_nth(List0, N, R, List) is true iff List is List0
    % with N'th element replaced with R.
```

```
    % Fails if N < 1 or if length of List0 < N.
    % (Position numbers start from 1.)
    %
:- pred replace_nth(one_or_more(T)::in, int::in, T::in, one_or_more(T)::out)
    is semidet.

    % det_replace_nth(List0, N, R) = List is true iff List is List0
    % with N'th element replaced with R.
    % Throws an exception if N < 1 or if length of List0 < N.
    % (Position numbers start from 1.)
    %
:- func det_replace_nth(one_or_more(T), int, T) = one_or_more(T).
:- pred det_replace_nth(one_or_more(T)::in, int::in, T::in,
    one_or_more(T)::out) is det.

%-----------------------------------------------------%

    % remove_dups(L0) = L:
    %
    % L is the result of deleting the second and subsequent occurrences
    % of every element that occurs twice in L0.
    %
:- func remove_dups(one_or_more(T)) = one_or_more(T).
:- pred remove_dups(one_or_more(T)::in, one_or_more(T)::out) is det.

    % remove_adjacent_dups(L0) = L:
    %
    % L is the result of replacing every sequence of duplicate elements in L0
    % with a single such element.
    %
:- func remove_adjacent_dups(one_or_more(T)) = one_or_more(T).
:- pred remove_adjacent_dups(one_or_more(T)::in, one_or_more(T)::out) is det.

    % remove_adjacent_dups(P, L0, L) is true iff L is the result
    % of replacing every sequence of elements in L0 which are equivalent
    % with respect to the ordering, with the first occurrence in L0 of
    % such an element.
    %
:- pred remove_adjacent_dups(comparison_pred(T)::in(comparison_pred),
    one_or_more(T)::in, one_or_more(T)::out) is det.

%-----------------------------------------------------%

    % merge(L1, L2) = L:
    %
    % L is the result of merging the elements of L1 and L2, in ascending order.
    % L1 and L2 must be sorted.
```

```
    %
:- func merge(one_or_more(T), one_or_more(T)) = one_or_more(T).
:- pred merge(one_or_more(T)::in, one_or_more(T)::in, one_or_more(T)::out)
    is det.

    % merge(Compare, As, Bs) = Sorted is true iff, assuming As and
    % Bs are sorted with respect to the ordering defined by Compare,
    % Sorted is a list containing the elements of As and Bs which is
    % also sorted. For elements which are equivalent in the ordering,
    % if they come from the same list then they appear in the same
    % sequence in Sorted as they do in that list, otherwise the elements
    % from As appear before the elements from Bs.
    %
:- func merge(comparison_func(T), one_or_more(T), one_or_more(T))
    = one_or_more(T).
:- pred merge(comparison_pred(T)::in(comparison_pred),
    one_or_more(T)::in, one_or_more(T)::in, one_or_more(T)::out) is det.

    % merge_and_remove_dups(L1, L2) = L:
    %
    % L is the result of merging the elements of L1 and L2, in ascending order,
    % and eliminating any duplicates. L1 and L2 must be sorted and must each
    % not contain any duplicates.
    %
:- func merge_and_remove_dups(one_or_more(T), one_or_more(T)) = one_or_more(T).
:- pred merge_and_remove_dups(one_or_more(T)::in, one_or_more(T)::in,
    one_or_more(T)::out) is det.

    % merge_and_remove_dups(Compare, As, Bs) = Sorted is true iff,
    % assuming As and Bs are sorted with respect to the ordering defined
    % by Compare and neither contains any duplicates, Sorted is a list
    % containing the elements of As and Bs which is also sorted and
    % contains no duplicates. If an element from As is duplicated in
    % Bs (that is, they are equivalent in the ordering), then the element
    % from As is the one that appears in Sorted.
    %
:- func merge_and_remove_dups(comparison_func(T),
    one_or_more(T), one_or_more(T)) = one_or_more(T).
:- pred merge_and_remove_dups(comparison_pred(T)::in(comparison_pred),
    one_or_more(T)::in, one_or_more(T)::in, one_or_more(T)::out) is det.

%-----------------------------------------------%

    % sort(List) = SortedList:
    %
    % SortedList is List sorted.
    %
```

```
:- func sort(one_or_more(T)) = one_or_more(T).
:- pred sort(one_or_more(T)::in, one_or_more(T)::out) is det.

    % sort_and_remove_dups(List) = SortedList:
    %
    % SortedList is List sorted with the second and subsequent occurrence of
    % any duplicates removed.
    %
:- func sort_and_remove_dups(one_or_more(T)) = one_or_more(T).
:- pred sort_and_remove_dups(one_or_more(T)::in, one_or_more(T)::out) is det.

%-----------------------------------------------%

    % sort(Compare, Unsorted) = Sorted is true iff Sorted is a
    % list containing the same elements as Unsorted, where Sorted is
    % sorted with respect to the ordering defined by Compare,
    % and the elements that are equivalent in this ordering appear
    % in the same sequence in Sorted as they do in Unsorted
    % (that is, the sort is stable).
    %
:- func sort(comparison_func(T), one_or_more(T)) = one_or_more(T).
:- pred sort(comparison_pred(T)::in(comparison_pred),
    one_or_more(T)::in, one_or_more(T)::out) is det.

    % sort_and_remove_dups(Compare, Unsorted, Sorted) is true iff
    % Sorted is a list containing the same elements as Unsorted, where
    % Sorted is sorted with respect to the ordering defined by the
    % predicate term Compare, except that if two elements in Unsorted
    % are equivalent with respect to this ordering only the one which
    % occurs first will be in Sorted.
    %
:- pred sort_and_remove_dups(comparison_pred(T)::in(comparison_pred),
    one_or_more(T)::in, one_or_more(T)::out) is det.

%-----------------------------------------------%

    % split_list(N, List, Start, End):
    %
    % splits 'List' into a prefix 'Start' of length 'N', and a remainder
    % 'End'. Fails if 'N' is not in '0 .. length(List)'.
    % See also: take, drop and split_upto.
    %
:- pred split_list(int::in, one_or_more(T)::in, list(T)::out, list(T)::out)
    is semidet.

    % det_split_list(N, List, Start, End):
    %
```

```
    % A deterministic version of split_list, which throws an exception
    % instead of failing if 'N' is not in 0 .. length(List).
    %
:- pred det_split_list(int::in, one_or_more(T)::in, list(T)::out, list(T)::out)
    is det.

    % split_upto(N, List, Start, End):
    %
    % splits 'List' into a prefix 'Start' of length 'min(N, length(List))',
    % and a remainder 'End'. Throws an exception if 'N' < 0.
    % See also: split_list, take, drop.
    %
:- pred split_upto(int::in, one_or_more(T)::in, list(T)::out, list(T)::out)
    is det.

%-----------------------------------------------------%

    % last(List, Last) is true if Last is the last element of List.
    %
:- func last(one_or_more(T)) = T.
:- pred last(one_or_more(T)::in, T::out) is det.

    % split_last(List, AllButLast, Last) is true if Last is the
    % last element of List and AllButLast is the list of elements before it.
    %
:- pred split_last(one_or_more(T)::in, list(T)::out, T::out) is det.

%-----------------------------------------------------%

    % all_same(List) is true if all elements of the list are the same.
    %
:- pred all_same(one_or_more(T)::in) is semidet.

%-----------------------------------------------------%

    % condense(ListOfOoMs) = List:
    %
    % 'List' is the result of concatenating all the elements of 'ListOfOoMs'.
    %
:- func condense(list(one_or_more(T))) = list(T).
:- pred condense(list(one_or_more(T))::in, list(T)::out) is det.

    % chunk(List, ChunkSize) = Chunks:
    %
    % Takes a list 'List' and breaks it into a list of lists 'Chunks',
    % such that the length of each list in 'Chunks' is at most 'ChunkSize.
    % (More precisely, the length of each list in 'Chunks' other than the
```

```
    % last one is exactly ‘ChunkSize’, and the length of the last list in
    % ‘Chunks’ is between one and ‘ChunkSize’.)
    %
:- func chunk(one_or_more(T), int) = one_or_more(one_or_more(T)).
:- pred chunk(one_or_more(T)::in, int::in, one_or_more(one_or_more(T))::out)
    is det.

%---------------------------------------------------%

    % zip(ListA, ListB) = List:
    %
    % List is the result of alternating the elements of ListA and ListB,
    % starting with the first element of ListA (followed by the first element
    % of ListB, then the second element of listA, then the second element
    % of ListB, etc.). When there are no more elements remaining in one of
    % the lists, the remainder of the nonempty list is appended.
    %
:- func zip(one_or_more(T), one_or_more(T)) = one_or_more(T).
:- pred zip(one_or_more(T)::in, one_or_more(T)::in,
    one_or_more(T)::out) is det.

%---------------------------------------------------%

    % perm(List0, List):
    %
    % True iff ‘List’ is a permutation of ‘List0’.
    %
:- pred perm(one_or_more(T)::in, one_or_more(T)::out) is multi.

%---------------------------------------------------%

    % Convert a list to a pretty_printer.doc for formatting.
    %
:- func one_or_more_to_doc(one_or_more(T)) = pretty_printer.doc.

%---------------------------------------------------%
%
% The following group of predicates use higher-order terms to simplify
% various list processing tasks. They implement pretty much standard
% sorts of operations provided by standard libraries for functional languages.
%
%---------------------------------------------------%

    % find_first_match(Pred, List, FirstMatch) takes a closure with one
    % input argument. It returns the first element X of the list (if any)
    % for which Pred(X) is true.
    %
```

```
:- pred find_first_match(pred(T)::in(pred(in) is semidet), one_or_more(T)::in,
    T::out) is semidet.

    % any_true(Pred, List):
    % Succeeds iff Pred succeeds for at least one element of List.
    % Same as ‘not all_false(Pred, List)’.
    %
:- pred any_true(pred(T)::in(pred(in) is semidet), one_or_more(T)::in)
    is semidet.

    % any_false(Pred, List):
    % Succeeds iff Pred fails for at least one element of List.
    % Same as ‘not all_true(Pred, List)’.
    %
:- pred any_false(pred(T)::in(pred(in) is semidet), one_or_more(T)::in)
    is semidet.

    % all_true(Pred, List) takes a closure with one input argument.
    % If Pred succeeds for every member of List, all_true succeeds.
    % If Pred fails for any member of List, all_true fails.
    %
:- pred all_true(pred(T)::in(pred(in) is semidet), one_or_more(T)::in)
    is semidet.

    % all_false(Pred, List) takes a closure with one input argument.
    % If Pred fails for every member of List, all_false succeeds.
    % If Pred succeeds for any member of List, all_false fails.
    %
:- pred all_false(pred(T)::in(pred(in) is semidet), one_or_more(T)::in)
    is semidet.

    % all_true_corresponding(Pred, ListA, ListB):
    % Succeeds if Pred succeeds for every corresponding pair of elements from
    % ListA and ListB. Fails if Pred fails for any pair of corresponding
    % elements.
    %
    % An exception is raised if the list arguments differ in length.
    %
:- pred all_true_corresponding(pred(X, Y)::in(pred(in, in) is semidet),
    one_or_more(X)::in, one_or_more(Y)::in) is semidet.

    % all_false_corresponding(Pred, ListA, ListB):
    % Succeeds if Pred fails for every corresponding pair of elements from
    % ListA and ListB. Fails if Pred succeeds for any pair of corresponding
    % elements.
    %
    % An exception is raised if the list arguments differ in length.
```

```
    %
:- pred all_false_corresponding(pred(X, Y)::in(pred(in, in) is semidet),
    one_or_more(X)::in, one_or_more(Y)::in) is semidet.

%-----------------------------------------------%

    % filter(Pred, List) = TrueList takes a closure with one
    % input argument and for each member X of List, calls the closure.
    % X is included in TrueList iff Pred(X) is true.
    %
:- func filter(pred(T)::in(pred(in) is semidet), one_or_more(T)::in)
    = (list(T)::out) is det.
:- pred filter(pred(T)::in(pred(in) is semidet), one_or_more(T)::in,
    list(T)::out) is det.

    % filter(Pred, List, TrueList, FalseList) takes a closure with one
    % input argument and for each member X of List, calls the closure.
    % X is included in TrueList iff Pred(T) is true.
    % X is included in FalseList iff Pred(T) is false.
    %
:- pred filter(pred(T)::in(pred(in) is semidet), one_or_more(T)::in,
    list(T)::out, list(T)::out) is det.

    % negated_filter(Pred, List) = FalseList takes a closure with one
    % input argument and for each member of List 'X', calls the closure.
    % X is included in FalseList iff Pred(X) is false.
    %
:- func negated_filter(pred(T)::in(pred(in) is semidet), one_or_more(T)::in)
    = (list(T)::out) is det.
:- pred negated_filter(pred(T)::in(pred(in) is semidet), one_or_more(T)::in,
    list(T)::out) is det.

    % filter_map(Transformer, List, TrueList) takes a semidet function
    % and calls it with each element of List. If a call succeeds, then
    % its return value is included in TrueList.
    %
:- func filter_map(func(X) = Y, one_or_more(X)) = list(Y).
:- mode filter_map(func(in) = out is semidet, in) = out is det.

    % filter_map(Transformer, List, TrueList) takes a predicate
    % with one input argument and one output argument, and calls it
    % with each element of List. If a call succeeds, then
    % its output is included in TrueList.
    %
:- pred filter_map(pred(X, Y)::in(pred(in, out) is semidet),
    one_or_more(X)::in, list(Y)::out) is det.
```

```
    % filter_map(Transformer, List, TrueList, FalseList) takes
    % a predicate with one input argument and one output argument.
    % It is called with each element of List. If a call succeeds,
    % then the output is included in TrueList; otherwise, the failing
    % input is included in FalseList.
    %
:- pred filter_map(pred(X, Y)::in(pred(in, out) is semidet),
    one_or_more(X)::in, list(Y)::out, list(X)::out) is det.

    % Same as filter_map/3 except that it only returns the first match:
    %
    %   find_first_map(X, Y, Z) <=> filter_map(X, Y, [Z | _])
    %
:- pred find_first_map(pred(X, Y)::in(pred(in, out) is semidet),
    one_or_more(X)::in, Y::out) is semidet.

    % Same as find_first_map, except with two outputs.
    %
:- pred find_first_map2(pred(X, A, B)::in(pred(in, out, out) is semidet),
    one_or_more(X)::in, A::out, B::out) is semidet.

    % Same as find_first_map, except with three outputs.
    %
:- pred find_first_map3(
    pred(X, A, B, C)::in(pred(in, out, out, out) is semidet),
    one_or_more(X)::in, A::out, B::out, C::out) is semidet.

    % find_index_of_match(Match, List, Index0, Index)
    %
    % Find the index of the first item in List for which Match is true,
    % where the first element in the list has the index Index0.
    % (Index0 is *not* the number of items to skip at the head of List.)
    %
:- pred find_index_of_match(pred(T), one_or_more(T), int, int).
:- mode find_index_of_match(pred(in) is semidet, in, in, out) is semidet.

%-----------------------------------------------%

    % map(T, L) = M:
    % map(T, L, M):
    %
    % Apply the closure T to transform the elements of L
    % into the elements of M.
    %
:- func map(func(X) = Y, one_or_more(X)) = one_or_more(Y).
:- pred map(pred(X, Y), one_or_more(X), one_or_more(Y)).
:- mode map(pred(in, out) is det, in, out) is det.
```

```
:- mode map(pred(in, out) is cc_multi, in, out) is cc_multi.
:- mode map(pred(in, out) is semidet, in, out) is semidet.
:- mode map(pred(in, out) is multi, in, out) is multi.
:- mode map(pred(in, out) is nondet, in, out) is nondet.
:- mode map(pred(in, in) is semidet, in, in) is semidet.

    % map2(T, L, M1, M2) uses the closure T
    % to transform the elements of L into the elements of M1 and M2.
    %
:- pred map2(pred(A, B, C), one_or_more(A), one_or_more(B), one_or_more(C)).
:- mode map2(pred(in, out, out) is det, in, out, out) is det.
:- mode map2(pred(in, out, out) is cc_multi, in, out, out) is cc_multi.
:- mode map2(pred(in, out, out) is semidet, in, out, out) is semidet.
:- mode map2(pred(in, out, out) is multi, in, out, out) is multi.
:- mode map2(pred(in, out, out) is nondet, in, out, out) is nondet.
:- mode map2(pred(in, in, in) is semidet, in, in, in) is semidet.

    % map3(T, L, M1, M2, M3) uses the closure T
    % to transform the elements of L into the elements of M1, M2 and M3.
    %
:- pred map3(pred(A, B, C, D),
    one_or_more(A), one_or_more(B), one_or_more(C), one_or_more(D)).
:- mode map3(pred(in, out, out, out) is det, in, out, out, out) is det.
:- mode map3(pred(in, out, out, out) is cc_multi, in, out, out, out)
    is cc_multi.
:- mode map3(pred(in, out, out, out) is semidet, in, out, out, out)
    is semidet.
:- mode map3(pred(in, out, out, out) is multi, in, out, out, out)
    is multi.
:- mode map3(pred(in, out, out, out) is nondet, in, out, out, out)
    is nondet.
:- mode map3(pred(in, in, in, in) is semidet, in, in, in, in) is semidet.

    % map4(T, L, M1, M2, M3, M4) uses the closure T
    % to transform the elements of L into the elements of M1, M2, M3 and M4.
    %
:- pred map4(pred(A, B, C, D, E),
    one_or_more(A), one_or_more(B), one_or_more(C), one_or_more(D),
    one_or_more(E)).
:- mode map4(pred(in, out, out, out, out) is det, in, out, out, out, out)
    is det.
:- mode map4(pred(in, out, out, out, out) is cc_multi, in, out, out, out,
    out) is cc_multi.
:- mode map4(pred(in, out, out, out, out) is semidet, in, out, out, out,
    out) is semidet.
:- mode map4(pred(in, out, out, out, out) is multi, in, out, out, out,
    out) is multi.
```

```
:- mode map4(pred(in, out, out, out, out) is nondet, in, out, out, out,
    out) is nondet.
:- mode map4(pred(in, in, in, in, in) is semidet, in, in, in, in, in)
    is semidet.

    % map5(T, L, M1, M2, M3, M4, M5) uses the closure T
    % to transform the elements of L into the elements of M1, M2, M3, M4
    % and M5.
    %
:- pred map5(pred(A, B, C, D, E, F),
    one_or_more(A), one_or_more(B), one_or_more(C), one_or_more(D),
    one_or_more(E), one_or_more(F)).
:- mode map5(pred(in, out, out, out, out, out) is det, in, out, out, out,
    out, out) is det.
:- mode map5(pred(in, out, out, out, out, out) is cc_multi, in, out, out,
    out, out, out) is cc_multi.
:- mode map5(pred(in, out, out, out, out, out) is semidet, in, out, out,
    out, out, out) is semidet.
:- mode map5(pred(in, out, out, out, out, out) is multi, in, out, out,
    out, out, out) is multi.
:- mode map5(pred(in, out, out, out, out, out) is nondet, in, out, out,
    out, out, out) is nondet.
:- mode map5(pred(in, in, in, in, in, in) is semidet, in, in, in, in, in,
    in) is semidet.

    % map6(T, L, M1, M2, M3, M4, M5, M6) uses the closure T
    % to transform the elements of L into the elements of M1, M2, M3, M4,
    % M5 and M6.
    %
:- pred map6(pred(A, B, C, D, E, F, G),
    one_or_more(A), one_or_more(B), one_or_more(C), one_or_more(D),
    one_or_more(E), one_or_more(F), one_or_more(G)).
:- mode map6(pred(in, out, out, out, out, out, out) is det, in, out, out,
    out, out, out, out) is det.
:- mode map6(pred(in, out, out, out, out, out, out) is cc_multi, in, out,
    out, out, out, out, out) is cc_multi.
:- mode map6(pred(in, out, out, out, out, out, out) is semidet, in, out,
    out, out, out, out, out) is semidet.
:- mode map6(pred(in, out, out, out, out, out, out) is multi, in, out,
    out, out, out, out, out) is multi.
:- mode map6(pred(in, out, out, out, out, out, out) is nondet, in, out,
    out, out, out, out, out) is nondet.
:- mode map6(pred(in, in, in, in, in, in, in) is semidet, in, in, in, in,
    in, in, in) is semidet.

    % map7(T, L, M1, M2, M3, M4, M5, M6, M7) uses the closure T
    % to transform the elements of L into the elements of M1, M2, M3, M4,
```

```
    % M5, M6 and M7.
    %
:- pred map7(pred(A, B, C, D, E, F, G, H),
    one_or_more(A), one_or_more(B), one_or_more(C), one_or_more(D),
    one_or_more(E), one_or_more(F), one_or_more(G), one_or_more(H)).
:- mode map7(pred(in, out, out, out, out, out, out, out) is det,
    in, out, out, out, out, out, out, out) is det.
:- mode map7(pred(in, out, out, out, out, out, out, out) is cc_multi,
    in, out, out, out, out, out, out, out) is cc_multi.
:- mode map7(pred(in, out, out, out, out, out, out, out) is semidet,
    in, out, out, out, out, out, out, out) is semidet.
:- mode map7(pred(in, out, out, out, out, out, out, out) is multi,
    in, out, out, out, out, out, out, out) is multi.
:- mode map7(pred(in, out, out, out, out, out, out, out) is nondet,
    in, out, out, out, out, out, out, out) is nondet.
:- mode map7(pred(in, in, in, in, in, in, in, in) is semidet,
    in, in, in, in, in, in, in, in) is semidet.

    % map8(T, L, M1, M2, M3, M4, M5, M6, M7) uses the closure T
    % to transform the elements of L into the elements of M1, M2, M3, M4,
    % M5, M6, M7 and M8.
    %
:- pred map8(pred(A, B, C, D, E, F, G, H, I),
    one_or_more(A), one_or_more(B), one_or_more(C), one_or_more(D),
    one_or_more(E), one_or_more(F), one_or_more(G), one_or_more(H),
    one_or_more(I)).
:- mode map8(pred(in, out, out, out, out, out, out, out, out) is det,
    in, out, out, out, out, out, out, out, out) is det.
:- mode map8(pred(in, out, out, out, out, out, out, out, out) is cc_multi,
    in, out, out, out, out, out, out, out, out) is cc_multi.
:- mode map8(pred(in, out, out, out, out, out, out, out, out) is semidet,
    in, out, out, out, out, out, out, out, out) is semidet.
:- mode map8(pred(in, out, out, out, out, out, out, out, out) is multi,
    in, out, out, out, out, out, out, out, out) is multi.
:- mode map8(pred(in, out, out, out, out, out, out, out, out) is nondet,
    in, out, out, out, out, out, out, out, out) is nondet.
:- mode map8(pred(in, in, in, in, in, in, in, in, in) is semidet,
    in, in, in, in, in, in, in, in, in) is semidet.

%-----------------------------------------------%

    % map_corresponding(F, [A1, .. An], [B1, .. Bn]) =
    %   [F(A1, B1), .., F(An, Bn)].
    %
    % An exception is raised if the list arguments differ in length.
    %
:- func map_corresponding(func(A, B) = R, one_or_more(A), one_or_more(B))
```

```
    = one_or_more(R).
:- pred map_corresponding(pred(A, B, R), one_or_more(A), one_or_more(B),
    one_or_more(R)).
:- mode map_corresponding(in(pred(in, in, out) is det), in, in, out)
    is det.
:- mode map_corresponding(in(pred(in, in, out) is semidet), in, in, out)
    is semidet.

    % map_corresponding3(F, [A1, .. An], [B1, .. Bn], [C1, .. Cn]) =
    %   [F(A1, B1, C1), .., F(An, Bn, Cn)].
    %
    % An exception is raised if the list arguments differ in length.
    %
:- func map_corresponding3(func(A, B, C) = R,
    one_or_more(A), one_or_more(B), one_or_more(C)) = one_or_more(R).
:- pred map_corresponding3(pred(A, B, C, R), one_or_more(A), one_or_more(B),
    one_or_more(C), one_or_more(R)).
:- mode map_corresponding3(in(pred(in, in, in, out) is det),
    in, in, in, out) is det.
:- mode map_corresponding3(in(pred(in, in, in, out) is semidet),
    in, in, in, out) is semidet.

%-----------------------------------------------%

    % filter_map_corresponding/3 is like map_corresponding/3
    % except the function argument is semidet and the output list
    % consists of only those applications of the function argument that
    % succeeded.
    %
:- func filter_map_corresponding(
    (func(A, B) = R)::(func(in, in) = out is semidet),
    one_or_more(A)::in, one_or_more(B)::in) = (list(R)::out) is det.
:- pred filter_map_corresponding(
    pred(A, B, R)::in(pred(in, in, out) is semidet),
    one_or_more(A)::in, one_or_more(B)::in, list(R)::out) is det.

    % filter_map_corresponding3/4 is like map_corresponding3/4
    % except the function argument is semidet and the output list
    % consists of only those applications of the function argument that
    % succeeded.
    %
:- func filter_map_corresponding3(
    (func(A, B, C) = R)::in(func(in, in, in) = out is semidet),
    one_or_more(A)::in, one_or_more(B)::in, one_or_more(C)::in)
    = (list(R)::out) is det.
:- pred filter_map_corresponding3(
    pred(A, B, C, R)::in(pred(in, in, in, out) is semidet),
```

```
    one_or_more(A)::in, one_or_more(B)::in, one_or_more(C)::in,
    list(R)::out) is det.

%-----------------------------------------------%

    % foldl(Func, List, Start) = End calls Func with each element of List
    % (working left-to-right) and an accumulator (with the initial value
    % of Start), and returns the final value in End.
    %
:- func foldl(func(L, A) = A, one_or_more(L), A) = A.

    % foldl(Pred, List, Start, End) calls Pred with each element of List
    % (working left-to-right) and an accumulator (with the initial value
    % of Start), and returns the final value in End.
    %
:- pred foldl(pred(L, A, A), one_or_more(L), A, A).
:- mode foldl(pred(in, in, out) is det, in, in, out) is det.
:- mode foldl(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode foldl(pred(in, di, uo) is det, in, di, uo) is det.
:- mode foldl(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode foldl(pred(in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode foldl(pred(in, di, uo) is semidet, in, di, uo) is semidet.
:- mode foldl(pred(in, in, out) is multi, in, in, out) is multi.
:- mode foldl(pred(in, in, out) is nondet, in, in, out) is nondet.
:- mode foldl(pred(in, mdi, muo) is nondet, in, mdi, muo) is nondet.
:- mode foldl(pred(in, in, out) is cc_multi, in, in, out) is cc_multi.
:- mode foldl(pred(in, di, uo) is cc_multi, in, di, uo) is cc_multi.

    % foldl2(Pred, List, !Acc1, !Acc2):
    % Does the same job as foldl, but with two accumulators.
    % (Although no more expressive than foldl, this is often
    % a more convenient format, and a little more efficient).
    %
:- pred foldl2(pred(L, A, A, Z, Z), one_or_more(L), A, A, Z, Z).
:- mode foldl2(pred(in, in, out, in, out) is det,
    in, in, out, in, out) is det.
:- mode foldl2(pred(in, in, out, mdi, muo) is det,
    in, in, out, mdi, muo) is det.
:- mode foldl2(pred(in, in, out, di, uo) is det,
    in, in, out, di, uo) is det.
:- mode foldl2(pred(in, di, uo, di, uo) is det,
    in, di, uo, di, uo) is det.
:- mode foldl2(pred(in, in, out, in, out) is semidet,
    in, in, out, in, out) is semidet.
:- mode foldl2(pred(in, in, out, mdi, muo) is semidet,
    in, in, out, mdi, muo) is semidet.
:- mode foldl2(pred(in, in, out, di, uo) is semidet,
```

```
    in, in, out, di, uo) is semidet.
:- mode foldl2(pred(in, in, out, in, out) is nondet,
    in, in, out, in, out) is nondet.
:- mode foldl2(pred(in, in, out, mdi, muo) is nondet,
    in, in, out, mdi, muo) is nondet.
:- mode foldl2(pred(in, in, out, in, out) is cc_multi,
    in, in, out, in, out) is cc_multi.
:- mode foldl2(pred(in, in, out, mdi, muo) is cc_multi,
    in, in, out, mdi, muo) is cc_multi.
:- mode foldl2(pred(in, in, out, di, uo) is cc_multi,
    in, in, out, di, uo) is cc_multi.
:- mode foldl2(pred(in, di, uo, di, uo) is cc_multi,
    in, di, uo, di, uo) is cc_multi.

    % foldl3(Pred, List, !Acc1, !Acc2, !Acc3):
    % Does the same job as foldl, but with three accumulators.
    % (Although no more expressive than foldl, this is often
    % a more convenient format, and a little more efficient).
    %
:- pred foldl3(pred(L, A, A, B, B, C, C), one_or_more(L),
    A, A, B, B, C, C).
:- mode foldl3(pred(in, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out) is det.
:- mode foldl3(pred(in, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, mdi, muo) is det.
:- mode foldl3(pred(in, in, out, in, out, di, uo) is det,
    in, in, out, in, out, di, uo) is det.
:- mode foldl3(pred(in, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out) is semidet.
:- mode foldl3(pred(in, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, mdi, muo) is semidet.
:- mode foldl3(pred(in, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, di, uo) is semidet.
:- mode foldl3(pred(in, in, out, in, out, in, out) is nondet,
    in, in, out, in, out, in, out) is nondet.
:- mode foldl3(pred(in, in, out, in, out, mdi, muo) is nondet,
    in, in, out, in, out, mdi, muo) is nondet.
:- mode foldl3(pred(in, in, out, in, out, in, out) is cc_multi,
    in, in, out, in, out, in, out) is cc_multi.
:- mode foldl3(pred(in, in, out, in, out, di, uo) is cc_multi,
    in, in, out, in, out, di, uo) is cc_multi.

    % foldl4(Pred, List, !Acc1, !Acc2, !Acc3, !Acc4):
    % Does the same job as foldl, but with four accumulators.
    % (Although no more expressive than foldl, this is often
    % a more convenient format, and a little more efficient).
    %
```

```
:- pred foldl4(pred(L, A, A, B, B, C, C, D, D), one_or_more(L),
    A, A, B, B, C, C, D, D).
:- mode foldl4(pred(in, in, out, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out, in, out) is det.
:- mode foldl4(pred(in, in, out, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, in, out, mdi, muo) is det.
:- mode foldl4(pred(in, in, out, in, out, in, out, di, uo) is det,
    in, in, out, in, out, in, out, di, uo) is det.
:- mode foldl4(pred(in, in, out, in, out, in, out, in, out) is cc_multi,
    in, in, out, in, out, in, out, in, out) is cc_multi.
:- mode foldl4(pred(in, in, out, in, out, in, out, di, uo) is cc_multi,
    in, in, out, in, out, in, out, di, uo) is cc_multi.
:- mode foldl4(pred(in, in, out, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out, in, out) is semidet.
:- mode foldl4(pred(in, in, out, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode foldl4(pred(in, in, out, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, in, out, di, uo) is semidet.
:- mode foldl4(pred(in, in, out, in, out, in, out, in, out) is nondet,
    in, in, out, in, out, in, out, in, out) is nondet.
:- mode foldl4(pred(in, in, out, in, out, in, out, mdi, muo) is nondet,
    in, in, out, in, out, in, out, mdi, muo) is nondet.

    % foldl5(Pred, List, !Acc1, !Acc2, !Acc3, !Acc4, !Acc5):
    % Does the same job as foldl, but with five accumulators.
    % (Although no more expressive than foldl, this is often
    % a more convenient format, and a little more efficient).
    %
:- pred foldl5(pred(L, A, A, B, B, C, C, D, D, E, E), one_or_more(L),
    A, A, B, B, C, C, D, D, E, E).
:- mode foldl5(pred(in, in, out, in, out, in, out, in, out, in, out)
    is det,
    in, in, out, in, out, in, out, in, out, in, out) is det.
:- mode foldl5(pred(in, in, out, in, out, in, out, in, out, mdi, muo)
    is det,
    in, in, out, in, out, in, out, in, out, mdi, muo) is det.
:- mode foldl5(pred(in, in, out, in, out, in, out, in, out, di, uo)
    is det,
    in, in, out, in, out, in, out, in, out, di, uo) is det.
:- mode foldl5(pred(in, in, out, in, out, in, out, in, out, in, out)
    is semidet,
    in, in, out, in, out, in, out, in, out, in, out) is semidet.
:- mode foldl5(pred(in, in, out, in, out, in, out, in, out, mdi, muo)
    is semidet,
    in, in, out, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode foldl5(pred(in, in, out, in, out, in, out, in, out, di, uo)
    is semidet,
```

```
    in, in, out, in, out, in, out, in, out, di, uo) is semidet.
:- mode foldl5(pred(in, in, out, in, out, in, out, in, out, in, out)
    is nondet,
    in, in, out, in, out, in, out, in, out, in, out) is nondet.
:- mode foldl5(pred(in, in, out, in, out, in, out, in, out, mdi, muo)
    is nondet,
    in, in, out, in, out, in, out, in, out, mdi, muo) is nondet.
:- mode foldl5(pred(in, in, out, in, out, in, out, in, out, in, out)
    is cc_multi,
    in, in, out, in, out, in, out, in, out, in, out) is cc_multi.
:- mode foldl5(pred(in, in, out, in, out, in, out, in, out, di, uo)
    is cc_multi,
    in, in, out, in, out, in, out, in, out, di, uo) is cc_multi.

    % foldl6(Pred, List, !Acc1, !Acc2, !Acc3, !Acc4, !Acc5, !Acc6):
    % Does the same job as foldl, but with six accumulators.
    % (Although no more expressive than foldl, this is often
    % a more convenient format, and a little more efficient).
    %
:- pred foldl6(pred(L, A, A, B, B, C, C, D, D, E, E, F, F), one_or_more(L),
    A, A, B, B, C, C, D, D, E, E, F, F).
:- mode foldl6(pred(in, in, out, in, out, in, out, in, out, in, out,
    in, out) is det,
    in, in, out, in, out, in, out, in, out, in, out, in, out) is det.
:- mode foldl6(pred(in, in, out, in, out, in, out, in, out, in, out,
    mdi, muo) is det,
    in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is det.
:- mode foldl6(pred(in, in, out, in, out, in, out, in, out, in, out,
    di, uo) is det,
    in, in, out, in, out, in, out, in, out, in, out, di, uo) is det.
:- mode foldl6(pred(in, in, out, in, out, in, out, in, out, in, out,
    in, out) is cc_multi,
    in, in, out, in, out, in, out, in, out, in, out, in, out) is cc_multi.
:- mode foldl6(pred(in, in, out, in, out, in, out, in, out, in, out,
    di, uo) is cc_multi,
    in, in, out, in, out, in, out, in, out, in, out, di, uo) is cc_multi.
:- mode foldl6(pred(in, in, out, in, out, in, out, in, out, in, out,
    in, out) is semidet,
    in, in, out, in, out, in, out, in, out, in, out, in, out) is semidet.
:- mode foldl6(pred(in, in, out, in, out, in, out, in, out, in, out,
    mdi, muo) is semidet,
    in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode foldl6(pred(in, in, out, in, out, in, out, in, out, in, out,
    di, uo) is semidet,
    in, in, out, in, out, in, out, in, out, in, out, di, uo) is semidet.
:- mode foldl6(pred(in, in, out, in, out, in, out, in, out, in, out,
    in, out) is nondet,
```

```
    in, in, out, in, out, in, out, in, out, in, out, in, out) is nondet.

%-----------------------------------------------%

    % foldr(Func, List, Start) = End calls Func with each element of List
    % (working right-to-left) and an accumulator (with the initial value
    % of Start), and returns the final value in End.
    %
:- func foldr(func(L, A) = A, one_or_more(L), A) = A.

    % foldr(Pred, List, Start, End) calls Pred with each element of List
    % (working right-to-left) and an accumulator (with the initial value
    % of Start), and returns the final value in End.
    %
:- pred foldr(pred(L, A, A), one_or_more(L), A, A).
:- mode foldr(pred(in, in, out) is det, in, in, out) is det.
:- mode foldr(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode foldr(pred(in, di, uo) is det, in, di, uo) is det.
:- mode foldr(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode foldr(pred(in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode foldr(pred(in, di, uo) is semidet, in, di, uo) is semidet.
:- mode foldr(pred(in, in, out) is multi, in, in, out) is multi.
:- mode foldr(pred(in, in, out) is nondet, in, in, out) is nondet.
:- mode foldr(pred(in, mdi, muo) is nondet, in, mdi, muo) is nondet.
:- mode foldr(pred(in, di, uo) is cc_multi, in, di, uo) is cc_multi.
:- mode foldr(pred(in, in, out) is cc_multi, in, in, out) is cc_multi.

    % foldr2(Pred, List, !Acc1, !Acc2):
    % Does the same job as foldr, but with two accumulators.
    % (Although no more expressive than foldl, this is often
    % a more convenient format, and a little more efficient).
    %
:- pred foldr2(pred(L, A, A, B, B), one_or_more(L), A, A, B, B).
:- mode foldr2(pred(in, in, out, in, out) is det, in, in, out,
    in, out) is det.
:- mode foldr2(pred(in, in, out, mdi, muo) is det, in, in, out,
    mdi, muo) is det.
:- mode foldr2(pred(in, in, out, di, uo) is det, in, in, out,
    di, uo) is det.
:- mode foldr2(pred(in, in, out, in, out) is semidet, in, in, out,
    in, out) is semidet.
:- mode foldr2(pred(in, in, out, mdi, muo) is semidet, in, in, out,
    mdi, muo) is semidet.
:- mode foldr2(pred(in, in, out, di, uo) is semidet, in, in, out,
    di, uo) is semidet.
:- mode foldr2(pred(in, in, out, in, out) is nondet, in, in, out,
    in, out) is nondet.
```

```
:- mode foldr2(pred(in, in, out, mdi, muo) is nondet, in, in, out,
    mdi, muo) is nondet.

    % foldr3(Pred, List, !Acc1, !Acc2, !Acc3):
    % Does the same job as foldr, but with two accumulators.
    % (Although no more expressive than foldl, this is often
    % a more convenient format, and a little more efficient).
    %
:- pred foldr3(pred(L, A, A, B, B, C, C), one_or_more(L), A, A, B, B, C, C).
:- mode foldr3(pred(in, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out) is det.
:- mode foldr3(pred(in, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, mdi, muo) is det.
:- mode foldr3(pred(in, in, out, in, out, di, uo) is det, in,
    in, out, in, out, di, uo) is det.
:- mode foldr3(pred(in, in, out, in, out, in, out) is semidet, in,
    in, out, in, out, in, out) is semidet.
:- mode foldr3(pred(in, in, out, in, out, mdi, muo) is semidet, in,
    in, out, in, out, mdi, muo) is semidet.
:- mode foldr3(pred(in, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, di, uo) is semidet.
:- mode foldr3(pred(in, in, out, in, out, in, out) is nondet, in,
    in, out, in, out, in, out) is nondet.
:- mode foldr3(pred(in, in, out, in, out, mdi, muo) is nondet, in,
    in, out, in, out, mdi, muo) is nondet.

%-----------------------------------------------%

    % foldl_corresponding(P, As, Bs, !Acc):
    %
    % Does the same job as foldl, but works on two lists in parallel.
    % An exception is raised if the list arguments differ in length.
    %
:- pred foldl_corresponding(pred(A, B, C, C),
    one_or_more(A), one_or_more(B), C, C).
:- mode foldl_corresponding(pred(in, in, in, out) is det,
    in, in, in, out) is det.
:- mode foldl_corresponding(pred(in, in, mdi, muo) is det,
    in, in, mdi, muo) is det.
:- mode foldl_corresponding(pred(in, in, di, uo) is det,
    in, in, di, uo) is det.
:- mode foldl_corresponding(pred(in, in, in, out) is semidet,
    in, in, in, out) is semidet.
:- mode foldl_corresponding(pred(in, in, mdi, muo) is semidet,
    in, in, mdi, muo) is semidet.
:- mode foldl_corresponding(pred(in, in, di, uo) is semidet,
    in, in, di, uo) is semidet.
```

```
:- mode foldl_corresponding(pred(in, in, in, out) is nondet,
    in, in, in, out) is nondet.
:- mode foldl_corresponding(pred(in, in, mdi, muo) is nondet,
    in, in, mdi, muo) is nondet.
:- mode foldl_corresponding(pred(in, in, in, out) is cc_multi,
    in, in, in, out) is cc_multi.
:- mode foldl_corresponding(pred(in, in, di, uo) is cc_multi,
    in, in, di, uo) is cc_multi.

:- func foldl_corresponding(func(A, B, C) = C,
    one_or_more(A), one_or_more(B), C) = C.

    % foldl2_corresponding(F, As, Bs, !Acc1, !Acc2):
    % Does the same job as foldl_corresponding, but has two
    % accumulators.
    %
:- pred foldl2_corresponding(pred(A, B, C, C, D, D),
    one_or_more(A), one_or_more(B), C, C, D, D).
:- mode foldl2_corresponding(pred(in, in, in, out, in, out) is det,
    in, in, in, out, in, out) is det.
:- mode foldl2_corresponding(pred(in, in, in, out, mdi, muo) is det,
    in, in, in, out, mdi, muo) is det.
:- mode foldl2_corresponding(pred(in, in, in, out, di, uo) is det,
    in, in, in, out, di, uo) is det.
:- mode foldl2_corresponding(pred(in, in, in, out, in, out) is semidet,
    in, in, in, out, in, out) is semidet.
:- mode foldl2_corresponding(pred(in, in, in, out, mdi, muo) is semidet,
    in, in, in, out, mdi, muo) is semidet.
:- mode foldl2_corresponding(pred(in, in, in, out, di, uo) is semidet,
    in, in, in, out, di, uo) is semidet.
:- mode foldl2_corresponding(pred(in, in, in, out, in, out) is nondet,
    in, in, in, out, in, out) is nondet.
:- mode foldl2_corresponding(pred(in, in, in, out, mdi, muo) is nondet,
    in, in, in, out, mdi, muo) is nondet.
:- mode foldl2_corresponding(pred(in, in, in, out, in, out) is cc_multi,
    in, in, in, out, in, out) is cc_multi.
:- mode foldl2_corresponding(pred(in, in, in, out, di, uo) is cc_multi,
    in, in, in, out, di, uo) is cc_multi.

    % foldl3_corresponding(F, As, Bs, !Acc1, !Acc2, !Acc3):
    % Does the same job as foldl_corresponding, but has three
    % accumulators.
    %
:- pred foldl3_corresponding(pred(A, B, C, C, D, D, E, E),
    one_or_more(A), one_or_more(B), C, C, D, D, E, E).
:- mode foldl3_corresponding(
    pred(in, in, in, out, in, out, in, out) is det, in, in, in, out,
```

```
    in, out, in, out) is det.
:- mode foldl3_corresponding(
    pred(in, in, in, out, in, out, mdi, muo) is det, in, in, in, out,
    in, out, mdi, muo) is det.
:- mode foldl3_corresponding(
    pred(in, in, in, out, in, out, di, uo) is det, in, in, in, out,
    in, out, di, uo) is det.
:- mode foldl3_corresponding(
    pred(in, in, in, out, in, out, in, out) is semidet, in, in, in, out,
    in, out, in, out) is semidet.
:- mode foldl3_corresponding(
    pred(in, in, in, out, in, out, mdi, muo) is semidet, in, in, in, out,
    in, out, mdi, muo) is semidet.
:- mode foldl3_corresponding(
    pred(in, in, in, out, in, out, di, uo) is semidet, in, in, in, out,
    in, out, di, uo) is semidet.

    % foldl_corresponding3(P, As, Bs, Cs, !Acc):
    % Like foldl_corresponding but folds over three corresponding
    % lists.
    %
:- pred foldl_corresponding3(pred(A, B, C, D, D),
    one_or_more(A), one_or_more(B), one_or_more(C), D, D).
:- mode foldl_corresponding3(pred(in, in, in, in, out) is det,
    in, in, in, in, out) is det.
:- mode foldl_corresponding3(pred(in, in, in, mdi, muo) is det,
    in, in, in, mdi, muo) is det.
:- mode foldl_corresponding3(pred(in, in, in, di, uo) is det,
    in, in, in, di, uo) is det.
:- mode foldl_corresponding3(pred(in, in, in, in, out) is semidet,
    in, in, in, in, out) is semidet.
:- mode foldl_corresponding3(pred(in, in, in, mdi, muo) is semidet,
    in, in, in, mdi, muo) is semidet.
:- mode foldl_corresponding3(pred(in, in, in, di, uo) is semidet,
    in, in, in, di, uo) is semidet.

    % foldl2_corresponding3(P, As, Bs, Cs, !Acc1, !Acc2):
    % like foldl_corresponding3 but with two accumulators.
    %
:- pred foldl2_corresponding3(pred(A, B, C, D, D, E, E),
    one_or_more(A), one_or_more(B), one_or_more(C), D, D, E, E).
:- mode foldl2_corresponding3(pred(in, in, in, in, out, in, out) is det,
    in, in, in, in, out, in, out) is det.
:- mode foldl2_corresponding3(pred(in, in, in, in, out, mdi, muo) is det,
    in, in, in, in, out, mdi, muo) is det.
:- mode foldl2_corresponding3(pred(in, in, in, in, out, di, uo) is det,
    in, in, in, in, out, di, uo) is det.
```

```
:- mode foldl2_corresponding3(
    pred(in, in, in, in, out, in, out) is semidet,
    in, in, in, in, out, in, out) is semidet.
:- mode foldl2_corresponding3(
    pred(in, in, in, in, out, mdi, muo) is semidet,
    in, in, in, in, out, mdi, muo) is semidet.
:- mode foldl2_corresponding3(
    pred(in, in, in, in, out, di, uo) is semidet,
    in, in, in, in, out, di, uo) is semidet.

    % foldl3_corresponding3(P, As, Bs, Cs, !Acc1, !Acc2, !Acc3):
    % like foldl_corresponding3 but with three accumulators.
    %
:- pred foldl3_corresponding3(pred(A, B, C, D, D, E, E, F, F),
    one_or_more(A), one_or_more(B), one_or_more(C), D, D, E, E, F, F).
:- mode foldl3_corresponding3(
    pred(in, in, in, in, out, in, out, in, out) is det,
    in, in, in, in, out, in, out, in, out) is det.
:- mode foldl3_corresponding3(
    pred(in, in, in, in, out, in, out, mdi, muo) is det,
    in, in, in, in, out, in, out, mdi, muo) is det.
:- mode foldl3_corresponding3(
    pred(in, in, in, in, out, in, out, di, uo) is det,
    in, in, in, in, out, in, out, di, uo) is det.
:- mode foldl3_corresponding3(
    pred(in, in, in, in, out, in, out, in, out) is semidet,
    in, in, in, in, out, in, out, in, out) is semidet.
:- mode foldl3_corresponding3(
    pred(in, in, in, in, out, in, out, mdi, muo) is semidet,
    in, in, in, in, out, in, out, mdi, muo) is semidet.
:- mode foldl3_corresponding3(
    pred(in, in, in, in, out, in, out, di, uo) is semidet,
    in, in, in, in, out, in, out, di, uo) is semidet.

    % foldl4_corresponding3(P, As, Bs, Cs, !Acc1, !Acc2, !Acc3, !Acc4):
    % like foldl_corresponding3 but with four accumulators.
    %
:- pred foldl4_corresponding3(pred(A, B, C, D, D, E, E, F, F, G, G),
    one_or_more(A), one_or_more(B), one_or_more(C), D, D, E, E, F, F, G, G).
:- mode foldl4_corresponding3(
    pred(in, in, in, in, out, in, out, in, out, in, out) is det,
    in, in, in, in, out, in, out, in, out, in, out) is det.
:- mode foldl4_corresponding3(
    pred(in, in, in, in, out, in, out, in, out, mdi, muo) is det,
    in, in, in, in, out, in, out, in, out, mdi, muo) is det.
:- mode foldl4_corresponding3(
    pred(in, in, in, in, out, in, out, in, out, di, uo) is det,
```

```
    in, in, in, in, out, in, out, in, out, di, uo) is det.
:- mode foldl4_corresponding3(
    pred(in, in, in, in, out, in, out, in, out, in, out) is semidet,
    in, in, in, in, out, in, out, in, out, in, out) is semidet.
:- mode foldl4_corresponding3(
    pred(in, in, in, in, out, in, out, in, out, mdi, muo) is semidet,
    in, in, in, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode foldl4_corresponding3(
    pred(in, in, in, in, out, in, out, in, out, di, uo) is semidet,
    in, in, in, in, out, in, out, in, out, di, uo) is semidet.

%---------------------------------------------------%

    % map_foldl(Pred, InList, OutList, Start, End) calls Pred
    % with an accumulator (with the initial value of Start) on
    % each element of InList (working left-to-right) to transform
    % InList into OutList. The final value of the accumulator is
    % returned in End.
    %
:- pred map_foldl(pred(L, M, A, A), one_or_more(L), one_or_more(M), A, A).
:- mode map_foldl(pred(in, out, in, out) is det, in, out, in, out)
    is det.
:- mode map_foldl(pred(in, out, mdi, muo) is det, in, out, mdi, muo)
    is det.
:- mode map_foldl(pred(in, out, di, uo) is det, in, out, di, uo)
    is det.
:- mode map_foldl(pred(in, out, in, out) is semidet, in, out, in, out)
    is semidet.
:- mode map_foldl(pred(in, out, mdi, muo) is semidet, in, out, mdi, muo)
    is semidet.
:- mode map_foldl(pred(in, out, di, uo) is semidet, in, out, di, uo)
    is semidet.
:- mode map_foldl(pred(in, in, di, uo) is semidet, in, in, di, uo)
    is semidet.
:- mode map_foldl(pred(in, out, in, out) is nondet, in, out, in, out)
    is nondet.
:- mode map_foldl(pred(in, out, mdi, muo) is nondet, in, out, mdi, muo)
    is nondet.
:- mode map_foldl(pred(in, out, in, out) is cc_multi, in, out, in, out)
    is cc_multi.
:- mode map_foldl(pred(in, out, mdi, muo) is cc_multi, in, out, mdi, muo)
    is cc_multi.
:- mode map_foldl(pred(in, out, di, uo) is cc_multi, in, out, di, uo)
    is cc_multi.

    % Same as map_foldl, but with two accumulators.
    %
```

```
:- pred map_foldl2(pred(L, M, A, A, B, B),
    one_or_more(L), one_or_more(M), A, A, B, B).
:- mode map_foldl2(pred(in, out, in, out, in, out) is det,
    in, out, in, out, in, out) is det.
:- mode map_foldl2(pred(in, out, in, out, mdi, muo) is det,
    in, out, in, out, mdi, muo) is det.
:- mode map_foldl2(pred(in, out, in, out, di, uo) is det,
    in, out, in, out, di, uo) is det.
:- mode map_foldl2(pred(in, out, in, out, in, out) is semidet,
    in, out, in, out, in, out) is semidet.
:- mode map_foldl2(pred(in, out, in, out, mdi, muo) is semidet,
    in, out, in, out, mdi, muo) is semidet.
:- mode map_foldl2(pred(in, out, in, out, di, uo) is semidet,
    in, out, in, out, di, uo) is semidet.
:- mode map_foldl2(pred(in, in, in, out, di, uo) is semidet,
    in, in, in, out, di, uo) is semidet.
:- mode map_foldl2(pred(in, out, in, out, in, out) is cc_multi,
    in, out, in, out, in, out) is cc_multi.
:- mode map_foldl2(pred(in, out, in, out, mdi, muo) is cc_multi,
    in, out, in, out, mdi, muo) is cc_multi.
:- mode map_foldl2(pred(in, out, in, out, di, uo) is cc_multi,
    in, out, in, out, di, uo) is cc_multi.
:- mode map_foldl2(pred(in, out, in, out, in, out) is nondet,
    in, out, in, out, in, out) is nondet.

    % Same as map_foldl, but with three accumulators.
    %
:- pred map_foldl3(pred(L, M, A, A, B, B, C, C),
    one_or_more(L), one_or_more(M), A, A, B, B, C, C).
:- mode map_foldl3(pred(in, out, in, out, in, out, di, uo) is det,
    in, out, in, out, in, out, di, uo) is det.
:- mode map_foldl3(pred(in, out, in, out, in, out, in, out) is det,
    in, out, in, out, in, out, in, out) is det.
:- mode map_foldl3(pred(in, out, in, out, in, out, di, uo) is cc_multi,
    in, out, in, out, in, out, di, uo) is cc_multi.
:- mode map_foldl3(pred(in, out, in, out, in, out, in, out) is cc_multi,
    in, out, in, out, in, out, in, out) is cc_multi.
:- mode map_foldl3(pred(in, out, in, out, in, out, in, out) is semidet,
    in, out, in, out, in, out, in, out) is semidet.
:- mode map_foldl3(pred(in, out, in, out, in, out, in, out) is nondet,
    in, out, in, out, in, out, in, out) is nondet.

    % Same as map_foldl, but with four accumulators.
    %
:- pred map_foldl4(pred(L, M, A, A, B, B, C, C, D, D),
    one_or_more(L), one_or_more(M), A, A, B, B, C, C, D, D).
:- mode map_foldl4(pred(in, out, in, out, in, out, in, out, di, uo)
```

```
    is det,
    in, out, in, out, in, out, in, out, di, uo) is det.
:- mode map_foldl4(pred(in, out, in, out, in, out, in, out, in, out)
    is det,
    in, out, in, out, in, out, in, out, in, out) is det.
:- mode map_foldl4(pred(in, out, in, out, in, out, in, out, di, uo)
    is cc_multi,
    in, out, in, out, in, out, in, out, di, uo) is cc_multi.
:- mode map_foldl4(pred(in, out, in, out, in, out, in, out, in, out)
    is cc_multi,
    in, out, in, out, in, out, in, out, in, out) is cc_multi.
:- mode map_foldl4(pred(in, out, in, out, in, out, in, out, in, out)
    is semidet,
    in, out, in, out, in, out, in, out, in, out) is semidet.
:- mode map_foldl4(pred(in, out, in, out, in, out, in, out, in, out)
    is nondet,
    in, out, in, out, in, out, in, out, in, out) is nondet.

    % Same as map_foldl, but with five accumulators.
    %
:- pred map_foldl5(pred(L, M, A, A, B, B, C, C, D, D, E, E),
    one_or_more(L), one_or_more(M), A, A, B, B, C, C, D, D, E, E).
:- mode map_foldl5(pred(in, out, in, out, in, out, in, out, in, out,
    di, uo) is det,
    in, out, in, out, in, out, in, out, in, out, di, uo) is det.
:- mode map_foldl5(pred(in, out, in, out, in, out, in, out, in, out,
    in, out) is det,
    in, out, in, out, in, out, in, out, in, out, in, out) is det.
:- mode map_foldl5(pred(in, out, in, out, in, out, in, out, in, out,
    di, uo) is cc_multi,
    in, out, in, out, in, out, in, out, in, out, di, uo) is cc_multi.
:- mode map_foldl5(pred(in, out, in, out, in, out, in, out, in, out,
    in, out) is cc_multi,
    in, out, in, out, in, out, in, out, in, out, in, out) is cc_multi.
:- mode map_foldl5(pred(in, out, in, out, in, out, in, out, in, out,
    in, out) is semidet,
    in, out, in, out, in, out, in, out, in, out, in, out) is semidet.
:- mode map_foldl5(pred(in, out, in, out, in, out, in, out, in, out,
    in, out) is nondet,
    in, out, in, out, in, out, in, out, in, out, in, out) is nondet.

    % Same as map_foldl, but with six accumulators.
    %
:- pred map_foldl6(pred(L, M, A, A, B, B, C, C, D, D, E, E, F, F),
    one_or_more(L), one_or_more(M), A, A, B, B, C, C, D, D, E, E, F, F).
:- mode map_foldl6(pred(in, out, in, out, in, out, in, out, in, out,
    in, out, di, uo) is det,
```

```
    in, out, in, out, in, out, in, out, in, out, in, out, di, uo) is det.
:- mode map_foldl6(pred(in, out, in, out, in, out, in, out, in, out,
    in, out, in, out) is det,
    in, out, in, out, in, out, in, out, in, out, in, out, in, out) is det.
:- mode map_foldl6(pred(in, out, in, out, in, out, in, out, in, out,
    in, out, di, uo) is cc_multi,
    in, out, in, out, in, out, in, out, in, out, in, out, di, uo)
    is cc_multi.
:- mode map_foldl6(pred(in, out, in, out, in, out, in, out, in, out,
    in, out, in, out) is cc_multi,
    in, out, in, out, in, out, in, out, in, out, in, out, in, out)
    is cc_multi.
:- mode map_foldl6(pred(in, out, in, out, in, out, in, out, in, out,
    in, out, in, out) is semidet,
    in, out, in, out, in, out, in, out, in, out, in, out, in, out)
    is semidet.
:- mode map_foldl6(pred(in, out, in, out, in, out, in, out, in, out,
    in, out, in, out) is nondet,
    in, out, in, out, in, out, in, out, in, out, in, out, in, out)
    is nondet.

    % Same as map_foldl, but with two mapped outputs.
    %
:- pred map2_foldl(pred(L, M, N, A, A),
    one_or_more(L), one_or_more(M), one_or_more(N), A, A).
:- mode map2_foldl(pred(in, out, out, in, out) is det, in, out, out,
    in, out) is det.
:- mode map2_foldl(pred(in, out, out, mdi, muo) is det, in, out, out,
    mdi, muo) is det.
:- mode map2_foldl(pred(in, out, out, di, uo) is det, in, out, out,
    di, uo) is det.
:- mode map2_foldl(pred(in, out, out, in, out) is semidet, in, out, out,
    in, out) is semidet.
:- mode map2_foldl(pred(in, out, out, mdi, muo) is semidet, in, out, out,
    mdi, muo) is semidet.
:- mode map2_foldl(pred(in, out, out, di, uo) is semidet, in, out, out,
    di, uo) is semidet.
:- mode map2_foldl(pred(in, out, out, in, out) is nondet, in, out, out,
    in, out) is nondet.
:- mode map2_foldl(pred(in, out, out, mdi, muo) is nondet, in, out, out,
    mdi, muo) is nondet.
:- mode map2_foldl(pred(in, out, out, in, out) is cc_multi, in, out, out,
    in, out) is cc_multi.
:- mode map2_foldl(pred(in, out, out, di, uo) is cc_multi, in, out, out,
    di, uo) is cc_multi.

    % Same as map_foldl, but with two mapped outputs and two
```

```
    % accumulators.
    %
:- pred map2_foldl2(pred(L, M, N, A, A, B, B),
    one_or_more(L), one_or_more(M), one_or_more(N), A, A, B, B).
:- mode map2_foldl2(pred(in, out, out, in, out, di, uo) is det,
    in, out, out, in, out, di, uo) is det.
:- mode map2_foldl2(pred(in, out, out, in, out, in, out) is det,
    in, out, out, in, out, in, out) is det.
:- mode map2_foldl2(pred(in, out, out, in, out, di, uo) is cc_multi,
    in, out, out, in, out, di, uo) is cc_multi.
:- mode map2_foldl2(pred(in, out, out, in, out, in, out) is cc_multi,
    in, out, out, in, out, in, out) is cc_multi.
:- mode map2_foldl2(pred(in, out, out, in, out, in, out) is semidet,
    in, out, out, in, out, in, out) is semidet.
:- mode map2_foldl2(pred(in, out, out, in, out, in, out) is nondet,
    in, out, out, in, out, in, out) is nondet.

    % Same as map_foldl, but with two mapped outputs and three
    % accumulators.
    %
:- pred map2_foldl3(pred(L, M, N, A, A, B, B, C, C),
    one_or_more(L), one_or_more(M), one_or_more(N), A, A, B, B, C, C).
:- mode map2_foldl3(
    pred(in, out, out, in, out, in, out, in, out) is det,
    in, out, out, in, out, in, out, in, out) is det.
:- mode map2_foldl3(
    pred(in, out, out, in, out, in, out, di, uo) is det,
    in, out, out, in, out, in, out, di, uo) is det.
:- mode map2_foldl3(
    pred(in, out, out, in, out, in, out, in, out) is cc_multi,
    in, out, out, in, out, in, out, in, out) is cc_multi.
:- mode map2_foldl3(
    pred(in, out, out, in, out, in, out, di, uo) is cc_multi,
    in, out, out, in, out, in, out, di, uo) is cc_multi.
:- mode map2_foldl3(
    pred(in, out, out, in, out, in, out, in, out) is semidet,
    in, out, out, in, out, in, out, in, out) is semidet.
:- mode map2_foldl3(
    pred(in, out, out, in, out, in, out, in, out) is nondet,
    in, out, out, in, out, in, out, in, out) is nondet.

    % Same as map_foldl, but with two mapped outputs and four
    % accumulators.
    %
:- pred map2_foldl4(pred(L, M, N, A, A, B, B, C, C, D, D),
    one_or_more(L), one_or_more(M), one_or_more(N), A, A, B, B, C, C, D, D).
:- mode map2_foldl4(
```

```
    pred(in, out, out, in, out, in, out, in, out, in, out) is det,
    in, out, out, in, out, in, out, in, out, in, out) is det.
:- mode map2_foldl4(
    pred(in, out, out, in, out, in, out, in, out, di, uo) is det,
    in, out, out, in, out, in, out, in, out, di, uo) is det.
:- mode map2_foldl4(
    pred(in, out, out, in, out, in, out, in, out, in, out) is cc_multi,
    in, out, out, in, out, in, out, in, out, in, out) is cc_multi.
:- mode map2_foldl4(
    pred(in, out, out, in, out, in, out, in, out, di, uo) is cc_multi,
    in, out, out, in, out, in, out, in, out, di, uo) is cc_multi.
:- mode map2_foldl4(
    pred(in, out, out, in, out, in, out, in, out, in, out) is semidet,
    in, out, out, in, out, in, out, in, out, in, out) is semidet.
:- mode map2_foldl4(
    pred(in, out, out, in, out, in, out, in, out, in, out) is nondet,
    in, out, out, in, out, in, out, in, out, in, out) is nondet.

    % Same as map_foldl, but with three mapped outputs.
    %
:- pred map3_foldl(pred(L, M, N, O, A, A),
    one_or_more(L), one_or_more(M), one_or_more(N), one_or_more(O), A, A).
:- mode map3_foldl(pred(in, out, out, out, in, out) is det, in, out, out,
    out, in, out) is det.
:- mode map3_foldl(pred(in, out, out, out, mdi, muo) is det, in, out, out,
    out, mdi, muo) is det.
:- mode map3_foldl(pred(in, out, out, out, di, uo) is det, in, out, out,
    out, di, uo) is det.
:- mode map3_foldl(pred(in, out, out, out, in, out) is semidet, in, out,
    out, out, in, out) is semidet.
:- mode map3_foldl(pred(in, out, out, out, mdi, muo) is semidet, in, out,
    out, out, mdi, muo) is semidet.
:- mode map3_foldl(pred(in, out, out, out, di, uo) is semidet, in, out,
    out, out, di, uo) is semidet.
:- mode map3_foldl(pred(in, out, out, out, in, out) is nondet, in, out,
    out, out, in, out) is nondet.
:- mode map3_foldl(pred(in, out, out, out, mdi, muo) is nondet, in, out,
    out, out, mdi, muo) is nondet.
:- mode map3_foldl(pred(in, out, out, out, in, out) is cc_multi, in, out,
    out, out, in, out) is cc_multi.
:- mode map3_foldl(pred(in, out, out, out, di, uo) is cc_multi, in, out,
    out, out, di, uo) is cc_multi.

    % Same as map_foldl, but with three mapped outputs and two
    % accumulators.
    %
:- pred map3_foldl2(pred(L, M, N, O, A, A, B, B), one_or_more(L),
```

```
    one_or_more(M), one_or_more(N), one_or_more(O), A, A, B, B).
:- mode map3_foldl2(pred(in, out, out, out, in, out, di, uo) is det,
    in, out, out, out, in, out, di, uo) is det.
:- mode map3_foldl2(pred(in, out, out, out, in, out, in, out) is det,
    in, out, out, out, in, out, in, out) is det.
:- mode map3_foldl2(pred(in, out, out, out, in, out, di, uo) is cc_multi,
    in, out, out, out, in, out, di, uo) is cc_multi.
:- mode map3_foldl2(pred(in, out, out, out, in, out, in, out) is cc_multi,
    in, out, out, out, in, out, in, out) is cc_multi.
:- mode map3_foldl2(pred(in, out, out, out, in, out, in, out) is semidet,
    in, out, out, out, in, out, in, out) is semidet.
:- mode map3_foldl2(pred(in, out, out, out, in, out, in, out) is nondet,
    in, out, out, out, in, out, in, out) is nondet.

    % Same as map_foldl, but with four mapped outputs.
    %
:- pred map4_foldl(pred(L, M, N, O, P, A, A), one_or_more(L), one_or_more(M),
one_or_more(N),
    one_or_more(O), one_or_more(P), A, A).
:- mode map4_foldl(pred(in, out, out, out, out, in, out) is det,
    in, out, out, out, out, in, out) is det.
:- mode map4_foldl(pred(in, out, out, out, out, mdi, muo) is det,
    in, out, out, out, out, mdi, muo) is det.
:- mode map4_foldl(pred(in, out, out, out, out, di, uo) is det,
    in, out, out, out, out, di, uo) is det.
:- mode map4_foldl(pred(in, out, out, out, out, in, out) is semidet,
    in, out, out, out, out, in, out) is semidet.
:- mode map4_foldl(pred(in, out, out, out, out, mdi, muo) is semidet,
    in, out, out, out, out, mdi, muo) is semidet.
:- mode map4_foldl(pred(in, out, out, out, out, di, uo) is semidet,
    in, out, out, out, out, di, uo) is semidet.
:- mode map4_foldl(pred(in, out, out, out, out, in, out) is nondet,
    in, out, out, out, out, in, out) is nondet.
:- mode map4_foldl(pred(in, out, out, out, out, mdi, muo) is nondet,
    in, out, out, out, out, mdi, muo) is nondet.
:- mode map4_foldl(pred(in, out, out, out, out, in, out) is cc_multi,
    in, out, out, out, out, in, out) is cc_multi.
:- mode map4_foldl(pred(in, out, out, out, out, di, uo) is cc_multi,
    in, out, out, out, out, di, uo) is cc_multi.

%-----------------------------------------------%

    % map_foldr(Pred, InList, OutList, Start, End) calls Pred
    % with an accumulator (with the initial value of Start) on
    % each element of InList (working right-to-left) to transform
    % InList into OutList. The final value of the accumulator is
    % returned in End.
```

```
    %
:- pred map_foldr(pred(L, M, A, A), one_or_more(L), one_or_more(M), A, A).
:- mode map_foldr(pred(in, out, in, out) is det, in, out, in, out)
    is det.
:- mode map_foldr(pred(in, out, mdi, muo) is det, in, out, mdi, muo)
    is det.
:- mode map_foldr(pred(in, out, di, uo) is det, in, out, di, uo)
    is det.
:- mode map_foldr(pred(in, out, in, out) is semidet, in, out, in, out)
    is semidet.
:- mode map_foldr(pred(in, out, mdi, muo) is semidet, in, out, mdi, muo)
    is semidet.
:- mode map_foldr(pred(in, out, di, uo) is semidet, in, out, di, uo)
    is semidet.
:- mode map_foldr(pred(in, in, di, uo) is semidet, in, in, di, uo)
    is semidet.

%-----------------------------------------------%

    % map_corresponding_foldl/6 is like map_corresponding except
    % that it has an accumulator threaded through it.
    %
:- pred map_corresponding_foldl(pred(A, B, C, D, D),
    one_or_more(A), one_or_more(B), one_or_more(C), D, D).
:- mode map_corresponding_foldl(pred(in, in, out, in, out) is det,
    in, in, out, in, out) is det.
:- mode map_corresponding_foldl(pred(in, in, out, mdi, muo) is det,
    in, in, out, mdi, muo) is det.
:- mode map_corresponding_foldl(pred(in, in, out, di, uo) is det,
    in, in, out, di, uo) is det.
:- mode map_corresponding_foldl(pred(in, in, out, in, out) is semidet,
    in, in, out, in, out) is semidet.
:- mode map_corresponding_foldl(pred(in, in, out, mdi, muo) is semidet,
    in, in, out, mdi, muo) is semidet.
:- mode map_corresponding_foldl(pred(in, in, out, di, uo) is semidet,
    in, in, out, di, uo) is semidet.

    % Like map_corresponding_foldl/6 except that it has two
    % accumulators.
    %
:- pred map_corresponding_foldl2(pred(A, B, C, D, D, E, E),
    one_or_more(A), one_or_more(B), one_or_more(C), D, D, E, E).
:- mode map_corresponding_foldl2(
    pred(in, in, out, in, out, in, out) is det, in, in, out, in, out,
    in, out) is det.
:- mode map_corresponding_foldl2(
    pred(in, in, out, in, out, mdi, muo) is det, in, in, out, in, out,
```

```
    mdi, muo) is det.
:- mode map_corresponding_foldl2(
    pred(in, in, out, in, out, di, uo) is det, in, in, out, in, out,
    di, uo) is det.
:- mode map_corresponding_foldl2(
    pred(in, in, out, in, out, in, out) is semidet, in, in, out, in, out,
    in, out) is semidet.
:- mode map_corresponding_foldl2(
    pred(in, in, out, in, out, mdi, muo) is semidet, in, in, out, in, out,
    mdi, muo) is semidet.
:- mode map_corresponding_foldl2(
    pred(in, in, out, in, out, di, uo) is semidet, in, in, out, in, out,
    di, uo) is semidet.

    % Like map_corresponding_foldl/6 except that it has three
    % accumulators.
    %
:- pred map_corresponding_foldl3(pred(A, B, C, D, D, E, E, F, F),
    one_or_more(A), one_or_more(B), one_or_more(C), D, D, E, E, F, F).
:- mode map_corresponding_foldl3(
    pred(in, in, out, in, out, in, out, in, out) is det, in, in, out, in, out,
    in, out, in, out) is det.
:- mode map_corresponding_foldl3(
    pred(in, in, out, in, out, in, out, mdi, muo) is det, in, in, out, in, out,
    in, out, mdi, muo) is det.
:- mode map_corresponding_foldl3(
    pred(in, in, out, in, out, in, out, di, uo) is det, in, in, out, in, out,
    in, out, di, uo) is det.
:- mode map_corresponding_foldl3(
    pred(in, in, out, in, out, in, out, in, out) is semidet, in, in, out,
    in, out, in, out, in, out) is semidet.
:- mode map_corresponding_foldl3(
    pred(in, in, out, in, out, in, out, mdi, muo) is semidet, in, in, out,
    in, out, in, out, mdi, muo) is semidet.
:- mode map_corresponding_foldl3(
    pred(in, in, out, in, out, in, out, di, uo) is semidet, in, in, out,
    in, out, in, out, di, uo) is semidet.

    % map_corresponding3_foldl/7 is like map_corresponding3 except
    % that it has an accumulator threaded through it.
    %
:- pred map_corresponding3_foldl(pred(A, B, C, D, E, E),
    one_or_more(A), one_or_more(B), one_or_more(C), one_or_more(D), E, E).
:- mode map_corresponding3_foldl(pred(in, in, in, out, in, out) is det,
    in, in, in, out, in, out) is det.
:- mode map_corresponding3_foldl(pred(in, in, in, out, mdi, muo) is det,
    in, in, in, out, mdi, muo) is det.
```

```
:- mode map_corresponding3_foldl(pred(in, in, in, out, di, uo) is det,
    in, in, in, out, di, uo) is det.
:- mode map_corresponding3_foldl(
    pred(in, in, in, out, in, out) is semidet,
    in, in, in, out, in, out) is semidet.
:- mode map_corresponding3_foldl(
    pred(in, in, in, out, mdi, muo) is semidet,
    in, in, in, out, mdi, muo) is semidet.
:- mode map_corresponding3_foldl(
    pred(in, in, in, out, di, uo) is semidet,
    in, in, in, out, di, uo) is semidet.

%-----------------------------------------------%

    % filter_map_foldl(Transformer, List, TrueList, Start, End):
    % Takes a predicate with one input argument, one output argument and an
    % accumulator. It is called with each element of List. If a call succeeds,
    % then the output is included in TrueList and the accumulator is updated.
    %
:- pred filter_map_foldl(
    pred(X, Y, A, A)::in(pred(in, out, in, out) is semidet),
    one_or_more(X)::in, list(Y)::out, A::in, A::out) is det.

%-----------------------------------------------%

% The following is a list of the functions and predicates that are
% present in list.m but not in one_or_more.m, together with the reasons
% for their absences.
%
% - pred is_empty/1
% - pred is_not_empty/1
%
%   When applied to known-to-be-nonempty lists, the outcomes of these tests
%   are known statically.
%
% - func det_head/1
% - func det_tail/1
%
%   For nonempty lists, the simple head and tail functions are already
%   deterministic.
%
% - func append/2
%
%   You can append two one_or_mores using the ++ function, and the compiler
%   already gets confused by whether references to append with an arity
%   less than three is a reference to the function or to the predicate version.
%
```

```
% - pred remove_suffix/3
%
%   When the suffix is removed, there may be no element left.
%
% - pred reverse_prepend/3
%
%   In list.m, it is part of the implementation of reverse, but one_or_more.m
%   implements reverse by calling list.reverse.
%   XXX We could implement reverse_prepend nevertheless.
%
% - pred insert/3
%
%   In the reverse modes, where the caller is deleting an element,
%   there may be no element left.
%
% - func ../2
%
%   Depending on the values of the lower and upper bounds, the resulting list
%   may be empty.
%
% - func det_last/1
% - pred det_last/2
% - pred det_split_last/3
%
%   For nonempty lists, the simple last function and predicate,
%   and the split_last predicate, are already deterministic.
%
% - pred take/3
% - pred det_take/3
% - func take_upto/2
% - pred take_upto/3
% - pred drop/3
% - pred det_drop/3
%
%   Depending on the value of the count parameter, the resulting list
%   may contain no elements.
%
% - pred take_while/4
% - func take_while/2
% - pred take_while/3
% - func drop_while/2
% - pred drop_while/3
%
%   Depending on the value of the count parameter, the resulting list
%   may contain no elements.
%
% - func duplicate/1
```

```
% - pred duplicate/2
%
%   Depending on the value of the count parameter, the resulting list
%   may contain no elements.

%---------------------------------------------------%
%---------------------------------------------------%
```

51 one_or_more_map

```
%---------------------------------------------------%
% vim: ts=4 sw=4 et ft=mercury
%---------------------------------------------------%
% Copyright (C) 2020 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: one_or_more_map.m.
%
% This file provides a version of the 'multi_map' ADT.
% A map (also known as a dictionary or an associative array) is a collection
% of (Key, Value) pairs which allows you to look up any Value given the Key.
% A multi_map is similar, but it allows more than one Value for each Key.
% A multi_map represents this by using list(V) as the range type, which works,
% but does not express the invariant maintained by the relevant operations,
% which is that these lists are never empty. A one_or_more_map is a multi_map
% in which key the range type is one_or_more(V), which *does* express
% this invariant.
%
%---------------------------------------------------%

:- module one_or_more_map.
:- interface.

:- import_module assoc_list.
:- import_module list.
:- import_module one_or_more.
:- import_module map.
:- import_module set.

%---------------------------------------------------%

:- type one_or_more_map(K, V) == map(K, one_or_more(V)).

%---------------------------------------------------%
```

```
    % Return an empty one_or_more_map.
    %
:- func init = one_or_more_map(K, V).
:- pred init(one_or_more_map(K, V)::uo) is det.

    % Check whether the one_or_more_map is empty.
    %
:- pred is_empty(one_or_more_map(K, V)::in) is semidet.

%---------------------------------------------------%

    % Check whether the one_or_more_map has an entry for the given key.
    %
:- pred contains(one_or_more_map(K, V)::in, K::in) is semidet.

    % Succeed once for each key-value pair in the one_or_more_map.
    %
:- pred member(one_or_more_map(K, V)::in, K::out, V::out) is nondet.

    % If the one_or_more_map has an entry for the given key, return the
    % list of corresponding values.
    %
:- pred search(one_or_more_map(K, V)::in, K::in, one_or_more(V)::out)
    is semidet.

    % If the one_or_more_map has an entry for the given key,
    % succeed once for each of the corresponding values.
    %
:- pred nondet_search(one_or_more_map(K, V)::in, K::in, V::out) is nondet.

    % If the one_or_more_map has an entry for the given key,
    % succeed once for each of the corresponding values.
    % Otherwise, throw an exception.
    %
:- func lookup(one_or_more_map(K, V), K) = one_or_more(V).
:- pred lookup(one_or_more_map(K, V)::in, K::in, one_or_more(V)::out) is det.

    % If the one_or_more_map has an entry for the given key,
    % succeed once for each of the corresponding values.
    % Otherwise, throw an exception.
    %
:- pred nondet_lookup(one_or_more_map(K, V)::in, K::in, V::out) is nondet.

    % If the one_or_more_map has an entry for keys with the given value,
    % succeed once for each of those keys.
    %
```

```
    % NOTE: The implementation of this predicate is necessarily inefficient,
    % and so this predicate is intended for non-performance-critical uses only.
    %
:- pred inverse_search(one_or_more_map(K, V)::in, V::in, K::out) is nondet.

%---------------------------------------------------%

    % Add the given key-value pair to the one_or_more_map.
    % Fail if the key already exists.
    %
:- pred insert(K::in, V::in,
    one_or_more_map(K, V)::in, one_or_more_map(K, V)::out) is semidet.

    % Add the given key-value pair to the one_or_more_map.
    % Throw an exception if the key already exists.
    %
:- func det_insert(one_or_more_map(K, V), K, V) = one_or_more_map(K, V).
:- pred det_insert(K::in, V::in,
    one_or_more_map(K, V)::in, one_or_more_map(K, V)::out) is det.

    % Add the given key-value pair to the one_or_more_map.
    % Fail if the key does not already exist.
    %
:- pred update(K::in, V::in,
    one_or_more_map(K, V)::in, one_or_more_map(K, V)::out) is semidet.

    % Add the given key-value pair to the one_or_more_map.
    % Throw an exception if the key does not already exist.
    %
:- func det_update(one_or_more_map(K, V), K, V) = one_or_more_map(K, V).
:- pred det_update(K::in, V::in,
    one_or_more_map(K, V)::in, one_or_more_map(K, V)::out) is det.

    % Replace the list of values corresponding to the given key.
    % Fails if the key does not already exist.
    %
:- pred replace(K::in, one_or_more(V)::in,
    one_or_more_map(K, V)::in, one_or_more_map(K, V)::out) is semidet.

    % Replace the list of values corresponding to the given key.
    % Throws an exception if the key does not already exist.
    %
:- func det_replace(one_or_more_map(K, V), K,
    one_or_more(V)) = one_or_more_map(K, V).
:- pred det_replace(K::in, one_or_more(V)::in,
    one_or_more_map(K, V)::in, one_or_more_map(K, V)::out) is det.
```

```
    % Add the given key-value pair to the one_or_more_map.
    % (‘add’ is a synonym for ‘set’.)
    %
:- func set(one_or_more_map(K, V), K, V) = one_or_more_map(K, V).
:- pred set(K::in, V::in,
    one_or_more_map(K, V)::in, one_or_more_map(K, V)::out) is det.
:- func add(one_or_more_map(K, V), K, V) = one_or_more_map(K, V).
:- pred add(K::in, V::in,
    one_or_more_map(K, V)::in, one_or_more_map(K, V)::out) is det.

    % Add the given value-key pair to the one_or_more_map.
    %
:- func reverse_set(one_or_more_map(K, V), V, K) = one_or_more_map(K, V).
:- pred reverse_set(V::in, K::in,
    one_or_more_map(K, V)::in, one_or_more_map(K, V)::out) is det.

%---------------------------------------------%

    % Delete a key and its corresponding values from a one_or_more_map.
    % If the key is not present, leave the one_or_more_map unchanged.
    %
:- func delete(one_or_more_map(K, V), K) = one_or_more_map(K, V).
:- pred delete(K::in,
    one_or_more_map(K, V)::in, one_or_more_map(K, V)::out) is det.

    % Delete the given key-value pair from a one_or_more_map.
    % If the key is not present, leave the one_or_more_map unchanged.
    %
:- func delete(one_or_more_map(K, V), K, V) = one_or_more_map(K, V).
:- pred delete(K::in, V::in,
    one_or_more_map(K, V)::in, one_or_more_map(K, V)::out) is det.

    % Delete a key from a one_or_more_map and return the list of values
    % previously corresponding to it.
    % Fail if the key is not present.
    %
:- pred remove(K::in, one_or_more(V)::out,
    one_or_more_map(K, V)::in, one_or_more_map(K, V)::out) is semidet.

    % Delete a key from a one_or_more_map and return the list of values
    % previously corresponding to it.
    % Throw an exception if the key is not present.
    %
:- pred det_remove(K::in, one_or_more(V)::out,
    one_or_more_map(K, V)::in, one_or_more_map(K, V)::out) is det.

    % Remove the smallest key and its corresponding values from the
```

```
    % one_or_more_map.
    % Fails if the one_or_more_map is empty.
    %
:- pred remove_smallest(K::out, one_or_more(V)::out,
    one_or_more_map(K, V)::in, one_or_more_map(K, V)::out) is semidet.

%---------------------------------------------------%

    % Select takes a one_or_more_map and a set of keys and returns
    % a one_or_more_map containing only the keys in the set,
    % together with their corresponding values.
    %
:- func select(one_or_more_map(K, V), set(K)) = one_or_more_map(K, V).
:- pred select(one_or_more_map(K, V)::in, set(K)::in,
    one_or_more_map(K, V)::out) is det.

%---------------------------------------------------%

    % merge(MultiMapA, MultiMapB, MultiMap):
    %
    % Merge 'MultiMapA' and 'MultiMapB' so that
    %
    % - if a key occurs in both 'MultiMapA' and 'MultiMapB', then the values
    %   corresponding to that key in 'MultiMap' will be the concatenation
    %   of the values to that key from 'MultiMapA' and 'MultiMapB'; while
    % - if a key occurs in only one of 'MultiMapA' and 'MultiMapB', then
    %   the values corresponding to it in that map will be carried over
    %   to 'MultiMap'.
    %
:- func merge(one_or_more_map(K, V), one_or_more_map(K, V))
    = one_or_more_map(K, V).
:- pred merge(one_or_more_map(K, V)::in, one_or_more_map(K, V)::in,
    one_or_more_map(K, V)::out) is det.

%---------------------------------------------------%

    % Declaratively, a no-operation.
    % Operationally, a suggestion that the implementation optimize
    % the representation of the one_or_more_map, in the expectation that the
    % following operations will consist of searches and lookups
    % but (almost) no updates.
    %
:- func optimize(one_or_more_map(K, V)) = one_or_more_map(K, V).
:- pred optimize(one_or_more_map(K, V)::in, one_or_more_map(K, V)::out) is det.

%---------------------------------------------------%
```

```
    % Convert a one_or_more_map to an association list.
    %
:- func to_flat_assoc_list(one_or_more_map(K, V)) = assoc_list(K, V).
:- pred to_flat_assoc_list(one_or_more_map(K, V)::in,
    assoc_list(K, V)::out) is det.

    % Convert an association list to a one_or_more_map.
    %
:- func from_flat_assoc_list(assoc_list(K, V)) = one_or_more_map(K, V).
:- pred from_flat_assoc_list(assoc_list(K, V)::in,
    one_or_more_map(K, V)::out) is det.

    % Convert a one_or_more_map to an association list, with all the values
    % for each key in one element of the association list.
    %
:- func to_assoc_list(one_or_more_map(K, V)) = assoc_list(K, one_or_more(V)).
:- pred to_assoc_list(one_or_more_map(K, V)::in,
    assoc_list(K, one_or_more(V))::out) is det.

    % Convert an association list with all the values for each key
    % in one element of the list to a one_or_more_map.
    %
:- func from_assoc_list(assoc_list(K, one_or_more(V))) = one_or_more_map(K, V).
:- pred from_assoc_list(assoc_list(K, one_or_more(V))::in,
    one_or_more_map(K, V)::out) is det.

    % Convert a sorted association list to a one_or_more_map.
    %
:- func from_sorted_assoc_list(assoc_list(K, one_or_more(V)))
    = one_or_more_map(K, V).
:- pred from_sorted_assoc_list(assoc_list(K, one_or_more(V))::in,
    one_or_more_map(K, V)::out) is det.

    % Convert the corresponding elements of a list of keys and a
    % list of values (which must be of the same length) to a one_or_more_map.
    % A key may occur more than once in the list of keys.
    % Throw an exception if the two lists are not the same length.
    %
:- func from_corresponding_lists(list(K), list(V))
    = one_or_more_map(K, V).
:- pred from_corresponding_lists(list(K)::in, list(V)::in,
    one_or_more_map(K, V)::out) is det.

    % Convert the corresponding elements of a list of keys and a
    % *list of lists* of values to a one_or_more_map.
    % A key may *not* occur more than once in the list of keys.
    % Throw an exception if the two lists are not the same length,
```

```
    % or if a key does occur more than once in the list of keys.
    %
:- func from_corresponding_list_lists(list(K), list(one_or_more(V)))
    = one_or_more_map(K, V).
:- pred from_corresponding_list_lists(list(K)::in, list(one_or_more(V))::in,
    one_or_more_map(K, V)::out) is det.

%---------------------------------------------------%

    % Given a list of keys, produce a list of their values in a
    % specified one_or_more_map.
    %
:- func apply_to_list(list(K), one_or_more_map(K, V)) = list(V).
:- pred apply_to_list(list(K)::in, one_or_more_map(K, V)::in, list(V)::out)
    is det.

%---------------------------------------------------%

    % Given a one_or_more_map, return a list of all the keys in it.
    %
:- func keys(one_or_more_map(K, V)) = list(K).
:- pred keys(one_or_more_map(K, V)::in, list(K)::out) is det.

    % Given a one_or_more_map, return a list of all the keys in it
    % in sorted order.
    %
:- func sorted_keys(one_or_more_map(K, V)) = list(K).
:- pred sorted_keys(one_or_more_map(K, V)::in, list(K)::out) is det.

    % Given a one_or_more_map, return a list of all the keys in it
    % as a set
    %
:- func keys_as_set(one_or_more_map(K, V)) = set(K).
:- pred keys_as_set(one_or_more_map(K, V)::in, set(K)::out) is det.

    % Given a one_or_more_map, return a list of all the values in it.
    %
:- func values(one_or_more_map(K, V)) = list(V).
:- pred values(one_or_more_map(K, V)::in, list(V)::out) is det.

%---------------------------------------------------%

    % Count the number of keys in the one_or_more_map.
    %
:- func count(one_or_more_map(K, V)) = int.
:- pred count(one_or_more_map(K, V)::in, int::out) is det.
```

```
    % Count the number of key-value pairs in the one_or_more_map.
    %
:- func all_count(one_or_more_map(K, V)) = int.
:- pred all_count(one_or_more_map(K, V)::in, int::out) is det.

%---------------------------------------------------%
%---------------------------------------------------%
```

52 ops

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 1995-2008, 2010, 2012 The University of Melbourne.
% Copyright (C) 2014-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: ops.m.
% Main author: fjh.
% Stability: low.
%
% This module exports a typeclass `ops.op_table' which is used to define
% operator precedence tables for use by `parser.read_term_with_op_table'
% and `term_io.write_term_with_op_table'.
%
% It also exports an instance `ops.mercury_op_table' that implements the
% Mercury operator table defined in the Mercury Language Reference Manual.
%
% See samples/calculator2.m for an example program.
%
%---------------------------------------------------%
%---------------------------------------------------%

:- module ops.
:- interface.

:- import_module list.

%---------------------------------------------------%

    % An class describes what structure terms constructed with an operator
    % of that class are allowed to take.
:- type class
    --->    infix(ops.assoc, ops.assoc)             % term Op term
```

```
    ;       prefix(ops.assoc)                       % Op term
    ;       binary_prefix(ops.assoc, ops.assoc)     % Op term term
    ;       postfix(ops.assoc).                     % term Op

    % ‘x’ represents an argument whose priority must be
    % strictly lower than the priority of the operator.
    % ‘y’ represents an argument whose priority must be
    % lower than or equal to the priority of the operator.
:- type assoc
    --->    x
    ;       y.

    % Operators with a low "priority" bind more tightly than those
    % with a high "priority". For example, given that ‘+’ has
    % priority 500 and ‘*’ has priority 400, the term ‘2 * X + Y’
    % would parse as ‘(2 * X) + Y’.
    %
    % The lowest priority is 0.
    %
:- type priority == int.

:- type op_info
    --->    op_info(
                ops.class,
                ops.priority
            ).

%-----------------------------------------------------%

:- typeclass op_table(Table) where [

        % Check whether a string is the name of an infix operator,
        % and if it is, return its precedence and associativity.
        %
    pred lookup_infix_op(Table::in, string::in, ops.priority::out,
        ops.assoc::out, ops.assoc::out) is semidet,

        % Check whether a string is the name of a prefix operator,
        % and if it is, return its precedence and associativity.
        %
    pred lookup_prefix_op(Table::in, string::in,
        ops.priority::out, ops.assoc::out) is semidet,

        % Check whether a string is the name of a binary prefix operator,
        % and if it is, return its precedence and associativity.
        %
    pred lookup_binary_prefix_op(Table::in, string::in,
```

```
        ops.priority::out, ops.assoc::out, ops.assoc::out) is semidet,

        % Check whether a string is the name of a postfix operator,
        % and if it is, return its precedence and associativity.
        %
    pred lookup_postfix_op(Table::in, string::in, ops.priority::out,
        ops.assoc::out) is semidet,

        % Check whether a string is the name of an operator.
        %
    pred lookup_op(Table::in, string::in) is semidet,

        % Check whether a string is the name of an operator, and if it is,
        % return the op_info describing that operator in the third argument.
        % If the string is the name of more than one operator, return
        % information about its other guises in the last argument.
        %
    pred lookup_op_infos(Table::in, string::in,
        op_info::out, list(op_info)::out) is semidet,

        % Operator terms are terms of the form `X `Op` Y', where `Op' is
        % a variable or a name and `X' and `Y' are terms. If operator terms
        % are included in `Table', return their precedence and associativity.
        %
    pred lookup_operator_term(Table::in, ops.priority::out,
        ops.assoc::out, ops.assoc::out) is semidet,

        % Returns the highest priority number (the lowest is zero).
        %
    func max_priority(Table) = ops.priority,

        % The maximum priority of an operator appearing as the top-level
        % functor of an argument of a compound term.
        %
        % This will generally be the precedence of `,/2' less one.
        % If `,/2' does not appear in the op_table, `ops.max_priority' plus one
        % may be a reasonable value.
        %
    func arg_priority(Table) = ops.priority
].

%-----------------------------------------------------%

    % The table of Mercury operators.
    % See the "Builtin Operators" section of the "Syntax" chapter
    % of the Mercury Language Reference Manual for details.
    %
```

```
:- type mercury_op_table.
:- instance ops.op_table(ops.mercury_op_table).

:- func init_mercury_op_table = (ops.mercury_op_table::uo) is det.

%-----------------------------------------------------%
%-----------------------------------------------------%
```

53 pair

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 1994-2006 The University of Melbourne.
% Copyright (C) 2014-2015, 2017-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: pair.m.
% Main author: fjh.
% Stability: high.
%
% The "pair" type.  Useful for many purposes.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module pair.
:- interface.

:- type pair(T1, T2)
    --->    (T1 - T2).
:- type pair(T) ==  pair(T, T).

:- inst pair(I1, I2) for pair/2
    --->    (I1 - I2).
:- inst pair(I) == pair(I, I).

    % Return the first element of the pair.
    %
:- func fst(pair(X, Y)) = X.
:- pred fst(pair(X, Y)::in, X::out) is det.

    % Return the second element of the pair.
    %
```

```
:- func snd(pair(X, Y)) = Y.
:- pred snd(pair(X, Y)::in, Y::out) is det.

:- func pair(T1, T2) = pair(T1, T2).

%-----------------------------------------------------%
%-----------------------------------------------------%
```

54 parser

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 1995-2001, 2003-2008, 2011-2012 The University of Melbourne.
% Copyright (C) 2014-2020 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: parser.m.
% Main author: fjh.
% Stability: high.
%
% This file exports the predicate read_term, which reads a term from the
% current input stream. The read_term_from_*string predicates are the same as
% the read_term predicates, except that the term is read from a string rather
% than from the current input stream. The parse_tokens predicate is
% similar, but it takes a list of tokens rather than a string.
%
% The parser is a relatively straight-forward top-down recursive descent
% parser, made somewhat complicated by the need to handle operator
% precedences.  It uses `lexer.get_token_list' to read a list of tokens.
% It uses the routines from the module `ops' to look up operator precedences.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module parser.
:- interface.

:- import_module io.
:- import_module lexer.
:- import_module ops.
:- import_module term_io.

%-----------------------------------------------------%
```

```
    % read_term(Result, !IO):
    % read_term(Stream, Result, !IO):
    %
    % Reads a Mercury term from the current input stream or from Stream.
    %
:- pred read_term(read_term(T)::out, io::di, io::uo) is det.
:- pred read_term(io.text_input_stream::in, read_term(T)::out,
    io::di, io::uo) is det.

    % read_term_with_op_table(Ops, Result, !IO):
    % read_term_with_op_table(Stream, Ops, Result, !IO):
    %
    % Reads a term from the current input stream or from Stream,
    % using the given op_table to interpret the operators.
    %
:- pred read_term_with_op_table(Ops::in,
    read_term(T)::out, io::di, io::uo) is det <= op_table(Ops).
:- pred read_term_with_op_table(io.text_input_stream::in, Ops::in,
    read_term(T)::out, io::di, io::uo) is det <= op_table(Ops).

    % read_term_filename(FileName, Result, !IO):
    % read_term_filename(Stream, FileName, Result, !IO):
    %
    % Reads a term from the current input stream or from Stream.
    % The string is the filename to use for the stream; this is used
    % in constructing the term.contexts in the read term.
    % This interface is used to support the ':- pragma source_file' directive.
    %
:- pred read_term_filename(string::in,
    read_term(T)::out, io::di, io::uo) is det.
:- pred read_term_filename(io.text_input_stream::in, string::in,
    read_term(T)::out, io::di, io::uo) is det.

    % read_term_filename_with_op_table(Ops, FileName, Result, !IO):
    % read_term_filename_with_op_table(Stream, Ops, FileName, Result, !IO):
    %
    % As above but using the given op_table.
    %
:- pred read_term_filename_with_op_table(Ops::in,
    string::in, read_term(T)::out, io::di, io::uo) is det <= op_table(Ops).
:- pred read_term_filename_with_op_table(io.text_input_stream::in, Ops::in,
    string::in, read_term(T)::out, io::di, io::uo) is det <= op_table(Ops).

%---------------------------------------------------%

    % The read_term_from_string predicates are the same as the read_term
```

```
    % predicates, except that the term is read from a string rather than from
    % the current input stream. The returned value 'EndPos' is the position
    % one character past the end of the term read. The arguments 'MaxOffset'
    % and 'StartPos' in the read_term_from_substring* versions specify
    % the length of the string and the position within the string
    % at which to start parsing.

    % read_term_from_string(FileName, String, EndPos, Term).
    %
:- pred read_term_from_string(string::in, string::in, posn::out,
    read_term(T)::out) is det.

    % read_term_from_string_with_op_table(Ops, FileName, String, EndPos, Term).
    %
:- pred read_term_from_string_with_op_table(Ops::in, string::in,
    string::in, posn::out, read_term(T)::out) is det <= op_table(Ops).

    % read_term_from_substring(FileName, String, MaxOffset,
    %   StartPos, EndPos, Term).
    % read_term_from_linestr(FileName, String, MaxOffset,
    %   StartLineContext, EndLineContext, StartLinePosn, EndLinePosn, Term).
    %
:- pred read_term_from_substring(string::in, string::in, int::in,
    posn::in, posn::out, read_term(T)::out) is det.
:- pred read_term_from_linestr(string::in, string::in, int::in,
    line_context::in, line_context::out, line_posn::in, line_posn::out,
    read_term(T)::out) is det.

    % read_term_from_substring_with_op_table(Ops, FileName, String, MaxOffset,
    %   StartPos, EndPos, Term).
    % read_term_from_linestr_with_op_table(Ops, FileName, String, MaxOffset,
    %   StartLineContext, EndLineContext, StartLinePosn, EndLinePosn, Term).
    %
:- pred read_term_from_substring_with_op_table(Ops::in, string::in,
    string::in, int::in, posn::in, posn::out, read_term(T)::out) is det
    <= op_table(Ops).
:- pred read_term_from_linestr_with_op_table(Ops::in, string::in,
    string::in, int::in,
    line_context::in, line_context::out, line_posn::in, line_posn::out,
    read_term(T)::out) is det <= op_table(Ops).

%-----------------------------------------------------%

    % parse_tokens(FileName, TokenList, Result):
    %
:- pred parse_tokens(string::in, token_list::in, read_term(T)::out) is det.
```

```
    % parse_tokens(FileName, TokenList, Result):
    %
:- pred parse_tokens_with_op_table(Ops::in, string::in, token_list::in,
    read_term(T)::out) is det <= op_table(Ops).

%-----------------------------------------------------%
%-----------------------------------------------------%
```

55 parsing_utils

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 2009-2012 The University of Melbourne.
% Copyright (C) 2014-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: parsing_utils.m
% Authors: Ralph Becket <rafe@csse.unimelb.edu.au>, maclarty
% Stability: low
%
% Utilities for recursive descent parsers. Parsers take at least three
% arguments: a source (src) containing the input string, and an input/output
% pair of parser states (ps) tracking the current offset into the input.
%
% Call parse(InputString, SkipWS, Parser, Result) to parse an input string
% and return an error context and message if parsing failed.
% The SkipWS predicate is used by the primitive parsers to skip over any
% following whitespace (providing a skipping predicate allows users to define
% comments as whitespace).
% Alternatively, a new src and ps can be constructed by calling
% new_src_and_ps(InputString, SkipWS, Src, !:PS).
%
% Parsing predicates are semidet and typically take the form
% p(...parameters..., Src, Result, !PS). A parser matching variable
% assignments of the form ‘x = 42’ might be defined like this:
%
%   var_assignment(Src, {Var, Value}, !PS) :-
%       var(Src, Var, !PS),
%       punct(Src, "=", !PS),
%       expr(Src, Expr, !PS).
%
% where var/4 and expr/4 are parsers for variables and expressions respectively
% and punct/4 is provided by this module for matching punctuation.
```

```
%
%---------------------------------------------------%
%---------------------------------------------------%

:- module parsing_utils.
:- interface.

:- import_module char.
:- import_module list.
:- import_module maybe.
:- import_module unit.

%---------------------------------------------------%

    % The parser source (input string).
    %
:- type src.

    % The parser "state", passed around in DCG arguments.
    %
:- type ps.

    % These types and insts are useful for specifying "standard" parser
    % signatures.
    %
:- type parser(T) == pred(src, T, ps, ps).
:- inst parser == (pred(in, out, in, out) is semidet).

    % The following are for parsers that also transform a separate state value.
    %
:- type parser_with_state(T, S) == pred(src, T, S, S, ps, ps).
:- inst parser_with_state == (pred(in, out, in, out, in, out) is semidet).

    % Predicates of this type are used to skip whitespace in the primitive
    % parsers provided by this module.
    %
:- type skip_whitespace_pred == parser(unit).

:- type parse_result(T)
    --->    ok(T)
    ;       error(
                error_message :: maybe(string),
                error_line    :: int,
                error_col     :: int
            ).

    % parse(Input, SkipWS, Parser, Result).
```

```
    % Try to parse Input using Parser and SkipWS to consume whitespace.
    % If Parser succeeds then return ok with the parsed value,
    % otherwise return error. If there were any calls to fail_with_message
    % without any subsequent progress being made, then the error message
    % passed to the last call to fail_with_message will be returned in the
    % error result. Otherwise no message is returned and the furthest
    % position the parser got in the input string is returned.
    %
:- pred parse(string::in, skip_whitespace_pred::in(parser),
    parser(T)::in(parser), parse_result(T)::out) is cc_multi.

    % As above but using the default whitespace parser, whitespace/4.
    %
:- pred parse(string::in, parser(T)::in(parser), parse_result(T)::out)
    is cc_multi.

%-----------------------------------------------%

    % Construct a new parser source and state from a string, also specifying
    % a predicate for skipping over whitespace (several primitive parsers
    % use this predicate to consume whitespace after a token; this argument
    % allows the user to specify a predicate for, say, skipping over comments
    % as well).
    %
:- pred new_src_and_ps(string::in, skip_whitespace_pred::in(parser),
    src::out, ps::out) is det.

    % Construct a new parser source and state from a string.
    % The default whitespace parser, whitespace/4, is used.
    %
:- pred new_src_and_ps(string::in, src::out, ps::out) is det.

%-----------------------------------------------%

    % Return the input string and its length from the parser source.
    %
:- pred input_string(src::in, string::out, int::out) is det.

    % Obtain the current offset from the start of the input string
    % (the first character in the input has offset 0).
    %
:- pred current_offset(src::in, int::out, ps::in, ps::out) is det.

    % Return the parser to skip over whitespace from the parser source.
    %
:- pred get_skip_whitespace_pred(src::in, skip_whitespace_pred::out(parser))
    is det.
```

```
%---------------------------------------------------%

    % input_substring(Src, StartOffset, EndOffsetPlusOne, Substring):
    % Copy the substring from the input occupying the offsets
    % [StartOffset, EndOffsetPlusOne).
    %
:- pred input_substring(src::in, int::in, int::in, string::out) is semidet.

%---------------------------------------------------%

:- type line_numbers.

    % Compute a structure from the parser source which can be used to
    % convert offsets into line numbers and positions in the file (this
    % is useful for error reporting).
    %
:- func src_to_line_numbers(src) = line_numbers.

    % Convert an offset into a line number and position within the line
    % (the first line is number 1; the first character in a line is
    % position 1).
    %
:- pred offset_to_line_number_and_position(line_numbers::in, int::in,
    int::out, int::out) is det.

%---------------------------------------------------%

    % Read the next char.
    %
:- pred next_char(src::in, char::out, ps::in, ps::out) is semidet.

    % Read the next char but do not record progress information.
    % This is more efficient than next_char, but may produce less informative
    % error messages in case of a parse error.
    %
:- pred next_char_no_progress(src::in, char::out, ps::in, ps::out) is semidet.

%---------------------------------------------------%

    % Match a char from the given string.
    %
:- pred char_in_class(string::in, src::in, char::out,
    ps::in, ps::out) is semidet.

%---------------------------------------------------%
```

```
    % Match a string exactly and any subsequent whitespace.
    %
:- pred punct(string::in, src::in, unit::out, ps::in, ps::out) is semidet.

    % keyword(IdChars, Keyword, Src, _, !PS) matches Keyword exactly (i.e., it
    % must not be followed by any character in IdChars) and any subsequent
    % whitespace.
    %
:- pred keyword(string::in, string::in, src::in, unit::out,
    ps::in, ps::out) is semidet.

    % ikeyword(IdChars, Keyword, Src, _, !PS)
    % Case-insensitive version of keyword/6.
    % Only uppercase and lowercase letters in the ASCII range (A-Z, a-z)
    % are compared case insensitively.
    %
:- pred ikeyword(string::in, string::in, src::in, unit::out,
    ps::in, ps::out) is semidet.

    % identifier(InitIdChars, IdChars, Src, Identifier, !PS) matches the next
    % identifier (result in Identifier) comprising a char from InitIdChars
    % followed by zero or more chars from IdChars. Any subsequent whitespace
    % is consumed.
    %
:- pred identifier(string::in, string::in, src::in, string::out,
    ps::in, ps::out) is semidet.

    % Consume any whitespace (defined as a sequence of characters
    % satisfying char.is_whitespace).
    %
:- pred whitespace(src::in, unit::out,
    ps::in, ps::out) is semidet.

%---------------------------------------------------%

    % Consume any input up to, and including, the next newline character
    % marking the end of the current line.
    %
:- pred skip_to_eol(src::in, unit::out,
    ps::in, ps::out) is semidet.

    % Succeed if we have reached the end of the input.
    %
:- pred eof(src::in, unit::out, ps::in, ps::out) is semidet.

%---------------------------------------------------%
```

```
    % Parse a float literal matching [-][0-9]+[.][0-9]+([Ee][-+][0-9]+)?
    % followed by any whitespace. The float_literal_as_string version simply
    % returns the matched string. The float_literal version uses
    % string.to_float to convert the output of float_literal_as_string; this
    % may return an approximate answer since not all floating point numbers
    % can be perfectly represented as Mercury floats.
    %
:- pred float_literal_as_string(src::in, string::out,
    ps::in, ps::out) is semidet.
:- pred float_literal(src::in, float::out,
    ps::in, ps::out) is semidet.

    % Parse an int literal matching [-][0-9]+, not followed by [.][0-9]+,
    % followed by any whitespace. The int_literal_as_string version simply
    % returns the matched string. The int_literal version uses string.to_int
    % to convert the output of int_literal_as_string; this may fail if the
    % number in question cannot be represented as a Mercury int.
    %
:- pred int_literal_as_string(src::in, string::out,
    ps::in, ps::out) is semidet.
:- pred int_literal(src::in, int::out,
    ps::in, ps::out) is semidet.

    % Parse an string literal. The string argument is the quote character.
    % A backslash (\) character in the string makes the next character
    % literal (e.g., for embedding quotes). These 'escaped' characters
    % are included as-is in the result, along with the preceding backslash.
    % Any following whitespace is also consumed.
    %
:- pred string_literal(char::in, src::in, string::out,
    ps::in, ps::out) is semidet.

%-----------------------------------------------%

% Each basic parser combinators has a version that has a separate state
% argument is threaded through the computation, for parsers that e.g.
% incrementally construct a symbol table.

    % optional(P, Src, Result, !PS) returns Result = yes(X) if P(Src, X, !PS),
    % or Result = no if P does not succeed.
    %
:- pred optional(parser(T)::in(parser), src::in, maybe(T)::out,
    ps::in, ps::out) is semidet.

    % optional(P, Src, Result, !S, !PS) returns Result = yes(X)
    % if P(Src, X, !S, !PS), or Result = no if P does not succeed.
    %
```

```
:- pred optional(parser_with_state(T, S)::in(parser_with_state), src::in,
    maybe(T)::out, S::in, S::out, ps::in, ps::out) is semidet.

%-----------------------------------------------------%

    % zero_or_more(P, Src, Xs, !PS) returns the list of results Xs obtained
    % by repeatedly applying P until P fails. The nth item in Xs is
    % the result from the nth application of P.
    %
:- pred zero_or_more(parser(T)::in(parser), src::in, list(T)::out,
    ps::in, ps::out) is semidet.

    % zero_or_more(P, Src, Xs, !S, !PS) returns the list of results Xs obtained
    % by repeatedly applying P until P fails. The nth item in Xs is
    % the result from the nth application of P.
    %
:- pred zero_or_more(parser_with_state(T, S)::in(parser_with_state), src::in,
    list(T)::out, S::in, S::out, ps::in, ps::out) is semidet.

%-----------------------------------------------------%

    % one_or_more(P, Src, Xs, !PS) returns the list of results Xs obtained
    % by repeatedly applying P until P fails. The nth item in Xs is
    % the result from the nth application of P. P must succeed at least once.
    %
:- pred one_or_more(parser(T)::in(parser), src::in, list(T)::out,
    ps::in, ps::out) is semidet.

    % one_or_more(P, Src, Xs, !S, !PS) returns the list of results Xs obtained
    % by repeatedly applying P until P fails. The nth item in Xs is
    % the result from the nth application of P. P must succeed at least once.
    %
:- pred one_or_more(parser_with_state(T, S)::in(parser_with_state), src::in,
    list(T)::out, S::in, S::out, ps::in, ps::out) is semidet.

%-----------------------------------------------------%

    % brackets(L, R, P, Src, X, !PS) is equivalent to
    %   punct(L, Src, _, !PS), P(Src, X, !PS), punct(R, Src, _, !PS).
    %
:- pred brackets(string::in, string::in, parser(T)::in(parser), src::in,
    T::out, ps::in, ps::out) is semidet.

    % brackets(L, R, P, Src, X, !S, !PS) is equivalent to
    %   punct(L, Src, _, !PS), P(Src, X, !S, !PS), punct(R, Src, _, !PS).
    %
:- pred brackets(string::in, string::in,
```

```
    parser_with_state(T, S)::in(parser_with_state), src::in,
    T::out, S::in, S::out, ps::in, ps::out) is semidet.

%-----------------------------------------------%

    % separated_list(Separator, P, Src, Xs, !PS) is like
    % zero_or_more(P, Src, Xs, !PS) except that successive applications of
    % P must be separated by punct(Separator, Src, _, !PS).
    %
:- pred separated_list(string::in, parser(T)::in(parser), src::in,
    list(T)::out, ps::in, ps::out) is semidet.

    % separated_list(Separator, P, Src, Xs, !S, !PS) is like
    % zero_or_more(P, Src, Xs, !S, !PS) except that successive applica-
tions of
    % P must be separated by punct(Separator, Src, _, !PS).
    %
:- pred separated_list(string::in,
    parser_with_state(T, S)::in(parser_with_state),
    src::in, list(T)::out, S::in, S::out, ps::in, ps::out) is semidet.

%-----------------------------------------------%

    % comma_separated_list(P, Src, Xs) is the same as
    %   separated_list(",", P, Src, Xs).
    %
:- pred comma_separated_list(parser(T)::in(parser), src::in, list(T)::out,
    ps::in, ps::out) is semidet.

    % comma_separated_list(P, Src, Xs, !S, !PS) is the same as
    %   separated_list(",", P, Src, Xs, !S, !PS).
    %
:- pred comma_separated_list(parser_with_state(T, S)::in(parser_with_state),
    src::in, list(T)::out, S::in, S::out, ps::in, ps::out) is semidet.

%-----------------------------------------------%

    % Declaratively this predicate is equivalent to false. Operationally,
    % it will record an error message that will be returned by parse/4
    % if no further progress is made and then fail.
    %
:- pred fail_with_message(string::in, src::in, T::out, ps::in, ps::out)
    is semidet.

    % As above, but use the given offset for the context of the message.
    %
:- pred fail_with_message(string::in, int::in, src::in, T::out,
```

```
    ps::in, ps::out) is semidet.

%-----------------------------------------------------%
%-----------------------------------------------------%
```

56 pprint

```
%-----------------------------------------------------%
% vim:ts=4 sw=4 expandtab ft=mercury
%-----------------------------------------------------%
% Copyright (C) 2000-2007, 2010-2011 The University of Melbourne
% Copyright (C) 2014-2018, 2020 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: pprint.m
% Main author: rafe
% Stability: medium
%
% NOTE: this module has now been superceded by pretty_printer.m which is
% more economical, produces better output, has better control over
% the amount of output produced, and supports user-specifiable formatting
% for arbitrary types.
%
% ABOUT
% -----
%
% This started off as pretty much a direct transliteration of Philip Wadler's
% Haskell pretty printer described in "A Prettier Printer", available at
% http://cm.bell-labs.com/cm/cs/who/wadler/topics/recent.html
%
% Several changes have been made to the algorithm to preserve linear running
% time under a strict language and to ensure scalability to extremely large
% terms without thrashing the VM system.
%
% Wadler's approach has three main advantages:
%
% 1. the layout algebra is small and quite intuitive (more so than Hughes');
% 2. the pretty printer is optimal in the sense that it will never generate
%    output that over-runs the specified width unless that is unavoidable; and
% 3. the pretty printer is bounded in that it never needs to look more than
%    k characters ahead to make a formatting decision.
%
% I have made the following changes:
%
```

```
% (a) rather than having group/1 as a non-primitive function (for
% allowing line-breaks to be converted into spaces at the pretty
% printer's discretion) over docs, I have extended the doc type to
% include a 'GROUP' constructor and made the appropriate algorithmic
% changes. Because 'UNION' only arises as a consequence of processing
% a 'GROUP' it turns out to be simpler to do away with 'UNION'
% altogether and convert clauses that process 'UNION' terms to
% processing 'GROUP's.
%
% (b) Flattened 'line' breaks become empty strings rather than spaces.
%
% (c) The third change is the introduction of the 'LABEL' constructor,
% which acts much like 'NEST', except that indentation is defined
% using a string rather than a number of spaces. This is useful for,
% e.g., multi-line compiler errors and warnings that should be
% prefixed with the offending source file and line number.
%
% (d) The formatting decision procedure has been altered to preserve
% linear runtime behaviour in a strict language.
%
% (e) Naively marking up a term as a doc has the drawback that the
% resulting doc is significantly larger than the original term.
% Worse, any sharing structure in the original term leads to
% duplicated sub-docs, which can cause an exponential blow-up in the
% size of the doc w.r.t. the source term. To get around this problem
% I have introduced the 'DOC' constructor which causes on-demand
% conversion of arguments.
%
% [This is not true laziness in the sense that the 'DOC', once
% evaluated, will be overwritten with its value. This approach would
% lead to garbage retention and not solve the page thrashing behaviour
% otherwise experienced when converting extremely large terms.
% Instead, each 'DOC' is reevaluated each time it is examined.
% This trades off computation time for space.]
%
% I have added several obvious general purpose formatting functions.
%
%
% USAGE
% -----
%
% There are two stages in pretty printing an object of some type T:
% 1. convert the object to a pprint.doc using the constructor functions
%    described below or by simply calling pprint.to_doc/[1,2];
% 2. call pprint.write/[4,5] or pprint.to_string/2 passing the display width
%    and the doc.
%
```

```
%
% EXAMPLES
% --------
%
% The doc/1 type class has types string, char, int, float and doc as instances.
% Hence these types can all be converted to docs by applying doc/1.
% This happens automatically to the arguments of ++/2. Users may find it
% convenient to add other types as instances of the doc/1 type class.
%
% Below are some docs followed by the ways they might be displayed by the
% pretty printer given various line widths.
%
% 1. "Hello " ++ line ++ "world"
%
%   Hello
%   world
%
% 2. group("Hello " ++ line ++ "world")
%
%   Hello world
%
%   Hello
%   world
%
% 3. group("Hello " ++ nest(3, line ++ "world"))
%
%   Hello world
%
%   Hello
%      world
%
% 4. group("Goodbye " ++ nest(3, line ++ "cruel " ++ line ++ "world")
%
%   Goodbye cruel world
%
%   Goodbye
%      cruel
%      world
%
% 5. group("Goodbye " ++ nest(3, line ++ group("cruel " ++ line ++ "world")))
%
%   Goodbye cruel world
%
%   Goodbye
%      cruel world
%
%   Goodbye
```

```
%      cruel
%      world
%
% 6. label("Look! ", line ++
%        group("Goodbye " ++
%              nest(3, line ++ group("cruel " ++ line ++ "world"))))
%
%   Look! Goodbye cruel world
%
%   Look! Goodbye
%   Look!    cruel world
%
%   Look! Goodbye
%   Look!    cruel
%   Look!    world
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module pprint.
:- interface.

:- import_module char.
:- import_module io.
:- import_module list.
:- import_module stream.
:- import_module string.
:- import_module univ.

%-----------------------------------------------------%

    % Clients must translate data structures into docs for
    % the pretty printer to display.
    %
:- type doc.

    % This typeclass can be used to simplify the construction of docs.
    %
:- typeclass doc(T) where [
    % Convert a T to a doc, placing a limit on how much of the term
    % will be fully converted as follows:
    %
    % doc(_, f         ) = f
    % doc(N, f(A, B, C)) = f/3 if N =< 0
    % doc(N, f(A, B, C)) = some representation of the term whereby
    %   A is converted as doc(N - 1, A),
    %   B is converted as doc(N - 2, B), and
```

```
    %   C is converted as doc(N - 3, C)
    %   - if there are more than N arguments, the N+1th and subsequent
    %     arguments should be replaced with a single ellipsis.
    %
    func doc(int, T) = doc
].

:- instance doc(doc).
:- instance doc(string).
:- instance doc(int).
:- instance doc(int8).
:- instance doc(int16).
:- instance doc(int32).
:- instance doc(int64).
:- instance doc(uint).
:- instance doc(uint8).
:- instance doc(uint16).
:- instance doc(uint32).
:- instance doc(uint64).
:- instance doc(float).
:- instance doc(char).

    % Fully convert an instance of doc/1.
    %
:- func doc(T) = doc <= (doc(T)).

    % An alternative to the <>/2 concatenation operator that works
    % on members of the doc/1 typeclass.
    %
:- func T1 ++ T2 = doc <= (doc(T1), doc(T2)).

    % The empty document corresponding to the null string.
    %
:- func nil                 = doc.

    % The document consisting of a single string.
    %
    % NOTE: since string is now an instance of the doc/1
    % type class, it is simpler to just apply the doc/1
    % method.
    %
:- func text(string)        = doc.

    % The composition of two docs with no intervening space.
    %
    % NOTE: with the addition of the doc/1 type class, it is
    % simpler to construct compound docs using ++/2.
```

```
    %
:- func doc `<>` doc        = doc.

    % The newline document. In a group doc (see below) the pretty printer
    % may choose to instead 'flatten' all line docs into nil docs in order
    % to fit a doc on a single line.
    %
:- func line                = doc.

    % Any 'line' docs in the body that are not flattened out by the
    % pretty printer are followed by the given number of spaces
    % (nested 'nest's add up).
    %
:- func nest(int, T)        = doc <= (doc(T)).

    % Identical to a nest doc except that indentation is extended with
    % a string label rather than some number of spaces.
    %
:- func label(string, T)    = doc <= (doc(T)).

    % A group doc gives the pretty printer a choice: if the doc can be printed
    % without line wrapping then it does so (all line, label, nest and group
    % directives within the group are ignored); otherwise the pretty printer
    % treats the group body literally, although nested group docs remain as
    % choice points.
    %
:- func group(T)            = doc <= (doc(T)).

    % This function can be used to convert strings, chars, ints, uints and
    % floats to their text doc equivalents.
    %
    % NOTE: since these types are now instances of the doc/1 type class,
    % it is simpler to just apply the doc/1 method to these types.
    %
:- func poly(poly_type) = doc.

    % Shorthand for doc ++ line ++ doc.
    %
:- func doc `</>` doc       = doc.

    % Various bracketing functions.
    %
    %   bracketed(L, R, Doc) = L ++ Doc ++ R
    %       parentheses(Doc) = bracketed("(", ")", Doc)
    %          brackets(Doc) = bracketed("[", "]", Doc)
    %            braces(Doc) = bracketed("{", "}", Doc)
    %
```

```
:- func bracketed(T1, T2, T3)  = doc <= (doc(T1), doc(T2), doc(T3)).
:- func parentheses(T)         = doc <= (doc(T)).
:- func brackets(T)            = doc <= (doc(T)).
:- func braces(T)              = doc <= (doc(T)).

    % packed(Sep, [X1, X2, .., Xn]) = G1 `<>` G2 `<>` .. `<>` Gn where
    % Gi = group(line `<>` Xi `<>` Sep), except for Gn where
    % Gn = group(line `<>` Xn).
    %
    % For the singleton list case, packed(Sep, [X]) = group(line `<>` X).
    %
    % The resulting doc tries to pack as many items on a line as possible.
    %
:- func packed(T1, list(T2)) = doc <= (doc(T1), doc(T2)).

    % A variant of the above whereby only the first N elements of the list
    % are formatted and the rest are replaced by a single ellipsis.
    %
:- func packed(int, T1, list(T2)) = doc <= (doc(T1), doc(T2)).

    % packed_cs(Xs) = packed(comma_space, Xs).
    %
    % For example, to pretty print a Mercury list of docs one might use
    %
    %   brackets(nest(2, packed_cs(Xs)))
    %
:- func packed_cs(list(T)) = doc <= (doc(T)).

    % A variant of the above whereby only the first N elements of the list
    % are formatted and the rest are replaced by a single ellipsis.
    %
:- func packed_cs(int, list(T)) = doc <= (doc(T)).

    % This is like a depth-limited version of packed_cs/1 that first calls
    % to_doc/2 on each member of the argument list.
    %
:- func packed_cs_to_depth(int, list(T)) = doc.

    % This is like a version of packed_cs_to_depth/1 that first calls
    % univ_value/1 for each member of the argument list.
    %
:- func packed_cs_univ_args(int, list(univ)) = doc.

    % separated(PP, Sep, [X1,...,Xn]) =
    %   PP(X1) `<>` Sep `<>` ... Sep `<>` PP(Xn)
    %
:- func separated(func(T1) = doc, T2, list(T1)) = doc <= (doc(T2)).
```

```
    % Handy punctuation docs and versions with following
    % spaces and/or line breaks.
    %
:- func comma               = doc.
:- func semic               = doc.      % Semicolon.
:- func colon               = doc.
:- func space               = doc.
:- func comma_space         = doc.
:- func semic_space         = doc.
:- func colon_space         = doc.
:- func comma_line          = doc.
:- func semic_line          = doc.
:- func colon_line          = doc.
:- func space_line          = doc.
:- func comma_space_line    = doc.
:- func semic_space_line    = doc.
:- func colon_space_line    = doc.
:- func ellipsis            = doc.      % "...".

    % Performs word wrapping at the end of line, taking whitespace sequences
    % as delimiters separating words.
    %
    % See 'char.is_whitespace' for the definition of whitespace characters
    % used by this predicate.
    %
:- func word_wrapped(string) = doc.

    % Convert arbitrary terms to docs. This requires std_util.functor/3 to work
    % on all components of the object being converted. The second version
    % places a maximum depth on terms which are otherwise truncated in the
    % manner described in the documentation for the doc/2 method of the doc/1
    % type class.
    %
    % This may throw an exception or cause a runtime abort if the term
    % in question has user-defined equality.
    %
:- func to_doc(T)           = doc.
:- func to_doc(int, T)      = doc.

    % Convert docs to pretty printed strings. The int argument specifies
    % a line width in characters.
    %
:- func to_string(int, doc) = string.

    % Write docs out in pretty printed format. The int argument specifies
    % a page width in characters.
```

```
    %
:- pred write(int::in, T::in, io::di, io::uo) is det <= doc(T).

    % Write docs to the specified string writer stream in pretty printed
    % format. The int argument specifies a page width in characters.
    %
:- pred write(Stream::in, int::in, T::in, State::di, State::uo) is det
    <= ( doc(T), stream.writer(Stream, string, State) ).

%---------------------------------------------------%
%---------------------------------------------------%
```

57 pqueue

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 1994-1995, 1997, 1999, 2003-2007, 2009 The University of
% Melbourne.
% Copyright (C) 2013-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: pqueue.m.
% Main author: conway.
% Stability: high.
%
% This module implements a priority queue ADT.
%
% A pqueue is a priority queue. A priority queue holds a collection
% of key-value pairs; the interface provides operations to create
% an empty priority queue, to insert a key-value pair into a priority
% queue, and to remove the element with the lowest key.
%
% Insertion/removal is not guaranteed to be "stable"; that is,
% if you insert two values with the same key, the order in which
% they will be removed is unspecified.
%
%---------------------------------------------------%
%---------------------------------------------------%

:- module pqueue.
:- interface.

:- import_module assoc_list.
```

```
%---------------------------------------------------%

:- type pqueue(K, V).

    % Create an empty priority queue.
    %
:- func init = pqueue(K, V).
:- pred init(pqueue(K, V)::out) is det.

    % Succeed iff the priority queue is empty.
    %
:- pred is_empty(pqueue(K, V)::in) is semidet.

    % Extract the smallest key-value pair from the priority queue without
    % removing it. Fails if the priority queue is empty.
    %
:- pred peek(pqueue(K, V)::in, K::out, V::out) is semidet.

    % Extract the smallest key from the priority queue without removing it.
    % Fail if the priority queue is empty.
    %
:- pred peek_key(pqueue(K, V)::in, K::out) is semidet.

    % Extract the smallest value from the priority queue without remov-
ing it.
    % Fail if the priority queue is empty.
    %
:- pred peek_value(pqueue(K, V)::in, V::out) is semidet.

    % As above, but call error/1 if the priority queue is empty.
    %
:- pred det_peek(pqueue(K, V)::in, K::out, V::out) is det.
:- func det_peek_key(pqueue(K, V)) = K.
:- func det_peek_value(pqueue(K, V)) = V.

    % Insert a value V with key K into the given priority queue,
    % and return the updated priority queue.
    %
:- func insert(pqueue(K, V), K, V) = pqueue(K, V).
:- pred insert(K::in, V::in, pqueue(K, V)::in, pqueue(K, V)::out)
    is det.

    % Remove the smallest item from the priority queue.
    % Fail if the priority queue is empty.
    %
:- pred remove(K::out, V::out, pqueue(K, V)::in, pqueue(K, V)::out)
```

```
    is semidet.

    % As above, but calls error/1 if the priority queue is empty.
    %
:- pred det_remove(K::out, V::out, pqueue(K, V)::in, pqueue(K, V)::out)
    is det.

    % Merge all the entries of one priority queue with another,
    % returning the merged list.
    %
:- func merge(pqueue(K, V), pqueue(K, V)) = pqueue(K, V).
:- pred merge(pqueue(K, V)::in, pqueue(K, V)::in, pqueue(K, V)::out)
    is det.

    % Extract all the items from a priority queue by repeated removal,
    % and place them in an association list.
    %
:- func to_assoc_list(pqueue(K, V)) = assoc_list(K, V).
:- pred to_assoc_list(pqueue(K, V)::in, assoc_list(K, V)::out)
    is det.

    % Insert all the key-value pairs in an association list
    % into a priority queue.
    %
:- func assoc_list_to_pqueue(assoc_list(K, V)) = pqueue(K, V).
:- pred assoc_list_to_pqueue(assoc_list(K, V)::in, pqueue(K, V)::out)
    is det.

    % A synonym for assoc_list_to_pqueue/1.
    %
:- func from_assoc_list(assoc_list(K, V)) = pqueue(K, V).

    % length(PQueue) = Length.
    %
    % Length is the number of items in PQueue.
    %
:- func length(pqueue(K, V)) = int.

%---------------------------------------------------%
%---------------------------------------------------%
```

58 pretty_printer

```
%---------------------------------------------------%
% vim: ts=4 sw=4 expandtab ft=mercury
```

```
%-----------------------------------------------------%
% Copyright (C) 2007, 2009-2011 The University of Melbourne
% Copyright (C) 2014-2016, 2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: pretty_printer.m
% Main author: rafe
% Stability: medium
%
% This module defines a doc type for formatting and a pretty printer for
% displaying docs.
%
% The doc type includes data constructors for outputting strings, newlines,
% forming groups, indented blocks, and arbitrary values.
%
% The key feature of the algorithm is this: newlines in a group are ig-
nored if
% the group can fit on the remainder of the current line. (The algorithm is
% similar to those of Oppen and Wadler, although it uses neither corou-
tines or
% laziness.)
%
% When a newline is printed, indentation is also output according to the
% current indentation level.
%
% The pretty printer includes special support for formatting Mercury style
% terms in a way that respects Mercury's rules for operator precedence and
% bracketing.
%
% The pretty printer takes a parameter specifying a collection of user-
defined
% formatting functions for handling certain types rather than using the
% default built-in mechanism. This allows one to, say, format maps as
% sequences of (key -> value) pairs rather than exposing the underlying
% 234-tree structure.
%
% The amount of output produced is controlled via limit parameters.
% Three kinds of limits are supported: the output line width, the maximum
% number of lines to be output, and a limit on the depth for formatting
% arbitrary terms. Output is replaced with ellipsis ("...") when a limit
% has been exceeded.
%
%-----------------------------------------------------%

:- module pretty_printer.
:- interface.
```

```
:- import_module deconstruct.
:- import_module list.
:- import_module io.
:- import_module stream.
:- import_module type_desc.
:- import_module univ.

%-----------------------------------------------%

:- type doc
    --->    str(string)
            % Output a literal string. Strings containing newlines, hard tabs,
            % etc. will lead to strange output.

    ;       nl
            % Output a newline, followed by indentation, iff the enclosing
            % group does not fit on the current line and starting a new line
            % adds more space.

    ;       hard_nl
            % Always outputs a newline, followed by indentation.

    ;       docs(docs)
            % An embedded sequence of docs.

    ;       format_univ(univ)
            % Use a specialised formatter if available, otherwise use the
            % generic formatter.

    ;       format_list(list(univ), doc)
            % Pretty print a list of items using the given doc as a separator
            % between items.

    ;       format_term(string, list(univ))
            % Pretty print a term with zero or more arguments. If the term
            % corresponds to a Mercury operator it will be printed with
            % appropriate fixity and, if necessary, in parentheses. The term
            % name will be quoted and escaped if necessary.

    ;       format_susp((func) = doc)
            % The argument is a suspended computation used to lazily pro-
duce a
            % doc. If the formatting limit has been reached then just "..." is
            % output, otherwise the suspension is evaluated and the resulting
            % doc is used. This is useful for formatting large structures
            % without using more resources than required. Expanding a
```

```
                % suspended computation reduces the formatting limit by one.

        ;       pp_internal(pp_internal).
                % pp_internal docs are used in the implementation and cannot be
                % exploited by user code.

:- type docs == list(doc).

    % This type is private to the implementation and cannot be exploited
    % by user code.
    %
:- type pp_internal.

%---------------------------------------------------%
%
% Functions for constructing docs.
%

    % indent(IndentString, Docs):
    %
    % Append IndentString to the current indentation while printing Docs.
    % Indentation is printed after each newline that is output.
    %
:- func indent(string, docs) = doc.

    % indent(Docs) = indent("  ", Docs).
    %
    % A convenient abbreviation.
    %
:- func indent(docs) = doc.

    % group(Docs):
    %
    % If Docs can be output on the remainder of the current line by ignoring
    % any nls in Docs, then do so. Otherwise nls in Docs are printed
    % (followed by any indentation). The formatting test is applied recursively
    % for any subgroups in Docs.
    %
:- func group(list(doc)) = doc.

    % format(X) = format_univ(univ(X)):
    %
    % A convenient abbreviation.
    %
:- func format(T) = doc.

    % format_arg(Doc) has the effect of formatting any term in Doc as though
```

```
    % it were an argument in a Mercury term by enclosing it in parenthe-
ses if
    % necessary.
    %
:- func format_arg(doc) = doc.

%---------------------------------------------------%
%
% Functions for converting docs to strings and writing them out to streams.
%

    % write_doc(Doc, !IO):
    % write_doc(FileStream, Doc, !IO):
    %
    % Format Doc to io.stdout_stream or FileStream respectively using put_doc,
    % with include_details_cc, the default formatter_map, and the default
    % pp_params.
    %
:- pred write_doc(doc::in, io::di, io::uo) is det.
:- pred write_doc(io.output_stream::in, doc::in, io::di, io::uo) is det.

    % put_doc(Stream, Canonicalize, FMap, Params, Doc, !State):
    %
    % Format Doc to Stream. Format format_univ(_) docs using specialised
    % formatters Formatters, and using Params as the pretty printer parameters.
    % The Canonicalize argument controls how put_doc deconstructs values
    % of noncanonical types (see the documentation of the noncanon_handling
    % type for details).
    %
:- pred put_doc(Stream, noncanon_handling, formatter_map, pp_params,
    doc, State, State)
    <= stream.writer(Stream, string, State).
:- mode put_doc(in, in(canonicalize), in, in, in, di, uo) is det.
:- mode put_doc(in, in(include_details_cc), in, in, in, di, uo) is cc_multi.

%---------------------------------------------------%
%
% Mechanisms for controlling *how* docs are converted to strings.
%

    % The type of generic formatting functions.
    % The first argument is the univ of the value to be formatted.
    % The second argument is the list of argument type_descs for
    % the type of the first argument.
    %
:- type formatter == ( func(univ, list(type_desc)) = doc ).
```

```
    % A formatter_map maps types to pps. Types are identified by module name,
    % type name, and type arity.
    %
:- type formatter_map.

    % Construct a new formatter_map.
    %
:- func new_formatter_map = formatter_map.

    % set_formatter(ModuleName, TypeName, TypeArity, Formatter, !FMap):
    %
    % Update !FMap to use Formatter to format the type
    % ModuleName.TypeName/TypeArity.
    %
:- pred set_formatter(string::in, string::in, int::in, formatter::in,
    formatter_map::in, formatter_map::out) is det.

%---------------------------------------------------%

    % The func_symbol_limit type controls *how many* of the function symbols
    % stored in the term inside a format_univ, format_list, or format_term doc
    % the write_doc family of functions should include in the resulting string.
    %
    % A limit of linear(N) formats the first N functors before truncating
    % output to "...".
    %
    % A limit of triangular(N) formats a term t(X1, ..., Xn) by applying
    % the following limits:
    %
    % - triangular(N - 1) when formatting X1,
    % - triangular(N - 2) when formatting X2,
    % - ..., and
    % - triangular(N - n) when formatting Xn.
    %
    % The cost of formatting the term t(X1, ..., Xn) as a whole is just one,
    % so a sequence of terms T1, T2, ... is formatted with limits
    % triangular(N), triangular(N - 1), ... respectively. When the limit
    % is exhausted, terms are output as just "...".
    %
:- type func_symbol_limit
    --->    linear(int)
    ;       triangular(int).

    % The pp_params type contains the parameters of the prettyprinting process:
    %
    % - the width of each line,
    % - the maximum number of lines to print, and
```

```
    % - the controls for how many function symbols to print.
    %
:- type pp_params
    --->    pp_params(
                pp_line_width   :: int,
                pp_max_lines    :: int,
                pp_limit        :: func_symbol_limit
            ).

%-----------------------------------------------------%

    % A user-configurable default set of type-specific formatters and
    % formatting parameters is always attached to the I/O state.
    % The write_doc predicate (in both its arities) uses these settings.
    %
    % The get_default_formatter_map predicate reads the default formatter_map
    % from the current I/O state, while set_default_formatter_map writes
    % the specified formatter_map to the I/O state to become the new default.
    %
    % The initial value of the default formatter_map provides the means
    % to prettyprint the most commonly used types in the Mercury standard
    % library, such as arrays, chars, floats, ints, maps, strings, etc.
    %
    % The default formatter_map may also be updated by users' modules
    % (e.g. in initialisation goals).
    %
    % These defaults are thread local, and therefore changes made by one thread
    % to the default formatter_map will not be visible in another thread.
    %
:- pred get_default_formatter_map(formatter_map::out, io::di, io::uo) is det.
:- pred set_default_formatter_map(formatter_map::in, io::di, io::uo) is det.

    % set_default_formatter(ModuleName, TypeName, TypeArity, Formatter, !IO):
    %
    % Update the default formatter in the I/O state to use Formatter
    % to print values of the type ModuleName.TypeName/TypeArity.
    %
:- pred set_default_formatter(string::in, string::in, int::in, formatter::in,
    io::di, io::uo) is det.

    % Alongside the default formatter_map, the I/O state also always stores
    % a default set of pretty-printing parameters (pp_params) for use by
    % the write_doc predicate (in both its arities).
    %
    % The get_default_params predicate reads the default parameters
    % from the current I/O state, while set_default_params writes the specified
    % parameters to the I/O state to become the new default.
```

```
    %
    % The initial default parameters are pp_params(78, 100, triangular(100)).
    %
    % These defaults are thread local, and therefore changes made by one thread
    % to the default pp_params will not be visible in another thread.
    %
:- pred get_default_params(pp_params::out, io::di, io::uo) is det.
:- pred set_default_params(pp_params::in, io::di, io::uo) is det.

%-----------------------------------------------------%
%-----------------------------------------------------%
```

59 prolog

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 1997-2003, 2005-2006, 2012 The University of Melbourne.
% Copyright (C) 2014-2016, 2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: prolog.m.
% Main author: fjh.
% Stability: high.
%
% This file contains predicates that are intended to help people
% porting Prolog programs, or writing programs in the intersection
% of Mercury and Prolog.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module prolog.
:- interface.

:- import_module list.
:- import_module pair.
:- import_module univ.

%-----------------------------------------------------%
%
% Prolog arithmetic operators.
%
```

```
:- pred T =:= T.            % In Mercury, just use =
:- mode in =:= in is semidet.

:- pred T =\= T.            % In Mercury, just use \=
:- mode in =\= in is semidet.

% is/2 is currently defined in int.m, for historical reasons.
%
% :- pred is(T, T) is det.        % In Mercury, just use =
% :- mode is(uo, di) is det.
% :- mode is(out, in) is det.

%-----------------------------------------------%
%
% Prolog term comparison operators.
%

:- pred T == T.             % In Mercury, just use =
:- mode in == in is semidet.

:- pred T \== T.            % In Mercury, just use \=
:- mode in \== in is semidet.

% Prolog's so-called "univ" operator, `=..'.
% Note: this is not related to Mercury's "univ" type!
% In Mercury, use `deconstruct.deconstruct' instead.

:- pred T =.. univ_result.
:- mode in =.. out is det.
    %
    % Note that the Mercury =.. is a bit different to the Prolog
    % one.  We could make it slightly more similar by overloading '.'/2,
    % but that would cause ambiguities that might prevent type
    % inference in a lot of cases.
    %
% :- type univ_result ---> '.'(string, list(univ)).
:- type univ_result == pair(string, list(univ)).

    % arg/3.
    % In Mercury, use arg/4 (defined in module deconstruct) instead:
    %
    %   arg(ArgNum, Term, Data) :-
    %       deconstruct.arg(Term, canonicalize, ArgNum - 1, Data).
    %
:- pred arg(int::in, T::in, univ::out) is semidet.

    % det_arg/3: like arg/3, but calls error/1 rather than failing
```

```
    % if the index is out of range.
    %
:- pred det_arg(int::in, T::in, univ::out) is det.

%---------------------------------------------------%
%---------------------------------------------------%
```

60 psqueue

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 2014-2019 The Mercury Team
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: psqueue.m.
% Main author: Matthias Gdemann.
% Stability: low.
%
% This module implements priority search queues. A priority search queue,
% or psqueue for short, combines in a single ADT the functionality of both
% a map and a priority queue.
%
% Psqueues map from priorities to keys and back. This modules provide functions
% and predicates to lookup the priority of a key, to insert and to remove
% priority-key pairs, to adjust the priority of a given key, and to retrieve
% the priority/key pair with the highest conceptual priority. However,
% since in many applications of psqueues, a low number represents high
% priority; for example, Dijkstra's shortest path algorithm wants to process
% the nearest nodes first. Therefore, given two priorities PrioA and PrioB,
% this module considers priority PrioA to have the higher conceptual priority
% if compare(CMP, PrioA, PrioB) returns CMP = (<). If priorities are numerical,
% which is common but is not required, then higher priorities are represented
% by lower numbers.
%
% The operations in this module are based on the algorithms described in
% Ralf Hinze: A simple implementation technique for priority search queues,
% Proceedings of the International Conference on Functional Programming 2001,
% pages 110-121. They use a weight-balanced tree to store priority/key pairs,
% to allow the following operation complexities:
%
% psqueue.insert        insert new priority/key pair:   O(log n)
% psqueue.lookup        lookup the priority of a key:   O(log n)
% psqueue.adjust        adjust the priority of a key:   O(log n)
```

```
% psqueue.peek:         read highest priority pair:     O(1)
% psqueue.remove_least: remove highest priority pair:   O(log n)
% psqueue.remove        remove pair with given key:     O(log n)
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module psqueue.
:- interface.

:- import_module assoc_list.

%-----------------------------------------------------%

:- type psqueue(P, K).

    % Create an empty priority search queue.
    %
:- func init = psqueue(P, K).
:- pred init(psqueue(P, K)::out) is det.

    % True iff the priority search queue is empty.
    %
:- pred is_empty(psqueue(P, K)::in) is semidet.

    % Create a singleton psqueue.
    %
:- func singleton(P, K) = psqueue(P, K).
:- pred singleton(P::in, K::in, psqueue(P, K)::out) is det.

    % Insert key K with priority P into the given priority search queue.
    % Fail if the key already exists.
    %
:- pred insert(P::in, K::in, psqueue(P, K)::in, psqueue(P, K)::out) is semidet.
:- pragma type_spec(insert/4, P = int).

    % Insert key K with priority P into the given priority search queue.
    % Throw an exception if the key already exists.
    %
:- func det_insert(psqueue(P, K), P, K) = psqueue(P, K).
:- pred det_insert(P::in, K::in, psqueue(P, K)::in, psqueue(P, K)::out) is det.
:- pragma type_spec(det_insert/3, P = int).
:- pragma type_spec(det_insert/4, P = int).

    % Return the highest priority priority/key pair in the given queue.
    % Fail if the queue is empty.
    %
```

```
:- pred peek(psqueue(P, K)::in, P::out, K::out) is semidet.

    % Return the highest priority priority/key pair in the given queue.
    % Throw an exception if the queue is empty.
    %
:- pred det_peek(psqueue(P, K)::in, P::out, K::out) is det.

    % Remove the element with the top priority. If the queue is empty, fail.
    %
:- pred remove_least(P::out, K::out, psqueue(P, K)::in, psqueue(P, K)::out)
    is semidet.
:- pragma type_spec(remove_least/4, P = int).

    % Remove the element with the top priority. If the queue is empty,
    % throw an exception.
    %
:- pred det_remove_least(P::out, K::out, psqueue(P, K)::in, psqueue(P, K)::out)
    is det.
:- pragma type_spec(det_remove_least/4, P = int).

    % Create an association list from a priority search queue.
    % The returned list will be in ascending order, sorted first on priority,
    % and then on key.
    %
:- func to_assoc_list(psqueue(P, K)) = assoc_list(P, K).
:- pred to_assoc_list(psqueue(P, K)::in, assoc_list(P, K)::out) is det.
:- pragma type_spec(to_assoc_list/1, P = int).
:- pragma type_spec(to_assoc_list/2, P = int).

    % Create a priority search queue from an assoc_list of priority/key pairs.
    %
:- func from_assoc_list(assoc_list(P, K)) = psqueue(P, K).
:- pred from_assoc_list(assoc_list(P, K)::in, psqueue(P, K)::out) is det.
:- pragma type_spec(from_assoc_list/1, P = int).
:- pragma type_spec(from_assoc_list/2, P = int).

    % Remove the element with the given key from a priority queue.
    % Fail if it is not in the queue.
    %
:- pred remove(P::out, K::in, psqueue(P, K)::in, psqueue(P, K)::out)
    is semidet.
:- pragma type_spec(remove/4, P = int).

    % Remove the element with the given key from a priority queue.
    % Throw an exception if it is not in the queue.
    %
:- pred det_remove(P::out, K::in, psqueue(P, K)::in, psqueue(P, K)::out)
```

```
    is det.
:- pragma type_spec(det_remove/4, P = int).

    % Adjust the priority of the specified element; the new priority will be
    % the value returned by the given adjustment function on the old priority.
    % Fail if the element is not in the queue.
    %
:- pred adjust((func(P) = P)::in, K::in, psqueue(P, K)::in, psqueue(P, K)::out)
    is semidet.
:- pragma type_spec(adjust/4, P = int).

    % Search for the priority of the specified key. If it is not in the queue,
    % fail.
    %
:- pred search(psqueue(P, K)::in, K::in, P::out) is semidet.
:- pragma type_spec(search/3, P = int).

    % Search for the priority of the specified key. If it is not in the queue,
    % throw an exception.
    %
:- func lookup(psqueue(P, K), K) = P.
:- pred lookup(psqueue(P, K)::in, K::in, P::out) is det.
:- pragma type_spec(lookup/2, P = int).
:- pragma type_spec(lookup/3, P = int).

    % Return all priority/key pairs whose priority is less than or equal to
    % the given priority.
    %
:- func at_most(psqueue(P, K), P) = assoc_list(P, K).
:- pred at_most(psqueue(P, K)::in, P::in, assoc_list(P, K)::out) is det.
:- pragma type_spec(at_most/2, P = int).
:- pragma type_spec(at_most/3, P = int).

    % Return the number of priority/key pairs in the given queue.
    %
:- func size(psqueue(P, K)) = int.
:- pred size(psqueue(P, K)::in, int::out) is det.
:- pragma type_spec(size/1, P = int).
:- pragma type_spec(size/2, P = int).

%---------------------------------------------------%
```

61 queue

```
%---------------------------------------------------%
```

```
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 1994-1995, 1997-1999, 2003-2006, 2011 The University of Melbourne.
% Copyright (C) 2014-2016, 2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: queue.m.
% Main author: fjh.
% Stability: high.
%
% This file contains a ‘queue’ ADT.
% A queue holds a sequence of values, and provides operations
% to insert values at the end of the queue (put) and remove them from
% the front of the queue (get).
%
% This implementation is in terms of a pair of lists.
% The put and get operations are amortized constant-time.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module queue.
:- interface.

:- import_module list.

%-----------------------------------------------------%

:- type queue(T).

    % ‘init(Queue)’ is true iff ‘Queue’ is an empty queue.
    %
:- func init = queue(T).
:- pred init(queue(T)::out) is det.

    % 'queue_equal(Q1, Q2)' is true iff Q1 and Q2 contain the same
    % elements in the same order.
    %
:- pred equal(queue(T)::in, queue(T)::in) is semidet.

    % ‘is_empty(Queue)’ is true iff ‘Queue’ is an empty queue.
    %
:- pred is_empty(queue(T)::in) is semidet.

    % ‘is_full(Queue)’ is intended to be true iff ‘Queue’ is a queue
```

```
    % whose capacity is exhausted. This implementation allows arbitrary-
sized
    % queues, so is_full always fails.
    %
:- pred is_full(queue(T)::in) is semidet.

    % 'put(Elem, Queue0, Queue)' is true iff 'Queue' is the queue
    % which results from appending 'Elem' onto the end of 'Queue0'.
    %
:- func put(queue(T), T) = queue(T).
:- pred put(T::in, queue(T)::in, queue(T)::out) is det.

    % 'put_list(Elems, Queue0, Queue)' is true iff 'Queue' is the queue
    % which results from inserting the items in the list 'Elems' into 'Queue0'.
    %
:- func put_list(queue(T), list(T)) = queue(T).
:- pred put_list(list(T)::in, queue(T)::in, queue(T)::out) is det.

    % 'first(Queue, Elem)' is true iff 'Queue' is a non-empty queue
    % whose first element is 'Elem'.
    %
:- pred first(queue(T)::in, T::out) is semidet.

    % 'get(Elem, Queue0, Queue)' is true iff 'Queue0' is a non-empty
    % queue whose first element is 'Elem', and 'Queue' the queue which results
    % from removing that element from the front of 'Queue0'.
    %
:- pred get(T::out, queue(T)::in, queue(T)::out) is semidet.

    % 'length(Queue, Length)' is true iff 'Queue' is a queue
    % containing 'Length' elements.
    %
:- func length(queue(T)) = int.
:- pred length(queue(T)::in, int::out) is det.

    % 'list_to_queue(List, Queue)' is true iff 'Queue' is a queue
    % containing the elements of List, with the first element of List at
    % the head of the queue.
    %
:- func list_to_queue(list(T)) = queue(T).
:- pred list_to_queue(list(T)::in, queue(T)::out) is det.

    % A synonym for list_to_queue/1.
    %
:- func from_list(list(T)) = queue(T).

    % 'to_list(Queue) = List' is the inverse of from_list/1.
```

```
    %
:- func to_list(queue(T)) = list(T).

    % ‘delete_all(Elem, Queue0, Queue)’ is true iff ‘Queue’ is the same
    % queue as ‘Queue0’ with all occurrences of ‘Elem’ removed from it.
    %
:- func delete_all(queue(T), T) = queue(T).
:- pred delete_all(T::in, queue(T)::in, queue(T)::out) is det.

    % ‘put_on_front(Queue0, Elem) = Queue’ pushes ‘Elem’ on to
    % the front of ‘Queue0’, giving ‘Queue’.
    %
:- func put_on_front(queue(T), T) = queue(T).
:- pred put_on_front(T::in, queue(T)::in, queue(T)::out) is det.

    % ‘put_list_on_front(Queue0, Elems) = Queue’ pushes ‘Elems’
    % on to the front of ‘Queue0’, giving ‘Queue’ (the N’th member
    % of ‘Elems’ becomes the N’th member from the front of ‘Queue’).
    %
:- func put_list_on_front(queue(T), list(T)) = queue(T).
:- pred put_list_on_front(list(T)::in, queue(T)::in, queue(T)::out)
    is det.

    % ‘get_from_back(Elem, Queue0, Queue)’ removes ‘Elem’ from
    % the back of ‘Queue0’, giving ‘Queue’.
    %
:- pred get_from_back(T::out, queue(T)::in, queue(T)::out) is semidet.

%---------------------------------------------------%
%---------------------------------------------------%
```

62 random

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 1994-1998,2001-2006, 2011 The University of Melbourne.
% Copyright (C) 2015-2016, 2018-2019 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: random.m
% Main author: Mark Brown
%
% This module provides interfaces to several random number generators,
```

```
% implementations of which can be found in the submodules.
%
% The interfaces can be used in three styles:
%
%   - In the "ground" style, an instance of the random/1 typeclass is
%   passed through the code using 'in' and 'out' modes. This can be used
%   to generate random numbers, and since the value is ground it can also
%   easily be stored in larger data structures. The major drawback is that
%   generators in this style tend to be either fast or of good quality,
%   but not both.
%
%   - In the "unique" style, the urandom/2 typeclass is used instead. Each
%   instance consists of a "params" type which is passed into the code
%   using an 'in' mode, and a "state" type which is passed through the
%   code using modes 'di' and 'uo'. The uniqueness allows destructive
%   update, which means that these generators can be both fast and good.
%
%   - A generator can be attached to the I/O state. In this case, the
%   interface is the same as the unique style, with 'io' being used as
%   the unique state.
%
% Each generator defined in the submodules is natively one of the first
% two styles. Adaptors are defined below for converting between these,
% or from either of these to the third style.
%
%
% Example, ground style:
%
%   main(!IO) :-
%       R0 = sfc16.init,
%       roll(R0, R1, !IO),
%       roll(R1, _, !IO).
%
%   :- pred roll(R::in, R::out, io::di, io::uo) is det <= random(R).
%
%   roll(!R, !IO) :-
%       uniform_int_in_range(1, 6, N, !R),
%       io.format("You rolled a %d\n", [i(N)], !IO).
%
%
% Example, unique style:
%
%   main(!IO) :-
%       sfc64.init(P, S0),
%       roll(P, S0, S1, !IO),
%       roll(P, S1, _, !IO).
%
```

```
%   :- pred roll(P::in, S::di, S::uo, io::di, io::uo) is det <= urandom(P, S).
%
%   roll(P, !S, !IO) :-
%       uniform_int_in_range(P, 1, 6, N, !S),
%       io.format("You rolled a %d\n", [i(N)], !IO).
%
%
% Example, attached to I/O state:
%
%   main(!IO) :-
%       % Using a ground generator.
%       R = sfc16.init,
%       make_io_random(R, M1, !IO),
%       roll(M1, !IO),
%       roll(M1, !IO),
%
%       % Using a unique generator.
%       sfc64.init(P, S),
%       make_io_urandom(P, S, M2, !IO),
%       roll(M2, !IO),
%       roll(M2, !IO).
%
%   :- pred roll(M::in, io::di, io::uo) is det <= urandom(M, io).
%
%   roll(M, !IO) :-
%       uniform_int_in_range(M, 1, 6, N, !IO),
%       io.format("You rolled a %d\n", [i(N)], !IO).
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module random.
:- interface.

:- include_module sfc16.
:- include_module sfc32.
:- include_module sfc64.

:- import_module array.
:- import_module io.
:- import_module list.

%-----------------------------------------------------%

    % Interface to random number generators.
    %
:- typeclass random(R) where [
```

```
        % Generate a uniformly distributed pseudo-random unsigned integer
        % of 8, 16, 32 or 64 bits, respectively.
        %
    pred generate_uint8(uint8::out, R::in, R::out) is det,
    pred generate_uint16(uint16::out, R::in, R::out) is det,
    pred generate_uint32(uint32::out, R::in, R::out) is det,
    pred generate_uint64(uint64::out, R::in, R::out) is det

].

    % uniform_int_in_range(Start, Range, N, !R)
    %
    % Generate a pseudo-random integer that is uniformly distributed
    % in the range Start to (Start + Range - 1), inclusive.
    %
    % Throws an exception if Range < 1 or Range > uint32_max.
    %
:- pred uniform_int_in_range(int::in, int::in, int::out, R::in, R::out)
    is det <= random(R).

    % uniform_uint_in_range(Start, Range, N, !R)
    %
    % Generate a pseudo-random unsigned integer that is uniformly
    % distributed in the range Start to (Start + Range - 1), inclusive.
    %
    % Throws an exception if Range < 1 or Range > uint32_max.
    %
:- pred uniform_uint_in_range(uint::in, uint::in, uint::out, R::in, R::out)
    is det <= random(R).

    % uniform_float_in_range(Start, Range, N, !R)
    %
    % Generate a pseudo-random float that is uniformly distributed
    % in the interval [Start, Start + Range).
    %
:- pred uniform_float_in_range(float::in, float::in, float::out, R::in, R::out)
    is det <= random(R).

    % uniform_float_around_mid(Mid, Delta, N, !R)
    %
    % Generate a pseudo-random float that is uniformly distributed
    % in the interval (Mid - Delta, Mid + Delta).
    %
:- pred uniform_float_around_mid(float::in, float::in, float::out,
    R::in, R::out) is det <= random(R).
```

```
    % uniform_float_in_01(N, !R)
    %
    % Generate a pseudo-random float that is uniformly distributed
    % in the interval [0.0, 1.0).
    %
:- pred uniform_float_in_01(float::out, R::in, R::out) is det <= random(R).

    % normal_floats(M, SD, U, V, !R)
    %
    % Generate two pseudo-random floats from a normal (i.e., Gaussian)
    % distribution with mean M and standard deviation SD, using the
    % Box-Muller method.
    %
    % We generate two at a time for efficiency; they are independent of
    % each other.
    %
:- pred normal_floats(float::in, float::in, float::out, float::out,
    R::in, R::out) is det <= random(R).

    % normal_floats(U, V, !R)
    %
    % Generate two pseudo-random floats from a normal (i.e., Gaussian)
    % distribution with mean 0.0 and standard deviation 1.0, using the
    % Nox-Muller method.
    %
    % We generate two at a time for efficiency; they are independent of
    % each other.
    %
:- pred normal_floats(float::out, float::out, R::in, R::out) is det
    <= random(R).

    % Generate a random permutation of a list.
    %
:- pred shuffle_list(list(T)::in, list(T)::out, R::in, R::out) is det
    <= random(R).

    % Generate a random permutation of an array.
    %
:- pred shuffle_array(array(T)::array_di, array(T)::array_uo, R::in, R::out)
    is det <= random(R).

%-----------------------------------------------------%

    % Interface to unique random number generators. Callers need to
    % ensure they preserve the uniqueness of the random state, and in
    % turn instances can use destructive update on it.
    %
```

```
:- typeclass urandom(P, S) <= (P -> S) where [

        % Generate a uniformly distributed pseudo-random unsigned integer
        % of 8, 16, 32 or 64 bits, respectively.
        %
    pred generate_uint8(P::in, uint8::out, S::di, S::uo) is det,
    pred generate_uint16(P::in, uint16::out, S::di, S::uo) is det,
    pred generate_uint32(P::in, uint32::out, S::di, S::uo) is det,
    pred generate_uint64(P::in, uint64::out, S::di, S::uo) is det

].

:- typeclass urandom_dup(S) where [

        % urandom_dup(!S, !:Sdup)
        %
        % Create a duplicate random state that will generate the same
        % sequence of integers.
        %
    pred urandom_dup(S::di, S::uo, S::uo) is det

].

    % uniform_int_in_range(P, Start, Range, N, !S)
    %
    % Generate a pseudo-random integer that is uniformly distributed
    % in the range Start to (Start + Range - 1), inclusive.
    %
    % Throws an exception if Range < 1 or Range > uint32_max.
    %
:- pred uniform_int_in_range(P::in, int::in, int::in, int::out, S::di, S::uo)
    is det <= urandom(P, S).

    % uniform_uint_in_range(P, Start, Range, N, !S)
    %
    % Generate a pseudo-random unsigned integer that is uniformly
    % distributed in the range Start to (Start + Range - 1), inclusive.
    %
    % Throws an exception if Range < 1 or Range > uint32_max.
    %
:- pred uniform_uint_in_range(P::in, uint::in, uint::in, uint::out,
    S::di, S::uo) is det <= urandom(P, S).

    % uniform_float_in_range(P, Start, Range, N, !S)
    %
    % Generate a pseudo-random float that is uniformly distributed
    % in the interval [Start, Start + Range).
```

```
    %
:- pred uniform_float_in_range(P::in, float::in, float::in, float::out,
    S::di, S::uo) is det <= urandom(P, S).

    % uniform_float_around_mid(P, Mid, Delta, N, !S)
    %
    % Generate a pseudo-random float that is uniformly distributed
    % in the interval (Mid - Delta, Mid + Delta).
    %
:- pred uniform_float_around_mid(P::in, float::in, float::in, float::out,
    S::di, S::uo) is det <= urandom(P, S).

    % uniform_float_in_01(P, N, !S)
    %
    % Generate a pseudo-random float that is uniformly distributed
    % in the interval [0.0, 1.0).
    %
:- pred uniform_float_in_01(P::in, float::out, S::di, S::uo) is det
    <= urandom(P, S).

    % normal_floats(P, M, S, U, V, !S)
    %
    % Generate two pseudo-random floats from a normal (i.e., Gaussian)
    % distribution with mean M and standard deviation S, using the
    % Box-Muller method.
    %
    % We generate two at a time for efficiency; they are independent of
    % each other.
    %
:- pred normal_floats(P::in, float::in, float::in, float::out, float::out,
    S::di, S::uo) is det <= urandom(P, S).

    % normal_floats(P, U, V, !S)
    %
    % Generate two pseudo-random floats from a normal (i.e., Gaussian)
    % distribution with mean 0.0 and standard deviation 1.0, using the
    % Box-Muller method.
    %
    % We generate two at a time for efficiency; they are independent of
    % each other.
    %
:- pred normal_floats(P::in, float::out, float::out, S::di, S::uo) is det
    <= urandom(P, S).

    % Generate a random permutation of a list.
    %
:- pred shuffle_list(P::in, list(T)::in, list(T)::out, S::di, S::uo) is det
```

```
    <= urandom(P, S).

    % Generate a random permutation of an array.
    %
:- pred shuffle_array(P::in, array(T)::array_di, array(T)::array_uo,
    S::di, S::uo) is det <= urandom(P, S).

%---------------------------------------------------%
%---------------------------------------------------%

    % Convert any instance of random/1 into an instance of urandom/2.
    % This creates additional overhead in the form of additional
    % typeclass method calls.
    %
:- type urandom_params(R).
:- type urandom_state(R).

:- instance urandom(urandom_params(R), urandom_state(R)) <= random(R).
:- instance urandom_dup(urandom_state(R)) <= random(R).

:- pred make_urandom(R::in, urandom_params(R)::out, urandom_state(R)::uo)
    is det.

%---------------------------------------------------%

    % Convert any instance of urandom/2 and urandom_dup/1 into an
    % instance of random/1. This duplicates the state every time a
    % random number is generated, hence may use significantly more
    % memory than if the unique version were used directly.
    %
:- type shared_random(P, S).

:- instance random(shared_random(P, S)) <= (urandom(P, S), urandom_dup(S)).

:- func make_shared_random(P::in, S::di) = (shared_random(P, S)::out) is det.

%---------------------------------------------------%

    % Convert any instance of random/1 into an instance of urandom/2
    % where the state is the I/O state.
    %
:- type io_random(R).

:- instance urandom(io_random(R), io) <= random(R).

:- pred make_io_random(R::in, io_random(R)::out, io::di, io::uo) is det
    <= random(R).
```

```
%---------------------------------------------------%

    % Convert any instance of urandom/2 into an instance of urandom/2
    % where the state is the I/O state.
    %
:- type io_urandom(P, S).

:- instance urandom(io_urandom(P, S), io) <= urandom(P, S).

:- pred make_io_urandom(P::in, S::di, io_urandom(P, S)::out, io::di, io::uo)
    is det <= urandom(P, S).

%---------------------------------------------------%
%---------------------------------------------------%
%
% Interface to the older random number generator. This is now deprecated.
%
% Define a set of random number generator predicates. This implementation
% uses a threaded random-number supply.  The supply can be used in a
% non-unique way, which means that each thread returns the same list of
% random numbers.  However, this may not be desired so in the interests
% of safety it is also declared with (backtrackable) unique modes.
%
% The coefficients used in the implementation were taken from Numerical
% Recipes in C (Press et al), and are originally due to Knuth.  These
% coefficients are described as producing a "Quick and Dirty" random number
% generator, which generates the numbers very quickly but not necessarily
% with a high degree of quality.  As with all random number generators,
% the user is advised to consider carefully whether this generator meets
% their requirements in terms of "randomness".  For applications which have
% special needs (e.g. cryptographic key generation), a generator such as
% this is unlikely to be suitable.
%
% Note that random number generators of this type have several known
% pitfalls which the user may need to avoid:
%
%   1) The high bits tend to be more random than the low bits.  If
%   you wish to generate a random integer within a given range, you
%   should something like 'div' to reduce the random numbers to the
%   required range rather than something like 'mod' (or just use
%   random.random/5).
%
%   2) Similarly, you should not try to break a random number up into
%   components.  Instead, you should generate each number with a
%   separate call to this module.
%
```

```
%   3) There can be sequential correlation between successive calls,
%   so you shouldn't try to generate tuples of random numbers, for
%   example, by generating each component of the tuple in sequential
%   order.  If you do, it is likely that the resulting sequence will
%   not cover the full range of possible tuples.
%
%-----------------------------------------------%

    % The type 'supply' represents a supply of random numbers.
    %
:- type supply.

    % init(Seed, RS).
    %
    % Creates a supply of random numbers RS using the specified Seed.
    %
    % This predicate has been declared obsolete because all of the
    % interface from here on is deprecated. All code using this part
    % of the interface will need to be updated.
    %
:- pragma obsolete(init/2).
:- pred init(int::in, supply::uo) is det.

    % random(Num, !RS).
    %
    % Extracts a number Num in the range 0 .. RandMax from the random number
    % supply !RS.
    %
:- pred random(int, supply, supply).
:- mode random(out, in, out) is det.
:- mode random(out, mdi, muo) is det.

    % random(Low, Range, Num, !RS).
    %
    % Extracts a number Num in the range Low .. (Low + Range - 1) from the
    % random number supply !RS.  For best results, the value of Range should be
    % no greater than about 100.
    %
:- pred random(int, int, int, supply, supply).
:- mode random(in, in, out, in, out) is det.
:- mode random(in, in, out, mdi, muo) is det.

    % randmax(RandMax, !RS).
    %
    % Binds RandMax to the maximum random number that can be returned from the
    % random number supply !RS, the state of the supply is unchanged.
    %
```

```
:- pred randmax(int, supply, supply).
:- mode randmax(out, in, out) is det.
:- mode randmax(out, mdi, muo) is det.

    % randcount(RandCount, !RS).
    %
    % Binds RandCount to the number of distinct random numbers that can be
    % returned from the random number supply !RS.  The state of the sup-
ply is
    % unchanged.  This will be one more than the number returned by randmax/3.
    %
:- pred randcount(int, supply, supply).
:- mode randcount(out, in, out) is det.
:- mode randcount(out, mdi, muo) is det.

    % permutation(List0, List, !RS).
    %
    % Binds List to a random permutation of List0.
    %
:- pred permutation(list(T), list(T), supply, supply).
:- mode permutation(in, out, in, out) is det.
:- mode permutation(in, out, mdi, muo) is det.

%---------------------------------------------------%
%---------------------------------------------------%
```

63 random.sfc16

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 2019 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: random.sfc16.m
% Main author: Mark Brown
%
% 16-bit Small Fast Counting generator, by Chris Doty-Humphrey.
%
% http://pracrand.sourceforge.net/
%
% From the above:
% "[A] good small chaotic RNG driven by a bad smaller linear RNG. The
% combination gives it the strengths of each - good chaotic behavior,
```

```
% but enough structure to avoid short cycles."
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module random.sfc16.
:- interface.

%-----------------------------------------------------%

    % A fast, 16-bit SFC generator.
    %
:- type random.

:- instance random(random).

    % Initialise a 16-bit SFC generator with the default seed. The
    % resulting generator produces the same sequence every time.
    %
:- func init = random.

    % Initialise a 16-bit SFC generator with the given seed.
    %
:- func seed(uint64) = random.

    % Generate a uniformly distributed pseudo-random unsigned integer
    % of 8, 16, 32 or 64 bits, respectively.
    %
:- pred generate_uint8(uint8::out, random::in, random::out) is det.
:- pred generate_uint16(uint16::out, random::in, random::out) is det.
:- pred generate_uint32(uint32::out, random::in, random::out) is det.
:- pred generate_uint64(uint64::out, random::in, random::out) is det.

%-----------------------------------------------------%
%-----------------------------------------------------%
```

64 random.sfc32

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 2019 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
```

```
% File: random.sfc32.m
% Main author: Mark Brown
%
% 32-bit Small Fast Counting generator, by Chris Doty-Humphrey.
%
% http://pracrand.sourceforge.net/
%
% From the above:
% "[A] good small chaotic RNG driven by a bad smaller linear RNG. The
% combination gives it the strengths of each - good chaotic behavior,
% but enough structure to avoid short cycles."
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module random.sfc32.
:- interface.

%-----------------------------------------------------%

    % A fast, 32-bit SFC generator with unique state. This may achieve
    % better performance on 32-bit architectures, but generally does not
    % have the quality of the 64-bit generator or the low heap usage of
    % the 16-bit generator.
    %
:- type params.
:- type ustate.

:- instance urandom(params, ustate).
:- instance urandom_dup(ustate).

    % Initialise a 32-bit SFC generator with the default seed. The
    % resulting generator produces the same sequence every time.
    %
:- pred init(params::out, ustate::uo) is det.

    % Initialise a 32-bit SFC generator with the given seed.
    %
:- pred seed(uint32::in, uint32::in, uint32::in, params::out, ustate::uo)
    is det.

    % Generate a uniformly distributed pseudo-random unsigned integer
    % of 8, 16, 32 or 64 bits, respectively.
    %
:- pred generate_uint8(params::in, uint8::out,
    ustate::di, ustate::uo) is det.
:- pred generate_uint16(params::in, uint16::out,
```

```
    ustate::di, ustate::uo) is det.
:- pred generate_uint32(params::in, uint32::out,
    ustate::di, ustate::uo) is det.
:- pred generate_uint64(params::in, uint64::out,
    ustate::di, ustate::uo) is det.

    % Duplicate a 32-bit SFC state.
    %
:- pred urandom_dup(ustate::di, ustate::uo, ustate::uo) is det.

%---------------------------------------------------%

    % Generate a uniformly distributed pseudo-random unsigned integer
    % of 8, 16, 32 or 64 bits, respectively.
    %
    % As above, but does not require the params argument (which is a dummy
    % type only needed to satisfy the typeclass interface).
    %
:- pred generate_uint8(uint8::out, ustate::di, ustate::uo) is det.
:- pred generate_uint16(uint16::out, ustate::di, ustate::uo) is det.
:- pred generate_uint32(uint32::out, ustate::di, ustate::uo) is det.
:- pred generate_uint64(uint64::out, ustate::di, ustate::uo) is det.

%---------------------------------------------------%
%---------------------------------------------------%
```

65 random.sfc64

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 2019 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: random.sfc64.m
% Main author: Mark Brown
%
% 64-bit Small Fast Counting generator, by Chris Doty-Humphrey.
%
% http://pracrand.sourceforge.net/
%
% From the above:
% "[A] good small chaotic RNG driven by a bad smaller linear RNG. The
% combination gives it the strengths of each - good chaotic behavior,
```

```
% but enough structure to avoid short cycles."
%
%---------------------------------------------------%
%---------------------------------------------------%

:- module random.sfc64.
:- interface.

%---------------------------------------------------%

    % A fast, 64-bit SFC generator with unique state.
    %
:- type params.
:- type ustate.

:- instance urandom(params, ustate).
:- instance urandom_dup(ustate).

    % Initialise a 64-bit SFC generator with the default seed. The
    % resulting generator produces the same sequence every time.
    %
:- pred init(params::out, ustate::uo) is det.

    % Initialise a 64-bit SFC generator with the given seed.
    %
:- pred seed(uint64::in, uint64::in, uint64::in, params::out, ustate::uo)
    is det.

    % Generate a uniformly distributed pseudo-random unsigned integer
    % of 8, 16, 32 or 64 bits, respectively.
    %
:- pred generate_uint8(params::in, uint8::out,
    ustate::di, ustate::uo) is det.
:- pred generate_uint16(params::in, uint16::out,
    ustate::di, ustate::uo) is det.
:- pred generate_uint32(params::in, uint32::out,
    ustate::di, ustate::uo) is det.
:- pred generate_uint64(params::in, uint64::out,
    ustate::di, ustate::uo) is det.

    % Duplicate a 64-bit SFC state.
    %
:- pred urandom_dup(ustate::di, ustate::uo, ustate::uo) is det.

%---------------------------------------------------%

    % Generate a uniformly distributed pseudo-random unsigned integer
```

```
    % of 8, 16, 32 or 64 bits, respectively.
    %
    % As above, but does not require the params argument (which is a dummy
    % type only needed to satisfy the typeclass interface).
    %
:- pred generate_uint8(uint8::out, ustate::di, ustate::uo) is det.
:- pred generate_uint16(uint16::out, ustate::di, ustate::uo) is det.
:- pred generate_uint32(uint32::out, ustate::di, ustate::uo) is det.
:- pred generate_uint64(uint64::out, ustate::di, ustate::uo) is det.

%---------------------------------------------------%
%---------------------------------------------------%
```

66 ranges

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 2006-2009 The University of Melbourne.
% Copyright (C) 2013-2016 Opturion Pty Ltd.
% Copyright (C) 2017-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: ranges.m.
% Authors: Mark Brown.
% Stability: medium.
%
% This module defines the ranges abstract type.
%
%---------------------------------------------------%

:- module ranges.
:- interface.

:- import_module list.
:- import_module set.

%---------------------------------------------------%

    % Range lists represent sets of integers. Each contiguous block
    % of integers in the set is stored as an range which specifies
    % the bounds of the block, and these ranges are kept in a list-like
    % structure.
    %
```

```
:- type ranges.

    % empty returns the empty set.
    %
:- func empty = ranges.

    % is_empty(Set):
    % Succeeds iff Set is the empty set.
    %
:- pred is_empty(ranges::in) is semidet.

    % is_non_empty(Set):
    % Succeeds iff Set is not the empty set.
    %
:- pred is_non_empty(ranges::in) is semidet.

    % universe returns the largest set that can be handled by this module.
    % This is the set of integers (min_int+1)..max_int. Note that min_int
    % cannot be represented in any set.
    %
:- func universe = ranges.

    % range(Min, Max) is the set of all integers from Min to Max inclusive.
    %
:- func range(int, int) = ranges.

    % split(D, L, H, Rest) is true iff L..H is the first range
    % in D, and Rest is the domain D with this range removed.
    %
:- pred split(ranges::in, int::out, int::out, ranges::out) is semidet.

    % is_contiguous(R, L, H) <=> split(R, L, H, empty):
    % Test if the set is a contiguous set of integers and return the endpoints
    % of this set if this is the case.
    %
:- pred is_contiguous(ranges::in, int::out, int::out) is semidet.

    % Add an integer to the set.
    %
:- func insert(int, ranges) = ranges.
:- pred insert(int::in, ranges::in, ranges::out) is det.

    % Delete an integer from the set.
    %
:- func delete(int, ranges) = ranges.

    % Return the number of distinct integers which are in the ranges
```

```
    % (as opposed to the number of ranges).
    %
:- func size(ranges) = int.

    % Returns the median value of the set. In case of a tie, returns
    % the lower of the two options.
    %
:- func median(ranges) = int.

    % least(A, N) is true iff N is the least element of A.
    %
:- pred least(ranges::in, int::out) is semidet.

    % greatest(A, N) is true iff N is the greatest element of A.
    %
:- pred greatest(ranges::in, int::out) is semidet.

    % next(A, N0, N) is true iff N is the least element of A greater
    % than N0.
    %
:- pred next(ranges::in, int::in, int::out) is semidet.

    % Test set membership.
    %
:- pred member(int::in, ranges::in) is semidet.

    % Nondeterministically produce each range.
    %
:- pred range_member(int::out, int::out, ranges::in) is nondet.

    % Nondeterministically produce each element.
    %
:- pred nondet_member(int::out, ranges::in) is nondet.

    % subset(A, B) is true iff every value in A is in B.
    %
:- pred subset(ranges::in, ranges::in) is semidet.

    % disjoint(A, B) is true iff A and B have no values in common.
    %
:- pred disjoint(ranges::in, ranges::in) is semidet.

    % union(A, B) contains all the integers in either A or B.
    %
:- func union(ranges, ranges) = ranges.

    % intersection(A, B) contains all the integers in both A and B.
```

```
    %
:- func intersection(ranges, ranges) = ranges.

    % difference(A, B) contains all the integers which are in A but
    % not in B.
    %
:- func difference(ranges, ranges) = ranges.

    % restrict_min(Min, A) contains all integers in A which are greater
    % than or equal to Min.
    %
:- func restrict_min(int, ranges) = ranges.

    % restrict_max(Max, A) contains all integers in A which are less than
    % or equal to Max.
    %
:- func restrict_max(int, ranges) = ranges.

    % restrict_range(Min, Max, A) contains all integers I in A which
    % satisfy Min =< I =< Max.
    %
:- func restrict_range(int, int, ranges) = ranges.

    % prune_to_next_non_member(A0, A, N0, N):
    %
    % N is the smallest integer larger than or equal to N0 which is not
    % in A0. A is the set A0 restricted to values greater than N.
    %
:- pred prune_to_next_non_member(ranges::in, ranges::out,
    int::in, int::out) is det.

    % prune_to_prev_non_member(A0, A, N0, N):
    %
    % N is the largest integer smaller than or equal to N0 which is not
    % in A0. A is the set A0 restricted to values less than N.
    %
:- pred prune_to_prev_non_member(ranges::in, ranges::out,
    int::in, int::out) is det.

    % Negate all numbers:  A in R  <=>  -A in negate(R)
    %
:- func negate(ranges) = ranges.

    % The sum of two ranges.
    %
:- func plus(ranges, ranges) = ranges.
```

```
    % Shift a range by const C.
    %
:- func shift(ranges, int) = ranges.

    % Dilate a range by const C.
    %
:- func dilation(ranges, int) = ranges.

    % Contract a range by const C.
    %
:- func contraction(ranges, int) = ranges.

%-----------------------------------------------%

    % Convert to a sorted list of integers.
    %
:- func to_sorted_list(ranges) = list(int).

    % Convert from a list of integers.
    %
:- func from_list(list(int)) = ranges.

    % Convert from a set of integers.
    %
:- func from_set(set(int)) = ranges.

%-----------------------------------------------%

    % Compare the sets of integers given by the two ranges using lexicographic
    % ordering on the sorted set form.
    %
:- pred compare_lex(comparison_result::uo, ranges::in, ranges::in) is det.

%-----------------------------------------------%

:- pred foldl(pred(int, A, A), ranges, A, A).
:- mode foldl(pred(in, in, out) is det, in, in, out) is det.
:- mode foldl(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode foldl(pred(in, di, uo) is det, in, di, uo) is det.
:- mode foldl(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode foldl(pred(in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode foldl(pred(in, di, uo) is semidet, in, di, uo) is semidet.

:- pred foldl2(pred(int, A, A, B, B), ranges, A, A, B, B).
:- mode foldl2(pred(in, in, out, in, out) is det, in, in, out,
    in, out) is det.
:- mode foldl2(pred(in, in, out, mdi, muo) is det, in, in, out,
```

```
    mdi, muo) is det.
:- mode foldl2(pred(in, in, out, di, uo) is det, in, in, out,
    di, uo) is det.
:- mode foldl2(pred(in, in, out, in, out) is semidet, in, in, out,
    in, out) is semidet.
:- mode foldl2(pred(in, in, out, mdi, muo) is semidet, in, in, out,
    mdi, muo) is semidet.
:- mode foldl2(pred(in, in, out, di, uo) is semidet, in, in, out,
    di, uo) is semidet.

:- pred foldl3(pred(int, A, A, B, B, C, C), ranges, A, A, B, B, C, C).
:- mode foldl3(pred(in, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out) is det.
:- mode foldl3(pred(in, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, mdi, muo) is det.
:- mode foldl3(pred(in, in, out, in, out, di, uo) is det, in,
    in, out, in, out, di, uo) is det.
:- mode foldl3(pred(in, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, di, uo) is semidet.

:- pred foldr(pred(int, A, A), ranges, A, A).
:- mode foldr(pred(in, in, out) is det, in, in, out) is det.
:- mode foldr(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode foldr(pred(in, di, uo) is det, in, di, uo) is det.
:- mode foldr(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode foldr(pred(in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode foldr(pred(in, di, uo) is semidet, in, di, uo) is semidet.

%-----------------------------------------------%

    % For each range, call the predicate, passing it the lower and
    % upper bound and threading through an accumulator.
    %
:- pred range_foldl(pred(int, int, A, A), ranges, A, A).
:- mode range_foldl(pred(in, in, in, out) is det, in, in, out) is det.
:- mode range_foldl(pred(in, in, mdi, muo) is det, in, mdi, muo) is det.
:- mode range_foldl(pred(in, in, di, uo) is det, in, di, uo) is det.
:- mode range_foldl(pred(in, in, in, out) is semidet, in, in, out) is semidet.
:- mode range_foldl(pred(in, in, mdi, muo) is semidet, in, mdi, muo)
    is semidet.
:- mode range_foldl(pred(in, in, di, uo) is semidet, in, di, uo) is semidet.

    % As above, but with two accumulators.
    %
:- pred range_foldl2(pred(int, int, A, A, B, B), ranges, A, A, B, B).
:- mode range_foldl2(pred(in, in, in, out, in, out) is det,
    in, in, out, in, out) is det.
```

```
:- mode range_foldl2(pred(in, in, in, out, mdi, muo) is det,
    in, in, out, mdi, muo) is det.
:- mode range_foldl2(pred(in, in, in, out, di, uo) is det,
    in, in, out, di, uo) is det.
:- mode range_foldl2(pred(in, in, in, out, in, out) is semidet,
    in, in, out, in, out) is semidet.
:- mode range_foldl2(pred(in, in, in, out, mdi, muo) is semidet,
    in, in, out, mdi, muo) is semidet.
:- mode range_foldl2(pred(in, in, in, out, di, uo) is semidet,
    in, in, out, di, uo) is semidet.

:- pred range_foldr(pred(int, int, A, A), ranges, A, A).
:- mode range_foldr(pred(in, in, in, out) is det, in, in, out) is det.
:- mode range_foldr(pred(in, in, mdi, muo) is det, in, mdi, muo) is det.
:- mode range_foldr(pred(in, in, di, uo) is det, in, di, uo) is det.
:- mode range_foldr(pred(in, in, in, out) is semidet, in, in, out)
    is semidet.
:- mode range_foldr(pred(in, in, mdi, muo) is semidet, in, mdi, muo)
    is semidet.
:- mode range_foldr(pred(in, in, di, uo) is semidet, in, di, uo)
    is semidet.

%-----------------------------------------------%
%
% C interface to ranges.
%

% This section describes the C interface to the ranges/0 type that is exported
% by this module.
%
% In C the ranges/0 type is represented by the ML_Ranges type.
% The following operations are exported and may be called from C or C++
% code.
%
% ML_Ranges ML_ranges_empty(void);
% Return the empty set.
%
% ML_Ranges ML_ranges_universe(void);
% Return the set of integers from (min_int+1)..max_int.
%
% ML_Ranges ML_ranges_range(MR_Integer l, MR_Integer h);
% Return the set of integers from `l' to `h' inclusive.
%
% int ML_ranges_is_empty(ML_Ranges r);
% Return true iff `r` is the empty set.
%
% MR_Integer ML_ranges_size(ML_Ranges r);
```

```
% Return the number of distinct integers in ‘r’.
%
% int ML_ranges_split(ML_Ranges d, MR_Integer *l, MR_Integer *h,
%       ML_Ranges *rest);
% Return true if ‘d’ is not the empty set, setting ‘l’ and ‘h’ to the
% lower and upper bound of the first range in ‘d’, and setting ‘rest’
% to ‘d’ with the first range removed.
% Return false if ‘d’ is the empty set.
%
% ML_Ranges ML_ranges_insert(MR_Integer i, ML_ranges r);
% Return the ranges value that is the result of inserting the integer
% ‘i’ into the ranges value ‘r’.

%-----------------------------------------------%
%
% Java interface to ranges.
%

% This section describes the Java interface to the ranges/0 type that is
% exported by this module.
%
% In Java the ranges/0 type is represented by the ranges.Ranges_0 class.
% The following operations are exported as public static methods of the ranges
% module and may be called from Java code.
%
% ranges.Ranges_0 empty();
% Return the empty set.
%
% ranges.Ranges_0 universe();
% Return the set of integers from (min_int+1)..max_int.
%
% ranges.Ranges_0 range(int l, int, h);
% Return the set of integers from ‘l’ to ‘h’ inclusive.
%
% boolean is_empty(ranges.Ranges_0 r);
% Return true iff ‘r’ is the empty set.
%
% int size(ranges.Ranges_0 r);
% Return the number of distinct integers in ‘r’.
%
% boolean split(ranges.Ranges_0 d,
%     jmercury.runtime.Ref<Integer> l,
%     jmercury.runtime.Ref<Integer> h,
%     jmercury.runtime.Ref<ranges.Ranges_0> rest);
% Return true if ‘d’ is not the empty set, setting ‘l’ and ‘h’ to the
% lower and upper bound of the first range in ‘d’, and setting ‘rest’
% to ‘d’ with the first range removed.
```

```
% Return false if ‘d’ is the empty set.
%
% ranges.Ranges_0 insert(int i, ranges.Ranges_0 r);
% Return the ranges value that is the result of inserting the integer
% ‘i’ into the ranges value ‘r’.

%-----------------------------------------------------%
%-----------------------------------------------------%
```

67 rational

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 1997-1998, 2003-2006 The University of Melbourne.
% Copyright (C) 2014-2016, 2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: rational.m.
% Authors: aet Apr 1998. (with plagiarism from rat.m)
% Stability: high.
%
% Implements a rational number type and a set of basic operations on
% rational numbers.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module rational.
:- interface.

:- import_module integer.

%-----------------------------------------------------%

:- type rational.

:- func numer(rational) = integer.

:- func denom(rational) = integer.

:- func zero = rational.

:- func one = rational.
```

```
:- pred '<'(rational::in, rational::in) is semidet.

:- pred '>'(rational::in, rational::in) is semidet.

:- pred '=<'(rational::in, rational::in) is semidet.

:- pred '>='(rational::in, rational::in) is semidet.

:- func rational(int) = rational.

:- func rational(int, int) = rational.

:- func from_integer(integer) = rational.

:- func from_integers(integer, integer) = rational.

% :- func float(rational) = float.

:- func '+'(rational) = rational.

:- func '-'(rational) = rational.

:- func rational + rational = rational.

:- func rational - rational = rational.

:- func rational * rational = rational.

:- func rational / rational = rational.

:- func reciprocal(rational) = rational.

:- func abs(rational) = rational.

%-----------------------------------------------------%
%-----------------------------------------------------%
```

68 rbtree

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 1995-2000, 2003-2007, 2011 The University of Melbourne.
% Copyright (C) 2014-2018 The Mercury team.
```

```
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: rbtree.m.
% Main author: petdr.
% Stability: medium.
%
% Contains an implementation of red black trees.
%
% *** Exit conditions of main predicates ***
% insert:
%   fails if key already in tree.
% update:
%   changes value of key already in tree.  fails if key doesn't exist.
% transform_value:
%   looks up an existing value in the tree, applies a transformation to the
%   value and then updates the value.  fails if the key doesn't exist.
% set:
%   inserts or updates. Never fails.
%
% insert_duplicate:
%   inserts duplicate keys into the tree, never fails.  Search doesn't
%   yet support looking for duplicates.
%
% delete:
%   deletes a node from the tree if it exists.
% remove:
%   fails if node to remove doesn't exist in the tree.
%
% lookup:
%   Throws an exception if key looked up doesn't exist.
% search:
%   Fails if key looked up doesn't exist.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module rbtree.
:- interface.

:- import_module assoc_list.
:- import_module list.

%-----------------------------------------------------%

:- type rbtree(Key, Value).
```

```
    % Initialise the data structure.
    %
:- func init = rbtree(K, V).
:- pred init(rbtree(K, V)::uo) is det.

    % Check whether a tree is empty.
    %
:- pred is_empty(rbtree(K, V)::in) is semidet.

    % Initialise an rbtree containing the given key-value pair.
    %
:- func singleton(K, V) = rbtree(K, V).

    % Inserts a new key-value pair into the tree.
    % Fails if key already in the tree.
    %
:- pred insert(K::in, V::in, rbtree(K, V)::in, rbtree(K, V)::out) is semidet.

    % Updates the value associated with a key.
    % Fails if the key does not exist.
    %
:- pred update(K::in, V::in, rbtree(K, V)::in, rbtree(K, V)::out) is semidet.

    % Update the value at the given key by applying the supplied
    % transformation to it.  Fails if the key is not found.  This is faster
    % than first searching for the value and then updating it.
    %
:- pred transform_value(pred(V, V)::in(pred(in, out) is det), K::in,
    rbtree(K, V)::in, rbtree(K, V)::out) is semidet.

    % Sets a value regardless of whether key exists or not.
    %
:- func set(rbtree(K, V), K, V) = rbtree(K, V).
:- pred set(K::in, V::in, rbtree(K, V)::in, rbtree(K, V)::out) is det.

    % Insert a duplicate key into the tree.
    %
:- func insert_duplicate(rbtree(K, V), K, V) = rbtree(K, V).
:- pred insert_duplicate(K::in, V::in,
    rbtree(K, V)::in, rbtree(K, V)::out) is det.

:- pred member(rbtree(K, V)::in, K::out, V::out) is nondet.

    % Search for a key-value pair using the key.
    % Fails if the key does not exist.
    %
:- pred search(rbtree(K, V)::in, K::in, V::out) is semidet.
```

```
    % Lookup the value associated with a key.
    % Throws an exception if the key does not exist.
    %
:- func lookup(rbtree(K, V), K) = V.
:- pred lookup(rbtree(K, V)::in, K::in, V::out) is det.

    % Search for a key-value pair using the key.  If there is no entry
    % for the given key, returns the pair for the next lower key instead.
    % Fails if there is no key with the given or lower value.
    %
:- pred lower_bound_search(rbtree(K, V)::in, K::in, K::out, V::out)
    is semidet.

    % Search for a key-value pair using the key.  If there is no entry
    % for the given key, returns the pair for the next lower key instead.
    % Throws an exception if there is no key with the given or lower value.
    %
:- pred lower_bound_lookup(rbtree(K, V)::in, K::in, K::out, V::out) is det.

    % Search for a key-value pair using the key.  If there is no entry
    % for the given key, returns the pair for the next higher key instead.
    % Fails if there is no key with the given or higher value.
    %
:- pred upper_bound_search(rbtree(K, V)::in, K::in, K::out, V::out)
    is semidet.

    % Search for a key-value pair using the key.  If there is no entry
    % for the given key, returns the pair for the next higher key instead.
    % Throws an exception if there is no key with the given or higher value.
    %
:- pred upper_bound_lookup(rbtree(K, V)::in, K::in, K::out, V::out) is det.

    % Delete the key-value pair associated with a key.
    % Does nothing if the key does not exist.
    %
:- func delete(rbtree(K, V), K) = rbtree(K, V).
:- pred delete(K::in, rbtree(K, V)::in, rbtree(K, V)::out) is det.

    % Remove the key-value pair associated with a key.
    % Fails if the key does not exist.
    %
:- pred remove(K::in, V::out, rbtree(K, V)::in, rbtree(K, V)::out)
    is semidet.

    % Deletes the node with the minimum key from the tree,
    % and returns the key and value fields.
```

```
    %
:- pred remove_smallest(K::out, V::out,
    rbtree(K, V)::in, rbtree(K, V)::out) is semidet.

    % Deletes the node with the maximum key from the tree,
    % and returns the key and value fields.
    %
:- pred remove_largest(K::out, V::out,
    rbtree(K, V)::in, rbtree(K, V)::out) is semidet.

    % Returns an in-order list of all the keys in the rbtree.
    %
:- func keys(rbtree(K, V)) = list(K).
:- pred keys(rbtree(K, V)::in, list(K)::out) is det.

    % Returns a list of values such that the keys associated with the
    % values are in-order.
    %
:- func values(rbtree(K, V)) = list(V).
:- pred values(rbtree(K, V)::in, list(V)::out) is det.

    % Count the number of elements in the tree.
    %
:- func count(rbtree(K, V)) = int.
:- pred count(rbtree(K, V)::in, int::out) is det.

:- func assoc_list_to_rbtree(assoc_list(K, V)) = rbtree(K, V).
:- pred assoc_list_to_rbtree(assoc_list(K, V)::in, rbtree(K, V)::out) is det.

:- func from_assoc_list(assoc_list(K, V)) = rbtree(K, V).

:- func rbtree_to_assoc_list(rbtree(K, V)) = assoc_list(K, V).
:- pred rbtree_to_assoc_list(rbtree(K, V)::in, assoc_list(K, V)::out)
    is det.

:- func to_assoc_list(rbtree(K, V)) = assoc_list(K, V).

:- func foldl(func(K, V, T) = T, rbtree(K, V), T) = T.
:- pred foldl(pred(K, V, T, T), rbtree(K, V), T, T).
:- mode foldl(pred(in, in, in, out) is det, in, in, out) is det.
:- mode foldl(pred(in, in, mdi, muo) is det, in, mdi, muo) is det.
:- mode foldl(pred(in, in, di, uo) is det, in, di, uo) is det.
:- mode foldl(pred(in, in, in, out) is semidet, in, in, out)
    is semidet.
:- mode foldl(pred(in, in, mdi, muo) is semidet, in, mdi, muo)
    is semidet.
:- mode foldl(pred(in, in, di, uo) is semidet, in, di, uo)
```

```
    is semidet.

:- pred foldl2(pred(K, V, T, T, U, U), rbtree(K, V), T, T, U, U).
:- mode foldl2(pred(in, in, in, out, in, out) is det,
    in, in, out, in, out) is det.
:- mode foldl2(pred(in, in, in, out, mdi, muo) is det,
    in, in, out, mdi, muo) is det.
:- mode foldl2(pred(in, in, in, out, di, uo) is det,
    in, in, out, di, uo) is det.
:- mode foldl2(pred(in, in, di, uo, di, uo) is det,
    in, di, uo, di, uo) is det.
:- mode foldl2(pred(in, in, in, out, in, out) is semidet,
    in, in, out, in, out) is semidet.
:- mode foldl2(pred(in, in, in, out, mdi, muo) is semidet,
    in, in, out, mdi, muo) is semidet.
:- mode foldl2(pred(in, in, in, out, di, uo) is semidet,
    in, in, out, di, uo) is semidet.

:- pred foldl3(pred(K, V, T, T, U, U, W, W), rbtree(K, V),
    T, T, U, U, W, W).
:- mode foldl3(pred(in, in, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out) is det.
:- mode foldl3(pred(in, in, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out) is semidet.
:- mode foldl3(pred(in, in, in, out, in, out, di, uo) is det,
    in, in, out, in, out, di, uo) is det.
:- mode foldl3(pred(in, in, in, out, di, uo, di, uo) is det,
    in, in, out, di, uo, di, uo) is det.
:- mode foldl3(pred(in, in, di, uo, di, uo, di, uo) is det,
    in, di, uo, di, uo, di, uo) is det.

:- pred foldl_values(pred(V, A, A), rbtree(K, V), A, A).
:- mode foldl_values(pred(in, in, out) is det, in, in, out) is det.
:- mode foldl_values(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode foldl_values(pred(in, di, uo) is det, in, di, uo) is det.
:- mode foldl_values(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode foldl_values(pred(in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode foldl_values(pred(in, di, uo) is semidet, in, di, uo) is semidet.

:- pred foldl2_values(pred(V, A, A, B, B), rbtree(K, V), A, A, B, B).
:- mode foldl2_values(pred(in, in, out, in, out) is det, in, in, out,
    in, out) is det.
:- mode foldl2_values(pred(in, in, out, mdi, muo) is det, in, in, out,
    mdi, muo) is det.
:- mode foldl2_values(pred(in, in, out, di, uo) is det, in, in, out,
    di, uo) is det.
:- mode foldl2_values(pred(in, in, out, in, out) is semidet, in, in, out,
```

```
    in, out) is semidet.
:- mode foldl2_values(pred(in, in, out, mdi, muo) is semidet, in, in, out,
    mdi, muo) is semidet.
:- mode foldl2_values(pred(in, in, out, di, uo) is semidet, in, in, out,
    di, uo) is semidet.

:- func map_values(func(K, V) = W, rbtree(K, V)) = rbtree(K, W).
:- pred map_values(pred(K, V, W), rbtree(K, V), rbtree(K, W)).
:- mode map_values(pred(in, in, out) is det, in, out) is det.
:- mode map_values(pred(in, in, out) is semidet, in, out) is semidet.

%---------------------------------------------------%
%---------------------------------------------------%
```

69 require

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 1993-1999, 2003, 2005-2006, 2010-2011 The University of Melbourne.
% Copyright (C) 2014-2016, 2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: require.m.
% Main author: fjh.
% Stability: medium to high.
%
% This module provides features similar to <assert.h> in C.
%
%---------------------------------------------------%
%---------------------------------------------------%

:- module require.
:- interface.

    % error(Message):
    %
    % Throw a `software_error(Message)' exception.
    % This will normally cause execution to abort with an error message.
    %
:- pred error(string::in) is erroneous.

    % func_error(Message):
    %
```

```
    % An expression that results in a `software_error(Message)'
    % exception being thrown.
    %
:- func func_error(string) = _ is erroneous.

    % error(Pred, Message):
    % func_error(Pred, Message):
    %
    % Equivalent to invoking error or func_error on the string
    % Pred ++ ": " ++ Message.
    %
:- pred error(string::in, string::in) is erroneous.
:- func func_error(string, string) = _ is erroneous.

%-----------------------------------------------%

    % sorry(Module, What):
    %
    % Call error/1 with the string
    % "Module: Sorry, not implemented: What".
    %
    % Use this for features that should be implemented (or at least could be
    % implemented).
    %
:- func sorry(string, string) = _ is erroneous.
:- pred sorry(string::in, string::in) is erroneous.

    % sorry(Module, Proc, What):
    %
    % Call error/1 with the string
    % "Module: Proc: Sorry, not implemented: What".
    %
    % Use this for features that should be implemented,
    % or at least could be implemented.
    %
:- func sorry(string, string, string) = _ is erroneous.
:- pred sorry(string::in, string::in, string::in) is erroneous.

    % unexpected(Module, Message):
    %
    % Call error/1 with the string
    % "Module: Unexpected: What".
    %
    % Use this to handle cases which are not expected to arise (i.e. bugs).
    %
:- func unexpected(string, string) = _ is erroneous.
:- pred unexpected(string::in, string::in) is erroneous.
```

```
    % unexpected(Module, Proc, Message):
    %
    % Call error/1 with the string
    % "Module: Proc: Unexpected: What".
    %
    % Use this to handle cases which are not expected to arise (i.e. bugs).
    %
:- func unexpected(string, string, string) = _ is erroneous.
:- pred unexpected(string::in, string::in, string::in) is erroneous.

%-----------------------------------------------%

    % require(Goal, Message):
    %
    % Call goal, and call error(Message) if Goal fails.
    % This is not as useful as you might imagine, since it requires
    % that the goal not produce any output variables. In most circumstances,
    % you should use an explicit if-then-else with a call to error/1,
    % or one of its wrappers, in the "else".
    %
:- pred require((pred)::((pred) is semidet), string::in) is det.

    % expect(Goal, Module, Message):
    %
    % Call Goal, and call unexpected(Module, Message) if Goal fails.
    %
:- pred expect((pred)::((pred) is semidet), string::in, string::in) is det.

    % expect(Goal, Module, Proc, Message):
    %
    % Call Goal, and call unexpected(Module, Proc, Message) if Goal fails.
    %
:- pred expect((pred)::((pred) is semidet), string::in, string::in,
    string::in) is det.

    % expect_not(Goal, Module, Message):
    %
    % Call Goal, and call unexpected(Module, Message) if Goal succeeds.
    %
:- pred expect_not((pred)::((pred) is semidet), string::in, string::in) is det.

    % expect_not(Goal, Module, Proc, Message):
    %
    % Call Goal, and call unexpected(Module, Proc, Message) if Goal succeeds.
    %
:- pred expect_not((pred)::((pred) is semidet), string::in, string::in,
```

```
    string::in) is det.

%-----------------------------------------------------%

    % report_lookup_error(Message, Key):
    %
    % Call error/1 with an error message that is appropriate for
    % the failure of a lookup operation involving the specified Key.
    % The error message will include Message and information about Key.
    %
:- pred report_lookup_error(string::in, K::in) is erroneous.

    % report_lookup_error(Message, Key, Value):
    %
    % Call error/1 with an error message that is appropriate for
    % the failure of a lookup operation involving the specified Key and Value.
    % The error message will include Message and information about Key
    % and Value.
    %
:- pred report_lookup_error(string::in, K::in, V::unused) is erroneous.

%-----------------------------------------------------%
%-----------------------------------------------------%
```

70 rtree

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 2006-2007 The University of Melbourne.
% Copyright (C) 2014-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: rtree.m.
% Main author: gjd.
% Stability: low.
%
% This module provides a region tree (R-tree) ADT. A region tree associates
% values with regions in some space, e.g. rectangles in the 2D plane, or
% bounding spheres in 3D space. Region trees accept spatial queries, e.g. a
% typical usage is "find all pubs within a 2km radius".
%
% This module also provides the typeclass region(K) which allows the user to
% define new regions and spaces. Three "builtin" instances for region(K)
```

```
% are provided: region(interval), region(box) and region(box3d)
% corresponding to "square" regions in one, two and three dimensional spaces
% respectively.
%
%---------------------------------------------------%
%---------------------------------------------------%

:- module rtree.
:- interface.

:- import_module list.

%---------------------------------------------------%

:- type rtree(K, V).

:- typeclass region(K) where [

        % Succeeds iff two regions intersect.
        %
    pred intersects(K::in, K::in) is semidet,

        % Succeeds iff the first region is contained within the second.
        %
    pred contains(K::in, K::in) is semidet,

        % Returns the "size" of a region.
        % e.g. for a two dimensional box one possible measure of "size"
        %      would be the area.
        %
    func size(K) = float,

        % Return a region that contains both input regions.
        % The region returned should be minimal region that contains
        % both input regions.
        %
    func bounding_region(K, K) = K,

        % Computes the size of the bounding region returned by
        % bounding_region/2, i.e.
        %
        % bounding_region_size(K1, K2) = size(bounding_region(K1, K2)).
        %
        % While the above definition would suffice, a more efficient
        % implementation often exists, e.g. for intervals:
        %
        % bounding_region_size(interval(X0, X1), interval(Y0, Y1)) =
```

```
        %       max(X1, Y1) - min(X0, Y0).
        %
        % This version is more efficient since it does not create a
        % temporary interval.
        %
    func bounding_region_size(K, K) = float
].

%---------------------------------------------------%

    % Initialize an empty rtree.
    %
:- func init = (rtree(K, V)::uo) is det <= region(K).

    % Succeeds iff the given rtree is empty.
    %
:- pred is_empty(rtree(K, V)::in) is semidet.

    % Insert a new key and corresponding value into an rtree.
    %
:- func insert(K, V, rtree(K, V)) = rtree(K, V) <= region(K).
:- pred insert(K::in, V::in, rtree(K, V)::in, rtree(K, V)::out)
    is det <= region(K).

    % Delete a key-value pair from an rtree.
    % Assumes that K is either the key for V, or is contained in the key
    % for V.
    %
    % Fails if the key-value pair is not in the tree.
    %
:- pred delete(K::in, V::in, rtree(K, V)::in, rtree(K, V)::out)
    is semidet <= region(K).

    % Search for all values with keys that intersect the query key.
    %
:- func search_intersects(rtree(K, V), K) = list(V) <= region(K).

    % Search for all values with keys that contain the query key.
    %
:- func search_contains(rtree(K, V), K) = list(V) <= region(K).

    % search_general(KTest, VTest, T) = V.
    %
    % Search for all values V with associated keys K that satisfy
    % KTest(K) /\ VTest(V). The search assumes that for all K1, K2
    % such that K1 contains K2, then if KTest(K2) holds we have that
    % KTest(K1) also holds.
```

```
    %
    % We have that:
    %
    %   search_intersects(T, K, Vs)
    %       <=> search_general(intersects(K), true, T, Vs)
    %
    %   search_contains(T, K, Vs)
    %       <=> search_general(contains(K), true, T, Vs)
    %
:- func search_general(pred(K)::in(pred(in) is semidet),
    pred(V)::in(pred(in) is semidet), rtree(K, V)::in) = (list(V)::out)
    is det.

    % search_first(KTest, VTest, Max, T, V, L).
    %
    % Search for a value V with associated key K such that
    % KTest(K, _) /\ VTest(V, L) is satisfied and there does not exist a
    % V' with K' such that KTest(K', _) /\ VTest(V', L') /\ (L' < L) is
    % satisfied. Fail if no such key-value pair exists.
    %
    % The search assumes that for all K1, K2 such that
    % K1 contains K2, then if KTest(K2, L2) holds we have that
    % KTest(K1, L1) holds with L2 >= L1.
    %
    % If there exist multiple key-value pairs that satisfy the above
    % conditions, then one of the candidates is chosen arbitrarily.
    %
:- pred search_first(pred(K, L), pred(V, L), rtree(K, V), L, V, L).
:- mode search_first(pred(in, out) is semidet,
    pred(in, out) is semidet, in, in, out, out) is semidet.

    % search_general_fold(KTest, VPred, T, !A).
    %
    % Apply accumulator VPred to each key-value pair K-V that satisfies
    % KTest(K). The same assumptions for KTest from search_general apply
    % here.
    %
:- pred search_general_fold(pred(K), pred(K, V, A, A), rtree(K, V),
    A, A).
:- mode search_general_fold(pred(in) is semidet,
    pred(in, in, in, out) is det, in, in, out) is det.
:- mode search_general_fold(pred(in) is semidet,
    pred(in, in, di, uo) is det, in, di, uo) is det.

    % Perform a traversal of the rtree, applying an accumulator predicate
    % for each key-value pair.
    %
```

```
:- pred fold(pred(K, V, A, A), rtree(K, V), A, A).
:- mode fold(pred(in, in, in, out) is det, in, in, out) is det.
:- mode fold(pred(in, in, di, uo) is det, in, di, uo) is det.
:- mode fold(pred(in, in, in, out) is semidet, in, in, out)
    is semidet.

    % Apply a transformation predicate to all the values in an rtree.
    %
:- pred map_values(pred(K, V, W), rtree(K, V), rtree(K, W)).
:- mode map_values(pred(in, in, out) is det, in, out) is det.
:- mode map_values(pred(in, in, out) is semidet, in, out)
    is semidet.

%-----------------------------------------------------%
%
% Pre-defined regions.
%

    % An interval type represented as interval(Min, Max).
    %
:- type interval
    --->    interval(float, float).

    % A 2D axis aligned box represented as box(XMin, XMax, YMin, YMax).
    %
:- type box
    --->    box(float, float, float, float).

    % A 3D axis aligned box represented as
    % box(XMin, XMax, YMin, YMax, ZMin, ZMax).
    %
:- type box3d
    --->    box3d(float, float, float, float, float, float).

:- instance region(interval).
:- instance region(box).
:- instance region(box3d).

%-----------------------------------------------------%
%-----------------------------------------------------%
```

71 set

```
%-----------------------------------------------------%
% vim: ts=4 sw=4 et ft=mercury
```

```
%---------------------------------------------------%
% Copyright (C) 1994-1997, 1999-2012 The University of Melbourne.
% Copyright (C) 2014-2016, 2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: set.m.
% Main authors: conway, fjh, benyi.
% Stability: high.
%
% This module provides a set ADT.
% The implementation represents sets using ordered lists.
% This file just calls the equivalent predicates in set_ordlist.
%
%---------------------------------------------------%
%---------------------------------------------------%

:- module set.
:- interface.

:- import_module bool.
:- import_module list.

%---------------------------------------------------%

:- type set(T).

%---------------------------------------------------%
%
% Initial creation of sets.
%

    % `init(Set)' is true iff `Set' is an empty set.
    %
:- func init = set(T).
:- pred init(set(T)::uo) is det.

    % `singleton_set(Elem, Set)' is true iff `Set' is the set
    % containing just the single element `Elem'.
    %
:- pred singleton_set(T, set(T)).
:- mode singleton_set(in, out) is det.
:- mode singleton_set(out, in) is semidet.

:- func make_singleton_set(T) = set(T).

%---------------------------------------------------%
```

```
%
% Emptiness and singleton-ness tests.
%

    % 'empty(Set)' is true iff 'Set' is an empty set.
    % 'is_empty' is a synonym for 'empty'.
    %
:- pred empty(set(T)::in) is semidet.
:- pred is_empty(set(T)::in) is semidet.
:- pragma obsolete(empty/1, [is_empty/1]).

    % 'non_empty(Set)' is true iff 'Set' is not an empty set.
    % 'is_non_empty' is a synonym for 'non_empty'.
    %
:- pred non_empty(set(T)::in) is semidet.
:- pred is_non_empty(set(T)::in) is semidet.
:- pragma obsolete(non_empty/1, [is_non_empty/1]).

:- pred is_singleton(set(T)::in, T::out) is semidet.

%-----------------------------------------------%
%
% Membership tests.
%

    % 'member(X, Set)' is true iff 'X' is a member of 'Set'.
    %
:- pred member(T, set(T)).
:- mode member(in, in) is semidet.
:- mode member(out, in) is nondet.

    % 'is_member(X, Set, Result)' returns 'Result = yes'
    % iff 'X' is a member of 'Set'.
    %
:- pred is_member(T::in, set(T)::in, bool::out) is det.

    % 'contains(Set, X)' is true iff 'X' is a member of 'Set'.
    %
:- pred contains(set(T)::in, T::in) is semidet.

%-----------------------------------------------%
%
% Insertions and deletions.
%

    % 'insert(X, Set0, Set)' is true iff 'Set' is the union of
    % 'Set0' and the set containing only 'X'.
```

```
    %
:- func insert(set(T), T) = set(T).
:- pred insert(T::in, set(T)::in, set(T)::out) is det.

    % 'insert_new(X, Set0, Set)' is true iff 'Set0' does not contain
    % 'X', and 'Set' is the union of 'Set0' and the set containing only 'X'.
    %
:- pred insert_new(T::in, set(T)::in, set(T)::out) is semidet.

    % 'insert_list(Xs, Set0, Set)' is true iff 'Set' is the union of
    % 'Set0' and the set containing only the members of 'Xs'.
    %
:- func insert_list(set(T), list(T)) = set(T).
:- pred insert_list(list(T)::in, set(T)::in, set(T)::out) is det.

    % 'delete(X, Set0, Set)' is true iff 'Set' is the relative
    % complement of 'Set0' and the set containing only 'X', i.e.
    % if 'Set' is the set which contains all the elements of 'Set0'
    % except 'X'.
    %
:- func delete(set(T), T) = set(T).
:- pred delete(T::in, set(T)::in, set(T)::out) is det.

    % 'delete_list(Set0, Xs, Set)' is true iff 'Set' is the relative
    % complement of 'Set0' and the set containing only the members of
    % 'Xs'.
    %
:- func delete_list(set(T), list(T)) = set(T).
:- pred delete_list(list(T)::in, set(T)::in, set(T)::out) is det.

    % 'remove(X, Set0, Set)' is true iff 'Set0' contains 'X',
    % and 'Set' is the relative complement of 'Set0' and the set
    % containing only 'X', i.e.  if 'Set' is the set which contains
    % all the elements of 'Set0' except 'X'.
    %
    % The det_remove version throws an exception instead of failing.
    %
:- pred remove(T::in, set(T)::in, set(T)::out) is semidet.
:- pred det_remove(T::in, set(T)::in, set(T)::out) is det.

    % 'remove_list(Xs, Set0, Set)' is true iff 'Xs' does not
    % contain any duplicates, 'Set0' contains every member of 'Xs',
    % and 'Set' is the relative complement of 'Set0' and the set
    % containing only the members of 'Xs'.
    %
    % The det_remove_list version throws an exception instead of failing.
    %
```

```
:- pred remove_list(list(T)::in, set(T)::in, set(T)::out) is semidet.
:- pred det_remove_list(list(T)::in, set(T)::in, set(T)::out) is det.

    % 'remove_least(Elem, Set0, Set)' is true iff
    % 'Set0' is not empty, 'Elem' is the smallest element in 'Set0'
    % (with elements ordered using the standard ordering given
    % by compare/3), and 'Set' is the set containing all the
    % elements of 'Set0' except 'Elem'.
    %
:- pred remove_least(T::out, set(T)::in, set(T)::out) is semidet.

%-----------------------------------------------------%
%
% Comparisons between sets.
%

    % 'equal(SetA, SetB)' is true iff
    % 'SetA' and 'SetB' contain the same elements.
    %
:- pred equal(set(T)::in, set(T)::in) is semidet.

    % 'subset(SetA, SetB)' is true iff 'SetA' is a subset of 'SetB'.
    %
:- pred subset(set(T)::in, set(T)::in) is semidet.

    % 'superset(SetA, SetB)' is true iff 'SetA' is a
    % superset of 'SetB'.
    %
:- pred superset(set(T)::in, set(T)::in) is semidet.

%-----------------------------------------------------%
%
% Operations on two or more sets.
%

    % 'union(SetA, SetB, Set)' is true iff 'Set' is the union of
    % 'SetA' and 'SetB'.  If the sets are known to be of different
    % sizes, then for efficiency make 'SetA' the larger of the two.
    % (The current implementation using sorted lists with duplicates
    % removed is not sensitive to the ordering of the input arguments,
    % but other set implementations may be, so observing this convention
    % will make it less likely that you will encounter problems if
    % the implementation is changed.)
    %
:- func union(set(T), set(T)) = set(T).
:- pred union(set(T)::in, set(T)::in, set(T)::out) is det.
```

```
    % 'union_list(A, B)' is true iff 'B' is the union of
    % all the sets in 'A'.
    %
:- func union_list(list(set(T))) = set(T).

    % 'power_union(A, B)' is true iff 'B' is the union of
    % all the sets in 'A'.
    %
:- func power_union(set(set(T))) = set(T).
:- pred power_union(set(set(T))::in, set(T)::out) is det.

    % 'intersect(SetA, SetB, Set)' is true iff 'Set' is the
    % intersection of 'SetA' and 'SetB'. If the two sets are
    % known to be unequal in size, then making SetA be the larger
    % set will usually be more efficient.
    % (The current implementation, using sorted lists with duplicates
    % removed is not sensitive to the ordering of the input arguments
    % but other set implementations may be, so observing this convention
    % will make it less likely that you will encounter problems if
    % the implementation is changed.)
    %
:- func intersect(set(T), set(T)) = set(T).
:- pred intersect(set(T)::in, set(T)::in, set(T)::out) is det.

    % 'intersect_list(A, B)' is true iff 'B' is the intersection of
    % all the sets in 'A'.
    %
:- func intersect_list(list(set(T))) = set(T).

    % 'power_intersect(A, B)' is true iff 'B' is the intersection of
    % all the sets in 'A'.
    %
:- func power_intersect(set(set(T))) = set(T).
:- pred power_intersect(set(set(T))::in, set(T)::out) is det.

    % 'difference(SetA, SetB, Set)' is true iff 'Set' is the
    % set containing all the elements of 'SetA' except those that
    % occur in 'SetB'.
    %
:- func difference(set(T), set(T)) = set(T).
:- pred difference(set(T)::in, set(T)::in, set(T)::out) is det.

    % intersection_and_differences(SetA, SetB, InAandB, OnlyInA, OnlyInB):
    % Given SetA and SetB, return the elements that occur in both sets,
    % and those that occur only in one or the other.
    %
:- pred intersection_and_differences(set(T)::in, set(T)::in,
```

```
    set(T)::out, set(T)::out, set(T)::out) is det.

%---------------------------------------------------%
%
% Operations that divide a set into two parts.
%

    % divide(Pred, Set, TruePart, FalsePart):
    % TruePart consists of those elements of Set for which Pred succeeds;
    % FalsePart consists of those elements of Set for which Pred fails.
    %
:- pred divide(pred(T)::in(pred(in) is semidet), set(T)::in,
    set(T)::out, set(T)::out) is det.

    % divide_by_set(DivideBySet, Set, InPart, OutPart):
    % InPart consists of those elements of Set which are also in DivideBySet;
    % OutPart consists of those elements of which are not in DivideBySet.
    %
:- pred divide_by_set(set(T)::in, set(T)::in, set(T)::out, set(T)::out) is det.

%---------------------------------------------------%
%
% Converting lists to sets.
%

    % 'list_to_set(List, Set)' is true iff 'Set' is the set
    % containing only the members of 'List'.
    %
:- func list_to_set(list(T)) = set(T).
:- pred list_to_set(list(T)::in, set(T)::out) is det.

    % Synonyms for list_to_set/1.
    %
:- func from_list(list(T)) = set(T).
:- func set(list(T)) = set(T).
:- pragma obsolete(set/1, [list_to_set/1]).

    % 'sorted_list_to_set(List, Set)' is true iff 'Set' is the set
    % containing only the members of 'List'.  'List' must be sorted
    % and must not contain any duplicates.
    %
:- func sorted_list_to_set(list(T)) = set(T).
:- pred sorted_list_to_set(list(T)::in, set(T)::out) is det.

    % 'rev_sorted_list_to_set(List) = Set' is true iff 'Set' is the set
    % containing only the members of 'List'. 'List' must be sorted
    % in descending order and must not contain any duplicates.
```

```
    %
:- func rev_sorted_list_to_set(list(T)) = set(T).
:- pred rev_sorted_list_to_set(list(T)::in, set(T)::out) is det.

    % A synonym for sorted_list_to_set/1.
    %
:- func from_sorted_list(list(T)) = set(T).

%---------------------------------------------------%
%
% Converting sets to lists.
%

    % `to_sorted_list(Set, List)' is true iff `List' is the list
    % of all the members of `Set', in sorted order without any
    % duplicates.
    %
:- func to_sorted_list(set(T)) = list(T).
:- pred to_sorted_list(set(T)::in, list(T)::out) is det.

%---------------------------------------------------%
%
% Counting.
%

    % `count(Set, Count)' is true iff `Set' has `Count' elements.
    % i.e. `Count' is the cardinality (size) of the
    %
:- func count(set(T)) = int.
:- pred count(set(T)::in, int::out) is det.

%---------------------------------------------------%
%
% Standard higher order functions on collections.
%

    % all_true(Pred, Set) succeeds iff Pred(Element) succeeds
    % for all the elements of Set.
    %
:- pred all_true(pred(T)::in(pred(in) is semidet), set(T)::in) is semidet.

    % Return the set of items for which the given predicate succeeds.
    % filter(P, S) =
    %   sorted_list_to_set(list.filter(P, to_sorted_list(S))).
    %
:- func filter(pred(T1), set(T1)) = set(T1).
:- mode filter(pred(in) is semidet, in) = out is det.
```

```
:- pred filter(pred(T1), set(T1), set(T1)).
:- mode filter(pred(in) is semidet, in, out) is det.

    % Return the set of items for which the given predicate succeeds,
    % and the set of items for which it fails.
    %
:- pred filter(pred(T1), set(T1), set(T1), set(T1)).
:- mode filter(pred(in) is semidet, in, out, out) is det.

    % filter_map(PF, S) =
    %   list_to_set(list.filter_map(PF, to_sorted_list(S))).
    %
:- func filter_map(func(T1) = T2, set(T1)) = set(T2).
:- mode filter_map(func(in) = out is semidet, in) = out is det.
:- pred filter_map(pred(T1, T2), set(T1), set(T2)).
:- mode filter_map(in(pred(in, out) is semidet), in, out) is det.

    % map(F, S) =
    %   list_to_set(list.map(F, to_sorted_list(S))).
    %
:- func map(func(T1) = T2, set(T1)) = set(T2).
:- pred map(pred(T1, T2), set(T1), set(T2)).
:- mode map(pred(in, out) is det, in, out) is det.
:- mode map(pred(in, out) is cc_multi, in, out) is cc_multi.
:- mode map(pred(in, out) is semidet, in, out) is semidet.
:- mode map(pred(in, out) is multi, in, out) is multi.
:- mode map(pred(in, out) is nondet, in, out) is nondet.

    % map_fold(P, S0, S, A0, A) :-
    %   L0 = to_sorted_list(S0),
    %   list.map_foldl(P, L0, L, A0, A),
    %   S = list_to_set(L).
    %
:- pred map_fold(pred(T1, T2, T3, T3), set(T1), set(T2), T3, T3).
:- mode map_fold(pred(in, out, in, out) is det, in, out, in, out) is det.
:- mode map_fold(pred(in, out, mdi, muo) is det, in, out, mdi, muo) is det.
:- mode map_fold(pred(in, out, di, uo) is det, in, out, di, uo) is det.
:- mode map_fold(pred(in, out, in, out) is semidet, in, out,
    in, out) is semidet.
:- mode map_fold(pred(in, out, mdi, muo) is semidet, in, out,
    mdi, muo) is semidet.
:- mode map_fold(pred(in, out, di, uo) is semidet, in, out,
    di, uo) is semidet.

    % fold(F, S, A) =
    %   list.foldl(F, to_sorted_list(S), A).
    %
```

```
:- func fold(func(T, A) = A, set(T), A) = A.
:- func foldl(func(T, A) = A, set(T), A) = A.

:- pred fold(pred(T, A, A), set(T), A, A).
:- mode fold(pred(in, in, out) is det, in, in, out) is det.
:- mode fold(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode fold(pred(in, di, uo) is det, in, di, uo) is det.
:- mode fold(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode fold(pred(in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode fold(pred(in, di, uo) is semidet, in, di, uo) is semidet.

:- pred foldl(pred(T, A, A), set(T), A, A).
:- mode foldl(pred(in, in, out) is det, in, in, out) is det.
:- mode foldl(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode foldl(pred(in, di, uo) is det, in, di, uo) is det.
:- mode foldl(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode foldl(pred(in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode foldl(pred(in, di, uo) is semidet, in, di, uo) is semidet.

:- pred fold2(pred(T, A, A, B, B), set(T), A, A, B, B).
:- mode fold2(pred(in, in, out, in, out) is det, in,
    in, out, in, out) is det.
:- mode fold2(pred(in, in, out, mdi, muo) is det, in,
    in, out, mdi, muo) is det.
:- mode fold2(pred(in, in, out, di, uo) is det, in,
    in, out, di, uo) is det.
:- mode fold2(pred(in, in, out, in, out) is semidet,
    in, in, out, in, out) is semidet.
:- mode fold2(pred(in, in, out, mdi, muo) is semidet,
    in, in, out, mdi, muo) is semidet.
:- mode fold2(pred(in, in, out, di, uo) is semidet,
    in, in, out, di, uo) is semidet.

:- pred foldl2(pred(T, A, A, B, B), set(T), A, A, B, B).
:- mode foldl2(pred(in, in, out, in, out) is det, in,
    in, out, in, out) is det.
:- mode foldl2(pred(in, in, out, mdi, muo) is det, in,
    in, out, mdi, muo) is det.
:- mode foldl2(pred(in, in, out, di, uo) is det, in,
    in, out, di, uo) is det.
:- mode foldl2(pred(in, in, out, in, out) is semidet,
    in, in, out, in, out) is semidet.
:- mode foldl2(pred(in, in, out, mdi, muo) is semidet,
    in, in, out, mdi, muo) is semidet.
:- mode foldl2(pred(in, in, out, di, uo) is semidet,
    in, in, out, di, uo) is semidet.
```

```
:- pred fold3(pred(T, A, A, B, B, C, C), set(T), A, A, B, B, C, C).
:- mode fold3(pred(in, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out) is det.
:- mode fold3(pred(in, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, mdi, muo) is det.
:- mode fold3(pred(in, in, out, in, out, di, uo) is det, in,
    in, out, in, out, di, uo) is det.
:- mode fold3(pred(in, in, out, in, out, in, out) is semidet, in,
    in, out, in, out, in, out) is semidet.
:- mode fold3(pred(in, in, out, in, out, mdi, muo) is semidet, in,
    in, out, in, out, mdi, muo) is semidet.
:- mode fold3(pred(in, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, di, uo) is semidet.

:- pred foldl3(pred(T, A, A, B, B, C, C), set(T), A, A, B, B, C, C).
:- mode foldl3(pred(in, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out) is det.
:- mode foldl3(pred(in, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, mdi, muo) is det.
:- mode foldl3(pred(in, in, out, in, out, di, uo) is det, in,
    in, out, in, out, di, uo) is det.
:- mode foldl3(pred(in, in, out, in, out, in, out) is semidet, in,
    in, out, in, out, in, out) is semidet.
:- mode foldl3(pred(in, in, out, in, out, mdi, muo) is semidet, in,
    in, out, in, out, mdi, muo) is semidet.
:- mode foldl3(pred(in, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, di, uo) is semidet.

:- pred fold4(pred(T, A, A, B, B, C, C, D, D), set(T), A, A, B, B,
    C, C, D, D).
:- mode fold4(pred(in, in, out, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out, in, out) is det.
:- mode fold4(pred(in, in, out, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, in, out, mdi, muo) is det.
:- mode fold4(pred(in, in, out, in, out, in, out, di, uo) is det, in,
    in, out, in, out, in, out, di, uo) is det.
:- mode fold4(pred(in, in, out, in, out, in, out, in, out) is semidet, in,
    in, out, in, out, in, out, in, out) is semidet.
:- mode fold4(pred(in, in, out, in, out, in, out, mdi, muo) is semidet, in,
    in, out, in, out, in, out, mdi, muo) is semidet.
:- mode fold4(pred(in, in, out, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, in, out, di, uo) is semidet.

:- pred foldl4(pred(T, A, A, B, B, C, C, D, D), set(T), A, A, B, B,
    C, C, D, D).
:- mode foldl4(pred(in, in, out, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out, in, out) is det.
```

```
:- mode foldl4(pred(in, in, out, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, in, out, mdi, muo) is det.
:- mode foldl4(pred(in, in, out, in, out, in, out, di, uo) is det, in,
    in, out, in, out, in, out, di, uo) is det.
:- mode foldl4(pred(in, in, out, in, out, in, out, in, out) is semidet, in,
    in, out, in, out, in, out, in, out) is semidet.
:- mode foldl4(pred(in, in, out, in, out, in, out, mdi, muo) is semidet, in,
    in, out, in, out, in, out, mdi, muo) is semidet.
:- mode foldl4(pred(in, in, out, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, in, out, di, uo) is semidet.

:- pred fold5(pred(T, A, A, B, B, C, C, D, D, E, E), set(T), A, A, B, B,
    C, C, D, D, E, E).
:- mode fold5(
    pred(in, in, out, in, out, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out, in, out, in, out) is det.
:- mode fold5(
    pred(in, in, out, in, out, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, in, out, in, out, mdi, muo) is det.
:- mode fold5(
    pred(in, in, out, in, out, in, out, in, out, di, uo) is det,
    in, in, out, in, out, in, out, in, out, di, uo) is det.
:- mode fold5(
    pred(in, in, out, in, out, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out, in, out, in, out) is semidet.
:- mode fold5(
    pred(in, in, out, in, out, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode fold5(
    pred(in, in, out, in, out, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, in, out, in, out, di, uo) is semidet.

:- pred foldl5(pred(T, A, A, B, B, C, C, D, D, E, E), set(T), A, A, B, B,
    C, C, D, D, E, E).
:- mode foldl5(
    pred(in, in, out, in, out, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out, in, out, in, out) is det.
:- mode foldl5(
    pred(in, in, out, in, out, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, in, out, in, out, mdi, muo) is det.
:- mode foldl5(
    pred(in, in, out, in, out, in, out, in, out, di, uo) is det,
    in, in, out, in, out, in, out, in, out, di, uo) is det.
:- mode foldl5(
    pred(in, in, out, in, out, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out, in, out, in, out) is semidet.
:- mode foldl5(
```

```
    pred(in, in, out, in, out, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode foldl5(
    pred(in, in, out, in, out, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, in, out, in, out, di, uo) is semidet.

:- pred fold6(pred(T, A, A, B, B, C, C, D, D, E, E, F, F), set(T),
    A, A, B, B, C, C, D, D, E, E, F, F).
:- mode fold6(
    pred(in, in, out, in, out, in, out, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out, in, out, in, out, in, out) is det.
:- mode fold6(
    pred(in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is det.
:- mode fold6(
    pred(in, in, out, in, out, in, out, in, out, in, out, di, uo) is det,
    in, in, out, in, out, in, out, in, out, in, out, di, uo) is det.
:- mode fold6(
    pred(in, in, out, in, out, in, out, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out, in, out, in, out, in, out) is semidet.
:- mode fold6(
    pred(in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode fold6(
    pred(in, in, out, in, out, in, out, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, in, out, in, out, in, out, di, uo) is semidet.

:- pred foldl6(pred(T, A, A, B, B, C, C, D, D, E, E, F, F), set(T),
    A, A, B, B, C, C, D, D, E, E, F, F).
:- mode foldl6(
    pred(in, in, out, in, out, in, out, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out, in, out, in, out, in, out) is det.
:- mode foldl6(
    pred(in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is det.
:- mode foldl6(
    pred(in, in, out, in, out, in, out, in, out, in, out, di, uo) is det,
    in, in, out, in, out, in, out, in, out, in, out, di, uo) is det.
:- mode foldl6(
    pred(in, in, out, in, out, in, out, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out, in, out, in, out, in, out) is semidet.
:- mode foldl6(
    pred(in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode foldl6(
    pred(in, in, out, in, out, in, out, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, in, out, in, out, in, out, di, uo) is semidet.
```

```
%---------------------------------------------------%
%---------------------------------------------------%
```

72 set_bbbtree

```
%---------------------------------------------------%
% vim: ts=4 sw=4 et ft=mercury
%---------------------------------------------------%
% Copyright (C) 1995-1997, 1999-2006, 2010-2012 The University of Melbourne.
% Copyright (C) 2014-2015, 2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: set_bbbtree.m.
% Main authors: benyi.
% Stability: low.
%
% This module implements sets using bounded balanced binary trees.
%
%---------------------------------------------------%
%---------------------------------------------------%

:- module set_bbbtree.
:- interface.

:- import_module bool.
:- import_module list.

%---------------------------------------------------%

:- type set_bbbtree(T).

%---------------------------------------------------%
%
% Initial creation of sets.
%

    % `init(Set)' returns an initialized empty set.
    %
:- func init = set_bbbtree(T).
:- pred init(set_bbbtree(T)::uo) is det.

    % `singleton_set(X, Set)' is true iff `Set' is the set
    % containing just the single element `X'.
```

```
    %
:- pred singleton_set(T, set_bbbtree(T)).
:- mode singleton_set(in, out) is det.
:- mode singleton_set(in, in) is semidet.
:- mode singleton_set(out, in) is semidet.

:- func make_singleton_set(T) = set_bbbtree(T).

%-----------------------------------------------------%
%
% Emptiness and singleton-ness tests.
%

    % 'empty(Set)' is true iff 'Set' is an empty set.
    % 'is_empty' is a synonym for 'empty'.
    %
:- pred empty(set_bbbtree(T)::in) is semidet.
:- pred is_empty(set_bbbtree(T)::in) is semidet.
:- pragma obsolete(empty/1, [is_empty/1]).

    % 'non_empty(Set)' is true iff 'Set' is not an empty set.
    % 'is_non_empty' is a synonym for 'non_empty'.
    %
:- pred non_empty(set_bbbtree(T)::in) is semidet.
:- pred is_non_empty(set_bbbtree(T)::in) is semidet.
:- pragma obsolete(non_empty/1, [is_non_empty/1]).

:- pred is_singleton(set_bbbtree(T)::in, T::out) is semidet.

%-----------------------------------------------------%
%
% Membership tests.
%

    % 'member(X, Set)' is true iff 'X' is a member of 'Set'.
    % O(lg n) for (in, in) and O(1) for (out, in).
    %
:- pred member(T, set_bbbtree(T)).
:- mode member(in, in) is semidet.
:- mode member(out, in) is nondet.

    % 'is_member(X, Set, Result)' is true iff 'X' is a member
    % of 'Set'.
    %
:- pred is_member(T::in, set_bbbtree(T)::in, bool::out) is det.

    % 'contains(Set, X)' is true iff 'X' is a member of 'Set'.
```

```
    % O(lg n).
    %
:- pred contains(set_bbbtree(T)::in, T::in) is semidet.

    % 'least(Set, X)' is true iff 'X' is smaller than all
    % the other members of 'Set'.
    %
:- pred least(set_bbbtree(T), T).
:- mode least(in, out) is semidet.
:- mode least(in, in) is semidet.

    % 'largest(Set, X)' is true iff 'X' is larger than all
    % the other members of 'Set'.
    %
:- pred largest(set_bbbtree(T), T).
:- mode largest(in, out) is semidet.
:- mode largest(in, in) is semidet.

%-----------------------------------------------%
%
% Insertions and deletions.
%

    % 'insert(X, Set0, Set)' is true iff 'Set' is the union of
    % 'Set0' and the set containing only 'X'.
    %
:- func insert(set_bbbtree(T), T) = set_bbbtree(T).
:- pred insert(T, set_bbbtree(T), set_bbbtree(T)).
:- mode insert(di, di, uo) is det.
:- mode insert(in, in, out) is det.

    % 'insert_new(X, Set0, Set)' is true iff 'Set0' does not
    % contain 'X', and 'Set' is the union of 'Set0' and the set containing
    % only 'X'.
    %
:- pred insert_new(T::in,
    set_bbbtree(T)::in, set_bbbtree(T)::out) is semidet.

    % 'insert_list(Xs, Set0, Set)' is true iff 'Set' is
    % the union of 'Set0' and the set containing only the members of 'Xs'.
    %
:- func insert_list(set_bbbtree(T), list(T)) = set_bbbtree(T).
:- pred insert_list(list(T)::in,
    set_bbbtree(T)::in, set_bbbtree(T)::out) is det.

    % 'delete(X, Set0, Set)' is true iff 'Set' is the relative
    % complement of 'Set0' and the set containing only 'X', i.e.
```

```
    % if ‘Set’ is the set which contains all the elements of ‘Set0’
    % except ‘X’.
    %
:- func delete(set_bbbtree(T), T) = set_bbbtree(T).
:- pred delete(T, set_bbbtree(T), set_bbbtree(T)).
:- mode delete(in, di, uo) is det.
:- mode delete(in, in, out) is det.

    % ‘delete_list(Xs, Set0, Set)’ is true iff ‘Set’ is the
    % relative complement of ‘Set0’ and the set containing only the members
    % of ‘Xs’.
    %
:- func delete_list(set_bbbtree(T), list(T)) = set_bbbtree(T).
:- pred delete_list(list(T)::in,
    set_bbbtree(T)::in, set_bbbtree(T)::out) is det.

    % ‘remove(X, Set0, Set)’ is true iff ‘Set0’ contains ‘X’,
    % and ‘Set’ is the relative complement of ‘Set0’ and the set
    % containing only ‘X’, i.e.  if ‘Set’ is the set which contains
    % all the elements of ‘Set0’ except ‘X’.
    %
    % The det_remove version throws an exception instead of failing.
    %
:- pred remove(T::in, set_bbbtree(T)::in, set_bbbtree(T)::out) is semidet.
:- pred det_remove(T::in, set_bbbtree(T)::in, set_bbbtree(T)::out) is det.

    % ‘remove_list(Xs, Set0, Set)’ is true iff Xs does not
    % contain any duplicates, ‘Set0’ contains every member of ‘Xs’,
    % and ‘Set’ is the relative complement of ‘Set0’ and the set
    % containing only the members of ‘Xs’.
    %
    % The det_remove_list version throws an exception instead of failing.
    %
:- pred remove_list(list(T)::in,
    set_bbbtree(T)::in, set_bbbtree(T)::out) is semidet.
:- pred det_remove_list(list(T)::in,
    set_bbbtree(T)::in, set_bbbtree(T)::out) is det.

    % ‘remove_least(X, Set0, Set)’ is true iff the union if
    % ‘X’ and ‘Set’ is ‘Set0’ and ‘X’ is smaller than all the elements of
    % ‘Set’.
    %
:- pred remove_least(T::out,
    set_bbbtree(T)::in, set_bbbtree(T)::out) is semidet.

    % ‘remove_largest(X, Set0, Set)’ is true iff the union if
    % ‘X’ and ‘Set’ is ‘Set0’ and ‘X’ is larger than all the elements of
```

```
    % 'Set'.
    %
:- pred remove_largest(T::out,
    set_bbbtree(T)::in, set_bbbtree(T)::out) is semidet.

%-----------------------------------------------------%
%
% Comparisons between sets.
%

    % 'equal(SetA, SetB)' is true iff 'SetA' and 'SetB'
    % contain the same elements.
    %
:- pred equal(set_bbbtree(T)::in, set_bbbtree(T)::in) is semidet.

    % 'subset(SetA, SetB)' is true iff all the elements of
    % 'SetA' are also elements of 'SetB'.
    %
:- pred subset(set_bbbtree(T)::in, set_bbbtree(T)::in) is semidet.

    % 'superset(SetA, SetB)' is true iff all the elements of
    % 'SetB' are also elements of 'SetA'.
    %
:- pred superset(set_bbbtree(T)::in, set_bbbtree(T)::in) is semidet.

%-----------------------------------------------------%
%
% Operations on two or more sets.
%

    % 'union(SetA, SetB, Set)' is true iff 'Set' is the union
    % of 'SetA' and 'SetB'.
    %
:- func union(set_bbbtree(T), set_bbbtree(T)) = set_bbbtree(T).
:- pred union(set_bbbtree(T)::in, set_bbbtree(T)::in,
    set_bbbtree(T)::out) is det.

    % 'union_list(Sets) = Set' is true iff 'Set' is the union
    % of all the sets in 'Sets'
    %
:- func union_list(list(set_bbbtree(T))) = set_bbbtree(T).

    % 'power_union(Sets, Set)' is true iff 'Set' is the union
    % of all the sets in 'Sets'
    %
:- func power_union(set_bbbtree(set_bbbtree(T))) = set_bbbtree(T).
:- pred power_union(set_bbbtree(set_bbbtree(T))::in,
```

```
    set_bbbtree(T)::out) is det.

    % 'intersect(SetA, SetB, Set)' is true iff 'Set' is the
    % intersection of 'SetA' and 'SetB'.
    %
:- func intersect(set_bbbtree(T), set_bbbtree(T)) = set_bbbtree(T).
:- pred intersect(set_bbbtree(T)::in, set_bbbtree(T)::in,
    set_bbbtree(T)::out) is det.

    % 'intersect_list(Sets) = Set is true iff 'Set' is the
    % intersection of the sets in 'Sets'.
    %
:- func intersect_list(list(set_bbbtree(T))) = set_bbbtree(T).

    % 'power_intersect(Sets, Set) is true iff 'Set' is the
    % intersection of the sets in 'Sets'.
    %
:- func power_intersect(set_bbbtree(set_bbbtree(T))) = set_bbbtree(T).
:- pred power_intersect(set_bbbtree(set_bbbtree(T))::in, set_bbbtree(T)::out)
    is det.

    % 'set_bbtree.difference(SetA, SetB, Set)' is true iff 'Set' is the
    %  set containing all the elements of 'SetA' except those that
    % occur in 'SetB'.
    %
:- func difference(set_bbbtree(T), set_bbbtree(T)) = set_bbbtree(T).
:- pred difference(set_bbbtree(T)::in, set_bbbtree(T)::in,
    set_bbbtree(T)::out) is det.

%-----------------------------------------------------%
%
% Converting lists to sets.
%

    % 'list_to_set(List, Set)' is true iff 'Set' is the set
    % containing only the members of 'List'. O(n lg n)
    %
:- func list_to_set(list(T)) = set_bbbtree(T).
:- pred list_to_set(list(T)::in, set_bbbtree(T)::out) is det.

    % A synonym for set_bbtree.list_to_set/1.
    %
:- func from_list(list(T)) = set_bbbtree(T).

    % 'sorted_list_to_set(List, Set)' is true iff 'Set' is the
    % set containing only the members of 'List'.
    % 'List' must be sorted in ascending order, and must not contain
```

```
    % any duplicates. O(n).
    %
    % The sorted_list_to_set_len version allows the caller to provide
    % the length of List, which avoids the cost of computing it again.
    % This version will throw an exception if the length is incorrect.
    %
:- func sorted_list_to_set(list(T)) = set_bbbtree(T).
:- pred sorted_list_to_set(list(T)::in, set_bbbtree(T)::out) is det.
:- pred sorted_list_to_set_len(list(T)::in, set_bbbtree(T)::out,
    int::in) is det.

    % 'rev_sorted_list_to_set(List) = Set' is true iff 'Set' is the set
    % containing only the members of 'List'. 'List' must be sorted
    % in descending order.
    %
:- func rev_sorted_list_to_set(list(T)) = set_bbbtree(T).
:- pred rev_sorted_list_to_set(list(T)::in, set_bbbtree(T)::out) is det.
:- pred rev_sorted_list_to_set_len(list(T)::in, set_bbbtree(T)::out,
    int::in) is det.

    % A synonym for sorted_list_to_set/1.
    %
:- func from_sorted_list(list(T)) = set_bbbtree(T).

%---------------------------------------------------%
%
% Converting sets to lists.
%

    % 'to_sorted_list(Set, List)' is true iff 'List' is the
    % list of all the members of 'Set', in sorted order. O(n).
    %
:- func to_sorted_list(set_bbbtree(T)) = list(T).
:- pred to_sorted_list(set_bbbtree(T), list(T)).
:- mode to_sorted_list(di, uo) is det.
:- mode to_sorted_list(in, out) is det.

%---------------------------------------------------%
%
% Counting.
%

    % 'count(Set, Count)' is true iff 'Set' has 'Count' elements.
    % i.e. 'Count' is the cardinality (size) of the set.
    %
:- func count(set_bbbtree(T)) = int.
:- pred count(set_bbbtree(T)::in, int::out) is det.
```

```
%---------------------------------------------------%
%
% Standard higher order functions on collections.
%

    % all_true(Pred, Set) succeeds iff Pred(Element) succeeds
    % for all the elements of Set.
    %
:- pred all_true(pred(T)::in(pred(in) is semidet), set_bbbtree(T)::in)
    is semidet.

    % filter(Pred, Items, Trues):
    % Return the set of items for which Pred succeeds.
    %
:- pred filter(pred(T)::in(pred(in) is semidet),
    set_bbbtree(T)::in, set_bbbtree(T)::out) is det.

    % filter(Pred, Items, Trues, Falses):
    % Return the set of items for which Pred succeeds,
    % and the set for which it fails.
    %
:- pred filter(pred(T)::in(pred(in) is semidet),
    set_bbbtree(T)::in, set_bbbtree(T)::out, set_bbbtree(T)::out) is det.

:- func filter_map(func(T1) = T2, set_bbbtree(T1)) = set_bbbtree(T2).
:- mode filter_map(func(in) = out is semidet, in) = out is det.

:- func map(func(T1) = T2, set_bbbtree(T1)) = set_bbbtree(T2).

:- func fold(func(T1, T2) = T2, set_bbbtree(T1), T2) = T2.
:- pred fold(pred(T1, T2, T2), set_bbbtree(T1), T2, T2).
:- mode fold(pred(in, in, out) is det, in, in, out) is det.
:- mode fold(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode fold(pred(in, di, uo) is det, in, di, uo) is det.
:- mode fold(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode fold(pred(in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode fold(pred(in, di, uo) is semidet, in, di, uo) is semidet.

:- pred fold2(pred(T1, T2, T2, T3, T3), set_bbbtree(T1),
    T2, T2, T3, T3).
:- mode fold2(pred(in, in, out, in, out) is det, in,
    in, out, in, out) is det.
:- mode fold2(pred(in, in, out, mdi, muo) is det, in,
    in, out, mdi, muo) is det.
:- mode fold2(pred(in, in, out, di, uo) is det, in,
    in, out, di, uo) is det.
```

```
:- mode fold2(pred(in, in, out, in, out) is semidet, in,
    in, out, in, out) is semidet.
:- mode fold2(pred(in, in, out, mdi, muo) is semidet, in,
    in, out, mdi, muo) is semidet.
:- mode fold2(pred(in, in, out, di, uo) is semidet, in,
    in, out, di, uo) is semidet.

:- pred fold3(pred(T1, T2, T2, T3, T3, T4, T4),
    set_bbbtree(T1), T2, T2, T3, T3, T4, T4).
:- mode fold3(pred(in, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out) is det.
:- mode fold3(pred(in, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, mdi, muo) is det.
:- mode fold3(pred(in, in, out, in, out, di, uo) is det, in,
    in, out, in, out, di, uo) is det.
:- mode fold3(pred(in, in, out, in, out, in, out) is semidet, in,
    in, out, in, out, in, out) is semidet.
:- mode fold3(pred(in, in, out, in, out, mdi, muo) is semidet, in,
    in, out, in, out, mdi, muo) is semidet.
:- mode fold3(pred(in, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, di, uo) is semidet.

:- pred fold4(pred(T1, T2, T2, T3, T3, T4, T4, T5, T5),
    set_bbbtree(T1), T2, T2, T3, T3, T4, T4, T5, T5).
:- mode fold4(
    pred(in, in, out, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out, in, out) is det.
:- mode fold4(
    pred(in, in, out, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, in, out, mdi, muo) is det.
:- mode fold4(
    pred(in, in, out, in, out, in, out, di, uo) is det, in,
    in, out, in, out, in, out, di, uo) is det.
:- mode fold4(
    pred(in, in, out, in, out, in, out, in, out) is semidet, in,
    in, out, in, out, in, out, in, out) is semidet.
:- mode fold4(
    pred(in, in, out, in, out, in, out, mdi, muo) is semidet, in,
    in, out, in, out, in, out, mdi, muo) is semidet.
:- mode fold4(
    pred(in, in, out, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, in, out, di, uo) is semidet.

:- pred fold5(
    pred(T1, T2, T2, T3, T3, T4, T4, T5, T5, T6, T6),
    set_bbbtree(T1), T2, T2, T3, T3, T4, T4, T5, T5, T6, T6).
:- mode fold5(
```

```
    pred(in, in, out, in, out, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out, in, out, in, out) is det.
:- mode fold5(
    pred(in, in, out, in, out, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, in, out, in, out, mdi, muo) is det.
:- mode fold5(
    pred(in, in, out, in, out, in, out, in, out, di, uo) is det, in,
    in, out, in, out, in, out, in, out, di, uo) is det.
:- mode fold5(
    pred(in, in, out, in, out, in, out, in, out, in, out) is semidet, in,
    in, out, in, out, in, out, in, out, in, out) is semidet.
:- mode fold5(
    pred(in, in, out, in, out, in, out, in, out, mdi, muo) is semidet, in,
    in, out, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode fold5(
    pred(in, in, out, in, out, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, in, out, in, out, di, uo) is semidet.

:- pred fold6(
    pred(T1, T2, T2, T3, T3, T4, T4, T5, T5, T6, T6, T7, T7),
    set_bbbtree(T1), T2, T2, T3, T3, T4, T4, T5, T5, T6, T6, T7, T7).
:- mode fold6(
    pred(in, in, out, in, out, in, out, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out, in, out, in, out, in, out) is det.
:- mode fold6(
    pred(in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is det.
:- mode fold6(
    pred(in, in, out, in, out, in, out, in, out, in, out, di, uo) is det,
    in, in, out, in, out, in, out, in, out, in, out, di, uo) is det.
:- mode fold6(
    pred(in, in, out, in, out, in, out, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out, in, out, in, out, in, out) is semidet.
:- mode fold6(
    pred(in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode fold6(
    pred(in, in, out, in, out, in, out, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, in, out, in, out, in, out, di, uo) is semidet.

%---------------------------------------------------%
%---------------------------------------------------%
```

73 set_ctree234

```
%---------------------------------------------------%
% vim: ts=4 sw=4 et ft=mercury
%---------------------------------------------------%
% Copyright (C) 2005-2006, 2010-2012 The University of Melbourne.
% Copyright (C) 2014-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: set_ctree234.m.
% Author: zs.
% Stability: high.
%
% This module implements sets using 2-3-4 trees extended with element counts.
% This representation has higher constant factors for most operations than
% ordered lists, but it has much better worst-case complexity, and is likely
% to be faster for large sets. Specifically,
%
% - the cost of lookups is only logarithmic in the size of the set, not linear;
%
% - for operations that are intrinsically linear in the size of one input
%   operand or the other, the counts allow us to choose to be linear in the
%   size of the smaller set.
%
%---------------------------------------------------%
%---------------------------------------------------%

:- module set_ctree234.
:- interface.

:- import_module bool.
:- import_module list.

%---------------------------------------------------%

:- type set_ctree234(_T).

%---------------------------------------------------%
%
% Initial creation of sets.
%

    % 'init = Set' is true iff 'Set' is an empty set.
    %
:- func init = set_ctree234(T).
```

```
    % ‘singleton_set(Elem, Set)’ is true iff ‘Set’ is the set containing just
    % the single element ‘Elem’.
    %
:- pred singleton_set(T, set_ctree234(T)).
:- mode singleton_set(in, out) is det.
:- mode singleton_set(out, in) is semidet.

:- func make_singleton_set(T) = set_ctree234(T).

%-----------------------------------------------------%
%
% Emptiness and singleton-ness tests.
%

    % ‘empty(Set)’ is true iff ‘Set’ is an empty set.
    % ‘is_empty’ is a synonym for ‘empty’.
    %
:- pred empty(set_ctree234(_T)::in) is semidet.
:- pred is_empty(set_ctree234(_T)::in) is semidet.
:- pragma obsolete(empty/1, [is_empty/1]).

    % ‘non_empty(Set)’ is true iff ‘Set’ is not an empty set.
    % ‘is_non_empty’ is a synonym for ‘non_empty’.
    %
:- pred non_empty(set_ctree234(T)::in) is semidet.
:- pred is_non_empty(set_ctree234(T)::in) is semidet.
:- pragma obsolete(non_empty/1, [is_non_empty/1]).

:- pred is_singleton(set_ctree234(T)::in, T::out) is semidet.

%-----------------------------------------------------%
%
% Membership tests.
%

    % ‘member(X, Set)’ is true iff ‘X’ is a member of ‘Set’.
    %
:- pred member(T, set_ctree234(T)).
:- mode member(in, in) is semidet.
:- mode member(out, in) is nondet.

    % ‘one_member(Set, X)’ is true iff ‘X’ is a member of ‘Set’.
    %
:- pred one_member(set_ctree234(T)::in, T::out) is nondet.

    % ‘is_member(Set, X, Result)’ returns ‘Result = yes’ iff
    % ‘X’ is a member of ‘Set’.
```

```
    %
:- func is_member(set_ctree234(T), T) = bool.
:- pred is_member(set_ctree234(T)::in, T::in, bool::out) is det.

    % 'contains(Set, X)' is true iff 'X' is a member of 'Set'.
    %
:- pred contains(set_ctree234(T)::in, T::in) is semidet.

%-----------------------------------------------------%
%
% Insertions and deletions.
%

    % 'insert(X, Set0, Set)' is true iff 'Set' is the union of 'Set0'
    % and the set containing only 'X'.
    %
:- func insert(T, set_ctree234(T)) = set_ctree234(T).
:- pred insert(T::in, set_ctree234(T)::in, set_ctree234(T)::out) is det.

    % 'insert_new(X, Set0, Set)' is true iff 'Set0' does not contain 'X',
    % and 'Set' is the union of 'Set0' and the set containing only 'X'.
    %
:- pred insert_new(T::in, set_ctree234(T)::in, set_ctree234(T)::out)
    is semidet.

    % 'insert_list(Xs, Set0, Set)' is true iff 'Set' is the union of 'Set0'
    % and the set containing only the members of 'Xs'.
    %
:- func insert_list(list(T), set_ctree234(T)) = set_ctree234(T).
:- pred insert_list(list(T)::in, set_ctree234(T)::in, set_ctree234(T)::out)
    is det.

    % 'delete(X, Set0, Set)' is true iff 'Set' is the
    % relative complement of 'Set0' and the set containing only 'X', i.e.
    % if 'Set' is the set which contains all the elements of 'Set0'
    % except 'X'.
    %
:- func delete(T, set_ctree234(T)) = set_ctree234(T).
:- pred delete(T::in, set_ctree234(T)::in, set_ctree234(T)::out) is det.

    % 'delete_list(Xs, Set0, Set)' is true iff 'Set' is the relative complement
    % of 'Set0' and the set containing only the members of 'Xs'.
    %
:- func delete_list(list(T), set_ctree234(T)) = set_ctree234(T).
:- pred delete_list(list(T)::in, set_ctree234(T)::in, set_ctree234(T)::out)
    is det.
```

```
    % ‘remove(X, Set0, Set)’ is true iff ‘Set0’ contains ‘X’,
    % and ‘Set’ is the relative complement of ‘Set0’ and the set containing
    % only ‘X’, i.e. if ‘Set’ is the set which contains all the elements
    % of ‘Set0’ except ‘X’.
    %
    % The det_remove version throws an exception instead of failing.
    %
:- pred remove(T::in, set_ctree234(T)::in, set_ctree234(T)::out) is semidet.
:- pred det_remove(T::in, set_ctree234(T)::in, set_ctree234(T)::out) is det.

    % ‘remove_list(Xs, Set0, Set)’ is true iff Xs does not contain any
    % duplicates, ‘Set0’ contains every member of ‘Xs’, and ‘Set’ is the
    % relative complement of ‘Set0’ and the set containing only the mem-
bers of
    % ‘Xs’.
    %
    % The det_remove_list version throws an exception instead of failing.
    %
:- pred remove_list(list(T)::in, set_ctree234(T)::in, set_ctree234(T)::out)
    is semidet.
:- pred det_remove_list(list(T)::in, set_ctree234(T)::in, set_ctree234(T)::out)
    is det.

    % ‘remove_least(X, Set0, Set)’ is true iff ‘X’ is the least element in
    % ‘Set0’, and ‘Set’ is the set which contains all the elements of ‘Set0’
    % except ‘X’.
    %
:- pred remove_least(T::out, set_ctree234(T)::in, set_ctree234(T)::out)
    is semidet.

%---------------------------------------------------%
%
% Comparisons between sets.
%

    % ‘equal(SetA, SetB)’ is true iff ‘SetA’ and ‘SetB’ contain
    % the same elements.
    %
:- pred equal(set_ctree234(T)::in, set_ctree234(T)::in) is semidet.

    % ‘subset(SetA, SetB)’ is true iff ‘SetA’ is a subset of ‘SetB’.
    %
:- pred subset(set_ctree234(T)::in, set_ctree234(T)::in) is semidet.

    % ‘superset(SetA, SetB)’ is true iff ‘SetA’ is a superset of ‘SetB’.
    %
:- pred superset(set_ctree234(T)::in, set_ctree234(T)::in) is semidet.
```

```
%----------------------------------------------------%
%
% Operations on two or more sets.
%

    % 'union(SetA, SetB) = Set' is true iff 'Set' is the union of 'SetA' and
    % 'SetB'.
    %
:- func union(set_ctree234(T), set_ctree234(T)) = set_ctree234(T).
:- pred union(set_ctree234(T)::in, set_ctree234(T)::in, set_ctree234(T)::out)
    is det.

    % 'union_list(A, B)' is true iff 'B' is the union of
    % all the sets in 'A'
    %
:- func union_list(list(set_ctree234(T))) = set_ctree234(T).
:- pred union_list(list(set_ctree234(T))::in, set_ctree234(T)::out) is det.

    % 'power_union(A) = B' is true iff 'B' is the union of all the sets in 'A'.
    %
:- func power_union(set_ctree234(set_ctree234(T))) = set_ctree234(T).
:- pred power_union(set_ctree234(set_ctree234(T))::in, set_ctree234(T)::out)
    is det.

    % 'intersect(SetA, SetB) = Set' is true iff 'Set' is the intersection of
    % 'SetA' and 'SetB'.
    %
:- func intersect(set_ctree234(T), set_ctree234(T)) = set_ctree234(T).
:- pred intersect(set_ctree234(T)::in, set_ctree234(T)::in,
    set_ctree234(T)::out) is det.

    % 'intersect_list(A) = B' is true iff 'B' is the intersection
    % of all the sets in 'A'.
    %
:- func intersect_list(list(set_ctree234(T))) = set_ctree234(T).

    % 'power_intersect(A, B)' is true iff 'B' is the intersection
    % of all the sets in 'A'.
    %
:- func power_intersect(set_ctree234(set_ctree234(T))) = set_ctree234(T).

    % 'difference(SetA, SetB, Set)' is true iff 'Set' is the set containing
    % all the elements of 'SetA' except those that occur in 'SetB'.
    %
:- func difference(set_ctree234(T), set_ctree234(T)) = set_ctree234(T).
:- pred difference(set_ctree234(T)::in, set_ctree234(T)::in,
```

```
    set_ctree234(T)::out) is det.

    % intersection_and_differences(SetA, SetB, InAandB, OnlyInA, OnlyInB):
    % Given SetA and SetB, return the elements that occur in both sets,
    % and those that occur only in one or the other.
    %
:- pred intersection_and_differences(set_ctree234(T)::in, set_ctree234(T)::in,
    set_ctree234(T)::out, set_ctree234(T)::out, set_ctree234(T)::out) is det.

%-----------------------------------------------------%
%
% Operations that divide a set into two parts.
%

    % divide(Pred, Set, TruePart, FalsePart):
    % TruePart consists of those elements of Set for which Pred succeeds;
    % FalsePart consists of those elements of Set for which Pred fails.
    % NOTE: This is the same as filter/4.
    %
:- pred divide(pred(T)::in(pred(in) is semidet),
    set_ctree234(T)::in, set_ctree234(T)::out, set_ctree234(T)::out) is det.

    % divide_by_set(DivideBySet, Set, InPart, OutPart):
    % InPart consists of those elements of Set which are also in
    % DivideBySet; OutPart consists of those elements of which are
    % not in DivideBySet.
    %
:- pred divide_by_set(set_ctree234(T)::in, set_ctree234(T)::in,
    set_ctree234(T)::out, set_ctree234(T)::out) is det.

%-----------------------------------------------------%
%
% Converting lists to sets.
%

    % 'list_to_set(List) = Set' is true iff 'Set' is the set
    % containing only the members of 'List'.
    %
    % 'from_list' is a synonym for 'list_to_set'.
    %
:- func list_to_set(list(T)) = set_ctree234(T).
:- func from_list(list(T)) = set_ctree234(T).

    % 'sorted_list_to_set(List) = Set' is true iff 'Set' is the set
    % containing only the members of 'List'. 'List' must be sorted
    % in ascending order and must not contain any duplicates.
    %
```

```
:- func sorted_list_to_set(list(T)) = set_ctree234(T).

    % 'rev_sorted_list_to_set(List) = Set' is true iff 'Set' is the set
    % containing only the members of 'List'. 'List' must be sorted
    % in descending order and must not contain any duplicates.
    %
:- func rev_sorted_list_to_set(list(T)) = set_ctree234(T).

%-----------------------------------------------------%
%
% Converting sets to lists.
%

    % 'to_sorted_list(Set) = List' is true iff 'List' is the list
    % of all the members of 'Set', in sorted order.
    %
:- func to_sorted_list(set_ctree234(T)) = list(T).

%-----------------------------------------------------%
%
% Counting.
%

    % 'count(Set, Count)' is true iff 'Set' has 'Count' elements.
    %
:- func count(set_ctree234(T)) = int.

:- pred verify_depths(set_ctree234(T)::in, list(int)::out) is det.

%-----------------------------------------------------%
%
% Standard higher order functions on collections.
%

    % all_true(Pred, Set) succeeds iff Pred(Element) succeeds for all the
    % elements of Set.
    %
:- pred all_true(pred(T)::in(pred(in) is semidet),
    set_ctree234(T)::in) is semidet.

    % Return the set of items for which the predicate succeeds.
    %
:- pred filter(pred(T)::in(pred(in) is semidet),
    set_ctree234(T)::in, set_ctree234(T)::out) is det.

    % Return the set of items for which the predicate succeeds,
    % and the set for which it fails.
```

```
    %
:- pred filter(pred(T)::in(pred(in) is semidet),
    set_ctree234(T)::in, set_ctree234(T)::out, set_ctree234(T)::out) is det.

:- func filter_map(func(T1) = T2, set_ctree234(T1)) = set_ctree234(T2).
:- mode filter_map(func(in) = out is semidet, in) = out is det.

:- pred filter_map(pred(T1, T2)::in(pred(in, out) is semidet),
    set_ctree234(T1)::in, set_ctree234(T2)::out) is det.

:- func map(func(T1) = T2, set_ctree234(T1)) = set_ctree234(T2).
:- pred map(pred(T1, T2)::in(pred(in, out) is det),
    set_ctree234(T1)::in, set_ctree234(T2)::out) is det.

:- func fold(func(T1, T2) = T2, set_ctree234(T1), T2) = T2.
:- pred fold(pred(T1, T2, T2), set_ctree234(T1), T2, T2).
:- mode fold(pred(in, in, out) is det, in, in, out) is det.
:- mode fold(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode fold(pred(in, di, uo) is det, in, di, uo) is det.
:- mode fold(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode fold(pred(in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode fold(pred(in, di, uo) is semidet, in, di, uo) is semidet.

:- pred fold2(pred(T1, T2, T2, T3, T3), set_ctree234(T1),
    T2, T2, T3, T3).
:- mode fold2(pred(in, in, out, in, out) is det,
    in, in, out, in, out) is det.
:- mode fold2(pred(in, in, out, mdi, muo) is det,
    in, in, out, mdi, muo) is det.
:- mode fold2(pred(in, in, out, di, uo) is det,
    in, in, out, di, uo) is det.
:- mode fold2(pred(in, in, out, in, out) is semidet,
    in, in, out, in, out) is semidet.
:- mode fold2(pred(in, in, out, mdi, muo) is semidet,
    in, in, out, mdi, muo) is semidet.
:- mode fold2(pred(in, in, out, di, uo) is semidet,
    in, in, out, di, uo) is semidet.

:- pred fold3(
    pred(T1, T2, T2, T3, T3, T4, T4), set_ctree234(T1),
    T2, T2, T3, T3, T4, T4).
:- mode fold3(pred(in, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out) is det.
:- mode fold3(pred(in, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, mdi, muo) is det.
:- mode fold3(pred(in, in, out, in, out, di, uo) is det, in,
    in, out, in, out, di, uo) is det.
```

```
:- mode fold3(pred(in, in, out, in, out, in, out) is semidet, in,
    in, out, in, out, in, out) is semidet.
:- mode fold3(pred(in, in, out, in, out, mdi, muo) is semidet, in,
    in, out, in, out, mdi, muo) is semidet.
:- mode fold3(pred(in, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, di, uo) is semidet.

:- pred fold4(
    pred(T1, T2, T2, T3, T3, T4, T4, T5, T5), set_ctree234(T1),
    T2, T2, T3, T3, T4, T4, T5, T5).
:- mode fold4(pred(in, in, out, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out, in, out) is det.
:- mode fold4(pred(in, in, out, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, in, out, mdi, muo) is det.
:- mode fold4(pred(in, in, out, in, out, in, out, di, uo) is det,
    in, in, out, in, out, in, out, di, uo) is det.
:- mode fold4(
    pred(in, in, out, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out, in, out) is semidet.
:- mode fold4(
    pred(in, in, out, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode fold4(
    pred(in, in, out, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, in, out, di, uo) is semidet.

:- pred fold5(
    pred(T1, T2, T2, T3, T3, T4, T4, T5, T5, T6, T6),
    set_ctree234(T1), T2, T2, T3, T3, T4, T4, T5, T5, T6, T6).
:- mode fold5(
    pred(in, in, out, in, out, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out, in, out, in, out) is det.
:- mode fold5(
    pred(in, in, out, in, out, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, in, out, in, out, mdi, muo) is det.
:- mode fold5(
    pred(in, in, out, in, out, in, out, in, out, di, uo) is det,
    in, in, out, in, out, in, out, in, out, di, uo) is det.
:- mode fold5(
    pred(in, in, out, in, out, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out, in, out, in, out) is semidet.
:- mode fold5(
    pred(in, in, out, in, out, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode fold5(
    pred(in, in, out, in, out, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, in, out, in, out, di, uo) is semidet.
```

```
:- pred fold6(
    pred(T1, T2, T2, T3, T3, T4, T4, T5, T5, T6, T6, T7, T7),
    set_ctree234(T1), T2, T2, T3, T3, T4, T4, T5, T5, T6, T6, T7, T7).
:- mode fold6(
    pred(in, in, out, in, out, in, out, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out, in, out, in, out, in, out) is det.
:- mode fold6(
    pred(in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is det.
:- mode fold6(
    pred(in, in, out, in, out, in, out, in, out, in, out, di, uo) is det,
    in, in, out, in, out, in, out, in, out, in, out, di, uo) is det.
:- mode fold6(
    pred(in, in, out, in, out, in, out, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out, in, out, in, out, in, out) is semidet.
:- mode fold6(
    pred(in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode fold6(
    pred(in, in, out, in, out, in, out, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, in, out, in, out, in, out, di, uo) is semidet.

%-----------------------------------------------------%
%-----------------------------------------------------%
```

74 set_ordlist

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 1996-1997,1999-2002, 2004-2006, 2008-2012 The University of Melbourne.
% Copyright (C) 2014-2015, 2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: set_ordlist.m.
% Main authors: conway, fjh.
% Stability: medium.
%
% This file contains a 'set' ADT.
% Sets are implemented here as sorted lists without duplicates.
%
%-----------------------------------------------------%
%-----------------------------------------------------%
```

```
:- module set_ordlist.
:- interface.

:- import_module bool.
:- import_module list.

%-----------------------------------------------------%

:- type set_ordlist(_T).

%-----------------------------------------------------%
%
% Initial creation of sets.
%

    % 'init(Set)' is true iff 'Set' is an empty set.
    %
:- func init = set_ordlist(T).
:- pred init(set_ordlist(_T)::uo) is det.

    % 'singleton_set(Elem, Set)' is true iff 'Set' is the set containing just
    % the single element 'Elem'.
    %
:- pred singleton_set(T, set_ordlist(T)).
:- mode singleton_set(in, out) is det.
:- mode singleton_set(out, in) is semidet.

:- func make_singleton_set(T) = set_ordlist(T).

%-----------------------------------------------------%
%
% Emptiness and singleton-ness tests.
%

    % 'empty(Set)' is true iff 'Set' is an empty set.
    % 'is_empty' is a synonym for 'empty'.
    %
:- pred empty(set_ordlist(_T)::in) is semidet.
:- pred is_empty(set_ordlist(T)::in) is semidet.
:- pragma obsolete(empty/1, [is_empty/1]).

    % 'non_empty(Set)' is true iff 'Set' is not an empty set.
    % 'is_non_empty' is a synonym for 'non_empty'.
    %
:- pred non_empty(set_ordlist(T)::in) is semidet.
:- pred is_non_empty(set_ordlist(T)::in) is semidet.
```

```
:- pragma obsolete(non_empty/1, [is_non_empty/1]).

:- pred is_singleton(set_ordlist(T)::in, T::out) is semidet.

%-----------------------------------------------------%
%
% Membership tests.
%

    % 'member(X, Set)' is true iff 'X' is a member of 'Set'.
    %
:- pred member(T, set_ordlist(T)).
:- mode member(in, in) is semidet.
:- mode member(out, in) is nondet.

    % 'is_member(X, Set, Result)' returns 'Result = yes' iff 'X' is a mem-
ber of
    % 'Set'.
    %
:- pred is_member(T::in, set_ordlist(T)::in, bool::out) is det.

    % 'contains(Set, X)' is true iff 'X' is a member of 'Set'.
    %
:- pred contains(set_ordlist(T)::in, T::in) is semidet.

%-----------------------------------------------------%
%
% Insertions and deletions.
%

    % 'insert(X, Set0, Set)' is true iff 'Set' is the union
    % of 'Set0' and the set containing only 'X'.
    %
:- func insert(set_ordlist(T), T) = set_ordlist(T).
:- pred insert(T::in, set_ordlist(T)::in, set_ordlist(T)::out) is det.

    % 'insert_new(X, Set0, Set)' is true iff 'Set0' does not contain 'X', while
    % 'Set' is the union of 'Set0' and the set containing only 'X'.
    %
:- pred insert_new(T::in,
    set_ordlist(T)::in, set_ordlist(T)::out) is semidet.

    % 'insert_list(Xs, Set0, Set)' is true iff 'Set' is the union of 'Set0' and
    % the set containing only the members of 'Xs'.
    %
:- func insert_list(set_ordlist(T), list(T)) = set_ordlist(T).
:- pred insert_list(list(T)::in, set_ordlist(T)::in, set_ordlist(T)::out)
```

```
    is det.

    % 'delete(X, Set0, Set)' is true iff 'Set' is the
    % relative complement of 'Set0' and the set containing only 'X', i.e.
    % if 'Set' is the set which contains all the elements of 'Set0'
    % except 'X'.
    %
:- func delete(set_ordlist(T), T) = set_ordlist(T).
:- pred delete(T::in, set_ordlist(T)::in, set_ordlist(T)::out) is det.

    % 'delete_list(Xs, Set0, Set)' is true iff 'Set' is the relative complement
    % of 'Set0' and the set containing only the members of 'Xs'.
    %
:- func delete_list(set_ordlist(T), list(T)) = set_ordlist(T).
:- pred delete_list(list(T)::in, set_ordlist(T)::in, set_ordlist(T)::out)
    is det.

    % 'remove(X, Set0, Set)' is true iff 'Set0' contains 'X',
    % and 'Set' is the relative complement of 'Set0' and the set
    % containing only 'X', i.e. if 'Set' is the set which contains
    % all the elements of 'Set0' except 'X'.
    %
    % The det_remove version throws an exception instead of failing.
    %
:- pred remove(T::in, set_ordlist(T)::in, set_ordlist(T)::out) is semidet.
:- pred det_remove(T::in, set_ordlist(T)::in, set_ordlist(T)::out) is det.

    % 'remove_list(Xs, Set0, Set)' is true iff Xs does not contain any
    % duplicates, 'Set0' contains every member of 'Xs', and 'Set' is the
    % relative complement of 'Set0' and the set containing only the mem-
bers of
    % 'Xs'.
    %
    % The det_remove_list version throws an exception instead of failing.
    %
:- pred remove_list(list(T)::in, set_ordlist(T)::in, set_ordlist(T)::out)
    is semidet.
:- pred det_remove_list(list(T)::in, set_ordlist(T)::in, set_ordlist(T)::out)
    is det.

    % 'remove_least(X, Set0, Set)' is true iff 'X' is the least element in
    % 'Set0', and 'Set' is the set which contains all the elements of 'Set0'
    % except 'X'.
    %
:- pred remove_least(T::out, set_ordlist(T)::in, set_ordlist(T)::out)
    is semidet.
```

```
%-----------------------------------------------%
%
% Comparisons between sets.
%

    % 'equal(SetA, SetB)' is true iff 'SetA' and 'SetB' contain the same
    % elements.
    %
:- pred equal(set_ordlist(T)::in, set_ordlist(T)::in) is semidet.

    % 'subset(SetA, SetB)' is true iff 'SetA' is a subset of 'SetB'.
    %
:- pred subset(set_ordlist(T)::in, set_ordlist(T)::in) is semidet.

    % 'superset(SetA, SetB)' is true iff 'SetA' is a superset of 'SetB'.
    %
:- pred superset(set_ordlist(T)::in, set_ordlist(T)::in) is semidet.

%-----------------------------------------------%
%
% Operations on two or more sets.
%

    % 'union(SetA, SetB, Set)' is true iff 'Set' is the union
    % of 'SetA' and 'SetB'. The efficiency of the union operation is
    % O(card(SetA)+card(SetB)) and is not sensitive to the argument
    % ordering.
    %
:- func union(set_ordlist(T), set_ordlist(T)) = set_ordlist(T).
:- pred union(set_ordlist(T)::in, set_ordlist(T)::in, set_ordlist(T)::out)
    is det.

    % 'union_list(A, B)' is true iff 'B' is the union of all the sets in 'A'
    %
:- func union_list(list(set_ordlist(T))) = set_ordlist(T).
:- pred union_list(list(set_ordlist(T))::in, set_ordlist(T)::out) is det.

    % 'power_union(A, B)' is true iff 'B' is the union of
    % all the sets in 'A'
    %
:- func power_union(set_ordlist(set_ordlist(T))) = set_ordlist(T).
:- pred power_union(set_ordlist(set_ordlist(T))::in,
    set_ordlist(T)::out) is det.

    % 'intersect(SetA, SetB, Set)' is true iff 'Set' is the intersection of
    % 'SetA' and 'SetB'. The efficiency of the intersection operation is not
    % influenced by the argument order.
```

```
    %
:- func intersect(set_ordlist(T), set_ordlist(T)) = set_ordlist(T).
:- pred intersect(set_ordlist(T), set_ordlist(T), set_ordlist(T)).
:- mode intersect(in, in, out) is det.
:- mode intersect(in, in, in) is semidet.

    % 'intersect_list(A) = B' is true iff 'B' is the intersection of all the
    % sets in 'A'.
    %
:- func intersect_list(list(set_ordlist(T))) = set_ordlist(T).
:- pred intersect_list(list(set_ordlist(T))::in, set_ordlist(T)::out) is det.

    % 'power_intersect(A, B)' is true iff 'B' is the intersection of all the
    % sets in 'A'.
    %
:- func power_intersect(set_ordlist(set_ordlist(T)))
    = set_ordlist(T).
:- pred power_intersect(set_ordlist(set_ordlist(T))::in,
    set_ordlist(T)::out) is det.

    % 'difference(SetA, SetB, Set)' is true iff 'Set' is the
    % set containing all the elements of 'SetA' except those that
    % occur in 'SetB'.
    %
:- func difference(set_ordlist(T), set_ordlist(T)) = set_ordlist(T).
:- pred difference(set_ordlist(T)::in, set_ordlist(T)::in,
    set_ordlist(T)::out) is det.

    % intersection_and_differences(SetA, SetB, InAandB, OnlyInA, OnlyInB):
    % Given SetA and SetB, return the elements that occur in both sets,
    % and those that occur only in one or the other.
    %
:- pred intersection_and_differences(set_ordlist(T)::in, set_ordlist(T)::in,
    set_ordlist(T)::out, set_ordlist(T)::out, set_ordlist(T)::out) is det.

%-----------------------------------------------%
%
% Operations that divide a set into two parts.
%

    % divide(Pred, Set, TruePart, FalsePart):
    % TruePart consists of those elements of Set for which Pred succeeds;
    % FalsePart consists of those elements of Set for which Pred fails.
    %
:- pred divide(pred(T)::in(pred(in) is semidet),
    set_ordlist(T)::in, set_ordlist(T)::out, set_ordlist(T)::out) is det.
```

```
    % divide_by_set(DivideBySet, Set, InPart, OutPart):
    % InPart consists of those elements of Set which are also in DivideBySet;
    % OutPart consists of those elements of Set which are not in DivideBySet.
    %
:- pred divide_by_set(set_ordlist(T)::in, set_ordlist(T)::in,
    set_ordlist(T)::out, set_ordlist(T)::out) is det.

%-----------------------------------------------------%
%
% Converting lists to sets.
%

    % 'list_to_set(List, Set)' is true iff 'Set' is the set
    % containing only the members of 'List'.
    %
:- func list_to_set(list(T)) = set_ordlist(T).
:- pred list_to_set(list(T)::in, set_ordlist(T)::out) is det.

    % A synonym for list_to_set/1.
    %
:- func from_list(list(T)) = set_ordlist(T).

    % 'sorted_list_to_set(List, Set)' is true iff 'Set' is the set
    % containing only the members of 'List'. 'List' must be sorted
    % in ascending order.
    %
:- func sorted_list_to_set(list(T)) = set_ordlist(T).
:- pred sorted_list_to_set(list(T)::in, set_ordlist(T)::out) is det.

    % A synonym for sorted_list_to_set/1.
    %
:- func from_sorted_list(list(T)) = set_ordlist(T).

    % 'rev_sorted_list_to_set(List, Set)' is true iff 'Set' is the set
    % containing only the members of 'List'. 'List' must be sorted
    % in descending order and must not contain any duplicates.
    %
:- func rev_sorted_list_to_set(list(T)) = set_ordlist(T).
:- pred rev_sorted_list_to_set(list(T)::in, set_ordlist(T)::out) is det.

%-----------------------------------------------------%
%
% Converting sets to lists.
%

    % 'to_sorted_list(Set, List)' is true iff 'List' is the list of all the
    % members of 'Set', in sorted order.
```

```
    %
:- func to_sorted_list(set_ordlist(T)) = list(T).
:- pred to_sorted_list(set_ordlist(T)::in, list(T)::out) is det.

%-----------------------------------------------------%
%
% Counting.
%

    % 'count(Set, Count)' is true iff 'Set' has 'Count' elements.
    %
:- func count(set_ordlist(T)) = int.
:- pred count(set_ordlist(T)::in, int::out) is det.

%-----------------------------------------------------%
%
% Standard higher order functions on collections.
%

    % all_true(Pred, Set) succeeds iff Pred(Element) succeeds for all the
    % elements of Set.
    %
:- pred all_true(pred(T)::in(pred(in) is semidet), set_ordlist(T)::in)
    is semidet.

    % Return the set of items for which the given predicate succeeds.
    %
:- func filter(pred(T1), set_ordlist(T1)) = set_ordlist(T1).
:- mode filter(pred(in) is semidet, in) = out is det.
:- pred filter(pred(T1), set_ordlist(T1), set_ordlist(T1)).
:- mode filter(pred(in) is semidet, in, out) is det.

    % Return the set of items for which the given predicate succeeds, and the
    % set of items for which it fails.
    %
:- pred filter(pred(T1), set_ordlist(T1), set_ordlist(T1), set_ordlist(T1)).
:- mode filter(pred(in) is semidet, in, out, out) is det.

:- func filter_map(func(T1) = T2, set_ordlist(T1)) = set_ordlist(T2).
:- mode filter_map(func(in) = out is semidet, in) = out is det.
:- pred filter_map(pred(T1, T2), set_ordlist(T1), set_ordlist(T2)).
:- mode filter_map(pred(in, out) is semidet, in, out) is det.

:- func map(func(T1) = T2, set_ordlist(T1)) = set_ordlist(T2).

:- func fold(func(T1, T2) = T2, set_ordlist(T1), T2) = T2.
:- pred fold(pred(T1, T2, T2), set_ordlist(T1), T2, T2).
```

```
:- mode fold(pred(in, in, out) is det, in, in, out) is det.
:- mode fold(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode fold(pred(in, di, uo) is det, in, di, uo) is det.
:- mode fold(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode fold(pred(in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode fold(pred(in, di, uo) is semidet, in, di, uo) is semidet.

:- func foldl(func(T1, T2) = T2, set_ordlist(T1), T2) = T2.
:- pred foldl(pred(T1, T2, T2), set_ordlist(T1), T2, T2).
:- mode foldl(pred(in, in, out) is det, in, in, out) is det.
:- mode foldl(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode foldl(pred(in, di, uo) is det, in, di, uo) is det.
:- mode foldl(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode foldl(pred(in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode foldl(pred(in, di, uo) is semidet, in, di, uo) is semidet.

:- pred fold2(pred(T1, T2, T2, T3, T3), set_ordlist(T1),
    T2, T2, T3, T3).
:- mode fold2(pred(in, in, out, in, out) is det, in,
    in, out, in, out) is det.
:- mode fold2(pred(in, in, out, mdi, muo) is det, in,
    in, out, mdi, muo) is det.
:- mode fold2(pred(in, in, out, di, uo) is det, in,
    in, out, di, uo) is det.
:- mode fold2(pred(in, in, out, in, out) is semidet, in,
    in, out, in, out) is semidet.
:- mode fold2(pred(in, in, out, mdi, muo) is semidet, in,
    in, out, mdi, muo) is semidet.
:- mode fold2(pred(in, in, out, di, uo) is semidet, in,
    in, out, di, uo) is semidet.

:- pred foldl2(pred(T1, T2, T2, T3, T3), set_ordlist(T1),
    T2, T2, T3, T3).
:- mode foldl2(pred(in, in, out, in, out) is det, in,
    in, out, in, out) is det.
:- mode foldl2(pred(in, in, out, mdi, muo) is det, in,
    in, out, mdi, muo) is det.
:- mode foldl2(pred(in, in, out, di, uo) is det, in,
    in, out, di, uo) is det.
:- mode foldl2(pred(in, in, out, in, out) is semidet, in,
    in, out, in, out) is semidet.
:- mode foldl2(pred(in, in, out, mdi, muo) is semidet, in,
    in, out, mdi, muo) is semidet.
:- mode foldl2(pred(in, in, out, di, uo) is semidet, in,
    in, out, di, uo) is semidet.

:- pred fold3(pred(T1, T2, T2, T3, T3, T4, T4),
```

```
    set_ordlist(T1), T2, T2, T3, T3, T4, T4).
:- mode fold3(pred(in, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out) is det.
:- mode fold3(pred(in, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, mdi, muo) is det.
:- mode fold3(pred(in, in, out, in, out, di, uo) is det, in,
    in, out, in, out, di, uo) is det.
:- mode fold3(pred(in, in, out, in, out, in, out) is semidet, in,
    in, out, in, out, in, out) is semidet.
:- mode fold3(pred(in, in, out, in, out, mdi, muo) is semidet, in,
    in, out, in, out, mdi, muo) is semidet.
:- mode fold3(pred(in, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, di, uo) is semidet.

:- pred foldl3(pred(T1, T2, T2, T3, T3, T4, T4),
    set_ordlist(T1), T2, T2, T3, T3, T4, T4).
:- mode foldl3(pred(in, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out) is det.
:- mode foldl3(pred(in, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, mdi, muo) is det.
:- mode foldl3(pred(in, in, out, in, out, di, uo) is det, in,
    in, out, in, out, di, uo) is det.
:- mode foldl3(pred(in, in, out, in, out, in, out) is semidet, in,
    in, out, in, out, in, out) is semidet.
:- mode foldl3(pred(in, in, out, in, out, mdi, muo) is semidet, in,
    in, out, in, out, mdi, muo) is semidet.
:- mode foldl3(pred(in, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, di, uo) is semidet.

:- pred fold4(pred(T1, T2, T2, T3, T3, T4, T4, T5, T5),
    set_ordlist(T1), T2, T2, T3, T3, T4, T4, T5, T5).
:- mode fold4(
    pred(in, in, out, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out, in, out) is det.
:- mode fold4(
    pred(in, in, out, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, in, out, mdi, muo) is det.
:- mode fold4(
    pred(in, in, out, in, out, in, out, di, uo) is det, in,
    in, out, in, out, in, out, di, uo) is det.
:- mode fold4(
    pred(in, in, out, in, out, in, out, in, out) is semidet, in,
    in, out, in, out, in, out, in, out) is semidet.
:- mode fold4(
    pred(in, in, out, in, out, in, out, mdi, muo) is semidet, in,
    in, out, in, out, in, out, mdi, muo) is semidet.
:- mode fold4(
```

```
    pred(in, in, out, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, in, out, di, uo) is semidet.

:- pred foldl4(pred(T1, T2, T2, T3, T3, T4, T4, T5, T5),
    set_ordlist(T1), T2, T2, T3, T3, T4, T4, T5, T5).
:- mode foldl4(
    pred(in, in, out, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out, in, out) is det.
:- mode foldl4(
    pred(in, in, out, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, in, out, mdi, muo) is det.
:- mode foldl4(
    pred(in, in, out, in, out, in, out, di, uo) is det, in,
    in, out, in, out, in, out, di, uo) is det.
:- mode foldl4(
    pred(in, in, out, in, out, in, out, in, out) is semidet, in,
    in, out, in, out, in, out, in, out) is semidet.
:- mode foldl4(
    pred(in, in, out, in, out, in, out, mdi, muo) is semidet, in,
    in, out, in, out, in, out, mdi, muo) is semidet.
:- mode foldl4(
    pred(in, in, out, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, in, out, di, uo) is semidet.

:- pred fold5(
    pred(T1, T2, T2, T3, T3, T4, T4, T5, T5, T6, T6),
    set_ordlist(T1), T2, T2, T3, T3, T4, T4, T5, T5, T6, T6).
:- mode fold5(
    pred(in, in, out, in, out, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out, in, out, in, out) is det.
:- mode fold5(
    pred(in, in, out, in, out, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, in, out, in, out, mdi, muo) is det.
:- mode fold5(
    pred(in, in, out, in, out, in, out, in, out, di, uo) is det, in,
    in, out, in, out, in, out, in, out, di, uo) is det.
:- mode fold5(
    pred(in, in, out, in, out, in, out, in, out, in, out) is semidet, in,
    in, out, in, out, in, out, in, out, in, out) is semidet.
:- mode fold5(
    pred(in, in, out, in, out, in, out, in, out, mdi, muo) is semidet, in,
    in, out, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode fold5(
    pred(in, in, out, in, out, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, in, out, in, out, di, uo) is semidet.

:- pred foldl5(
```

```
    pred(T1, T2, T2, T3, T3, T4, T4, T5, T5, T6, T6),
    set_ordlist(T1), T2, T2, T3, T3, T4, T4, T5, T5, T6, T6).
:- mode foldl5(
    pred(in, in, out, in, out, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out, in, out, in, out) is det.
:- mode foldl5(
    pred(in, in, out, in, out, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, in, out, in, out, mdi, muo) is det.
:- mode foldl5(
    pred(in, in, out, in, out, in, out, in, out, di, uo) is det, in,
    in, out, in, out, in, out, in, out, di, uo) is det.
:- mode foldl5(
    pred(in, in, out, in, out, in, out, in, out, in, out) is semidet, in,
    in, out, in, out, in, out, in, out, in, out) is semidet.
:- mode foldl5(
    pred(in, in, out, in, out, in, out, in, out, mdi, muo) is semidet, in,
    in, out, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode foldl5(
    pred(in, in, out, in, out, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, in, out, in, out, di, uo) is semidet.

:- pred fold6(pred(T, A, A, B, B, C, C, D, D, E, E, F, F),
    set_ordlist(T), A, A, B, B, C, C, D, D, E, E, F, F).
:- mode fold6(
    pred(in, in, out, in, out, in, out, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out, in, out, in, out, in, out) is det.
:- mode fold6(
    pred(in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is det.
:- mode fold6(
    pred(in, in, out, in, out, in, out, in, out, in, out, di, uo) is det,
    in, in, out, in, out, in, out, in, out, in, out, di, uo) is det.
:- mode fold6(
    pred(in, in, out, in, out, in, out, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out, in, out, in, out, in, out) is semidet.
:- mode fold6(
    pred(in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode fold6(
    pred(in, in, out, in, out, in, out, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, in, out, in, out, in, out, di, uo) is semidet.

:- pred foldl6(pred(T, A, A, B, B, C, C, D, D, E, E, F, F),
    set_ordlist(T), A, A, B, B, C, C, D, D, E, E, F, F).
:- mode foldl6(
    pred(in, in, out, in, out, in, out, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out, in, out, in, out, in, out) is det.
```

```
:- mode foldl6(
    pred(in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is det.
:- mode foldl6(
    pred(in, in, out, in, out, in, out, in, out, in, out, di, uo) is det,
    in, in, out, in, out, in, out, in, out, in, out, di, uo) is det.
:- mode foldl6(
    pred(in, in, out, in, out, in, out, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out, in, out, in, out, in, out) is semidet.
:- mode foldl6(
    pred(in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode foldl6(
    pred(in, in, out, in, out, in, out, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, in, out, in, out, in, out, di, uo) is semidet.

%---------------------------------------------------%
%---------------------------------------------------%
```

75 set_tree234

```
%---------------------------------------------------%
% vim: ts=4 sw=4 et ft=mercury
%---------------------------------------------------%
% Copyright (C) 2005-2006, 2009-2012 The University of Melbourne.
% Copyright (C) 2014-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: set_tree234.m.
% Author: zs.
% Stability: high.
%
% This module implements sets using 2-3-4 trees.
%
%---------------------------------------------------%
%---------------------------------------------------%

:- module set_tree234.
:- interface.

:- import_module bool.
:- import_module list.
:- import_module set.
```

```
%---------------------------------------------------%

:- type set_tree234(_T).

%---------------------------------------------------%
%
% Initial creation of sets.
%

    % 'init = Set' is true iff 'Set' is an empty set.
    %
:- func init = set_tree234(T).

    % 'singleton_set(Elem, Set)' is true iff 'Set' is the set containing just
    % the single element 'Elem'.
    %
:- pred singleton_set(T, set_tree234(T)).
:- mode singleton_set(in, out) is det.
:- mode singleton_set(out, in) is semidet.

:- func make_singleton_set(T) = set_tree234(T).

%---------------------------------------------------%
%
% Emptiness and singleton-ness tests.
%

    % 'empty(Set)' is true iff 'Set' is an empty set.
    % 'is_empty' is a synonym for 'empty'.
    %
:- pred empty(set_tree234(_T)::in) is semidet.
:- pred is_empty(set_tree234(_T)::in) is semidet.
:- pragma obsolete(empty/1, [is_empty/1]).

    % 'non_empty(Set)' is true iff 'Set' is not an empty set.
    % 'is_non_empty' is a synonym for 'non_empty'.
    %
:- pred non_empty(set_tree234(T)::in) is semidet.
:- pred is_non_empty(set_tree234(T)::in) is semidet.
:- pragma obsolete(non_empty/1, [is_non_empty/1]).

:- pred is_singleton(set_tree234(T)::in, T::out) is semidet.

%---------------------------------------------------%
%
% Membership tests.
%
```

```
    % ‘member(X, Set)’ is true iff ‘X’ is a member of ‘Set’.
    %
:- pred member(T, set_tree234(T)).
:- mode member(in, in) is semidet.
:- mode member(out, in) is nondet.

    % ‘is_member(Set, X, Result)’ returns ‘Result = yes’ iff
    % ‘X’ is a member of ‘Set’.
    %
:- func is_member(set_tree234(T), T) = bool.
:- pred is_member(set_tree234(T)::in, T::in, bool::out) is det.

    % ‘contains(Set, X)’ is true iff ‘X’ is a member of ‘Set’.
    %
:- pred contains(set_tree234(T)::in, T::in) is semidet.

%-----------------------------------------------%
%
% Insertions and deletions.
%

    % ‘insert(X, Set0, Set)’ is true iff ‘Set’ is the union of ‘Set0’ and the
    % set containing only ‘X’.
    %
:- func insert(T, set_tree234(T)) = set_tree234(T).
:- pred insert(T::in, set_tree234(T)::in, set_tree234(T)::out) is det.

    % ‘insert_new(X, Set0, Set)’ is true iff ‘Set0’ does not contain ‘X’, while
    % ‘Set’ is the union of ‘Set0’ and the set containing only ‘X’.
    %
:- pred insert_new(T::in, set_tree234(T)::in, set_tree234(T)::out) is semidet.

    % ‘insert_list(Xs, Set0, Set)’ is true iff ‘Set’ is the union of ‘Set0’ and
    % the set containing only the members of ‘Xs’.
    %
:- func insert_list(list(T), set_tree234(T)) = set_tree234(T).
:- pred insert_list(list(T)::in, set_tree234(T)::in, set_tree234(T)::out)
    is det.

    % ‘delete(X, Set0, Set)’ is true iff ‘Set’ is the relative complement of
    % ‘Set0’ and the set containing only ‘X’, i.e.  if ‘Set’ is the set which
    % contains all the elements of ‘Set0’ except ‘X’.
    %
:- func delete(T, set_tree234(T)) = set_tree234(T).
:- pred delete(T::in, set_tree234(T)::in, set_tree234(T)::out) is det.
```

```
    % ‘delete_list(Xs, Set0, Set)’ is true iff ‘Set’ is the relative complement
    % of ‘Set0’ and the set containing only the members of ‘Xs’.
    %
:- func delete_list(list(T), set_tree234(T)) = set_tree234(T).
:- pred delete_list(list(T)::in, set_tree234(T)::in, set_tree234(T)::out)
    is det.

    % ‘remove(X, Set0, Set)’ is true iff ‘Set0’ contains ‘X’,
    % and ‘Set’ is the relative complement of ‘Set0’ and the set
    % containing only ‘X’, i.e.  if ‘Set’ is the set which contains
    % all the elements of ‘Set0’ except ‘X’.
    %
    % The det_remove version throws an exception instead of failing.
    %
:- pred remove(T::in, set_tree234(T)::in, set_tree234(T)::out) is semidet.
:- pred det_remove(T::in, set_tree234(T)::in, set_tree234(T)::out) is det.

    % ‘remove_list(Xs, Set0, Set)’ is true iff Xs does not contain any
    % duplicates, ‘Set0’ contains every member of ‘Xs’, and ‘Set’ is the
    % relative complement of ‘Set0’ and the set containing only the mem-
bers of
    % ‘Xs’.
    %
    % The det_remove_list version throws an exception instead of failing.
    %
:- pred remove_list(list(T)::in, set_tree234(T)::in, set_tree234(T)::out)
    is semidet.
:- pred det_remove_list(list(T)::in, set_tree234(T)::in, set_tree234(T)::out)
    is det.

    % ‘remove_least(X, Set0, Set)’ is true iff ‘X’ is the least element in
    % ‘Set0’, and ‘Set’ is the set which contains all the elements of ‘Set0’
    % except ‘X’.
    %
:- pred remove_least(T::out, set_tree234(T)::in, set_tree234(T)::out)
    is semidet.

%---------------------------------------------------%
%
% Comparisons between sets.
%

    % ‘equal(SetA, SetB)’ is true iff ‘SetA’ and ‘SetB’ contain the same
    % elements.
    %
:- pred equal(set_tree234(T)::in, set_tree234(T)::in) is semidet.
```

```
    % ‘subset(SetA, SetB)’ is true iff ‘SetA’ is a subset of ‘SetB’.
    %
:- pred subset(set_tree234(T)::in, set_tree234(T)::in) is semidet.

    % ‘superset(SetA, SetB)’ is true iff ‘SetA’ is a superset of ‘SetB’.
    %
:- pred superset(set_tree234(T)::in, set_tree234(T)::in) is semidet.

%-----------------------------------------------%
%
% Operations on two or more sets.
%

    % ‘union(SetA, SetB) = Set’ is true iff ‘Set’ is the union of ‘SetA’ and
    % ‘SetB’.
    %
:- func union(set_tree234(T), set_tree234(T)) = set_tree234(T).
:- pred union(set_tree234(T)::in, set_tree234(T)::in, set_tree234(T)::out)
    is det.

    % ‘union_list(A, B)’ is true iff ‘B’ is the union of all the sets in ‘A’
    %
:- func union_list(list(set_tree234(T))) = set_tree234(T).
:- pred union_list(list(set_tree234(T))::in, set_tree234(T)::out) is det.

    % ‘power_union(A) = B’ is true iff ‘B’ is the union of
    % all the sets in ‘A’
    %
:- func power_union(set_tree234(set_tree234(T))) = set_tree234(T).
:- pred power_union(set_tree234(set_tree234(T))::in, set_tree234(T)::out)
    is det.

    % ‘intersect(SetA, SetB) = Set’ is true iff ‘Set’ is the intersection of
    % ‘SetA’ and ‘SetB’.
    %
:- func intersect(set_tree234(T), set_tree234(T)) = set_tree234(T).
:- pred intersect(set_tree234(T)::in, set_tree234(T)::in, set_tree234(T)::out)
    is det.

    % ‘intersect_list(A, B)’ is true iff ‘B’ is the intersection of all the
    % sets in ‘A’.
    %
:- func intersect_list(list(set_tree234(T))) = set_tree234(T).
:- pred intersect_list(list(set_tree234(T))::in, set_tree234(T)::out) is det.

    % ‘power_intersect(A, B)’ is true iff ‘B’ is the intersection of all the
    % sets in ‘A’.
```

```
    %
:- func power_intersect(set_tree234(set_tree234(T))) = set_tree234(T).
:- pred power_intersect(set_tree234(set_tree234(T))::in, set_tree234(T)::out)
    is det.

    % ‘difference(SetA, SetB, Set)’ is true iff ‘Set’ is the set contain-
ing all
    % the elements of ‘SetA’ except those that occur in ‘SetB’.
    %
:- func difference(set_tree234(T), set_tree234(T)) = set_tree234(T).
:- pred difference(set_tree234(T)::in, set_tree234(T)::in, set_tree234(T)::out)
    is det.

    % intersection_and_differences(SetA, SetB, InAandB, OnlyInA, OnlyInB):
    % Given SetA and SetB, return the elements that occur in both sets,
    % and those that occur only in one or the other.
    %
:- pred intersection_and_differences(set_tree234(T)::in, set_tree234(T)::in,
    set_tree234(T)::out, set_tree234(T)::out, set_tree234(T)::out) is det.

%-----------------------------------------------%
%
% Operations that divide a set into two parts.
%

    % divide(Pred, Set, TruePart, FalsePart):
    % TruePart consists of those elements of Set for which Pred succeeds;
    % FalsePart consists of those elements of Set for which Pred fails.
    %
:- pred divide(pred(T)::in(pred(in) is semidet),
    set_tree234(T)::in, set_tree234(T)::out, set_tree234(T)::out) is det.

    % divide_by_set(DivideBySet, Set, InPart, OutPart):
    % InPart consists of those elements of Set which are also in
    % DivideBySet; OutPart consists of those elements of which are
    % not in DivideBySet.
    %
:- pred divide_by_set(set_tree234(T)::in, set_tree234(T)::in,
    set_tree234(T)::out, set_tree234(T)::out) is det.

%-----------------------------------------------%
%
% Converting lists to sets.
%

    % ‘list_to_set(List) = Set’ is true iff ‘Set’ is the set containing
    % only the members of ‘List’.
```

```
    %
:- func list_to_set(list(T)) = set_tree234(T).
:- pred list_to_set(list(T)::in, set_tree234(T)::out) is det.

:- func from_list(list(T)) = set_tree234(T).
:- pred from_list(list(T)::in, set_tree234(T)::out) is det.

    % 'sorted_list_to_set(List) = Set' is true iff 'Set' is the set
    % containing only the members of 'List'. 'List' must be sorted
    % in ascending order and must not contain duplicates.
    %
:- func sorted_list_to_set(list(T)) = set_tree234(T).
:- pred sorted_list_to_set(list(T)::in, set_tree234(T)::out) is det.

    % 'rev_sorted_list_to_set(List) = Set' is true iff 'Set' is the set
    % containing only the members of 'List'. 'List' must be sorted
    % in descending order and must not contain duplicates.
    %
:- func rev_sorted_list_to_set(list(T)) = set_tree234(T).
:- pred rev_sorted_list_to_set(list(T)::in, set_tree234(T)::out) is det.

%-----------------------------------------------%
%
% Converting sets to lists.
%

    % 'to_sorted_list(Set) = List' is true iff 'List' is the list of all the
    % members of 'Set', in sorted order.
    %
:- func to_sorted_list(set_tree234(T)) = list(T).
:- pred to_sorted_list(set_tree234(T)::in, list(T)::out) is det.

%-----------------------------------------------%
%
% Converting between different kinds of sets.
%

    % 'from_set(Set)' returns a set_tree234 containing only
    % the members of 'Set'. Takes O(card(Set)) time and space.
    %
:- func from_set(set.set(T)) = set_tree234(T).

    % 'to_sorted_list(Set)' returns a set.set containing all the members of
    % 'Set', in sorted order. Takes O(card(Set)) time and space.
    %
:- func to_set(set_tree234(T)) = set.set(T).
```

```
%---------------------------------------------------%
%
% Counting.
%

    % 'count(Set, Count)' is true iff 'Set' has 'Count' elements.
    %
:- func count(set_tree234(T)) = int.

%---------------------------------------------------%
%
% Standard higher order functions on collections.
%

    % all_true(Pred, Set) succeeds iff Pred(Element) succeeds for all the
    % elements of Set.
    %
:- pred all_true(pred(T)::in(pred(in) is semidet),
    set_tree234(T)::in) is semidet.

    % Return the set of items for which the predicate succeeds.
    %
:- func filter(pred(T)::in(pred(in) is semidet),
    set_tree234(T)::in) = (set_tree234(T)::out) is det.
:- pred filter(pred(T)::in(pred(in) is semidet),
    set_tree234(T)::in, set_tree234(T)::out) is det.

    % Return the set of items for which the predicate succeeds,
    % and the set for which it fails.
    %
:- pred filter(pred(T)::in(pred(in) is semidet),
    set_tree234(T)::in, set_tree234(T)::out, set_tree234(T)::out) is det.

:- func filter_map(func(T1) = T2, set_tree234(T1)) = set_tree234(T2).
:- mode filter_map(func(in) = out is semidet, in) = out is det.
:- pred filter_map(pred(T1, T2)::in(pred(in, out) is semidet),
    set_tree234(T1)::in, set_tree234(T2)::out) is det.

:- func map(func(T1) = T2, set_tree234(T1)) = set_tree234(T2).
:- pred map(pred(T1, T2)::in(pred(in, out) is det),
    set_tree234(T1)::in, set_tree234(T2)::out) is det.

:- func fold(func(T1, T2) = T2, set_tree234(T1), T2) = T2.
:- pred fold(pred(T1, T2, T2), set_tree234(T1), T2, T2).
:- mode fold(pred(in, in, out) is det, in, in, out) is det.
:- mode fold(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode fold(pred(in, di, uo) is det, in, di, uo) is det.
```

```
:- mode fold(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode fold(pred(in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode fold(pred(in, di, uo) is semidet, in, di, uo) is semidet.

:- func foldl(func(T1, T2) = T2, set_tree234(T1), T2) = T2.
:- pred foldl(pred(T1, T2, T2), set_tree234(T1), T2, T2).
:- mode foldl(pred(in, in, out) is det, in, in, out) is det.
:- mode foldl(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode foldl(pred(in, di, uo) is det, in, di, uo) is det.
:- mode foldl(pred(in, in, out) is semidet, in, in, out)
    is semidet.
:- mode foldl(pred(in, mdi, muo) is semidet, in, mdi, muo)
    is semidet.
:- mode foldl(pred(in, di, uo) is semidet, in, di, uo)
    is semidet.

:- pred fold2(pred(T1, T2, T2, T3, T3), set_tree234(T1),
    T2, T2, T3, T3).
:- mode fold2(pred(in, in, out, in, out) is det, in,
    in, out, in, out) is det.
:- mode fold2(pred(in, in, out, mdi, muo) is det, in,
    in, out, mdi, muo) is det.
:- mode fold2(pred(in, in, out, di, uo) is det, in,
    in, out, di, uo) is det.
:- mode fold2(pred(in, in, out, in, out) is semidet, in,
    in, out, in, out) is semidet.
:- mode fold2(pred(in, in, out, mdi, muo) is semidet, in,
    in, out, mdi, muo) is semidet.
:- mode fold2(pred(in, in, out, di, uo) is semidet, in,
    in, out, di, uo) is semidet.

:- pred foldl2(pred(T1, T2, T2, T3, T3), set_tree234(T1),
    T2, T2, T3, T3).
:- mode foldl2(pred(in, in, out, in, out) is det, in,
    in, out, in, out) is det.
:- mode foldl2(pred(in, in, out, mdi, muo) is det, in,
    in, out, mdi, muo) is det.
:- mode foldl2(pred(in, in, out, di, uo) is det, in,
    in, out, di, uo) is det.
:- mode foldl2(pred(in, in, out, in, out) is semidet, in,
    in, out, in, out) is semidet.
:- mode foldl2(pred(in, in, out, mdi, muo) is semidet, in,
    in, out, mdi, muo) is semidet.
:- mode foldl2(pred(in, in, out, di, uo) is semidet, in,
    in, out, di, uo) is semidet.

:- pred fold3(
```

```
    pred(T1, T2, T2, T3, T3, T4, T4), set_tree234(T1),
    T2, T2, T3, T3, T4, T4).
:- mode fold3(pred(in, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out) is det.
:- mode fold3(pred(in, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, mdi, muo) is det.
:- mode fold3(pred(in, in, out, in, out, di, uo) is det, in,
    in, out, in, out, di, uo) is det.
:- mode fold3(pred(in, in, out, in, out, in, out) is semidet, in,
    in, out, in, out, in, out) is semidet.
:- mode fold3(pred(in, in, out, in, out, mdi, muo) is semidet, in,
    in, out, in, out, mdi, muo) is semidet.
:- mode fold3(pred(in, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, di, uo) is semidet.

:- pred foldl3(
    pred(T1, T2, T2, T3, T3, T4, T4), set_tree234(T1),
    T2, T2, T3, T3, T4, T4).
:- mode foldl3(pred(in, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out) is det.
:- mode foldl3(pred(in, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, mdi, muo) is det.
:- mode foldl3(pred(in, in, out, in, out, di, uo) is det, in,
    in, out, in, out, di, uo) is det.
:- mode foldl3(pred(in, in, out, in, out, in, out) is semidet, in,
    in, out, in, out, in, out) is semidet.
:- mode foldl3(pred(in, in, out, in, out, mdi, muo) is semidet, in,
    in, out, in, out, mdi, muo) is semidet.
:- mode foldl3(pred(in, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, di, uo) is semidet.

:- pred fold4(
    pred(T1, T2, T2, T3, T3, T4, T4, T5, T5), set_tree234(T1),
    T2, T2, T3, T3, T4, T4, T5, T5).
:- mode fold4(pred(in, in, out, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out, in, out) is det.
:- mode fold4(pred(in, in, out, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, in, out, mdi, muo) is det.
:- mode fold4(pred(in, in, out, in, out, in, out, di, uo) is det,
    in, in, out, in, out, in, out, di, uo) is det.
:- mode fold4(
    pred(in, in, out, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out, in, out) is semidet.
:- mode fold4(
    pred(in, in, out, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode fold4(
```

```
    pred(in, in, out, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, in, out, di, uo) is semidet.

:- pred foldl4(
    pred(T1, T2, T2, T3, T3, T4, T4, T5, T5), set_tree234(T1),
    T2, T2, T3, T3, T4, T4, T5, T5).
:- mode foldl4(pred(in, in, out, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out, in, out) is det.
:- mode foldl4(pred(in, in, out, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, in, out, mdi, muo) is det.
:- mode foldl4(pred(in, in, out, in, out, in, out, di, uo) is det,
    in, in, out, in, out, in, out, di, uo) is det.
:- mode foldl4(
    pred(in, in, out, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out, in, out) is semidet.
:- mode foldl4(
    pred(in, in, out, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode foldl4(
    pred(in, in, out, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, in, out, di, uo) is semidet.

:- pred fold5(
    pred(T1, T2, T2, T3, T3, T4, T4, T5, T5, T6, T6),
    set_tree234(T1), T2, T2, T3, T3, T4, T4, T5, T5, T6, T6).
:- mode fold5(
    pred(in, in, out, in, out, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out, in, out, in, out) is det.
:- mode fold5(
    pred(in, in, out, in, out, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, in, out, in, out, mdi, muo) is det.
:- mode fold5(
    pred(in, in, out, in, out, in, out, in, out, di, uo) is det,
    in, in, out, in, out, in, out, in, out, di, uo) is det.
:- mode fold5(
    pred(in, in, out, in, out, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out, in, out, in, out) is semidet.
:- mode fold5(
    pred(in, in, out, in, out, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode fold5(
    pred(in, in, out, in, out, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, in, out, in, out, di, uo) is semidet.

:- pred foldl5(
    pred(T1, T2, T2, T3, T3, T4, T4, T5, T5, T6, T6),
    set_tree234(T1), T2, T2, T3, T3, T4, T4, T5, T5, T6, T6).
```

```
:- mode foldl5(
    pred(in, in, out, in, out, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out, in, out, in, out) is det.
:- mode foldl5(
    pred(in, in, out, in, out, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, in, out, in, out, mdi, muo) is det.
:- mode foldl5(
    pred(in, in, out, in, out, in, out, in, out, di, uo) is det,
    in, in, out, in, out, in, out, in, out, di, uo) is det.
:- mode foldl5(
    pred(in, in, out, in, out, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out, in, out, in, out) is semidet.
:- mode foldl5(
    pred(in, in, out, in, out, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode foldl5(
    pred(in, in, out, in, out, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, in, out, in, out, di, uo) is semidet.

:- pred fold6(
    pred(T1, T2, T2, T3, T3, T4, T4, T5, T5, T6, T6, T7, T7),
    set_tree234(T1), T2, T2, T3, T3, T4, T4, T5, T5, T6, T6, T7, T7).
:- mode fold6(
    pred(in, in, out, in, out, in, out, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out, in, out, in, out, in, out) is det.
:- mode fold6(
    pred(in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is det.
:- mode fold6(
    pred(in, in, out, in, out, in, out, in, out, in, out, di, uo) is det,
    in, in, out, in, out, in, out, in, out, in, out, di, uo) is det.
:- mode fold6(
    pred(in, in, out, in, out, in, out, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out, in, out, in, out, in, out) is semidet.
:- mode fold6(
    pred(in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode fold6(
    pred(in, in, out, in, out, in, out, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, in, out, in, out, in, out, di, uo) is semidet.

:- pred foldl6(
    pred(T1, T2, T2, T3, T3, T4, T4, T5, T5, T6, T6, T7, T7),
    set_tree234(T1), T2, T2, T3, T3, T4, T4, T5, T5, T6, T6, T7, T7).
:- mode foldl6(
    pred(in, in, out, in, out, in, out, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out, in, out, in, out, in, out) is det.
```

```
:- mode foldl6(
    pred(in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is det.
:- mode foldl6(
    pred(in, in, out, in, out, in, out, in, out, in, out, di, uo) is det,
    in, in, out, in, out, in, out, in, out, in, out, di, uo) is det.
:- mode foldl6(
    pred(in, in, out, in, out, in, out, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out, in, out, in, out, in, out) is semidet.
:- mode foldl6(
    pred(in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode foldl6(
    pred(in, in, out, in, out, in, out, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, in, out, in, out, in, out, di, uo) is semidet.

%---------------------------------------------------%
%---------------------------------------------------%
```

76 set_unordlist

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 1995-1997,1999-2002, 2004-2006, 2010-2012 The University of Melbourne.
% Copyright (C) 2014-2015, 2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: set_unordlist.m.
% Main authors: conway, fjh.
% Stability: medium.
%
% This file contains a 'set' ADT.
% Sets are implemented here as unsorted lists, which may contain duplicates.
%
%---------------------------------------------------%
%---------------------------------------------------%

:- module set_unordlist.
:- interface.

:- import_module bool.
:- import_module list.
```

```
%---------------------------------------------------%

:- type set_unordlist(_T).

%---------------------------------------------------%
%
% Initial creation of sets.
%

    % 'init(Set)' is true iff 'Set' is an empty set.
    %
:- func init = set_unordlist(T).
:- pred init(set_unordlist(_T)::uo) is det.

    % 'singleton_set(Elem, Set)' is true iff 'Set' is the set
    % containing just the single element 'Elem'.
    %
:- pred singleton_set(T, set_unordlist(T)).
:- mode singleton_set(in, out) is det.
:- mode singleton_set(in, in) is semidet.     % Implied.
:- mode singleton_set(out, in) is semidet.

:- func make_singleton_set(T) = set_unordlist(T).

%---------------------------------------------------%
%
% Emptiness and singleton-ness tests.
%

    % 'empty(Set)' is true iff 'Set' is an empty set.
    % 'is_empty' is a synonym of 'empty'.
    %
:- pred empty(set_unordlist(_T)::in) is semidet.
:- pred is_empty(set_unordlist(_T)::in) is semidet.
:- pragma obsolete(empty/1, [is_empty/1]).

    % 'non_empty(Set)' is true iff 'Set' is not an empty set.
    % 'is_non_empty' is a synonym of 'non_empty'.
    %
:- pred non_empty(set_unordlist(_T)::in) is semidet.
:- pred is_non_empty(set_unordlist(_T)::in) is semidet.
:- pragma obsolete(non_empty/1, [is_non_empty/1]).

:- pred is_singleton(set_unordlist(T)::in, T::out) is semidet.

%---------------------------------------------------%
%
```

```
% Membership tests.
%

    % 'member(X, Set)' is true iff 'X' is a member of 'Set'.
    %
:- pred member(T, set_unordlist(T)).
:- mode member(in, in) is semidet.
:- mode member(out, in) is nondet.

    % 'is_member(X, Set, Result)' returns 'Result = yes' iff 'X' is a mem-
ber of
    % 'Set'.
    %
:- pred is_member(T::in, set_unordlist(T)::in, bool::out) is det.

    % 'contains(Set, X)' is true iff 'X' is a member of 'Set'.
    %
:- pred contains(set_unordlist(T)::in, T::in) is semidet.

%-----------------------------------------------%
%
% Insertions and deletions.
%

    % 'insert(X, Set0, Set)' is true iff 'Set' is the union of 'Set0' and the
    % set containing only 'X'.
    %
:- func insert(set_unordlist(T), T) = set_unordlist(T).
:- pred insert(T, set_unordlist(T), set_unordlist(T)).
:- mode insert(di, di, uo) is det.
:- mode insert(in, in, out) is det.

    % 'insert_new(X, Set0, Set)' is true iff 'Set0' does not contain 'X', and
    % 'Set' is the union of 'Set0' and the set containing only 'X'.
    %
:- pred insert_new(T::in, set_unordlist(T)::in, set_unordlist(T)::out)
    is semidet.

    % 'insert_list(Xs, Set0, Set)' is true iff 'Set' is the
    % union of 'Set0' and the set containing only the members of 'Xs'.
    %
:- func insert_list(set_unordlist(T), list(T))
    = set_unordlist(T).
:- pred insert_list(list(T)::in,
    set_unordlist(T)::in, set_unordlist(T)::out) is det.

    % 'delete(X, Set0, Set)' is true iff 'Set' is the relative complement of
```

```
    % 'Set0' and the set containing only 'X', i.e. if 'Set' is the set which
    % contains all the elements of 'Set0' except 'X'.
    %
:- func delete(set_unordlist(T), T) = set_unordlist(T).
:- pred delete(T, set_unordlist(T), set_unordlist(T)).
:- mode delete(in, di, uo) is det.
:- mode delete(in, in, out) is det.

    % 'delete_list(Xs, Set0, Set)' is true iff 'Set' is the relative complement
    % of 'Set0' and the set containing only the members of 'Xs'.
    %
:- func delete_list(set_unordlist(T), list(T)) = set_unordlist(T).
:- pred delete_list(list(T)::in, set_unordlist(T)::in, set_unordlist(T)::out)
    is det.

    % 'remove(X, Set0, Set)' is true iff 'Set0' contains 'X',
    % and 'Set' is the relative complement of 'Set0' and the set
    % containing only 'X', i.e. if 'Set' is the set which contains
    % all the elements of 'Set0' except 'X'.
    %
    % The det_remove version throws an exception instead of failing.
    %
:- pred remove(T::in, set_unordlist(T)::in, set_unordlist(T)::out) is semidet.
:- pred det_remove(T::in, set_unordlist(T)::in, set_unordlist(T)::out) is det.

    % 'remove_list(Xs, Set0, Set)' is true iff Xs does not contain any
    % duplicates, 'Set0' contains every member of 'Xs', and 'Set' is the
    % relative complement of 'Set0' and the set containing only the mem-
bers of
    % 'Xs'.
    %
    % The det_remove_list version throws an exception instead of failing.
    %
:- pred remove_list(list(T)::in,
    set_unordlist(T)::in, set_unordlist(T)::out) is semidet.
:- pred det_remove_list(list(T)::in,
    set_unordlist(T)::in, set_unordlist(T)::out) is det.

    % 'remove_least(X, Set0, Set)' is true iff 'X' is the least element in
    % 'Set0', and 'Set' is the set which contains all the elements of 'Set0'
    % except 'X'.
    %
:- pred remove_least(T::out,
    set_unordlist(T)::in, set_unordlist(T)::out) is semidet.

%---------------------------------------------------%
%
```

```
% Comparisons between sets.
%

    % ‘equal(SetA, SetB)’ is true iff ‘SetA’ and ‘SetB’ contain the same
    % elements.
    %
:- pred equal(set_unordlist(T)::in, set_unordlist(T)::in) is semidet.

    % ‘subset(SetA, SetB)’ is true iff ‘SetA’ is a subset of ‘SetB’.
    %
:- pred subset(set_unordlist(T)::in, set_unordlist(T)::in) is semidet.

    % ‘superset(SetA, SetB)’ is true iff ‘SetA’ is a superset of ‘SetB’.
    %
:- pred superset(set_unordlist(T)::in, set_unordlist(T)::in) is semidet.

%---------------------------------------------%
%
% Operations on two or more sets.
%

    % ‘union(SetA, SetB, Set)’ is true iff ‘Set’ is the union of ‘SetA’ and
    % ‘SetB’. If the sets are known to be of different sizes, then for
    % efficiency make ‘SetA’ the larger of the two.
    %
:- func union(set_unordlist(T), set_unordlist(T)) = set_unordlist(T).
:- pred union(set_unordlist(T)::in, set_unordlist(T)::in,
    set_unordlist(T)::out) is det.

    % ‘union_list(A) = B’ is true iff ‘B’ is the union of all the sets in ‘A’
    %
:- func union_list(list(set_unordlist(T))) = set_unordlist(T).

    % ‘power_union(A, B)’ is true iff ‘B’ is the union of all the sets in ‘A’
    %
:- func power_union(set_unordlist(set_unordlist(T))) = set_unordlist(T).
:- pred power_union(set_unordlist(set_unordlist(T))::in,
    set_unordlist(T)::out) is det.

    % ‘intersect(SetA, SetB, Set)’ is true iff ‘Set’ is the intersection of
    % ‘SetA’ and ‘SetB’.
    %
:- func intersect(set_unordlist(T), set_unordlist(T)) = set_unordlist(T).
:- pred intersect(set_unordlist(T)::in, set_unordlist(T)::in,
    set_unordlist(T)::out) is det.

    % ‘intersect_list(A, B)’ is true iff ‘B’ is the intersection of all the
```

```
    % sets in 'A'
    %
:- func intersect_list(list(set_unordlist(T))) = set_unordlist(T).

    % 'power_intersect(A, B)' is true iff 'B' is the intersection of all the
    % sets in 'A'
    %
:- func power_intersect(set_unordlist(set_unordlist(T))) = set_unordlist(T).
:- pred power_intersect(set_unordlist(set_unordlist(T))::in,
    set_unordlist(T)::out) is det.

    % 'difference(SetA, SetB, Set)' is true iff 'Set' is the set contain-
ing all
    % the elements of 'SetA' except those that occur in 'SetB'
    %
:- func difference(set_unordlist(T), set_unordlist(T)) = set_unordlist(T).
:- pred difference(set_unordlist(T)::in, set_unordlist(T)::in,
    set_unordlist(T)::out) is det.

%---------------------------------------------------%
%
% Operations that divide a set into two parts.
%

    % divide(Pred, Set, TruePart, FalsePart):
    % TruePart consists of those elements of Set for which Pred succeeds;
    % FalsePart consists of those elements of Set for which Pred fails.
    % NOTE: this is the same as filter/4.
    %
:- pred divide(pred(T)::in(pred(in) is semidet),
    set_unordlist(T)::in, set_unordlist(T)::out, set_unordlist(T)::out) is det.

%---------------------------------------------------%
%
% Converting lists to sets.
%

    % 'list_to_set(List, Set)' is true iff 'Set' is the set
    % containing only the members of 'List'.
    %
:- func list_to_set(list(T)) = set_unordlist(T).
:- pred list_to_set(list(T)::in, set_unordlist(T)::out) is det.

    % A synonym for list_to_set/1.
    %
:- func from_list(list(T)) = set_unordlist(T).
```

```
    % ‘sorted_list_to_set(List, Set)’ is true iff ‘Set’ is the set
    % containing only the members of ‘List’. ‘List’ must be sorted
    % in ascending order.
    %
:- func sorted_list_to_set(list(T)) = set_unordlist(T).
:- pred sorted_list_to_set(list(T)::in, set_unordlist(T)::out) is det.

    % A synonym for sorted_list_to_set/1.
    %
:- func from_sorted_list(list(T)) = set_unordlist(T).

    % ‘rev_sorted_list_to_set(List, Set)’ is true iff ‘Set’ is the set
    % containing only the members of ‘List’. ‘List’ must be sorted
    % in descending order.
    %
:- func rev_sorted_list_to_set(list(T)) = set_unordlist(T).
:- pred rev_sorted_list_to_set(list(T)::in, set_unordlist(T)::out) is det.

%---------------------------------------------------%
%
% Converting sets to lists.
%

    % ‘to_sorted_list(Set, List)’ is true iff ‘List’ is the list of all the
    % members of ‘Set’, in sorted order.
    %
:- func to_sorted_list(set_unordlist(T)) = list(T).
:- pred to_sorted_list(set_unordlist(T)::in, list(T)::out) is det.

%---------------------------------------------------%
%
% Counting.
%

:- func count(set_unordlist(T)) = int.
:- pred count(set_unordlist(T)::in, int::out) is det.

%---------------------------------------------------%
%
% Standard higher order functions on collections.
%

    % all_true(Pred, Set) succeeds iff Pred(Element) succeeds for all the
    % elements of Set.
    %
:- pred all_true(pred(T)::in(pred(in) is semidet),
    set_unordlist(T)::in) is semidet.
```

```
    % Return the set of items for which the predicate succeeds.
    %
:- pred filter(pred(T)::in(pred(in) is semidet),
    set_unordlist(T)::in, set_unordlist(T)::out) is det.

    % Return the set of items for which the predicate succeeds,
    % and the set for which it fails.
    %
:- pred filter(pred(T)::in(pred(in) is semidet),
    set_unordlist(T)::in, set_unordlist(T)::out, set_unordlist(T)::out) is det.

:- func filter_map(func(T1) = T2, set_unordlist(T1)) = set_unordlist(T2).
:- mode filter_map(func(in) = out is semidet, in) = out is det.

:- func map(func(T1) = T2, set_unordlist(T1)) = set_unordlist(T2).

:- func fold(func(T1, T2) = T2, set_unordlist(T1), T2) = T2.
:- pred fold(pred(T1, T2, T2), set_unordlist(T1), T2, T2).
:- mode fold(pred(in, in, out) is det, in, in, out) is det.
:- mode fold(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode fold(pred(in, di, uo) is det, in, di, uo) is det.
:- mode fold(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode fold(pred(in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode fold(pred(in, di, uo) is semidet, in, di, uo) is semidet.

:- pred fold2(pred(T1, T2, T2, T3, T3), set_unordlist(T1),
    T2, T2, T3, T3).
:- mode fold2(pred(in, in, out, in, out) is det, in,
    in, out, in, out) is det.
:- mode fold2(pred(in, in, out, mdi, muo) is det, in,
    in, out, mdi, muo) is det.
:- mode fold2(pred(in, in, out, di, uo) is det, in,
    in, out, di, uo) is det.
:- mode fold2(pred(in, in, out, in, out) is semidet, in,
    in, out, in, out) is semidet.
:- mode fold2(pred(in, in, out, mdi, muo) is semidet, in,
    in, out, mdi, muo) is semidet.
:- mode fold2(pred(in, in, out, di, uo) is semidet, in,
    in, out, di, uo) is semidet.

:- pred fold3(pred(T1, T2, T2, T3, T3, T4, T4),
    set_unordlist(T1), T2, T2, T3, T3, T4, T4).
:- mode fold3(pred(in, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out) is det.
:- mode fold3(pred(in, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, mdi, muo) is det.
```

```
:- mode fold3(pred(in, in, out, in, out, di, uo) is det, in,
    in, out, in, out, di, uo) is det.
:- mode fold3(pred(in, in, out, in, out, in, out) is semidet, in,
    in, out, in, out, in, out) is semidet.
:- mode fold3(pred(in, in, out, in, out, mdi, muo) is semidet, in,
    in, out, in, out, mdi, muo) is semidet.
:- mode fold3(pred(in, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, di, uo) is semidet.

:- pred fold4(pred(T1, T2, T2, T3, T3, T4, T4, T5, T5),
    set_unordlist(T1), T2, T2, T3, T3, T4, T4, T5, T5).
:- mode fold4(
    pred(in, in, out, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out, in, out) is det.
:- mode fold4(
    pred(in, in, out, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, in, out, mdi, muo) is det.
:- mode fold4(
    pred(in, in, out, in, out, in, out, di, uo) is det, in,
    in, out, in, out, in, out, di, uo) is det.
:- mode fold4(
    pred(in, in, out, in, out, in, out, in, out) is semidet, in,
    in, out, in, out, in, out, in, out) is semidet.
:- mode fold4(
    pred(in, in, out, in, out, in, out, mdi, muo) is semidet, in,
    in, out, in, out, in, out, mdi, muo) is semidet.
:- mode fold4(
    pred(in, in, out, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, in, out, di, uo) is semidet.

:- pred fold5(
    pred(T1, T2, T2, T3, T3, T4, T4, T5, T5, T6, T6),
    set_unordlist(T1), T2, T2, T3, T3, T4, T4, T5, T5, T6, T6).
:- mode fold5(
    pred(in, in, out, in, out, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out, in, out, in, out) is det.
:- mode fold5(
    pred(in, in, out, in, out, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, in, out, in, out, mdi, muo) is det.
:- mode fold5(
    pred(in, in, out, in, out, in, out, in, out, di, uo) is det, in,
    in, out, in, out, in, out, in, out, di, uo) is det.
:- mode fold5(
    pred(in, in, out, in, out, in, out, in, out, in, out) is semidet, in,
    in, out, in, out, in, out, in, out, in, out) is semidet.
:- mode fold5(
    pred(in, in, out, in, out, in, out, in, out, mdi, muo) is semidet, in,
```

```
    in, out, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode fold5(
    pred(in, in, out, in, out, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, in, out, in, out, di, uo) is semidet.

:- pred fold6(
    pred(T1, T2, T2, T3, T3, T4, T4, T5, T5, T6, T6, T7, T7),
    set_unordlist(T1), T2, T2, T3, T3, T4, T4, T5, T5, T6, T6, T7, T7).
:- mode fold6(
    pred(in, in, out, in, out, in, out, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out, in, out, in, out, in, out) is det.
:- mode fold6(
    pred(in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is det.
:- mode fold6(
    pred(in, in, out, in, out, in, out, in, out, in, out, di, uo) is det,
    in, in, out, in, out, in, out, in, out, in, out, di, uo) is det.
:- mode fold6(
    pred(in, in, out, in, out, in, out, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out, in, out, in, out, in, out) is semidet.
:- mode fold6(
    pred(in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode fold6(
    pred(in, in, out, in, out, in, out, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, in, out, in, out, in, out, di, uo) is semidet.

%-----------------------------------------------------%
%-----------------------------------------------------%
```

77 solutions

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 1994-2007 The University of Melbourne.
% Copyright (C) 2014-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: solutions.m.
% Main author: fjh.
% Stability: medium.
%
%-----------------------------------------------------%
```

```
%-----------------------------------------------------%

:- module solutions.
:- interface.

:- import_module bool.
:- import_module list.
:- import_module set.

%-----------------------------------------------------%

    % solutions/2 collects all the solutions to a predicate and returns
    % them as a list in sorted order, with duplicates removed.
    % solutions_set/2 returns them as a set. unsorted_solutions/2 returns
    % them as an unsorted list with possible duplicates; since there are
    % an infinite number of such lists, this must be called from a context
    % in which only a single solution is required.
    %
:- pred solutions(pred(T), list(T)).
:- mode solutions(pred(out) is multi, out(non_empty_list)) is det.
:- mode solutions(pred(out) is nondet, out) is det.

:- func solutions(pred(T)) = list(T).
:- mode solutions(pred(out) is multi) = out(non_empty_list) is det.
:- mode solutions(pred(out) is nondet) = out is det.

:- func solutions_set(pred(T)) = set(T).
:- mode solutions_set(pred(out) is multi) = out is det.
:- mode solutions_set(pred(out) is nondet) = out is det.

:- pred solutions_set(pred(T), set(T)).
:- mode solutions_set(pred(out) is multi, out) is det.
:- mode solutions_set(pred(out) is nondet, out) is det.

:- pred unsorted_solutions(pred(T), list(T)).
:- mode unsorted_solutions(pred(out) is multi, out(non_empty_list))
    is cc_multi.
:- mode unsorted_solutions(pred(out) is nondet, out)
    is cc_multi.

:- func aggregate(pred(T), func(T, U) = U, U) = U.
:- mode aggregate(pred(out) is multi, func(in, in) = out is det, in)
    = out is det.
:- mode aggregate(pred(out) is nondet, func(in, in) = out is det, in)
    = out is det.

    % aggregate/4 generates all the solutions to a predicate,
```

```
    % sorts them and removes duplicates, then applies an accumulator
    % predicate to each solution in turn:
    %
    % aggregate(Generator, AccumulatorPred, Acc0, Acc) <=>
    %   solutions(Generator, Solutions),
    %   list.foldl(AccumulatorPred, Solutions, Acc0, Acc).
    %
:- pred aggregate(pred(T), pred(T, U, U), U, U).
:- mode aggregate(pred(out) is multi, pred(in, in, out) is det,
    in, out) is det.
:- mode aggregate(pred(out) is multi, pred(in, di, uo) is det,
    di, uo) is det.
:- mode aggregate(pred(out) is nondet, pred(in, di, uo) is det,
    di, uo) is det.
:- mode aggregate(pred(out) is nondet, pred(in, in, out) is det,
    in, out) is det.

    % aggregate2/6 generates all the solutions to a predicate,
    % sorts them and removes duplicates, then applies an accumulator
    % predicate to each solution in turn:
    %
    % aggregate2(Generator, AccumulatorPred, AccA0, AccA, AccB0, AccB) <=>
    %   solutions(Generator, Solutions),
    %   list.foldl2(AccumulatorPred, Solutions, AccA0, AccA, AccB0, AccB).
    %
:- pred aggregate2(pred(T), pred(T, U, U, V, V), U, U, V, V).
:- mode aggregate2(pred(out) is multi, pred(in, in, out, in, out) is det,
    in, out, in, out) is det.
:- mode aggregate2(pred(out) is multi, pred(in, in, out, di, uo) is det,
    in, out, di, uo) is det.
:- mode aggregate2(pred(out) is nondet, pred(in, in, out, di, uo) is det,
    in, out, di, uo) is det.
:- mode aggregate2(pred(out) is nondet, pred(in, in, out, in, out) is det,
    in, out, in, out) is det.

    % unsorted_aggregate/4 generates all the solutions to a predicate
    % and applies an accumulator predicate to each solution in turn.
    % Declaratively, the specification is as follows:
    %
    % unsorted_aggregate(Generator, AccumulatorPred, Acc0, Acc) <=>
    %   unsorted_solutions(Generator, Solutions),
    %   list.foldl(AccumulatorPred, Solutions, Acc0, Acc).
    %
    % Operationally, however, unsorted_aggregate/4 will call the
    % AccumulatorPred for each solution as it is obtained, rather than
    % first building a list of all the solutions.
    %
```

```
:- pred unsorted_aggregate(pred(T), pred(T, U, U), U, U).
:- mode unsorted_aggregate(pred(out) is multi, pred(in, in, out) is det,
    in, out) is cc_multi.
:- mode unsorted_aggregate(pred(out) is multi, pred(in, in, out) is cc_multi,
    in, out) is cc_multi.
:- mode unsorted_aggregate(pred(out) is multi, pred(in, di, uo) is det,
    di, uo) is cc_multi.
:- mode unsorted_aggregate(pred(out) is multi, pred(in, di, uo) is cc_multi,
    di, uo) is cc_multi.
:- mode unsorted_aggregate(pred(muo) is multi, pred(mdi, di, uo) is det,
    di, uo) is cc_multi.
:- mode unsorted_aggregate(pred(out) is nondet, pred(in, di, uo) is det,
    di, uo) is cc_multi.
:- mode unsorted_aggregate(pred(out) is nondet, pred(in, di, uo) is cc_multi,
    di, uo) is cc_multi.
:- mode unsorted_aggregate(pred(out) is nondet, pred(in, in, out) is det,
    in, out) is cc_multi.
:- mode unsorted_aggregate(pred(out) is nondet, pred(in, in, out) is cc_multi,
    in, out) is cc_multi.
:- mode unsorted_aggregate(pred(muo) is nondet, pred(mdi, di, uo) is det,
    di, uo) is cc_multi.

    % unsorted_aggregate2/6 generates all the solutions to a predicate
    % and applies an accumulator predicate to each solution in turn.
    % Declaratively, the specification is as follows:
    %
    % unsorted_aggregate2(Generator, AccumulatorPred, !Acc1, !Acc2) <=>
    %   unsorted_solutions(Generator, Solutions),
    %   list.foldl2(AccumulatorPred, Solutions, !Acc1, !Acc2).
    %
    % Operationally, however, unsorted_aggregate2/6 will call the
    % AccumulatorPred for each solution as it is obtained, rather than
    % first building a list of all the solutions.
    %
:- pred unsorted_aggregate2(pred(T), pred(T, U, U, V, V), U, U, V, V).
:- mode unsorted_aggregate2(pred(out) is multi,
    pred(in, in, out, in, out) is det, in, out, in, out) is cc_multi.
:- mode unsorted_aggregate2(pred(out) is multi,
    pred(in, in, out, in, out) is cc_multi, in, out, in, out) is cc_multi.
:- mode unsorted_aggregate2(pred(out) is multi,
    pred(in, in, out, di, uo) is det, in, out, di, uo) is cc_multi.
:- mode unsorted_aggregate2(pred(out) is multi,
    pred(in, in, out, di, uo) is cc_multi, in, out, di, uo) is cc_multi.
:- mode unsorted_aggregate2(pred(out) is nondet,
    pred(in, in, out, in, out) is det, in, out, in, out) is cc_multi.
:- mode unsorted_aggregate2(pred(out) is nondet,
    pred(in, in, out, in, out) is cc_multi, in, out, in, out) is cc_multi.
```

```
:- mode unsorted_aggregate2(pred(out) is nondet,
    pred(in, in, out, di, uo) is det, in, out, di, uo) is cc_multi.
:- mode unsorted_aggregate2(pred(out) is nondet,
    pred(in, in, out, di, uo) is cc_multi, in, out, di, uo) is cc_multi.

    % This is a generalization of unsorted_aggregate which allows the
    % iteration to stop before all solutions have been found.
    % Declaratively, the specification is as follows:
    %
    %   do_while(Generator, Filter, !Acc) :-
    %       unsorted_solutions(Generator, Solutions),
    %       do_while_2(Solutions, Filter, !Acc).
    %
    %   do_while_2([], _, !Acc).
    %   do_while_2([X | Xs], Filter, !Acc) :-
    %       Filter(X, More, !Acc),
    %       (
    %           More = yes,
    %           do_while_2(Xs, Filter, !Acc)
    %       ;
    %           More = no
    %       ).
    %
    % Operationally, however, do_while/4 will call the Filter
    % predicate for each solution as it is obtained, rather than
    % first building a list of all the solutions.
    %
:- pred do_while(pred(T), pred(T, bool, T2, T2), T2, T2).
:- mode do_while(pred(out) is multi, pred(in, out, in, out) is det,
    in, out) is cc_multi.
:- mode do_while(pred(out) is multi, pred(in, out, di, uo) is det,
    di, uo) is cc_multi.
:- mode do_while(pred(out) is multi, pred(in, out, di, uo) is cc_multi,
    di, uo) is cc_multi.
:- mode do_while(pred(out) is nondet, pred(in, out, in, out) is det,
    in, out) is cc_multi.
:- mode do_while(pred(out) is nondet, pred(in, out, di, uo) is det,
    di, uo) is cc_multi.
:- mode do_while(pred(out) is nondet, pred(in, out, di, uo) is cc_multi,
    di, uo) is cc_multi.

%-----------------------------------------------%
%-----------------------------------------------%
```

78 sparse_bitset

```
%-----------------------------------------------------%
% vim: ts=4 sw=4 et ft=mercury
%-----------------------------------------------------%
% Copyright (C) 2000-2007, 2011-2012 The University of Melbourne.
% Copyright (C) 2014-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: sparse_bitset.m.
% Author: stayl.
% Stability: medium.
%
% This module provides an ADT for storing sets of integers.
% If the integers stored are closely grouped, a sparse_bitset
% is much more compact than the representation provided by set.m,
% and the operations will be much faster.
%
% Efficiency notes:
%
% A sparse bitset is represented as a sorted list of pairs of integers.
% For a pair `Offset - Bits', `Offset' is a multiple of `int.bits_per_int'.
% The bits of `Bits' describe which of the elements of the range
% `Offset' .. `Offset + bits_per_int - 1' are in the set.
% Pairs with the same value of `Offset' are merged.
% Pairs in which `Bits' is zero are removed.
%
% The values of `Offset' in the list need not be *contiguous* multiples
% of `bits_per_int', hence the name *sparse* bitset.
%
% A sparse_bitset is suitable for storing sets of integers which
% can be represented using only a few `Offset - Bits' pairs.
% In the worst case, where the integers stored are not closely grouped,
% a sparse_bitset will take more memory than an ordinary set, but
% the operations should not be too much slower.
%
% In the asymptotic complexities of the operations below,
% `rep_size(Set)' is the number of pairs needed to represent `Set',
% and `card(Set)' is the number of elements in `Set'.
%
%-----------------------------------------------------%

:- module sparse_bitset.
:- interface.

:- import_module enum.
```

```
:- import_module list.
:- import_module term.

:- use_module set.

%-----------------------------------------------------%

:- type sparse_bitset(T). % <= enum(T).

%-----------------------------------------------------%
%
% Initial creation of sets.
%

    % Return an empty set.
    %
:- func init = sparse_bitset(T).
:- pred init(sparse_bitset(T)::out) is det.

    % Note: set.m contains the reverse mode of this predicate, but it is
    % difficult to implement both modes using the representation in this
    % module.
    %
:- pred singleton_set(sparse_bitset(T)::out, T::in) is det <= enum(T).

    % 'make_singleton_set(Elem)' returns a set containing just the single
    % element 'Elem'.
    %
:- func make_singleton_set(T) = sparse_bitset(T) <= enum(T).

%-----------------------------------------------------%
%
% Emptiness and singleton-ness tests.
%

:- pred empty(sparse_bitset(T)).
:- mode empty(in) is semidet.
:- mode empty(out) is det.
:- pragma obsolete(empty/1, [init/0, is_empty/1]).

:- pred is_empty(sparse_bitset(T)::in) is semidet.

:- pred is_non_empty(sparse_bitset(T)::in) is semidet.

    % Is the given set a singleton, and if yes, what is the element?
    %
:- pred is_singleton(sparse_bitset(T)::in, T::out) is semidet <= enum(T).
```

```
%---------------------------------------------------%
%
% Membership tests.
%

    % 'member(X, Set)' is true iff 'X' is a member of 'Set'.
    % Takes O(rep_size(Set)) time.
    %
:- pred member(T, sparse_bitset(T)) <= enum(T).
:- mode member(in, in) is semidet.
:- mode member(out, in) is nondet.

    % 'contains(Set, X)' is true iff 'X' is a member of 'Set'.
    % Takes O(rep_size(Set)) time.
    %
:- pred contains(sparse_bitset(T)::in, T::in) is semidet <= enum(T).

%---------------------------------------------------%
%
% Insertions and deletions.
%

    % 'insert(Set, X)' returns the union of 'Set' and the set containing
    % only 'X'. Takes O(rep_size(Set)) time and space.
    %
:- func insert(sparse_bitset(T), T) = sparse_bitset(T) <= enum(T).
:- pred insert(T::in, sparse_bitset(T)::in, sparse_bitset(T)::out)
    is det <= enum(T).

    % 'insert_new(X, Set0, Set)' returns the union of 'Set' and the set
    % containing only 'X' if 'Set0' does not already contain 'X'; if it does,
    % it fails. Takes O(rep_size(Set)) time and space.
    %
:- pred insert_new(T::in, sparse_bitset(T)::in, sparse_bitset(T)::out)
    is semidet <= enum(T).

    % 'insert_list(Set, X)' returns the union of 'Set' and the set containing
    % only the members of 'X'. Same as 'union(Set, list_to_set(X))', but may be
    % more efficient.
    %
:- func insert_list(sparse_bitset(T), list(T)) = sparse_bitset(T) <= enum(T).
:- pred insert_list(list(T)::in, sparse_bitset(T)::in, sparse_bitset(T)::out)
    is det <= enum(T).

%---------------------------------------------------%
```

```
    % 'delete(Set, X)' returns the difference of 'Set' and the set containing
    % only 'X'. Takes O(rep_size(Set)) time and space.
    %
:- func delete(sparse_bitset(T), T) = sparse_bitset(T) <= enum(T).
:- pred delete(T::in, sparse_bitset(T)::in, sparse_bitset(T)::out)
    is det <= enum(T).

    % 'delete_list(Set, X)' returns the difference of 'Set' and the set
    % containing only the members of 'X'. Same as
    % 'difference(Set, list_to_set(X))', but may be more efficient.
    %
:- func delete_list(sparse_bitset(T), list(T)) = sparse_bitset(T) <= enum(T).
:- pred delete_list(list(T)::in, sparse_bitset(T)::in, sparse_bitset(T)::out)
    is det <= enum(T).

    % 'remove(X, Set0, Set)' returns in 'Set' the difference of 'Set0'
    % and the set containing only 'X', failing if 'Set0' does not contain 'X'.
    % Takes O(rep_size(Set)) time and space.
    %
:- pred remove(T::in, sparse_bitset(T)::in, sparse_bitset(T)::out)
    is semidet <= enum(T).

    % 'remove_list(X, Set0, Set)' returns in 'Set' the difference of 'Set0'
    % and the set containing all the elements of 'X', failing if any element
    % of 'X' is not in 'Set0'. Same as 'subset(list_to_set(X), Set0),
    % difference(Set0, list_to_set(X), Set)', but may be more efficient.
    %
:- pred remove_list(list(T)::in, sparse_bitset(T)::in, sparse_bitset(T)::out)
    is semidet <= enum(T).

    % 'remove_leq(Set, X)' returns 'Set' with all elements less than or equal
    % to 'X' removed. In other words, it returns the set containing all the
    % elements of 'Set' which are greater than 'X'.
    %
:- func remove_leq(sparse_bitset(T), T) = sparse_bitset(T) <= enum(T).
:- pred remove_leq(T::in, sparse_bitset(T)::in, sparse_bitset(T)::out)
    is det <= enum(T).

    % 'remove_gt(Set, X)' returns 'Set' with all elements greater than 'X'
    % removed. In other words, it returns the set containing all the elements
    % of 'Set' which are less than or equal to 'X'.
    %
:- func remove_gt(sparse_bitset(T), T) = sparse_bitset(T) <= enum(T).
:- pred remove_gt(T::in, sparse_bitset(T)::in, sparse_bitset(T)::out)
    is det <= enum(T).

    % 'remove_least(Set0, X, Set)' is true iff 'X' is the least element in
```

```
    % ‘Set0’, and ‘Set’ is the set which contains all the elements of ‘Set0’
    % except ‘X’. Takes O(1) time and space.
    %
:- pred remove_least(T::out, sparse_bitset(T)::in, sparse_bitset(T)::out)
    is semidet <= enum(T).

%-----------------------------------------------------%
%
% Comparisons between sets.
%

    % ‘equal(SetA, SetB’ is true iff ‘SetA’ and ‘SetB’ contain the same
    % elements. Takes O(min(rep_size(SetA), rep_size(SetB))) time.
    %
:- pred equal(sparse_bitset(T)::in, sparse_bitset(T)::in) is semidet.

    % ‘subset(Subset, Set)’ is true iff ‘Subset’ is a subset of ‘Set’.
    % Same as ‘intersect(Set, Subset, Subset)’, but may be more efficient.
    %
:- pred subset(sparse_bitset(T)::in, sparse_bitset(T)::in) is semidet.

    % ‘superset(Superset, Set)’ is true iff ‘Superset’ is a superset of ‘Set’.
    % Same as ‘intersect(Superset, Set, Set)’, but may be more efficient.
    %
:- pred superset(sparse_bitset(T)::in, sparse_bitset(T)::in) is semidet.

%-----------------------------------------------------%
%
% Operations on two or more sets.
%

    % ‘union(SetA, SetB)’ returns the union of ‘SetA’ and ‘SetB’. The
    % efficiency of the union operation is not sensitive to the argument
    % ordering. Takes O(rep_size(SetA) + rep_size(SetB)) time and space.
    %
:- func union(sparse_bitset(T), sparse_bitset(T)) = sparse_bitset(T).
:- pred union(sparse_bitset(T)::in, sparse_bitset(T)::in,
    sparse_bitset(T)::out) is det.

    % ‘union_list(Sets, Set)’ returns the union of all the sets in Sets.
    %
:- func union_list(list(sparse_bitset(T))) = sparse_bitset(T).
:- pred union_list(list(sparse_bitset(T))::in, sparse_bitset(T)::out) is det.

    % ‘intersect(SetA, SetB)’ returns the intersection of ‘SetA’ and ‘SetB’.
    % The efficiency of the intersection operation is not sensitive to the
    % argument ordering. Takes O(rep_size(SetA) + rep_size(SetB)) time and
```

```
    % O(min(rep_size(SetA)), rep_size(SetB)) space.
    %
:- func intersect(sparse_bitset(T), sparse_bitset(T)) = sparse_bitset(T).
:- pred intersect(sparse_bitset(T)::in, sparse_bitset(T)::in,
    sparse_bitset(T)::out) is det.

    % 'intersect_list(Sets, Set)' returns the intersection of all the sets
    % in Sets.
    %
:- func intersect_list(list(sparse_bitset(T))) = sparse_bitset(T).
:- pred intersect_list(list(sparse_bitset(T))::in, sparse_bitset(T)::out)
    is det.

    % 'difference(SetA, SetB)' returns the set containing all the elements
    % of 'SetA' except those that occur in 'SetB'. Takes
    % O(rep_size(SetA) + rep_size(SetB)) time and O(rep_size(SetA)) space.
    %
:- func difference(sparse_bitset(T), sparse_bitset(T)) = sparse_bitset(T).
:- pred difference(sparse_bitset(T)::in, sparse_bitset(T)::in,
    sparse_bitset(T)::out) is det.

%---------------------------------------------------%
%
% Operations that divide a set into two parts.
%

    % divide(Pred, Set, InPart, OutPart):
    % InPart consists of those elements of Set for which Pred succeeds;
    % OutPart consists of those elements of Set for which Pred fails.
    %
:- pred divide(pred(T)::in(pred(in) is semidet), sparse_bitset(T)::in,
    sparse_bitset(T)::out, sparse_bitset(T)::out) is det <= enum(T).

    % divide_by_set(DivideBySet, Set, InPart, OutPart):
    % InPart consists of those elements of Set which are also in DivideBySet;
    % OutPart consists of those elements of Set which are not in DivideBySet.
    %
:- pred divide_by_set(sparse_bitset(T)::in, sparse_bitset(T)::in,
    sparse_bitset(T)::out, sparse_bitset(T)::out) is det <= enum(T).

%---------------------------------------------------%
%
% Converting lists to sets.
%

    % 'list_to_set(List)' returns a set containing only the members of 'List'.
    % In the worst case this will take O(length(List)^2) time and space.
```

```
    % If the elements of the list are closely grouped, it will be closer
    % to O(length(List)).
    %
:- func list_to_set(list(T)) = sparse_bitset(T) <= enum(T).
:- pred list_to_set(list(T)::in, sparse_bitset(T)::out) is det <= enum(T).

    % 'sorted_list_to_set(List)' returns a set containing only the members
    % of 'List'. 'List' must be sorted. Takes O(length(List)) time and space.
    %
:- func sorted_list_to_set(list(T)) = sparse_bitset(T) <= enum(T).
:- pred sorted_list_to_set(list(T)::in, sparse_bitset(T)::out)
    is det <= enum(T).

%-----------------------------------------------------%
%
% Converting sets to lists.
%

    % 'to_sorted_list(Set)' returns a list containing all the members of 'Set',
    % in sorted order. Takes O(card(Set)) time and space.
    %
:- func to_sorted_list(sparse_bitset(T)) = list(T) <= enum(T).
:- pred to_sorted_list(sparse_bitset(T)::in, list(T)::out) is det <= enum(T).

%-----------------------------------------------------%
%
% Converting between different kinds of sets.
%

    % 'from_set(Set)' returns a bitset containing only the members of 'Set'.
    % Takes O(card(Set)) time and space.
    %
:- func from_set(set.set(T)) = sparse_bitset(T) <= enum(T).

    % 'to_sorted_list(Set)' returns a set.set containing all the members
    % of 'Set', in sorted order. Takes O(card(Set)) time and space.
    %
:- func to_set(sparse_bitset(T)) = set.set(T) <= enum(T).

%-----------------------------------------------------%
%
% Counting.
%

    % 'count(Set)' returns the number of elements in 'Set'.
    % Takes O(card(Set)) time.
    %
```

```
:- func count(sparse_bitset(T)) = int <= enum(T).

%-----------------------------------------------%
%
% Standard higher order functions on collections.
%

    % all_true(Pred, Set) succeeds iff Pred(Element) succeeds
    % for all the elements of Set.
    %
:- pred all_true(pred(T)::in(pred(in) is semidet), sparse_bitset(T)::in)
    is semidet <= enum(T).

    % 'filter(Pred, Set) = TrueSet' returns the elements of Set for which
    % Pred succeeds.
    %
:- func filter(pred(T), sparse_bitset(T)) = sparse_bitset(T) <= enum(T).
:- mode filter(pred(in) is semidet, in) = out is det.

    % 'filter(Pred, Set, TrueSet, FalseSet)' returns the elements of Set
    % for which Pred succeeds, and those for which it fails.
    %
:- pred filter(pred(T), sparse_bitset(T), sparse_bitset(T), sparse_bitset(T))
    <= enum(T).
:- mode filter(pred(in) is semidet, in, out, out) is det.

    % 'foldl(Func, Set, Start)' calls Func with each element of 'Set'
    % (in sorted order) and an accumulator (with the initial value of 'Start'),
    % and returns the final value. Takes O(card(Set)) time.
    %
:- func foldl(func(T, U) = U, sparse_bitset(T), U) = U <= enum(T).

:- pred foldl(pred(T, U, U), sparse_bitset(T), U, U) <= enum(T).
:- mode foldl(pred(in, in, out) is det, in, in, out) is det.
:- mode foldl(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode foldl(pred(in, di, uo) is det, in, di, uo) is det.
:- mode foldl(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode foldl(pred(in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode foldl(pred(in, di, uo) is semidet, in, di, uo) is semidet.
:- mode foldl(pred(in, in, out) is nondet, in, in, out) is nondet.
:- mode foldl(pred(in, in, out) is cc_multi, in, in, out) is cc_multi.
:- mode foldl(pred(in, di, uo) is cc_multi, in, di, uo) is cc_multi.

:- pred foldl2(pred(T, U, U, V, V), sparse_bitset(T), U, U, V, V) <= enum(T).
:- mode foldl2(pred(in, in, out, in, out) is det, in, in, out, in, out) is det.
:- mode foldl2(pred(in, in, out, mdi, muo) is det, in, in, out, mdi, muo)
    is det.
```

```
:- mode foldl2(pred(in, in, out, di, uo) is det, in, in, out, di, uo) is det.
:- mode foldl2(pred(in, di, uo, di, uo) is det, in, di, uo, di, uo) is det.
:- mode foldl2(pred(in, in, out, in, out) is semidet, in, in, out, in, out)
    is semidet.
:- mode foldl2(pred(in, in, out, mdi, muo) is semidet, in, in, out, mdi, muo)
    is semidet.
:- mode foldl2(pred(in, in, out, di, uo) is semidet, in, in, out, di, uo)
    is semidet.
:- mode foldl2(pred(in, in, out, in, out) is nondet, in, in, out, in, out)
    is nondet.
:- mode foldl2(pred(in, in, out, in, out) is cc_multi, in, in, out, in, out)
    is cc_multi.
:- mode foldl2(pred(in, in, out, di, uo) is cc_multi, in, in, out, di, uo)
    is cc_multi.
:- mode foldl2(pred(in, di, uo, di, uo) is cc_multi, in, di, uo, di, uo)
    is cc_multi.

    % 'foldr(Func, Set, Start)' calls Func with each element of 'Set'
    % (in reverse sorted order) and an accumulator (with the initial value
    % of 'Start'), and returns the final value. Takes O(card(Set)) time.
    %
:- func foldr(func(T, U) = U, sparse_bitset(T), U) = U <= enum(T).

:- pred foldr(pred(T, U, U), sparse_bitset(T), U, U) <= enum(T).
:- mode foldr(pred(in, in, out) is det, in, in, out) is det.
:- mode foldr(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode foldr(pred(in, di, uo) is det, in, di, uo) is det.
:- mode foldr(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode foldr(pred(in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode foldr(pred(in, di, uo) is semidet, in, di, uo) is semidet.
:- mode foldr(pred(in, in, out) is nondet, in, in, out) is nondet.
:- mode foldr(pred(in, in, out) is cc_multi, in, in, out) is cc_multi.
:- mode foldr(pred(in, di, uo) is cc_multi, in, di, uo) is cc_multi.

:- pred foldr2(pred(T, U, U, V, V), sparse_bitset(T), U, U, V, V) <= enum(T).
:- mode foldr2(pred(in, in, out, in, out) is det, in, in, out, in, out) is det.
:- mode foldr2(pred(in, in, out, mdi, muo) is det, in, in, out, mdi, muo)
    is det.
:- mode foldr2(pred(in, in, out, di, uo) is det, in, in, out, di, uo) is det.
:- mode foldr2(pred(in, di, uo, di, uo) is det, in, di, uo, di, uo) is det.
:- mode foldr2(pred(in, in, out, in, out) is semidet, in, in, out, in, out)
    is semidet.
:- mode foldr2(pred(in, in, out, mdi, muo) is semidet, in, in, out, mdi, muo)
    is semidet.
:- mode foldr2(pred(in, in, out, di, uo) is semidet, in, in, out, di, uo)
    is semidet.
:- mode foldr2(pred(in, in, out, in, out) is nondet, in, in, out, in, out)
```

```
    is nondet.
:- mode foldr2(pred(in, di, uo, di, uo) is cc_multi, in, di, uo, di, uo)
    is cc_multi.
:- mode foldr2(pred(in, in, out, di, uo) is cc_multi, in, in, out, di, uo)
    is cc_multi.
:- mode foldr2(pred(in, in, out, in, out) is cc_multi, in, in, out, in, out)
    is cc_multi.

%---------------------------------------------------%
%---------------------------------------------------%
```

79 stack

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 1994-1995, 1997-1999, 2005-2006, 2011-2012 The University of Melbourne
% Copyright (C) 2014-2016, 2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: stack.m.
% Main author: fjh.
% Stability: high.
%
% This file contains a 'stack' ADT.
% Stacks are implemented here using lists.
%
%---------------------------------------------------%

:- module stack.
:- interface.
:- import_module list.

%---------------------------------------------------%

:- type stack(T).

    % init = Stack:
    % init(Stack):
    %
    % True iff Stack is an empty stack.
    %
:- func init = stack(T).
:- pred init(stack(T)::out) is det.
```

```
    % is_empty(Stack):
    %
    % True iff Stack is an empty stack.
    %
:- pred is_empty(stack(T)::in) is semidet.

    % is_full(Stack):
    %
    % This is intended to be true iff Stack is a stack whose capacity
    % is exhausted. This implementation allows arbitrary-sized stacks,
    % so is_full always fails.
    %
:- pred is_full(stack(T)::in) is semidet.

    % push(Stack0, Elem) = Stack:
    % push(Elem, Stack0, Stack):
    %
    % True iff Stack is the stack which results from pushing Elem
    % onto the top of Stack0.
    %
:- func push(stack(T), T) = stack(T).
:- pred push(T::in, stack(T)::in, stack(T)::out) is det.

    % push_list(Stack0, Elems) = Stack:
    % push_list(Elems, Stack0, Stack):
    %
    % True iff Stack is the stack which results from pushing the elements of
    % the list Elems onto the top of Stack0.
    %
:- func push_list(stack(T), list(T)) = stack(T).
:- pred push_list(list(T)::in, stack(T)::in, stack(T)::out) is det.

    % top(Stack, Elem):
    %
    % True iff Stack is a non-empty stack whose top element is Elem.
    %
:- pred top(stack(T)::in, T::out) is semidet.

    % det_top is like top except that it will call error/1 rather than
    % failing if given an empty stack.
    %
:- func det_top(stack(T)) = T.
:- pred det_top(stack(T)::in, T::out) is det.

    % pop(Elem, Stack0, Stack):
    %
```

```
    % True iff Stack0 is a non-empty stack whose top element is Elem,
    % and Stack the stack which results from popping Elem off Stack0.
    %
:- pred pop(T::out, stack(T)::in, stack(T)::out) is semidet.

    % det_pop is like pop except that it will call error/1 rather than
    % failing if given an empty stack.
    %
:- pred det_pop(T::out, stack(T)::in, stack(T)::out) is det.

    % depth(Stack) = Depth:
    % depth(Stack, Depth):
    %
    % True iff Stack is a stack containing Depth elements.
    %
:- func depth(stack(T)) = int.
:- pred depth(stack(T)::in, int::out) is det.

%---------------------------------------------------%
%---------------------------------------------------%
```

80 std_util

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 1994-2006, 2008 The University of Melbourne.
% Copyright (C) 2016, 2018 The Mercury Team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: std_util.m.
% Main author: fjh.
% Stability: high.
%
% This file contains higher-order programming constructs and other
% useful standard utilities.
%
%---------------------------------------------------%
%---------------------------------------------------%

:- module std_util.
:- interface.

:- import_module maybe.
```

```
%---------------------------------------------------%
%
% General purpose higher-order programming constructs
%

    % compose(F, G, X) = F(G(X))
    %
    % Function composition.
    %
:- func compose(func(T2) = T3, func(T1) = T2, T1) = T3.

    % converse(F, X, Y) = F(Y, X).
    %
:- func converse(func(T1, T2) = T3, T2, T1) = T3.

    % pow(F, N, X) = F^N(X)
    %
    % Function exponentiation.
    %
:- func pow(func(T) = T, int, T) = T.

    % The identity function.
    %
:- func id(T) = T.

%---------------------------------------------------%

    % isnt(Pred, X) <=> not Pred(X)
    %
    % This is useful in higher order programming, e.g.
    %   Odds  = list.filter(odd, Xs)
    %   Evens = list.filter(isnt(odd), Xs)
    %
:- pred isnt(pred(T)::in(pred(in) is semidet), T::in) is semidet.

    % negate(Pred) <=> not Pred
    %
    % This is useful in higher order programming, e.g.
    %   expect(negate(Pred), ...)
    %
:- pred negate((pred)::in((pred) is semidet)) is semidet.

%---------------------------------------------------%

    % maybe_pred(Pred, X, Y) takes a closure Pred which transforms an
    % input semideterministically. If calling the closure with the input
```

```
    % X succeeds, Y is bound to 'yes(Z)' where Z is the output of the
    % call, or to 'no' if the call fails.
    %
    % Use maybe.pred_to_maybe instead.
:- pragma obsolete(maybe_pred/3).
:- pred maybe_pred(pred(T1, T2), T1, maybe(T2)).
:- mode maybe_pred(pred(in, out) is semidet, in, out) is det.

    % Use maybe.pred_to_maybe instead.
:- pragma obsolete(maybe_func/2).
:- func maybe_func(func(T1) = T2, T1) = maybe(T2).
:- mode maybe_func(func(in) = out is semidet, in) = out is det.

%-----------------------------------------------------%
%-----------------------------------------------------%
```

81 store

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 1994-1997, 2000-2008, 2010-2011 The University of Melbourne.
% Copyright (C) 2013-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: store.m.
% Main author: fjh.
% Stability: low.
%
% This file provides facilities for manipulating mutable stores.
% A store can be considered a mapping from abstract keys to their values.
% A store holds a set of nodes, each of which may contain a value of any
% type.
%
% Stores may be used to implement cyclic data structures such as circular
% linked lists, etc.
%
% Stores can have two different sorts of keys:
% mutable variables (mutvars) and references (refs).
% The difference between mutvars and refs is that mutvars can only be updated
% atomically, whereas it is possible to update individual fields of a
% reference one at a time (presuming the reference refers to a structured
% term).
%
```

```
%----------------------------------------------------%
%----------------------------------------------------%

:- module store.
:- interface.

:- import_module io.

%----------------------------------------------------%

    % Stores and keys are indexed by a type S of typeclass store(S) that
    % is used to distinguish between different stores. By using an
    % existential type declaration for 'init'/1 (see below), we use the
    % type system to ensure at compile time that you never attempt to use
    % a key from one store to access a different store.
    %
:- typeclass store(T) where [].
:- type store(S).

:- instance store(io).
:- instance store(store(S)).

    % Initialize a new store.
    %
:- some [S] pred init(store(S)::uo) is det.

%----------------------------------------------------%
%
% Mutvars
%

    % generic_mutvar(T, S):
    % A mutable variable holding a value of type T in store S.
    %
    % The mutable variable interface is inherently not thread-safe.
    % It is the programmer's responsibility to synchronise accesses to a
    % mutable variable from multiple threads where that is possible,
    % namely variables attached to the I/O state.
    %
:- type generic_mutvar(T, S).
:- type io_mutvar(T) == generic_mutvar(T, io).
:- type store_mutvar(T, S) == generic_mutvar(T, store(S)).

    % Create a new mutable variable, initialized with the specified value.
    %
:- pred new_mutvar(T::in, generic_mutvar(T, S)::out, S::di, S::uo)
    is det <= store(S).
```

```
    % copy_mutvar(OldMutvar, NewMutvar, S0, S) is equivalent to the sequence
    %   get_mutvar(OldMutvar, Value, S0, S1),
    %   new_mutvar(NewMutvar, Value, S1, S )
    %
:- pred copy_mutvar(generic_mutvar(T, S)::in, generic_mutvar(T, S)::out,
    S::di, S::uo) is det <= store(S).

    % Lookup the value stored in a given mutable variable.
    %
:- pred get_mutvar(generic_mutvar(T, S)::in, T::out,
    S::di, S::uo) is det <= store(S).

    % Replace the value stored in a given mutable variable.
    %
:- pred set_mutvar(generic_mutvar(T, S)::in, T::in,
    S::di, S::uo) is det <= store(S).

    % new_cyclic_mutvar(Func, Mutvar, !S):
    %
    % Create a new mutable variable, whose value is initialized with the value
    % returned from the specified function 'Func'. The argument passed to the
    % function is the mutvar itself, whose value has not yet been initialized
    % (this is safe because the function does not get passed the store, so it
    % cannot examine the uninitialized value).
    %
    % This predicate is useful for creating self-referential values such as
    % circular linked lists. For example:
    %
    %   :- type clist(T, S)
    %       --->    node(T, generic_mutvar(clist(T, S), S)).
    %
    %   :- pred init_cl(T::in, clist(T, S)::out, S::di, S::uo)
    %       is det <= store(S).
    %
    %   init_cl(X, CList, !Store) :-
    %       new_cyclic_mutvar(func(CL) = node(X, CL), CListVar, !Store),
    %       get_mutvar(CListVar, CList, !Store).
    %
:- pred new_cyclic_mutvar((func(generic_mutvar(T, S)) = T)::in,
    generic_mutvar(T, S)::out, S::di, S::uo) is det <= store(S).

%---------------------------------------------------%
%
% References
%
```

```
    % generic_ref(T, S):
    %
    % A reference to value of type T in store S.
    %
    % The reference interface is inherently not thread-safe.
    % It is the programmer's responsibility to synchronise accesses to a
    % reference from multiple threads where that is possible,
    % namely references attached to the I/O state.
    %
:- type generic_ref(T, S).
:- type io_ref(T, S) == generic_ref(T, io).
:- type store_ref(T, S) == generic_ref(T, store(S)).

    % new_ref(Val, Ref):
    %   /* In C: Ref = malloc(...); *Ref = Val; */
    %
    % Given a value of any type 'T', insert a copy of the term
    % into the store and return a new reference to that term.
    % (This does not actually perform a copy, it just returns a view
    % of the representation of that value.
    % It does however allocate one cell to hold the reference;
    % you can use new_arg_ref to avoid that.)
    %
:- pred new_ref(T::di, generic_ref(T, S)::out,
    S::di, S::uo) is det <= store(S).

    % ref_functor(Ref, Functor, Arity):
    %
    % Given a reference to a term, return the functor and arity of that term.
    %
:- pred ref_functor(generic_ref(T, S)::in, string::out, int::out,
    S::di, S::uo) is det <= store(S).

    % arg_ref(Ref, ArgNum, ArgRef):
    %   /* Pseudo-C code: ArgRef = &Ref[ArgNum]; */
    %
    % Given a reference to a term, return a reference to
    % the specified argument (field) of that term
    % (argument numbers start from zero).
    % It is an error if the argument number is out of range,
    % or if the argument reference has the wrong type.
    %
:- pred arg_ref(generic_ref(T, S)::in, int::in,
    generic_ref(ArgT, S)::out, S::di, S::uo) is det <= store(S).

    % new_arg_ref(Val, ArgNum, ArgRef):
    %   /* Pseudo-C code: ArgRef = &Val[ArgNum]; */
```

```
    %
    % Equivalent to `new_ref(Val, Ref), arg_ref(Ref, ArgNum, ArgRef)',
    % except that it is more efficient.
    % It is an error if the argument number is out of range,
    % or if the argument reference has the wrong type.
    %
:- pred new_arg_ref(T::di, int::in, generic_ref(ArgT, S)::out,
    S::di, S::uo) is det <= store(S).

    % set_ref(Ref, ValueRef):
    %   /* Pseudo-C code: *Ref = *ValueRef; */
    %
    % Given a reference to a term (Ref),
    % a reference to another term (ValueRef),
    % update the store so that the term referred to by Ref
    % is replaced with the term referenced by ValueRef.
    %
:- pred set_ref(generic_ref(T, S)::in, generic_ref(T, S)::in,
    S::di, S::uo) is det <= store(S).

    % set_ref_value(Ref, Value):
    %   /* Pseudo-C code: *Ref = Value; */
    %
    % Given a reference to a term (Ref), and a value (Value),
    % update the store so that the term referred to by Ref
    % is replaced with Value.
    %
:- pred set_ref_value(generic_ref(T, S)::in, T::di,
    S::di, S::uo) is det <= store(S).

    % Given a reference to a term, return that term.
    % Note that this requires making a copy, so this pred may
    % be inefficient if used to return large terms; it
    % is most efficient with atomic terms.
    % XXX current implementation buggy (does shallow copy)
    %
:- pred copy_ref_value(generic_ref(T, S)::in, T::uo,
    S::di, S::uo) is det <= store(S).

    % Same as above, but without making a copy. Destroys the store.
    %
:- pred extract_ref_value(S::di, generic_ref(T, S)::in, T::out)
    is det <= store(S).

%-----------------------------------------------%
%
% Nasty performance hacks
```

```
%
% WARNING: use of these procedures is dangerous!
% Use them only as a last resort, only if performance is critical, and only if
% profiling shows that using the safe versions is a bottleneck.
%
% These procedures may vanish in some future version of Mercury.

    % 'unsafe_arg_ref' is the same as 'arg_ref',
    % and 'unsafe_new_arg_ref' is the same as 'new_arg_ref'
    % except that they doesn't check for errors,
    % and they don't work for 'no_tag' types (types with
    % exactly one functor which has exactly one argument),
    % and they don't work for arguments which occupy a word with other
    % arguments,
    % and they don't work for types with >4 functors.
    % If the argument number is out of range,
    % or if the argument reference has the wrong type,
    % or if the argument is a 'no_tag' type,
    % or if the argument uses a packed representation,
    % then the behaviour is undefined, and probably harmful.

:- pred unsafe_arg_ref(generic_ref(T, S)::in, int::in,
    generic_ref(ArgT, S)::out, S::di, S::uo) is det <= store(S).

:- pred unsafe_new_arg_ref(T::di, int::in, generic_ref(ArgT, S)::out,
    S::di, S::uo) is det <= store(S).

%---------------------------------------------------%
%---------------------------------------------------%
```

82 stream

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 2006-2007, 2010 The University of Melbourne.
% Copyright (C) 2014-2020 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: stream.m.
% Authors: juliensf, maclarty.
% Stability: low
%
% This module provides a family of type classes for defining streams
```

```
% in Mercury. It also provides some generic predicates that operate
% on instances of these type classes.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module stream.
:- interface.

:- import_module bool.
:- import_module char.
:- import_module list.

:- include_module string_writer.

%-----------------------------------------------------%
%
% Types used by streams.
%

:- type name == string.

:- type result(Error)
    --->    ok
    ;       eof
    ;       error(Error).

:- type result(T, Error)
    --->    ok(T)
    ;       eof
    ;       error(Error).

:- type res(Error)
    --->    ok
    ;       error(Error).

:- type res(T, Error)
    --->    ok(T)
    ;       error(Error).

    % maybe_partial_res is used when it is possible to return a partial result
    % when an error occurs.
    %
:- type maybe_partial_res(T, Error)
    --->    ok(T)
    ;       error(T, Error).
```

```
%---------------------------------------------------%
%
% Stream errors.
%

:- typeclass error(Error) where
[

    % Convert a stream error into a human-readable format.
    % e.g. for use in error messages.
    %
    func error_message(Error) = string
].

%---------------------------------------------------%
%
% Streams.
%

    % A stream consists of a handle type and a state type.
    % The state type is threaded through, and destructively updated by,
    % the stream operations.
    %
:- typeclass stream(Stream, State) <= (Stream -> State) where
[
    % Returns a descriptive name for the stream.
    % Intended for use in error messages.
    %
    pred name(Stream::in, name::out, State::di, State::uo) is det
].

%---------------------------------------------------%
%
% Input streams.
%

    % An input stream is a source of data.
    %
:- typeclass input(Stream, State) <= stream(Stream, State) where [].

    % A reader stream is a subclass of a specific input stream that can be
    % used to read data of a specific type from that input stream.
    % A single input stream can support multiple reader subclasses.
    %
:- typeclass reader(Stream, Unit, State, Error)
    <= (input(Stream, State), error(Error), (Stream, Unit -> Error)) where
[
```

```
    % Get the next unit from the given stream.
    %
    % The get operation should block until the next unit is available,
    % or the end of the stream or an error is detected.
    %
    % If a call to get/4 returns 'eof', all further calls to get/4,
    % unboxed_get/5 or bulk_get/9 for that stream return 'eof'. If a call to
    % get/4 returns 'error(...)', all further calls to get/4, unboxed_get/5 or
    % bulk_get/4 for that stream return an error, although not necessar-
ily the
    % same one.
    %
    % XXX We should provide an interface to allow the user to reset the
    % error status to try again if an error is transient.
    %
    pred get(Stream::in, result(Unit, Error)::out,
        State::di, State::uo) is det
].

    % An unboxed_reader stream is like a reader stream except that it provides
    % an interface that avoids a memory allocation when there is no error.
    %
:- typeclass unboxed_reader(Stream, Unit, State, Error)
    <= (input(Stream, State), error(Error), (Stream, Unit -> Error)) where
[
    % Get the next unit from the given stream. On error or eof an arbitrary
    % value of type Unit is returned.
    %
    % The unboxed_get operation should block until the next unit is available,
    % or the end of the stream or an error is detected.
    %
    % If a call to unboxed_get/5 returns 'eof', all further calls to get/4,
    % unboxed_get/5 or bulk_get/9 for that stream return 'eof'. If a call to
    % unboxed_get/5 returns 'error(...)', all further calls to get/4,
    % unboxed_get/5 or bulk_get/4 for that stream return an error, although not
    % necessarily the same one.
    %
    % XXX We should provide an interface to allow the user to reset the
    % error status to try again if an error is transient.
    %
    pred unboxed_get(Stream::in, result(Error)::out, Unit::out,
        State::di, State::uo) is det
].

    % A bulk_reader stream is a subclass of specific input stream that can
    % be used to read multiple items of data of a specific type from that
    % input stream into a specified container. For example, binary input
```

```
    % streams may be able to efficiently read bytes into a bitmap.
    % A single input stream can support multiple bulk_reader subclasses.
    %
:- typeclass bulk_reader(Stream, Index, Store, State, Error)
    <= (input(Stream, State), error(Error),
        (Stream, Index, Store -> Error)) where
[
    % bulk_get(Stream, Index, NumItems, !Store, NumItemsRead, Result, !State).
    %
    % Read at most NumItems items into the given Store starting at the
    % given index, returning the number of items read.
    %
    % If the read succeeds, Result is 'ok' and NumItemsRead equals NumItems.
    %
    % On end-of-stream, bulk_get/9 puts as many items as it can into !Store.
    % NumItemsRead is less than NumItems, and Result is 'ok'.
    %
    % If an error is detected, bulk_get/9 puts as many items as it can into
    % !Store. NumItemsRead is less than NumItems, and Result is 'error(Err)'.
    %
    % Blocks until NumItems items are available or the end of the stream
    % is reached or an error is detected.
    %
    % Throws an exception if Index given is out of range or NumItems units
    % starting at Index will not fit in !Store.
    %
    % If a call to bulk_get/4 returns less than NumItems items, all further
    % calls to get/4, unboxed_get/5 or bulk_get/4 for that stream return no
    % items. If a call to bulk_get/9 returns 'error(...)', all further calls to
    % get/4, unboxed_get/5 or bulk_get/9 for that stream return an error,
    % although not necessarily the same one.
    %
    pred bulk_get(Stream::in, Index::in, int::in,
        Store::bulk_get_di, Store::bulk_get_uo,
        int::out, res(Error)::out, State::di, State::uo) is det
].

    % XXX These should be di and uo, but with the current state of the mode
    % system, an unsafe_promise_unique call would be required at each call
    % to bulk_get.
:- mode bulk_get_di == in.
:- mode bulk_get_uo == out.

%---------------------------------------------------%
%
% Output streams.
%
```

```
    % An output stream is a destination for data.
    % Note that unlike input streams, output streams do not include
    % an explicit error type. They should handle errors by throwing exceptions.
    %
:- typeclass output(Stream, State)
    <= stream(Stream, State) where
[
    % For buffered output streams, completely write out any data in the buffer.
    % For unbuffered streams, this operation is a no-op.
    %
    pred flush(Stream::in, State::di, State::uo) is det
].

    % A writer stream is a subclass of specific output stream that can be
    % used to write data of a specific type to that output stream.
    % A single output stream can support multiple writer subclasses.
    %
:- typeclass writer(Stream, Unit, State)
    <= output(Stream, State) where
[
    % Write the next unit to the given stream.
    % Blocks if the whole unit cannot be written to the stream at the time
    % of the call (for example because a buffer is full).
    %
    pred put(Stream::in, Unit::in, State::di, State::uo) is det
].

%---------------------------------------------------%
%
% Duplex streams.
%

    % A duplex stream is a stream that can act as both a source and
    % destination of data, i.e. it is a both an input and an output stream.
    %
:- typeclass duplex(Stream, State)
    <= (input(Stream, State), output(Stream, State)) where
[
].

%---------------------------------------------------%
%
% Putback streams.
%

    % A putback stream is an input stream that allows data to be pushed back
```

```
    % onto the stream. As with reader subclasses it is possible to define
    % multiple putback subclasses for a single input stream.
    %
:- typeclass putback(Stream, Unit, State, Error)
    <= reader(Stream, Unit, State, Error) where
[
    % Un-gets a unit from the specified input stream.
    % Only one unit of putback is guaranteed to be successful.
    %
    pred unget(Stream::in, Unit::in, State::di, State::uo) is det
].

    % As above, but guarantees that an unlimited number of units may be pushed
    % back onto the stream.
    %
:- typeclass unbounded_putback(Stream, Unit, State, Error)
    <= putback(Stream, Unit, State, Error) where
[
].

%---------------------------------------------------%
%
% Seekable streams.
%

    % whence denotes the base for a seek operation.
    %   set - seek relative to the start of the file
    %   cur - seek relative to the current position in the file
    %   end - seek relative to the end of the file.
    %
:- type whence
    --->    set
    ;       cur
    ;       end.

:- typeclass seekable(Stream, State) <= stream(Stream, State)
    where
[
    % Seek to an offset relative to whence on the specified stream.
    % The offset is measured in bytes.
    %
    pred seek(Stream::in, whence::in, int::in, State::di, State::uo) is det,

    % As above, but the offset is always a 64-bit value.
    %
    pred seek64(Stream::in, whence::in, int64::in, State::di, State::uo) is det
].
```

```
%-----------------------------------------------------%
%
% Line oriented streams.
%

    % A line oriented stream is a stream that keeps track of line numbers.
    %
:- typeclass line_oriented(Stream, State) <= stream(Stream, State)
    where
[
    % Get the current line number for the specified stream.
    %
    pred get_line(Stream::in, int::out, State::di, State::uo) is det,

    % Set the current line number of the specified stream.
    %
    pred set_line(Stream::in, int::in,  State::di, State::uo) is det
].

%-----------------------------------------------------%
%
% Generic folds over input streams.
%

    % Applies the given closure to each Unit read from the input stream in
    % turn, until eof or error.
    %
:- pred input_stream_fold(Stream, pred(Unit, T, T), T,
    maybe_partial_res(T, Error), State, State)
    <= reader(Stream, Unit, State, Error).
:- mode input_stream_fold(in, in(pred(in, in, out) is det),
    in, out, di, uo) is det.
:- mode input_stream_fold(in, in(pred(in, in, out) is cc_multi),
    in, out, di, uo) is cc_multi.

    % Applies the given closure to each Unit read from the input stream in
    % turn, until eof or error.
    %
:- pred input_stream_fold_state(Stream, pred(Unit, State, State),
    res(Error), State, State)
    <= reader(Stream, Unit, State, Error).
:- mode input_stream_fold_state(in, in(pred(in, di, uo) is det),
    out, di, uo) is det.
:- mode input_stream_fold_state(in, in(pred(in, di, uo) is cc_multi),
    out, di, uo) is cc_multi.
```

```
    % Applies the given closure to each Unit read from the input stream
    % in turn, until eof or error.
    %
:- pred input_stream_fold2_state(Stream,
    pred(Unit, T, T, State, State), T, maybe_partial_res(T, Error),
    State, State) <= reader(Stream, Unit, State, Error).
:- mode input_stream_fold2_state(in,
    in(pred(in, in, out, di, uo) is det),
    in, out, di, uo) is det.
:- mode input_stream_fold2_state(in,
    in(pred(in, in, out, di, uo) is cc_multi),
    in, out, di, uo) is cc_multi.

    % Applies the given closure to each Unit read from the input stream
    % in turn, until eof or error, or the closure returns 'no' as its
    % second argument.
    %
:- pred input_stream_fold2_state_maybe_stop(Stream,
    pred(Unit, bool, T, T, State, State),
    T, maybe_partial_res(T, Error), State, State)
    <= reader(Stream, Unit, State, Error).
:- mode input_stream_fold2_state_maybe_stop(in,
    in(pred(in, out, in, out, di, uo) is det), in, out, di, uo) is det.
:- mode input_stream_fold2_state_maybe_stop(in,
    in(pred(in, out, in, out, di, uo) is cc_multi), in, out, di, uo)
    is cc_multi.

%---------------------------------------------------%
%
% Misc. operations on input streams.
%

    % Discard all the whitespace characters satisfying 'char.is_whitespace'
    % from the specified stream.
    %
:- pred ignore_whitespace(Stream::in, result(Error)::out,
    State::di, State::uo)
    is det <= putback(Stream, char, State, Error).

%---------------------------------------------------%
%
% Misc. operations on output streams.
%

    % put_list(Stream, Write, Sep, List, !State).
    %
    % Write all the elements List to Stream separated by Sep.
```

```
    %
:- pred put_list(Stream, pred(Stream, T, State, State),
    pred(Stream, State, State), list(T), State, State)
    <= output(Stream, State).
:- mode put_list(in, pred(in, in, di, uo) is det, pred(in, di, uo) is det,
    in, di, uo) is det.
:- mode put_list(in, pred(in, in, di, uo) is cc_multi,
    pred(in, di, uo) is cc_multi, in, di, uo) is cc_multi.
:- mode put_list(in, pred(in, in, di, uo) is cc_multi,
    pred(in, di, uo) is det, in, di, uo) is cc_multi.

%---------------------------------------------------%
%---------------------------------------------------%
```

83 stream.string_writer

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 2006-2007, 2011 The University of Melbourne.
% Copyright (C) 2014-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: stream.string_writer.m.
% Authors: trd, fjh, stayl
%
% Predicates to write to streams that accept strings.
%
%---------------------------------------------------%
%---------------------------------------------------%

:- module stream.string_writer.
:- interface.

:- import_module char.
:- import_module deconstruct.
:- import_module io.
:- import_module list.
:- import_module string.
:- import_module univ.

%---------------------------------------------------%

:- pred put_int(Stream::in, int::in, State::di, State::uo) is det
```

```
    <= stream.writer(Stream, string, State).

:- pred put_uint(Stream::in, uint::in, State::di, State::uo) is det
    <= stream.writer(Stream, string, State).

:- pred put_int8(Stream::in, int8::in, State::di, State::uo) is det
    <= stream.writer(Stream, string, State).

:- pred put_uint8(Stream::in, uint8::in, State::di, State::uo) is det
    <= stream.writer(Stream, string, State).

:- pred put_int16(Stream::in, int16::in, State::di, State::uo) is det
    <= stream.writer(Stream, string, State).

:- pred put_uint16(Stream::in, uint16::in, State::di, State::uo) is det
    <= stream.writer(Stream, string, State).

:- pred put_int32(Stream::in, int32::in, State::di, State::uo) is det
    <= stream.writer(Stream, string, State).

:- pred put_uint32(Stream::in, uint32::in, State::di, State::uo) is det
    <= stream.writer(Stream, string, State).

:- pred put_int64(Stream::in, int64::in, State::di, State::uo) is det
    <= stream.writer(Stream, string, State).

:- pred put_uint64(Stream::in, uint64::in, State::di, State::uo) is det
    <= stream.writer(Stream, string, State).

:- pred put_float(Stream::in, float::in, State::di, State::uo) is det
    <= stream.writer(Stream, string, State).

:- pred put_char(Stream::in, char::in, State::di, State::uo) is det
    <= stream.writer(Stream, string, State).

    % A version of io.format that works for arbitrary string writers.
    %
:- pred format(Stream::in, string::in, list(poly_type)::in,
    State::di, State::uo) is det <= stream.writer(Stream, string, State).

:- pred nl(Stream::in, State::di, State::uo) is det
    <= stream.writer(Stream, string, State).

    % print/4 writes its second argument to the string writer stream specified
    % in its first argument. In all cases, the argument to output can be of
    % any type. It is output in a format that is intended to be human readable.
    %
```

```
    % If the argument is just a single string or character, it will be printed
    % out exactly as is (unquoted). If the argument is of type integer (i.e.
    % an arbitrary precision integer), then its decimal representation will be
    % printed. If the argument is of type univ, then the value stored in the
    % the univ will be printed out, but not the type. If the argument is of
    % type date_time, it will be printed out in the same form as the string
    % returned by the function date_to_string/1. If the argument is of type
    % duration, it will be printed out in the same form as the string
    % returned by the function duration_to_string/1.
    %
    % print/5 is the same as print/4 except that it allows the caller to
    % specify how non-canonical types should be handled. print/4 implicitly
    % specifies 'canonicalize' as the method for handling non-canonical types.
    % This means that for higher-order types, or types with user-defined
    % equality axioms, or types defined using the foreign language interface
    % (i.e. pragma foreign_type), the text output will only describe the type
    % that is being printed, not the value.
    %
    % print_cc/4 is the same as print/4 except that it specifies
    % 'include_details_cc' rather than 'canonicalize'. This means that it will
    % print the details of non-canonical types. However, it has determinism
    % 'cc_multi'.
    %
    % Note that even if 'include_details_cc' is specified, some implementations
    % may not be able to print all the details for higher-order types or types
    % defined using the foreign language interface.
    %
:- pred print(Stream::in, T::in, State::di, State::uo) is det
    <= (stream.writer(Stream, string, State),
    stream.writer(Stream, char, State)).

:- pred print_cc(Stream::in, T::in, State::di, State::uo) is cc_multi
    <= (stream.writer(Stream, string, State),
    stream.writer(Stream, char, State)).

:- pred print(Stream, deconstruct.noncanon_handling, T, State, State)
    <= (stream.writer(Stream, string, State),
    stream.writer(Stream, char, State)).
:- mode print(in, in(do_not_allow), in, di, uo) is det.
:- mode print(in, in(canonicalize), in, di, uo) is det.
:- mode print(in, in(include_details_cc), in, di, uo) is cc_multi.
:- mode print(in, in, in, di, uo) is cc_multi.

    % write/4 writes its second argument to the string writer stream specified
    % in its first argument. In all cases, the argument to output may be of
    % any type. The argument is written in a format that is intended to be
    % valid Mercury syntax whenever possible.
```

```
    %
    % Strings and characters are always printed out in quotes, using backslash
    % escapes if necessary and backslash or octal escapes for all characters
    % for which char.is_control/1 is true. For higher-order types, or for types
    % defined using the foreign language interface (pragma foreign_type), the
    % text output will only describe the type that is being printed, not the
    % value, and the result may not be parsable by 'read'. For the types
    % containing existential quantifiers, the type 'type_desc' and closure
    % types, the result may not be parsable by 'read', either. But in all
    % other cases the format used is standard Mercury syntax, and if you append
    % a period and newline (".\n"), then the results can be read in again using
    % 'read'.
    %
    % write/5 is the same as write/4 except that it allows the caller to
    % specify how non-canonical types should be handled. write_cc/4 is the
    % same as write/4 except that it specifies 'include_details_cc' rather than
    % 'canonicalize'.
    %
:- pred write(Stream::in, T::in, State::di, State::uo) is det
    <= (stream.writer(Stream, string, State),
    stream.writer(Stream, char, State)).

:- pred write_cc(Stream::in, T::in, State::di, State::uo) is cc_multi
    <= (stream.writer(Stream, string, State),
    stream.writer(Stream, char, State)).

:- pred write(Stream, deconstruct.noncanon_handling, T, State, State)
    <= (stream.writer(Stream, string, State),
    stream.writer(Stream, char, State)).
:- mode write(in, in(do_not_allow), in, di, uo) is det.
:- mode write(in, in(canonicalize), in, di, uo) is det.
:- mode write(in, in(include_details_cc), in, di, uo) is cc_multi.
:- mode write(in, in, in, di, uo) is cc_multi.

%---------------------------------------------------%
%---------------------------------------------------%
```

84 string.builder

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 2006-2007 The University of Melbourne.
% Copyright (C) 2014-2015, 2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
```

```
%-----------------------------------------------%
%
% File: string.builder.m.
% Main author: maclarty.
%
% This module implements a string builder stream.  It can be used to
% build up a string using string or character writers.
%
% To build up a string using this module, you first construct an initial
% string builder state by calling the init function.  You can then use
% any instances of stream.writer that write strings or characters to up-
date the
% string builder state, using string.builder.handle as the stream argument.
% Once you've finished writing to the string builder you can get the final
% string by calling string.builder.to_string/1.
%
% For example:
%
%     State0 = string.builder.init,
%     stream.string_writer.put_int(string.builder.handle, 5, State0, State),
%     Str = string.builder.to_string(State),  % Str = "5".
%
%-----------------------------------------------%

:- module string.builder.
:- interface.

:- import_module char.
:- import_module stream.

%-----------------------------------------------%

:- type handle
    --->    handle.

:- type state.

:- func init = (string.builder.state::uo) is det.

:- instance stream.stream(string.builder.handle, string.builder.state).

:- instance stream.output(string.builder.handle, string.builder.state).

:- instance stream.writer(string.builder.handle, string, string.builder.state).
:- instance stream.writer(string.builder.handle, char, string.builder.state).

:- func to_string(string.builder.state::di) = (string::uo) is det.
```

```
%-----------------------------------------------------%
%-----------------------------------------------------%
```

85 string

```
%-----------------------------------------------------%
% vim: ts=4 sw=4 et ft=mercury
%-----------------------------------------------------%
% Copyright (C) 1993-2012 The University of Melbourne.
% Copyright (C) 2013-2020 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: string.m.
% Main authors: fjh, petdr, wangp.
% Stability: medium to high.
%
% This modules provides basic string handling facilities.
%
% Mercury strings are Unicode strings in either UTF-8 or UTF-16 encoding
% depending on the target language.
%
% When Mercury is compiled to C, strings are UTF-8 encoded, with a null
% character as the string terminator. A single code point requires one to four
% bytes (code units) to encode.
%
% When Mercury is compiled to Java or C#, strings are represented using the
% Java 'String' or C# 'System.String' types, both using UTF-16 encoding.
% A single code point requires one or two 16-bit integers (code units)
% to encode.
%
% When Mercury is compiled to Erlang, strings are represented as Erlang
% binaries using UTF-8 encoding.
%
% The Mercury compiler will only allow well-formed UTF-8 or UTF-16 string
% constants. However, it is possible to produce strings containing invalid
% UTF-8 or UTF-16 via I/O, foreign code, and substring operations.
% Predicates or functions that inspect strings may fail, throw an exception,
% or else behave in some special way when an ill-formed code unit sequence is
% encountered.
%
% Unexpected null characters embedded in the middle of strings can be a source
% of security vulnerabilities, so the Mercury library predicates and functions
% which create strings from (lists of) characters throw an exception if a null
```

```
% character is detected. Programmers must not create strings that might
% contain null characters using the foreign language interface.
%
% The builtin comparison operation on strings is also implementation
% dependent. The current implementation performs string comparison using
%
% - C's strcmp() function, when compiling to C;
% - Java's String.compareTo() method, when compiling to Java;
% - C#'s System.String.CompareOrdinal() method, when compiling to C#; and
% - Erlang's term comparison operator, when compiling to Erlang.
%
%-----------------------------------------------------%
%
% This module is divided into several sections. These sections are:
%
% - Wrapper types that associate particular semantics with raw strings.
% - Converting between strings and lists of characters.
% - Reading characters from strings.
% - Writing characters to strings.
% - Determining the lengths of strings.
% - Computing hashes of strings.
% - Tests on strings.
% - Appending strings.
% - Splitting up strings.
% - Dealing with prefixes and suffixes.
% - Transformations of strings.
% - Folds over the characters in strings.
% - Formatting tables.
% - Converting strings to docs.
% - Converting strings to values of builtin types.
% - Converting values of builtin types to strings.
% - Converting values of arbitrary types to strings.
% - Converting values to strings based on a format string.
%
%-----------------------------------------------------%

:- module string.
:- interface.

:- include_module builder.

:- import_module assoc_list.
:- import_module char.
:- import_module deconstruct.
:- import_module list.
:- import_module maybe.
:- import_module ops.
```

```
:- import_module pretty_printer.

%---------------------------------------------------%
%
% Wrapper types that associate particular semantics with raw strings.
%
% These types are used for defining stream typeclass instances
% where you want different instances for strings representing different
% semantic entities. Using the string type itself, without a wrapper,
% would be ambiguous in such situations.
%
% While each module that associates semantics with strings could define
% its own wrapper types, the notions of lines and text files are so common
% that it is simpler to define them just once, and this is the logical
% place to do that.
%

    % A line is:
    %
    % - a possibly empty sequence of non-newline characters terminated by a
    %   newline character; or
    % - a non-empty sequence of non-newline characters terminated by the end
    %   of the file.
    %
:- type line
    --->    line(string).

    % A text file is a possibly empty sequence of characters
    % terminated by the end of the file.
    %
:- type text_file
    --->    text_file(string).

%---------------------------------------------------%
%
% Conversions between strings and lists of characters.
%

    % Convert the string to a list of characters (code points).
    %
    % In the forward mode:
    % If strings use UTF-8 encoding then each code unit in an ill-formed
    % sequence is replaced by U+FFFD REPLACEMENT CHARACTER in the list.
    % If strings use UTF-16 encoding then each unpaired surrogate code point
    % is returned as a separate code point in the list.
    %
    % The reverse mode of the predicate throws an exception if the list
```

```
    % contains a null character or code point that cannot be encoded in a
    % string (namely, surrogate code points cannot be encoded in UTF-8
    % strings).
    %
    % The reverse mode of to_char_list/2 is deprecated because the implied
    % ability to round trip convert a string to a list then back to the same
    % string does not hold in the presence of ill-formed code unit sequences.
    %
:- pragma obsolete_proc(to_char_list(uo, in), [from_char_list/2]).
:- func to_char_list(string) = list(char).
:- pred to_char_list(string, list(char)).
:- mode to_char_list(in, out) is det.
:- mode to_char_list(uo, in) is det.

    % Convert the string to a list of characters (code points) in reverse
    % order.
    %
    % In the forward mode:
    % If strings use UTF-8 encoding then each code unit in an ill-formed
    % sequence is replaced by U+FFFD REPLACEMENT CHARACTER in the list.
    % If strings use UTF-16 encoding then each unpaired surrogate code point
    % is returned as a separate code point in the list.
    %
    % The reverse mode of the predicate throws an exception if the list
    % contains a null character or code point that cannot be encoded in a
    % string (namely, surrogate code points cannot be encoded in UTF-8
    % strings).
    %
    % The reverse mode of to_rev_char_list/2 is deprecated because the implied
    % ability to round trip convert a string to a list then back to the same
    % string does not hold in the presence of ill-formed code unit sequences.
    %
:- pragma obsolete_proc(to_rev_char_list(uo, in), [from_rev_char_list/2]).
:- func to_rev_char_list(string) = list(char).
:- pred to_rev_char_list(string, list(char)).
:- mode to_rev_char_list(in, out) is det.
:- mode to_rev_char_list(uo, in) is det.

    % Convert a list of characters (code points) to a string.
    % Throws an exception if the list contains a null character or code point
    % that cannot be encoded in a string (namely, surrogate code points cannot
    % be encoded in UTF-8 strings).
    %
    % The reverse mode of from_char_list/2 is deprecated because the implied
    % ability to round trip convert a string to a list then back to the same
    % string does not hold in the presence of ill-formed code unit sequences.
    %
```

```
:- pragma obsolete_proc(from_char_list(out, in), [to_char_list/2]).
:- func from_char_list(list(char)::in) = (string::uo) is det.
:- pred from_char_list(list(char), string).
:- mode from_char_list(in, uo) is det.
:- mode from_char_list(out, in) is det.

    % As above, but fail instead of throwing an exception if the list contains
    % a null character or code point that cannot be encoded in a string.
    %
:- pred semidet_from_char_list(list(char)::in, string::uo) is semidet.

    % Same as from_char_list, except that it reverses the order
    % of the characters.
    % Throws an exception if the list contains a null character or code point
    % that cannot be encoded in a string (namely, surrogate code points cannot
    % be encoded in UTF-8 strings).
    %
:- func from_rev_char_list(list(char)::in) = (string::uo) is det.
:- pred from_rev_char_list(list(char)::in, string::uo) is det.

    % As above, but fail instead of throwing an exception if the list contains
    % a null character or code point that cannot be encoded in a string.
    %
:- pred semidet_from_rev_char_list(list(char)::in, string::uo) is semidet.

    % Convert a string into a list of code units of the string encoding used
    % by the current process.
    %
:- pred to_code_unit_list(string::in, list(int)::out) is det.

    % Convert a string into a list of UTF-8 code units.
    % Throws an exception if the string contains an unpaired surrogate code
    % point, as the encoding of surrogate code points is prohibited in UTF-
8.
    %
:- pred to_utf8_code_unit_list(string::in, list(int)::out) is det.

    % Convert a string into a list of UTF-16 code units.
    % Throws an exception if strings use UTF-8 encoding and the given string
    % contains an ill-formed code unit sequence, as arbitrary bytes can-
not be
    % represented in UTF-16 (even allowing for ill-formed sequences).
    %
:- pred to_utf16_code_unit_list(string::in, list(int)::out) is det.

    % Convert a list of code units to a string.
    % Fails if the list does not contain a valid encoding of a string
```

```
    % (in the encoding expected by the current process),
    % or if the string would contain a null character.
    %
:- pred from_code_unit_list(list(int)::in, string::uo) is semidet.

    % Convert a list of code units to a string.
    % The resulting string may contain ill-formed sequences.
    % Fails if the list contains a code unit that is out of range
    % or if the string would contain a null character.
    %
:- pred from_code_unit_list_allow_ill_formed(list(int)::in, string::uo)
    is semidet.

    % Convert a list of UTF-8 code units to a string.
    % Fails if the list does not contain a valid encoding of a string
    % or if the string would contain a null character.
    %
:- pred from_utf8_code_unit_list(list(int)::in, string::uo) is semidet.

    % Convert a list of UTF-16 code units to a string.
    % Fails if the list does not contain a valid encoding of a string
    % or if the string would contain a null character.
    %
:- pred from_utf16_code_unit_list(list(int)::in, string::uo) is semidet.

    % duplicate_char(Char, Count, String):
    %
    % Construct a string consisting of 'Count' occurrences of 'Char'
    % code points in sequence.
    %
:- func duplicate_char(char::in, int::in) = (string::uo) is det.
:- pred duplicate_char(char::in, int::in, string::uo) is det.

%-----------------------------------------------------%
%
% Reading characters from strings.
%

    % This type is used by the _repl indexing predicates to distinguish a
    % U+FFFD code point that is actually in a string from a U+FFFD code point
    % generated when the predicate encounters an ill-formed code unit sequence
    % in a UTF-8 string.
    %
:- type maybe_replaced
    --->    not_replaced
    ;       replaced_code_unit(uint8).
```

```
    % index(String, Index, Char):
    %
    % If ‘Index’ is the initial code unit offset of a well-formed code unit
    % sequence in ‘String’ then ‘Char’ is the code point encoded by that
    % sequence.
    %
    % Otherwise, if ‘Index’ is in range, ‘Char’ is either a U+FFFD REPLACEMENT
    % CHARACTER (when strings are UTF-8 encoded) or the unpaired surrogate
    % code point at ‘Index’ (when strings are UTF-16 encoded).
    %
    % Fails if ‘Index’ is out of range (negative, or greater than or equal to
    % the length of ‘String’).
    %
:- pred index(string::in, int::in, char::uo) is semidet.

    % det_index(String, Index, Char):
    %
    % Like index/3 but throws an exception if ‘Index’ is out of range
    % (negative, or greater than or equal to the length of ‘String’).
    %
:- func det_index(string, int) = char.
:- pred det_index(string::in, int::in, char::uo) is det.

    % unsafe_index(String, Index, Char):
    %
    % Like index/3 but does not check that ‘Index’ is in range.
    %
    % WARNING: behavior is UNDEFINED if ‘Index’ is out of range
    % (negative, or greater than or equal to the length of ‘String’).
    % This version is constant time, whereas det_index
    % may be linear in the length of the string. Use with care!
    %
:- func unsafe_index(string, int) = char.
:- pred unsafe_index(string::in, int::in, char::uo) is det.

    % A synonym for det_index/2:
    % String ^ elem(Index) = det_index(String, Index).
    %
:- func string ^ elem(int) = char.

    % A synonym for unsafe_index/2:
    % String ^ unsafe_elem(Index) = unsafe_index(String, Index).
    %
:- func string ^ unsafe_elem(int) = char.

    % index_next(String, Index, NextIndex, Char):
    %
```

```
    % If ‘Index’ is the initial code unit offset of a well-formed code unit
    % sequence in ‘String’ then ‘Char’ is the code point encoded by that
    % sequence, and ‘NextIndex’ is the offset immediately following that
    % sequence.
    %
    % Otherwise, if ‘Index’ is in range, ‘Char’ is either a U+FFFD REPLACEMENT
    % CHARACTER (when strings are UTF-8 encoded) or the unpaired surrogate
    % code point at ‘Index’ (when strings are UTF-16 encoded), and ‘NextIndex’
    % is Index + 1.
    %
    % Fails if ‘Index’ is out of range (negative, or greater than or equal to
    % the length of ‘String’).
    %
:- pred index_next(string::in, int::in, int::out, char::uo) is semidet.

    % index_next_repl(String, Index, NextIndex, Char, MaybeReplaced):
    %
    % Like index_next/4 but ‘MaybeReplaced’ is ‘replaced_code_unit(CodeUnit)’
    % iff ‘Char’ is the Unicode REPLACEMENT CHARACTER (U+FFFD) but ‘String’
    % does NOT contain an encoding of U+FFFD beginning at ‘Index’;
    % ‘CodeUnit’ is the code unit at ‘Index’.
    % (‘MaybeReplaced’ is always ‘not_replaced’ when strings are UTF-16
    % encoded.)
    %
:- pred index_next_repl(string::in, int::in, int::out, char::uo,
    maybe_replaced::out) is semidet.

    % unsafe_index_next(String, Index, NextIndex, Char):
    %
    % Like index_next/4 but does not check that ‘Index’ is in range.
    % Fails if ‘Index’ is equal to the length of ‘String’.
    %
    % WARNING: behavior is UNDEFINED if ‘Index’ is out of range
    % (negative, or greater than the length of ‘String’).
    %
:- pred unsafe_index_next(string::in, int::in, int::out, char::uo) is semidet.

    % unsafe_index_next_repl(String, Index, NextIndex, Char, MaybeReplaced):
    %
    % Like index_next_repl/5 but does not check that ‘Index’ is in range.
    % Fails if ‘Index’ is equal to the length of ‘String’.
    %
    % WARNING: behavior is UNDEFINED if ‘Index’ is out of range
    % (negative, or greater than the length of ‘String’).
    %
:- pred unsafe_index_next_repl(string::in, int::in, int::out, char::uo,
    maybe_replaced::out) is semidet.
```

```
    % prev_index(String, Index, PrevIndex, Char):
    %
    % If ‘Index - 1’ is the final code unit offset of a well-formed se-
quence in
    % ‘String’ then ‘Char’ is the code point encoded by that sequence, and
    % ‘PrevIndex’ is the initial code unit offset of that sequence.
    %
    % Otherwise, if ‘Index’ is in range, ‘Char’ is either a U+FFFD REPLACEMENT
    % CHARACTER (when strings are UTF-8 encoded) or the unpaired surrogate
    % code point at ‘Index - 1’ (when strings are UTF-16 encoded), and
    % ‘PrevIndex’ is ‘Index - 1’.
    %
    % Fails if ‘Index’ is out of range (non-positive, or greater than the
    % length of ‘String’).
    %
:- pred prev_index(string::in, int::in, int::out, char::uo) is semidet.

    % prev_index_repl(String, Index, PrevIndex, Char, MaybeReplaced):
    %
    % Like prev_index/4 but ‘MaybeReplaced’ is ‘replaced_code_unit(CodeUnit)’
    % iff ‘Char’ is the Unicode REPLACEMENT CHARACTER (U+FFFD) but ‘String’
    % does NOT contain an encoding of U+FFFD ending at ‘Index - 1’;
    % ‘CodeUnit’ is the code unit at ‘Index - 1’.
    % (‘MaybeReplaced’ is always ‘not_replaced’ when strings are UTF-16
    % encoded.)
    %
:- pred prev_index_repl(string::in, int::in, int::out, char::uo,
    maybe_replaced::out) is semidet.

    % unsafe_prev_index(String, Index, PrevIndex, Char):
    %
    % Like prev_index/4 but does not check that ‘Index’ is in range.
    % Fails if ‘Index’ is zero.
    %
    % WARNING: behavior is UNDEFINED if ‘Index’ is out of range
    % (negative, or greater than the length of ‘String’).
    %
:- pred unsafe_prev_index(string::in, int::in, int::out, char::uo) is semidet.

    % unsafe_prev_index_repl(String, Index, PrevIndex, Char, MaybeReplaced):
    %
    % Like prev_index_repl/5 but does not check that ‘Index’ is in range.
    % Fails if ‘Index’ is zero.
    %
    % WARNING: behavior is UNDEFINED if ‘Index’ is out of range
    % (negative, or greater than the length of ‘String’).
```

```
    %
:- pred unsafe_prev_index_repl(string::in, int::in, int::out, char::uo,
    maybe_replaced::out) is semidet.

    % unsafe_index_code_unit(String, Index, CodeUnit):
    %
    % ‘CodeUnit’ is the code unit in ‘String’ at the offset ‘Index’.
    % WARNING: behavior is UNDEFINED if ‘Index’ is out of range
    % (negative, or greater than or equal to the length of ‘String’).
    %
:- pred unsafe_index_code_unit(string::in, int::in, int::out) is det.

%-----------------------------------------------%
%
% Writing characters to strings.
%

    % set_char(Char, Index, String0, String):
    %
    % ‘String’ is ‘String0’, with the code unit sequence beginning at ‘Index’
    % replaced by the encoding of ‘Char’. If the code unit at ‘Index’ is the
    % initial code unit in a valid encoding of a code point, then that entire
    % code unit sequence is replaced. Otherwise, only the code unit at ‘Index’
    % is replaced.
    %
    % Fails if ‘Index’ is out of range (negative, or greater than or equal to
    % the length of ‘String0’).
    %
    % Throws an exception if ‘Char’ is the null character or a code point that
    % cannot be encoded in a string (namely, surrogate code points can-
not be
    % encoded in UTF-8 strings).
    %
:- pred set_char(char, int, string, string).
:- mode set_char(in, in, in, out) is semidet.
% NOTE This mode is disabled because the compiler puts constant strings
% into static data even when they might be updated.
% :- mode set_char(in, in, di, uo) is semidet.

    % det_set_char(Char, Index, String0, String):
    %
    % Same as set_char/4 but throws an exception if ‘Index’ is out of range
    % (negative, or greater than or equal to the length of ‘String0’).
    %
:- func det_set_char(char, int, string) = string.
:- pred det_set_char(char, int, string, string).
:- mode det_set_char(in, in, in, out) is det.
```

```
% NOTE This mode is disabled because the compiler puts constant strings
% into static data even when they might be updated.
% :- mode det_set_char(in, in, di, uo) is det.

    % unsafe_set_char(Char, Index, String0, String):
    %
    % Same as set_char/4 but does not check if ‘Index’ is in range.
    % WARNING: behavior is UNDEFINED if ‘Index’ is out of range
    % (negative, or greater than or equal to the length of ‘String0’).
    % Use with care!
    %
:- func unsafe_set_char(char, int, string) = string.
:- mode unsafe_set_char(in, in, in) = out is det.
% NOTE This mode is disabled because the compiler puts constant strings
% into static data even when they might be updated.
% :- mode unsafe_set_char(in, in, di) = uo is det.
:- pred unsafe_set_char(char, int, string, string).
:- mode unsafe_set_char(in, in, in, out) is det.
% NOTE This mode is disabled because the compiler puts constant strings
% into static data even when they might be updated.
% :- mode unsafe_set_char(in, in, di, uo) is det.

%-----------------------------------------------%
%
% Determining the lengths of strings.
%

    % Determine the length of a string, in code units.
    % An empty string has length zero.
    %
    % NOTE: code points (characters) are encoded using one or more code units,
    % i.e. bytes for UTF-8; 16-bit integers for UTF-16.
    %
:- func length(string::in) = (int::uo) is det.
:- pred length(string, int).
:- mode length(in, uo) is det.
:- mode length(ui, uo) is det.

    % Synonyms for length.
    %
:- func count_code_units(string) = int.
:- pred count_code_units(string::in, int::out) is det.

    % Determine the number of code points in a string.
    %
    % Each valid code point, and each code unit that is part of an ill-
formed
```

```
    % sequence, contributes one to the result.
    % (This matches the number of steps it would take to iterate over the
    % string using string.index_next or string.prev_index.)
    %
    % NOTE The names of this predicate and several other predicates
    % may be changed in the future to refer to code_points, not codepoints,
    % for consistency with predicate names that talk about code_units.
    %
:- func count_codepoints(string) = int.
:- pred count_codepoints(string::in, int::out) is det.

    % count_utf8_code_units(String) = Length:
    %
    % Return the number of code units required to represent a string in
    % UTF-8 encoding (with allowance for ill-formed sequences).
    % Equivalent to 'Length = length(to_utf8_code_unit_list(String))'.
    %
    % Throws an exception if strings use UTF-16 encoding but the given string
    % contains an unpaired surrogate code point. Surrogate code points cannot
    % be represented in UTF-8.
    %
:- func count_utf8_code_units(string) = int.

    % codepoint_offset(String, StartOffset, Count, Offset):
    %
    % Let 'S' be the substring of 'String' from code unit 'StartOffset' to the
    % end of the string. 'Offset' is code unit offset after advancing 'Count'
    % steps in 'S', where each step skips over either:
    %  - one encoding of a Unicode code point, or
    %  - one code unit that is part of an ill-formed sequence.
    %
    % Fails if 'StartOffset' is out of range (negative, or greater than the
    % length of 'String'), or if there are fewer than 'Count' steps possible
    % in 'S'.
    %
:- pred codepoint_offset(string::in, int::in, int::in, int::out) is semidet.

    % codepoint_offset(String, Count, Offset):
    %
    % Same as 'codepoint_offset(String, 0, Count, Offset)'.
    %
:- pred codepoint_offset(string::in, int::in, int::out) is semidet.

%---------------------------------------------------%
%
% Computing hashes of strings.
%
```

```
    % Compute a hash value for a string.
    %
:- func hash(string) = int.
:- pred hash(string::in, int::out) is det.

    % Two other hash functions for strings.
    %
:- func hash2(string) = int.
:- func hash3(string) = int.

    % Cross-compilation-friendly versions of hash, hash2 and hash3
    % respectively.
:- func hash4(string) = int.
:- func hash5(string) = int.
:- func hash6(string) = int.

%-----------------------------------------------------%
%
% Tests on strings.
%

    % True if string is the empty string.
    %
:- pred is_empty(string::in) is semidet.

    % True if the string is a valid UTF-8 or UTF-16 string.
    % In target languages that use UTF-8 string encoding, `is_well_formed(S)'
    % is true iff S consists of a well-formed UTF-8 code unit sequence.
    % In target languages that use UTF-16 string encoding, `is_well_formed(S)'
    % is true iff S consists of a well-formed UTF-16 code unit sequence.
    %
:- pred is_well_formed(string::in) is semidet.

    % True if string contains only alphabetic characters [A-Za-z].
    %
:- pred is_all_alpha(string::in) is semidet.

    % True if string contains only alphabetic characters [A-Za-z] and digits
    % [0-9].
    %
:- pred is_all_alnum(string::in) is semidet.

    % True if string contains only alphabetic characters [A-Za-z] and
    % underscores.
    %
:- pred is_all_alpha_or_underscore(string::in) is semidet.
```

```
    % True if string contains only alphabetic characters [A-Za-z],
    % digits [0-9], and underscores.
    %
:- pred is_all_alnum_or_underscore(string::in) is semidet.

    % True if the string contains only decimal digits (0-9).
    %
:- pred is_all_digits(string::in) is semidet.

    % all_match(TestPred, String):
    %
    % True iff ‘String’ is empty or contains only code points that satisfy
    % ‘TestPred’.
    %
:- pred all_match(pred(char)::in(pred(in) is semidet), string::in) is semidet.

    % contains_char(String, Char):
    %
    % Succeed if the code point ‘Char’ occurs in ‘String’.
    %
:- pred contains_char(string::in, char::in) is semidet.

    % compare_substrings(Res, X, StartX, Y, StartY, Length):
    %
    % Compare two substrings by code unit order. The two substrings are
    % the substring of ‘X’ between ‘StartX’ and ‘StartX + Length’, and
    % the substring of ‘Y’ between ‘StartY’ and ‘StartY + Length’.
    % ‘StartX’, ‘StartY’ and ‘Length’ are all in terms of code units.
    %
    % Fails if ‘StartX’ or ‘StartX + Length’ are not within [0, length(X)],
    % or if ‘StartY’ or ‘StartY + Length’ are not within [0, length(Y)],
    % or if ‘Length’ is negative.
    %
:- pred compare_substrings(comparison_result::uo, string::in, int::in,
    string::in, int::in, int::in) is semidet.

    % unsafe_compare_substrings(Res, X, StartX, Y, StartY, Length):
    %
    % Same as compare_between/4 but without range checks.
    % WARNING: if any of ‘StartX’, ‘StartY’, ‘StartX + Length’ or
    % ‘StartY + Length’ are out of range, or if ‘Length’ is negative,
    % then the behaviour is UNDEFINED. Use with care!
    %
:- pred unsafe_compare_substrings(comparison_result::uo, string::in, int::in,
    string::in, int::in, int::in) is det.
```

```
    % compare_ignore_case_ascii(Res, X, Y):
    %
    % Compare two strings by code unit order, ignoring the case of letters
    % (A-Z, a-z) in the ASCII range.
    % Equivalent to ‘compare(Res, to_lower(X), to_lower(Y))’
    % but more efficient.
    %
:- pred compare_ignore_case_ascii(comparison_result::uo,
    string::in, string::in) is det.

    % prefix_length(Pred, String):
    %
    % The length (in code units) of the maximal prefix of ‘String’ consisting
    % entirely of code points satisfying ‘Pred’.
    %
:- func prefix_length(pred(char)::in(pred(in) is semidet), string::in)
    = (int::out) is det.

    % suffix_length(Pred, String):
    %
    % The length (in code units) of the maximal suffix of ‘String’ consisting
    % entirely of code points satisfying ‘Pred’.
    %
:- func suffix_length(pred(char)::in(pred(in) is semidet), string::in)
    = (int::out) is det.

    % sub_string_search(String, SubString, Index):
    %
    % ‘Index’ is the code unit position in ‘String’ where the first
    % occurrence of ‘SubString’ begins. Indices start at zero, so if
    % ‘SubString’ is a prefix of ‘String’, this will return Index = 0.
    %
:- pred sub_string_search(string::in, string::in, int::out) is semidet.

    % sub_string_search_start(String, SubString, BeginAt, Index):
    %
    % ‘Index’ is the code unit position in ‘String’ where the first
    % occurrence of ‘SubString’ occurs such that ’Index’ is greater than or
    % equal to ‘BeginAt’. Indices start at zero.
    % Fails if either ‘BeginAt’ is negative, or greater than
    % length(String) - length(SubString).
    %
:- pred sub_string_search_start(string::in, string::in, int::in, int::out)
    is semidet.

    % unsafe_sub_string_search_start(String, SubString, BeginAt, Index):
    %
```

```
    % Same as sub_string_search_start/4 but does not check that 'BeginAt'
    % is in range.
    % WARNING: if 'BeginAt' is either negative, or greater than length(String),
    % then the behaviour is UNDEFINED. Use with care!
    %
:- pred unsafe_sub_string_search_start(string::in, string::in, int::in,
    int::out) is semidet.

%-----------------------------------------------------%
%
% Appending strings.
%

    % Append two strings together.
    %
:- func append(string::in, string::in) = (string::uo) is det.

    % append(S1, S2, S3):
    %
    % Append two strings together. S3 consists of the code units of S1
    % followed by the code units of S2, in order.
    %
    % An ill-formed code unit sequence at the end of S1 may join with an
    % ill-formed code unit sequence at the start of S2 to produce a valid
    % encoding of a code point in S3.
    %
    % The append(out, out, in) mode is deprecated because it does not match
    % the semantics of the forwards modes in the presence of ill-formed code
    % unit sequences. Use nondet_append/3 instead.
    %
:- pragma obsolete_proc(append(out, out, in), [nondet_append/3]).
:- pred append(string, string, string).
:- mode append(in, in, in) is semidet.  % implied
:- mode append(in, uo, in) is semidet.
:- mode append(in, in, uo) is det.
:- mode append(uo, in, in) is semidet.
:- mode append(out, out, in) is multi.

    % nondet_append(S1, S2, S3):
    %
    % Non-deterministically return S1 and S2, where S1 ++ S2 = S3.
    % S3 is split after each code point or code unit in an ill-formed sequence.
    %
:- pred nondet_append(string, string, string).
:- mode nondet_append(out, out, in) is multi.

    % S1 ++ S2 = S :- append(S1, S2, S).
```

```
    %
    % Append two strings together using nicer inline syntax.
    %
:- func string ++ string = string.
:- mode in ++ in = uo is det.

    % Append a list of strings together.
    %
:- func append_list(list(string)::in) = (string::uo) is det.
:- pred append_list(list(string)::in, string::uo) is det.

    % join_list(Separator, Strings) = JoinedString:
    %
    % Append together the strings in Strings, putting Separator between
    % each pair of adjacent strings. If Strings is the empty list,
    % return the empty string.
    %
:- func join_list(string::in, list(string)::in) = (string::uo) is det.

%-----------------------------------------------------%
%
% Making strings from smaller pieces.
%

:- type string_piece
    --->    string(string)
    ;       substring(string, int, int).    % string, start, end offset

    % append_string_pieces(Pieces, String):
    %
    % Append together the strings and substrings in 'Pieces' into a string.
    % Throws an exception if 'Pieces' contains an element
    % 'substring(S, Start, End)' where 'Start' or 'End' are not within
    % the range [0, length(S)], or if 'Start' > 'End'.
    %
:- pred append_string_pieces(list(string_piece)::in, string::uo) is det.

    % Same as append_string_pieces/2 but without range checks.
    % WARNING: if any piece 'substring(S, Start, End)' has 'Start' or 'End'
    % outside the range [0, length(S)], or if 'Start' > 'End',
    % then the behaviour is UNDEFINED. Use with care!
    %
:- pred unsafe_append_string_pieces(list(string_piece)::in, string::uo)
    is det.

%-----------------------------------------------------%
%
```

```
% Splitting up strings.
%

    % first_char(String, Char, Rest) is true iff ‘String’ begins with a
    % well-formed code unit sequence, ‘Char’ is the code point encoded by
    % that sequence, and ‘Rest’ is the rest of ‘String’ after that sequence.
    %
    % The (uo, in, in) mode throws an exception if ‘Char’ cannot be en-
coded in
    % a string, or if ‘Char’ is a surrogate code point (for consistency with
    % the other modes).
    %
    % WARNING: first_char makes a copy of Rest because the garbage collector
    % doesn't handle references into the middle of an object, at least not the
    % way we use it. This means that repeated use of first_char to iterate
    % over a string will result in very poor performance. If you want to
    % iterate over the characters in a string, use foldl or to_char_list
    % instead.
    %
:- pred first_char(string, char, string).
:- mode first_char(in, in, in) is semidet.  % implied
:- mode first_char(in, uo, in) is semidet.  % implied
:- mode first_char(in, in, uo) is semidet.  % implied
:- mode first_char(in, uo, uo) is semidet.
:- mode first_char(uo, in, in) is det.

    % split(String, Index, LeftSubstring, RightSubstring):
    %
    % Split a string into two substrings at the code unit offset ‘Index’.
    % (If ‘Index’ is out of the range [0, length of ‘String’], it is treated
    % as if it were the nearest end-point of that range.)
    %
:- pred split(string::in, int::in, string::out, string::out) is det.

    % split_by_codepoint(String, Count, LeftSubstring, RightSubstring):
    %
    % ‘LeftSubstring’ is the left-most ‘Count’ code points of ‘String’,
    % and ‘RightSubstring’ is the remainder of ‘String’.
    % (If ‘Count’ is out of the range [0, length of ‘String’], it is treated
    % as if it were the nearest end-point of that range.)
    %
:- pred split_by_codepoint(string::in, int::in, string::out, string::out)
    is det.

    % left(String, Count, LeftSubstring):
    %
    % ‘LeftSubstring’ is the left-most ‘Count’ code units of ‘String’.
```

```
    % (If ‘Count’ is out of the range [0, length of ‘String’], it is treated
    % as if it were the nearest end-point of that range.)
    %
:- func left(string::in, int::in) = (string::out) is det.
:- pred left(string::in, int::in, string::out) is det.

    % left_by_codepoint(String, Count, LeftSubstring):
    %
    % ‘LeftSubstring’ is the left-most ‘Count’ code points of ‘String’.
    % (If ‘Count’ is out of the range [0, length of ‘String’], it is treated
    % as if it were the nearest end-point of that range.)
    %
:- func left_by_codepoint(string::in, int::in) = (string::out) is det.
:- pred left_by_codepoint(string::in, int::in, string::out) is det.

    % right(String, Count, RightSubstring):
    %
    % ‘RightSubstring’ is the right-most ‘Count’ code units of ‘String’.
    % (If ‘Count’ is out of the range [0, length of ‘String’], it is treated
    % as if it were the nearest end-point of that range.)
    %
:- func right(string::in, int::in) = (string::out) is det.
:- pred right(string::in, int::in, string::out) is det.

    % right_by_codepoint(String, Count, RightSubstring):
    %
    % ‘RightSubstring’ is the right-most ‘Count’ code points of ‘String’.
    % (If ‘Count’ is out of the range [0, length of ‘String’], it is treated
    % as if it were the nearest end-point of that range.)
    %
:- func right_by_codepoint(string::in, int::in) = (string::out) is det.
:- pred right_by_codepoint(string::in, int::in, string::out) is det.

    % between(String, Start, End, Substring):
    %
    % ‘Substring’ consists of the segment of ‘String’ within the half-open
    % interval [Start, End), where ‘Start’ and ‘End’ are code unit offsets.
    % (If ‘Start’ is out of the range [0, length of ‘String’], it is treated
    % as if it were the nearest end-point of that range.
    % If ‘End’ is out of the range [‘Start’, length of ‘String’],
    % it is treated as if it were the nearest end-point of that range.)
    %
:- func between(string::in, int::in, int::in) = (string::uo) is det.
:- pred between(string::in, int::in, int::in, string::uo) is det.

    % between_codepoints(String, Start, End, Substring):
    %
```

```
    % 'Substring' is the part of 'String' between the code point positions
    % 'Start' and 'End'. The result is equivalent to:
    %
    %   between(String, StartOffset, EndOffset, Substring)
    %
    % where:
    %
    %   StartOffset is from codepoint_offset(String, Start, StartOffset)
    %     if Start is in [0, count_codepoints(String)],
    %   StartOffset = 0 if Start < 0,
    %   StartOffset = length(String) otherwise;
    %
    %   EndOffset is from codepoint_offset(String, End, EndOffset)
    %     if End is in [0, count_codepoints(String)],
    %   EndOffset = 0 if End < 0,
    %   EndOffset = length(String) otherwise.
    %
    % between/4 will enforce StartOffset =< EndOffset.
    %
:- func between_codepoints(string::in, int::in, int::in)
    = (string::uo) is det.
:- pred between_codepoints(string::in, int::in, int::in, string::uo) is det.

    % unsafe_between(String, Start, End, Substring):
    %
    % 'Substring' consists of the segment of 'String' within the half-open
    % interval [Start, End), where 'Start' and 'End' are code unit offsets.
    % WARNING: if 'Start' is out of the range [0, length of 'String'] or
    % 'End' is out of the range ['Start', length of 'String']
    % then the behaviour is UNDEFINED. Use with care!
    % This version takes time proportional to the length of the substring,
    % whereas substring may take time proportional to the length
    % of the whole string.
    %
:- func unsafe_between(string::in, int::in, int::in) = (string::uo) is det.
:- pred unsafe_between(string::in, int::in, int::in, string::uo) is det.

    % words_separator(SepP, String) returns the list of non-empty
    % substrings of String (in first to last order) that are delimited
    % by non-empty sequences of code points matched by SepP.
    % For example,
    %
    % words_separator(char.is_whitespace, " the cat  sat on the  mat") =
    %   ["the", "cat", "sat", "on", "the", "mat"]
    %
    % Note the difference to split_at_separator.
    %
```

```
:- func words_separator(pred(char), string) = list(string).
:- mode words_separator(pred(in) is semidet, in) = out is det.

    % words(String) =
    %   words_separator(char.is_whitespace, String).
    %
:- func words(string) = list(string).

    % split_at_separator(SepP, String) returns the list of (possibly empty)
    % substrings of String (in first to last order) that are delimited
    % by code points matched by SepP. For example,
    %
    % split_at_separator(char.is_whitespace, " a cat  sat on the  mat")
    %   = ["", "a", "cat", "", "sat", "on", "the", "", "mat"]
    %
    % Note the difference to words_separator.
    %
:- func split_at_separator(pred(char), string) = list(string).
:- mode split_at_separator(pred(in) is semidet, in) = out is det.

    % split_at_char(Char, String) =
    %     split_at_separator(unify(Char), String)
    %
:- func split_at_char(char, string) = list(string).

    % split_at_string(Separator, String) returns the list of substrings
    % of String that are delimited by Separator. For example,
    %
    % split_at_string("|||", "|||fld2|||fld3") = ["", "fld2", [fld3"]
    %
    % Always the first match of Separator is used to break the String, for
    % example: split_at_string("aa", "xaaayaaaz") = ["x", "ay", "az"]
    %
:- func split_at_string(string, string) = list(string).

%-----------------------------------------------------%
%
% Dealing with prefixes and suffixes.
%

    % prefix(String, Prefix) is true iff Prefix is a prefix of String.
    % Same as append(Prefix, _, String).
    %
:- pragma obsolete_proc(prefix(in, out)).
:- pred prefix(string, string).
:- mode prefix(in, in) is semidet.
:- mode prefix(in, out) is multi.
```

```
    % suffix(String, Suffix) is true iff Suffix is a suffix of String.
    % Same as append(_, Suffix, String).
    %
:- pragma obsolete_proc(suffix(in, out)).
:- pred suffix(string, string).
:- mode suffix(in, in) is semidet.
:- mode suffix(in, out) is multi.

    % remove_prefix(Prefix, String, Suffix):
    %
    % This is a synonym for append(Prefix, Suffix, String) but with the
    % arguments in a more convenient order for use with higher-order code.
    %
    % WARNING: the argument order differs from remove_suffix.
    %
:- pred remove_prefix(string::in, string::in, string::out) is semidet.

    % det_remove_prefix(Prefix, String, Suffix):
    %
    % This is a synonym for append(Prefix, Suffix, String) but with the
    % arguments in a more convenient order for use with higher-order code.
    %
    % WARNING: the argument order differs from remove_suffix.
    %
:- pred det_remove_prefix(string::in, string::in, string::out) is det.

    % remove_prefix_if_present(Prefix, String) = Suffix returns 'String' minus
    % 'Prefix' if 'String' begins with 'Prefix', and 'String' if it doesn't.
    %
:- func remove_prefix_if_present(string, string) = string.

    % remove_suffix(String, Suffix, Prefix):
    %
    % The same as append(Prefix, Suffix, String).
    %
    % WARNING: the argument order differs from both remove_prefix and
    % remove_suffix_if_present.
    %
:- pred remove_suffix(string::in, string::in, string::out) is semidet.

    % det_remove_suffix(String, Suffix) returns the same value as
    % remove_suffix, except it throws an exception if String does not end
    % with Suffix.
    %
    % WARNING: the argument order differs from both remove_prefix and
    % remove_suffix_if_present.
```

```
    %
:- func det_remove_suffix(string, string) = string.

    % remove_suffix_if_present(Suffix, String) returns ‘String’ minus ‘Suffix’
    % if ‘String’ ends with ‘Suffix’, and ‘String’ if it doesn’t.
    %
    % WARNING: the argument order differs from remove_suffix and
    % det_remove_suffix.
    %
:- func remove_suffix_if_present(string, string) = string.

%---------------------------------------------------%
%
% Transformations of strings.
%

    % Convert the first character (if any) of a string to uppercase.
    % Only letters (a-z) in the ASCII range are converted.
    %
    % This function transforms the initial code point of a string,
    % whether or not the code point occurs as part of a combining sequence.
    %
:- func capitalize_first(string) = string.
:- pred capitalize_first(string::in, string::out) is det.

    % Convert the first character (if any) of a string to lowercase.
    % Only letters (A-Z) in the ASCII range are converted.
    %
    % This function transforms the initial code point of a string,
    % whether or not the code point occurs as part of a combining sequence.
    %
:- func uncapitalize_first(string) = string.
:- pred uncapitalize_first(string::in, string::out) is det.

    % Converts a string to uppercase.
    % Only letters (A-Z) in the ASCII range are converted.
    %
    % This function transforms each code point individually.
    % Letters that occur within a combining sequence will be converted,
    % whereas the precomposed character equivalent to the combining
    % sequence would not be converted. For example:
    %
    %   to_upper("a\u0301") ==> "A\u0301"   %  decomposed
    %   to_upper("\u00E1")  ==> "\u00E1"    %  precomposed
    %
:- func to_upper(string::in) = (string::uo) is det.
:- pred to_upper(string, string).
```

```
:- mode to_upper(in, uo) is det.
:- mode to_upper(in, in) is semidet.        % implied

    % Converts a string to lowercase.
    % Only letters (a-z) in the ASCII range are converted.
    %
    % This function transforms each code point individually.
    % Letters that occur within a combining sequence will be converted,
    % whereas the precomposed character equivalent to the combining
    % sequence would not be converted. For example:
    %
    %   to_lower("A\u0301") ==> "a\u0301"   %  decomposed
    %   to_lower("\u00C1")  ==> "\u00C1"    %  precomposed
    %
:- func to_lower(string::in) = (string::uo) is det.
:- pred to_lower(string, string).
:- mode to_lower(in, uo) is det.
:- mode to_lower(in, in) is semidet.        % implied

    % pad_left(String0, PadChar, Width, String):
    %
    % Insert ‘PadChar’s at the left of ‘String0’ until it is at least as long
    % as ‘Width’, giving ‘String’. Width is currently measured as the number
    % of code points.
    %
:- func pad_left(string, char, int) = string.
:- pred pad_left(string::in, char::in, int::in, string::out) is det.

    % pad_right(String0, PadChar, Width, String):
    %
    % Insert ‘PadChar’s at the right of ‘String0’ until it is at least as long
    % as ‘Width’, giving ‘String’. Width is currently measured as the number
    % of code points.
    %
:- func pad_right(string, char, int) = string.
:- pred pad_right(string::in, char::in, int::in, string::out) is det.

    % chomp(String):
    %
    % Return ‘String’ minus any single trailing newline character.
    %
:- func chomp(string) = string.

    % strip(String):
    %
    % Returns ‘String’ minus any initial and trailing ASCII whitespace
    % characters, i.e. characters satisfying ‘char.is_whitespace’.
```

```
    %
:- func strip(string) = string.

    % lstrip(String):
    %
    % Return ‘String’ minus any initial ASCII whitespace characters,
    % i.e. characters satisfying ‘char.is_whitespace’.
    %
:- func lstrip(string) = string.

    % rstrip(String):
    %
    % Returns ‘String’ minus any trailing ASCII whitespace characters,
    % i.e. characters satisfying ‘char.is_whitespace’.
    %
:- func rstrip(string) = string.

    % lstrip_pred(Pred, String):
    %
    % Returns ‘String’ minus the maximal prefix consisting entirely
    % of code points satisfying ‘Pred’.
    %
:- func lstrip_pred(pred(char)::in(pred(in) is semidet), string::in)
    = (string::out) is det.

    % rstrip_pred(Pred, String):
    %
    % Returns ‘String’ minus the maximal suffix consisting entirely
    % of code points satisfying ‘Pred’.
    %
:- func rstrip_pred(pred(char)::in(pred(in) is semidet), string::in)
    = (string::out) is det.

    % replace(String0, Pattern, Subst, String):
    %
    % Replaces the first occurrence of Pattern in String0 with Subst to give
    % String. Fails if Pattern does not occur in String0.
    %
:- pred replace(string::in, string::in, string::in, string::uo) is semidet.

    % replace_all(String0, Pattern, Subst, String):
    %
    % Replaces any occurrences of Pattern in String0 with Subst to give
    % String.
    %
    % If Pattern is the empty string then Subst is inserted at every point
    % in String0 except between two code units in an encoding of a code point.
```

```
    % For example, these are true:
    %
    %   replace_all("", "", "|", "|")
    %   replace_all("a", "", "|", "|a|")
    %   replace_all("ab", "", "|", "|a|b|")
    %
:- func replace_all(string::in, string::in, string::in) = (string::uo) is det.
:- pred replace_all(string::in, string::in, string::in, string::uo) is det.

    % word_wrap(Str, N) = Wrapped:
    %
    % Wrapped is Str with newlines inserted between words (separated by ASCII
    % space characters) so that at most N code points appear on any line,
    % and each line contains as many whole words as possible subject to that
    % constraint. If any one word exceeds N code points in length, then
    % it will be broken over two (or more) lines. Sequences of whitespace
    % characters are replaced by a single space.
    %
    % See ‘char.is_whitespace’ for the definition of whitespace characters
    % used by this predicate.
    %
:- func word_wrap(string, int) = string.

    % word_wrap_separator(Str, N, WordSeparator) = Wrapped:
    %
    % word_wrap_separator/3 is like word_wrap/2, except that words that
    % need to be broken up over multiple lines have WordSeparator inserted
    % between each piece. If the length of WordSeparator is greater than
    % or equal to N code points, then no separator is used.
    %
:- func word_wrap_separator(string, int, string) = string.

%---------------------------------------------------%
%
% Folds over the characters in strings.
%

    % foldl(Closure, String, !Acc):
    %
    % ‘Closure’ is an accumulator predicate which is to be called for each
    % code point of the string ‘String’ in turn.
    % If ‘String’ contains ill-formed sequences, ‘Closure’ is called for each
    % code unit in an ill-formed sequence. If strings use UTF-8 encoding,
    % U+FFFD is passed to ‘Closure’ in place of each such code unit.
    % If strings use UTF-16 encoding, each code unit in an ill-formed sequence
    % is an unpaired surrogate code point, which will be passed to ‘Closure’.
    %
```

```
    % The initial value of the accumulator is '!.Acc' and the final value is
    % '!:Acc'.
    % (foldl(Closure, String, !Acc)  is equivalent to
    %   to_char_list(String, Chars),
    %   list.foldl(Closure, Chars, !Acc)
    % but is implemented more efficiently.)
    %
:- func foldl(func(char, A) = A, string, A) = A.
:- pred foldl(pred(char, A, A), string, A, A).
:- mode foldl(pred(in, di, uo) is det, in, di, uo) is det.
:- mode foldl(pred(in, in, out) is det, in, in, out) is det.
:- mode foldl(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode foldl(pred(in, in, out) is nondet, in, in, out) is nondet.
:- mode foldl(pred(in, in, out) is multi, in, in, out) is multi.

    % foldl2(Closure, String, !Acc1, !Acc2):
    % A variant of foldl with two accumulators.
    %
:- pred foldl2(pred(char, A, A, B, B), string, A, A, B, B).
:- mode foldl2(pred(in, di, uo, di, uo) is det,
    in, di, uo, di, uo) is det.
:- mode foldl2(pred(in, in, out, di, uo) is det,
    in, in, out, di, uo) is det.
:- mode foldl2(pred(in, in, out, in, out) is det,
    in, in, out, in, out) is det.
:- mode foldl2(pred(in, in, out, in, out) is semidet,
    in, in, out, in, out) is semidet.
:- mode foldl2(pred(in, in, out, in, out) is nondet,
    in, in, out, in, out) is nondet.
:- mode foldl2(pred(in, in, out, in, out) is multi,
    in, in, out, in, out) is multi.

    % foldl_between(Closure, String, Start, End, !Acc)
    % is equivalent to foldl(Closure, SubString, !Acc)
    % where SubString = between(String, Start, End).
    %
    % 'Start' and 'End' are in terms of code units.
    %
:- func foldl_between(func(char, A) = A, string, int, int, A) = A.
:- pred foldl_between(pred(char, A, A), string, int, int, A, A).
:- mode foldl_between(pred(in, in, out) is det, in, in, in,
    in, out) is det.
:- mode foldl_between(pred(in, di, uo) is det, in, in, in,
    di, uo) is det.
:- mode foldl_between(pred(in, in, out) is semidet, in, in, in,
    in, out) is semidet.
:- mode foldl_between(pred(in, in, out) is nondet, in, in, in,
```

```
    in, out) is nondet.
:- mode foldl_between(pred(in, in, out) is multi, in, in, in,
    in, out) is multi.

    % foldl2_between(Closure, String, Start, End, !Acc1, !Acc2)
    % A variant of foldl_between with two accumulators.
    %
    % 'Start' and 'End' are in terms of code units.
    %
:- pred foldl2_between(pred(char, A, A, B, B),
    string, int, int, A, A, B, B).
:- mode foldl2_between(pred(in, di, uo, di, uo) is det,
    in, in, in, di, uo, di, uo) is det.
:- mode foldl2_between(pred(in, in, out, di, uo) is det,
    in, in, in, in, out, di, uo) is det.
:- mode foldl2_between(pred(in, in, out, in, out) is det,
    in, in, in, in, out, in, out) is det.
:- mode foldl2_between(pred(in, in, out, in, out) is semidet,
    in, in, in, in, out, in, out) is semidet.
:- mode foldl2_between(pred(in, in, out, in, out) is nondet,
    in, in, in, in, out, in, out) is nondet.
:- mode foldl2_between(pred(in, in, out, in, out) is multi,
    in, in, in, in, out, in, out) is multi.

    % foldr(Closure, String, !Acc):
    % As foldl/4, except that processing proceeds right-to-left.
    %
:- func foldr(func(char, T) = T, string, T) = T.
:- pred foldr(pred(char, T, T), string, T, T).
:- mode foldr(pred(in, in, out) is det, in, in, out) is det.
:- mode foldr(pred(in, di, uo) is det, in, di, uo) is det.
:- mode foldr(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode foldr(pred(in, in, out) is nondet, in, in, out) is nondet.
:- mode foldr(pred(in, in, out) is multi, in, in, out) is multi.

    % foldr_between(Closure, String, Start, End, !Acc)
    % is equivalent to foldr(Closure, SubString, !Acc)
    % where SubString = between(String, Start, End).
    %
    % 'Start' and 'End' are in terms of code units.
    %
:- func foldr_between(func(char, T) = T, string, int, int, T) = T.
:- pred foldr_between(pred(char, T, T), string, int, int, T, T).
:- mode foldr_between(pred(in, in, out) is det, in, in, in,
    in, out) is det.
:- mode foldr_between(pred(in, di, uo) is det, in, in, in,
    di, uo) is det.
```

```
:- mode foldr_between(pred(in, in, out) is semidet, in, in, in,
    in, out) is semidet.
:- mode foldr_between(pred(in, in, out) is nondet, in, in, in,
    in, out) is nondet.
:- mode foldr_between(pred(in, in, out) is multi, in, in, in,
    in, out) is multi.

%-----------------------------------------------%
%
% Formatting tables.
%

:- type justified_column
    --->    left(list(string))
    ;       right(list(string)).

    % format_table(Columns, Separator) = Table:
    %
    % This function takes a list of columns and a column separator,
    % and returns a formatted table, where each field in each column
    % has been aligned and fields are separated with Separator.
    % There will be a newline character between each pair of rows.
    % Throws an exception if the columns are not all the same length.
    % Lengths are currently measured in terms of code points.
    %
    % For example:
    %
    % format_table([right(["a", "bb", "ccc"]), left(["1", "22", "333"])],
    %   " * ")
    % would return the table:
    %   a * 1
    %  bb * 22
    % ccc * 333
    %
:- func format_table(list(justified_column), string) = string.

    % format_table_max(Columns, Separator) does the same job as format_table,
    % but allows the caller to associate a maximum width with each column.
    %
:- func format_table_max(assoc_list(justified_column, maybe(int)), string)
    = string.

%-----------------------------------------------%
%
% Converting strings to docs.
%
```

```
    % Convert a string to a pretty_printer.doc for formatting.
    %
:- func string_to_doc(string) = pretty_printer.doc.

%-----------------------------------------------%
%
% Converting strings to values of builtin types.
%

    % Convert a string to an int. The string must contain only digits [0-
9],
    % optionally preceded by a plus or minus sign. If the string does
    % not match this syntax or the number is not in the range
    % [min_int + 1, max_int], to_int fails.
    %
:- pred to_int(string::in, int::out) is semidet.

    % Convert a signed base 10 string to an int. Throws an exception if the
    % string argument does not match the regexp [+-]?[0-9]+ or the num-
ber is
    % not in the range [min_int + 1, max_int].
    %
:- func det_to_int(string) = int.

    % Convert a string in the specified base (2-36) to an int. The string
    % must contain one or more digits in the specified base, optionally
    % preceded by a plus or minus sign. For bases > 10, digits 10 to 35
    % are represented by the letters A-Z or a-z. If the string does not match
    % this syntax or the number is not in the range [min_int, max_int],
    % the predicate fails.
    %
:- pred base_string_to_int(int::in, string::in, int::out) is semidet.

    % Convert a signed base N string to an int. Throws an exception
    % if the string argument is not precisely an optional sign followed by
    % a non-empty string of base N digits and the number is in the range
    % [min_int, max_int].
    %
:- func det_base_string_to_int(int, string) = int.

    % Convert a string to a float, returning infinity or -infinity if the
    % conversion overflows. Fails if the string is not a syntactically correct
    % float literal.
    %
:- pred to_float(string::in, float::out) is semidet.

    % Convert a string to a float, returning infinity or -infinity if the
```

```
    % conversion overflows. Throws an exception if the string is not a
    % syntactically correct float literal.
    %
:- func det_to_float(string) = float.

%---------------------------------------------------%
%
% Converting values of builtin types to strings.
%

    % char_to_string(Char, String):
    %
    % Converts a character to a string, or vice versa.
    % True if 'String' is the well-formed string that encodes the code point
    % 'Char'; or, if strings are UTF-16 encoded, 'Char' is a surrogate code
    % point and 'String' is the string that contains only that surrogate code
    % point. Otherwise, 'char_to_string(Char, String)' is false.
    %
    % Throws an exception if 'Char' is the null character or a code point that
    % cannot be encoded in a string (namely, surrogate code points can-
not be
    % encoded in UTF-8 strings).
    %
:- func char_to_string(char::in) = (string::uo) is det.
:- pred char_to_string(char, string).
:- mode char_to_string(in, uo) is det.
:- mode char_to_string(out, in) is semidet.

    % A synonym for char_to_string/1.
    %
:- func from_char(char::in) = (string::uo) is det.

    % Convert an integer to a string in base 10.
    % See int_to_base_string for the string format.
    %
:- func int_to_string(int::in) = (string::uo) is det.
:- pred int_to_string(int::in, string::uo) is det.

    % A synonym for int_to_string/1.
    %
:- func from_int(int::in) = (string::uo) is det.

    % int_to_base_string(Int, Base, String):
    %
    % Convert an integer to a string in a given Base.
    % 'String' will consist of a minus sign (U+002D HYPHEN-MINUS)
    % if 'Int' is negative, followed by one or more decimal digits (0-9)
```

```
    % or uppercase letters (A-Z). There will be no leading zeros.
    %
    % Base must be between 2 and 36, both inclusive; if it is not,
    % the predicate will throw an exception.
    %
:- func int_to_base_string(int::in, int::in) = (string::uo) is det.
:- pred int_to_base_string(int::in, int::in, string::uo) is det.

    % Convert an integer to a string in base 10 with commas as thousand
    % separators.
    %
:- func int_to_string_thousands(int::in) = (string::uo) is det.

    % int_to_base_string_group(Int, Base, GroupLength, Separator, String):
    %
    % Convert an integer to a string in a given Base,
    % in the same format as int_to_base_string,
    % with Separator inserted between every GroupLength digits
    % (grouping from the end of the string).
    % If GroupLength is less than one, no separators will appear
    % in the output. Useful for formatting numbers like "1,300,000".
    %
    % Base must be between 2 and 36, both inclusive; if it is not,
    % the predicate will throw an exception.
    %
:- func int_to_base_string_group(int, int, int, string) = string.
:- mode int_to_base_string_group(in, in, in, in) = uo is det.

    % Convert an unsigned integer to a string.
    %
:- func uint_to_string(uint::in) = (string::uo) is det.

    % Convert a signed/unsigned 8/16/32/64 bit integer to a string.
    %
:- func int8_to_string(int8::in) = (string::uo) is det.
:- func uint8_to_string(uint8::in) = (string::uo) is det.
:- func int16_to_string(int16::in) = (string::uo) is det.
:- func uint16_to_string(uint16::in) = (string::uo) is det.
:- func int32_to_string(int32::in) = (string::uo) is det.
:- func uint32_to_string(uint32::in) = (string::uo) is det.
:- func int64_to_string(int64::in) = (string::uo) is det.
:- func uint64_to_string(uint64::in) = (string::uo) is det.

    % Convert a float to a string.
    % In the current implementation, the resulting float will be in the form
    % that it was printed using the format string "%#.<prec>g".
    % <prec> will be in the range p to (p+2)
```

```
    % where p = floor(mantissa_digits * log2(base_radix) / log2(10)).
    % The precision chosen from this range will be such as to allow
    % a successful decimal -> binary conversion of the float.
    %
:- func float_to_string(float::in) = (string::uo) is det.
:- pred float_to_string(float::in, string::uo) is det.

    % A synonym for float_to_string/1.
    %
:- func from_float(float::in) = (string::uo) is det.

    % Convert a c_pointer to a string. The format is "c_pointer(0xXXXX)"
    % where XXXX is the hexadecimal representation of the pointer.
    %
:- func c_pointer_to_string(c_pointer::in) = (string::uo) is det.
:- pred c_pointer_to_string(c_pointer::in, string::uo) is det.

    % A synonym for c_pointer_to_string/1.
    %
:- func from_c_pointer(c_pointer::in) = (string::uo) is det.

%-----------------------------------------------------%
%
% Converting values of arbitrary types to strings.
%

    % string(X): Returns a canonicalized string representation of the value X
    % using the standard Mercury operators.
    %
:- func string(T) = string.

    % As above, but using the supplied table of operators.
    %
:- func string_ops(ops.table, T) = string.

    % string_ops_noncanon(NonCanon, OpsTable, X, String)
    %
    % As above, but the caller specifies what behaviour should occur for
    % non-canonical terms (i.e. terms where multiple representations
    % may compare as equal):
    %
    % - 'do_not_allow' will throw an exception if (any subterm of)
    %    the argument is not canonical;
    % - 'canonicalize' will substitute a string indicating the presence
    %    of a non-canonical subterm;
    % - 'include_details_cc' will show the structure of any non-canonical
    %   subterms, but can only be called from a committed choice context.
```

```
    %
:- pred string_ops_noncanon(noncanon_handling, ops.table, T, string).
:- mode string_ops_noncanon(in(do_not_allow), in, in, out) is det.
:- mode string_ops_noncanon(in(canonicalize), in, in, out) is det.
:- mode string_ops_noncanon(in(include_details_cc), in, in, out) is cc_multi.
:- mode string_ops_noncanon(in, in, in, out) is cc_multi.

%-----------------------------------------------------%
%
% Converting values to strings based on a format string.
%

:- type poly_type
    --->    f(float)
    ;       i(int)
    ;       u(uint)
    ;       s(string)
    ;       c(char).

    % A function similar to sprintf() in C.
    %
    % For example,
    %   format("%s %i %c %f\n",
    %       [s("Square-root of"), i(2), c('='), f(1.41)], String)
    % will return
    %   String = "Square-root of 2 = 1.41\n".
    %
    % The following options available in C are supported: flags [0+-# ],
    % a field width (or *), and a precision (could be a ".*").
    %
    % Valid conversion character types are {dioxXucsfeEgGp%}. %n is not
    % supported. format will not return the length of the string.
    %
    % conv  var         output form.      effect of '#'.
    % char. type(s).
    %
    % d     int         signed integer
    % i     int         signed integer
    % o     int, uint   unsigned octal    with '0' prefix
    % x,X   int, uint   unsigned hex      with '0x', '0X' prefix
    % u     int, uint   unsigned integer
    % c     char        character
    % s     string      string
    % f     float       rational number   with '.', if precision 0
    % e,E   float       [-]m.dddddE+-xx   with '.', if precision 0
    % g,G   float       either e or f     with trailing zeros.
    % p     int, uint   integer
```

```
    %
    % An option of zero will cause any padding to be zeros rather than spaces.
    % A '-' will cause the output to be left-justified in its 'space'.
    % (Without a `-', the default is for fields to be right-justified.)
    % A '+' forces a sign to be printed. This is not sensible for string
    % and character output. A ' ' causes a space to be printed before a thing
    % if there is no sign there. The other option is the '#', which modifies
    % the output string's format. These options are normally put directly
    % after the '%'.
    %
    % Notes:
    %
    % %#.0e, %#.0E now prints a '.' before the 'e'.
    %
    % Asking for more precision than a float actually has will result in
    % potentially misleading output.
    %
    % Numbers are now rounded by precision value, not truncated as previously.
    %
    % The implementation uses the sprintf() function in C grades,
    % so the actual output will depend on the C standard library.
    %
:- func format(string, list(poly_type)) = string.
:- pred format(string::in, list(poly_type)::in, string::out) is det.

%-----------------------------------------------------%
%-----------------------------------------------------%
```

86 table_statistics

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 2007 The University of Melbourne.
% Copyright (C) 2014-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: table_statistics.m.
% Author: zs.
% Stability: low.
%
% This file is automatically imported, as if via `use_module', into every
% module that contains a `pragma memo' that asks the compiler to create
% a predicate for returning statistics about the memo table. It defines
```

```
% the data structure that this predicate will return, and some operations
% on this data structure.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module table_statistics.
:- interface.

:- import_module io.
:- import_module list.
:- import_module maybe.

:- type proc_table_statistics
    --->    proc_table_statistics(
                call_table_stats            :: table_stats_curr_prev,
                maybe_answer_table_stats    :: maybe(table_stats_curr_prev)
            ).

:- type table_stats_curr_prev
    --->    table_stats_curr_prev(
                current_stats               :: table_stats,
                stats_at_last_call          :: table_stats
            ).

:- type table_stats
    --->    table_stats(
                num_lookups                 :: int,
                num_lookups_is_dupl         :: int,
                step_statistics             :: list(table_step_stats)
            ).

    % The definition of this type be an enum whose implementation matches
    % the type MR_TableTrieStep in runtime/mercury_tabling.h. It should also
    % be kept in sync with the type table_trie_step in hlds_pred.m.
    %
:- type table_step_kind
    --->    table_step_dummy
    ;       table_step_int
    ;       table_step_char
    ;       table_step_string
    ;       table_step_float
    ;       table_step_enum
    ;       table_step_foreign_enum
    ;       table_step_general
    ;       table_step_general_addr
    ;       table_step_general_poly
```

```
    ;       table_step_general_poly_addr
    ;       table_step_typeinfo
    ;       table_step_typeclassinfo
    ;       table_step_promise_implied
    ;       table_step_int8
    ;       table_step_uint8
    ;       table_step_int16
    ;       table_step_uint16
    ;       table_step_int32
    ;       table_step_uint32
    ;       table_step_int64
    ;       table_step_uint64.

:- type table_step_stats
    --->    table_step_stats(
                table_step_var_name                 :: string,
                table_step_num_lookups              :: int,
                table_step_num_lookups_is_dupl      :: int,
                table_step_detail                   :: table_step_stat_details
            ).

:- type table_step_stat_details
    --->    step_stats_none
    ;       step_stats_start(
                start_num_node_allocs               :: int,
                start_num_node_bytes                :: int
            )
    ;       step_stats_enum(
                enum_num_node_allocs                :: int,
                enum_num_node_bytes                 :: int
            )
    ;       step_stats_hash(
                hash_num_table_allocs               :: int,
                hash_num_table_bytes                :: int,
                hash_num_link_chunk_allocs          :: int,
                hash_num_link_chunk_bytes           :: int,
                hash_num_num_key_compares_not_dupl  :: int,
                hash_num_num_key_compares_dupl      :: int,
                hash_num_resizes                    :: int,
                hash_resizes_num_old_entries        :: int,
                hash_resizes_num_new_entries        :: int
            )
    ;       step_stats_du(
                du_num_node_allocs                  :: int,
                du_num_node_bytes                   :: int,
                du_num_arg_lookups                  :: int,
                du_num_exist_lookups                :: int,
```

```
                du_enum_num_node_allocs             :: int,
                du_enum_num_node_bytes              :: int,

                du_hash_num_table_allocs            :: int,
                du_hash_num_table_bytes             :: int,
                du_hash_num_link_chunk_allocs       :: int,
                du_hash_num_link_chunk_bytes        :: int,
                du_hash_num_num_key_compares_not_dupl :: int,
                du_hash_num_num_key_compares_dupl   :: int,
                du_hash_num_resizes                 :: int,
                du_hash_resizes_num_old_entries     :: int,
                du_hash_resizes_num_new_entries     :: int
            )
    ;       step_stats_poly(
                poly_du_num_node_allocs             :: int,
                poly_du_num_node_bytes              :: int,
                poly_du_num_arg_lookups             :: int,
                poly_du_num_exist_lookups           :: int,

                poly_enum_num_node_allocs           :: int,
                poly_enum_num_node_bytes            :: int,

                poly_hash_num_table_allocs          :: int,
                poly_hash_num_table_bytes           :: int,
                poly_hash_num_link_chunk_allocs     :: int,
                poly_hash_num_link_chunk_bytes      :: int,
                poly_hash_num_num_key_compares_not_dupl :: int,
                poly_hash_num_num_key_compares_dupl :: int,
                poly_hash_num_resizes               :: int,
                poly_hash_resizes_num_old_entries   :: int,
                poly_hash_resizes_num_new_entries   :: int
            ).

:- func table_stats_difference(table_stats, table_stats) = table_stats.

:- pred write_table_stats(table_stats::in, io::di, io::uo) is det.
:- pred write_table_stats(io.text_output_stream::in, table_stats::in,
    io::di, io::uo) is det.

    % In grades that don't support tabling, all calls to get tabling stats
    % will return these dummy statistics.
    %
:- func dummy_proc_table_statistics = proc_table_statistics.

%---------------------------------------------------%
```

87 term

```
%----------------------------------------------------%
% vim: ts=4 sw=4 et ft=mercury
%----------------------------------------------------%
% Copyright (C) 1993-2000,2003-2009,2011-2012 The University of Melbourne.
% Copyright (C) 2014-2020 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%----------------------------------------------------%
%
% File: term.m.
% Main author: fjh.
% Stability: medium.
%
% This file provides a type 'term' used to represent Herbrand terms,
% and various predicates to manipulate terms and substitutions.
%
%----------------------------------------------------%
%----------------------------------------------------%

:- module term.
:- interface.

:- import_module enum.
:- import_module integer.
:- import_module list.
:- import_module map.

%----------------------------------------------------%
%
% The term type represents logic terms (Herbrand terms, in the terminology
% of logic programming theory).
%
% The term type is polymorphic. The intention is to allow terms representing
% different kinds of things to specify a different type parameter. Since
% e.g. term(type_a) is a different type from e.g. term(type_b), this should
% prevent terms of different kinds from being accidentally mixed up.
%
% For the predicates that operate on more than one term, such as unify_term,
% all the terms must use variables from the same varset.
% (You can use varset.merge_renaming to combine two different varsets.)
%

:- type term(T)
    --->    functor(
                const,
                list(term(T)),
```

```
                term.context
            )
    ;       variable(
                var(T),
                term.context
            ).

:- type var(T).

:- type const
    --->    atom(string)
    ;       integer(
                integer_base       :: integer_base,
                integer_value      :: integer,
                integer_signedness :: signedness,
                integer_size       :: integer_size
            )
    ;       string(string)
    ;       float(float)
    ;       implementation_defined(string).

:- type integer_base
    --->    base_2
    ;       base_8
    ;       base_10
    ;       base_16.

:- type signedness
    --->    signed
    ;       unsigned.

:- type integer_size
    --->    size_word
    ;       size_8_bit
    ;       size_16_bit
    ;       size_32_bit
    ;       size_64_bit.

:- type generic
    --->    generic.

:- type term ==  term(generic).
:- type var  ==  var(generic).

%-----------------------------------------------%
%
% These predicates manage the supply of variables.
```

```
%

:- type var_supply(T).

    % init_var_supply(VarSupply):
    %
    % Returns a fresh var_supply for producing fresh variables.
    %
:- func init_var_supply = var_supply(T).
:- pred init_var_supply(var_supply(T)).
:- mode init_var_supply(out) is det.
:- mode init_var_supply(in) is semidet. % implied

    % create_var(Var, !VarSupply):
    %
    % Create a fresh variable (var) and return the updated var_supply.
    %
:- pred create_var(var(T)::out, var_supply(T)::in, var_supply(T)::out) is det.

%-----------------------------------------------%

    % from_int/1 should only be applied to integers returned by to_int/1.
:- instance enum(var(_)).

    % var_id(Variable):
    %
    % Returns a unique number associated with this variable w.r.t.
    % its originating var_supply.
    %
:- func var_to_int(var(T)) = int.
:- pred var_to_int(var(T)::in, int::out) is det.

    % var_id(Variable):
    %
    % Returns a unique number associated with this variable w.r.t.
    % its originating var_supply.
    %
:- func var_id(var(T)) = int.
:- pragma obsolete(var_id/1, [var_to_int/1]).

%-----------------------------------------------%

:- type renaming(T) == map(var(T), var(T)).
:- type renaming    == renaming(generic).

:- type substitution(T) == map(var(T), term(T)).
:- type substitution    == substitution(generic).
```

```
%-----------------------------------------------%

:- pred term_to_int(term(T)::in, int::out) is semidet.

:- pred term_to_int8(term(T)::in, int8::out) is semidet.

:- pred term_to_int16(term(T)::in, int16::out) is semidet.

:- pred term_to_int32(term(T)::in, int32::out) is semidet.

:- pred term_to_int64(term(T)::in, int64::out) is semidet.

:- pred term_to_uint(term(T)::in, uint::out) is semidet.

:- pred term_to_uint8(term(T)::in, uint8::out) is semidet.

:- pred term_to_uint16(term(T)::in, uint16::out) is semidet.

:- pred term_to_uint32(term(T)::in, uint32::out) is semidet.

:- pred term_to_uint64(term(T)::in, uint64::out) is semidet.

:- pred decimal_term_to_int(term(T)::in, int::out) is semidet.

:- func int_to_decimal_term(int, context) = term(T).

:- func int8_to_decimal_term(int8, context) = term(T).

:- func int16_to_decimal_term(int16, context) = term(T).

:- func int32_to_decimal_term(int32, context) = term(T).

:- func int64_to_decimal_term(int64, context) = term(T).

:- func uint_to_decimal_term(uint, context) = term(T).

:- func uint8_to_decimal_term(uint8, context) = term(T).

:- func uint16_to_decimal_term(uint16, context) = term(T).

:- func uint32_to_decimal_term(uint32, context) = term(T).

:- func uint64_to_decimal_term(uint64, context) = term(T).

%-----------------------------------------------%
%
```

```
% Predicates to unify terms.
%

    % unify_term(TermA, TermB, !Subst):
    %
    % Unify (with occur check) two terms with respect to the current
    % substitution, and update that substitution as necessary.
    %
:- pred unify_term(term(T)::in, term(T)::in,
    substitution(T)::in, substitution(T)::out) is semidet.

    % unify_term_list(TermsA, TermsB, !Subst):
    %
    % Unify (with occur check) two lists of terms with respect to the current
    % substitution, and update that substitution as necessary.
    % Fail if the lists are not of equal length.
    %
:- pred unify_term_list(list(term(T))::in, list(term(T))::in,
    substitution(T)::in, substitution(T)::out) is semidet.

    % unify_term_dont_bind(TermA, TermB, DontBindVars, !Subst):
    %
    % Do the same job as unify_term(TermA, TermB, !Subst), but fail
    % if any of the variables in DontBindVars would become bound
    % by the unification.
    %
:- pred unify_term_dont_bind(term(T)::in, term(T)::in,
    list(var(T))::in, substitution(T)::in, substitution(T)::out) is semidet.

    % unify_term_list_dont_bind(TermsA, TermsB, DontBindVars, !Subst):
    %
    % Do the same job as unify_term_list(TermsA, TermsB, !Subst), but fail
    % if any of the variables in DontBindVars would become bound
    % by the unification.
    %
:- pred unify_term_list_dont_bind(list(term(T))::in, list(term(T))::in,
    list(var(T))::in, substitution(T)::in, substitution(T)::out) is semidet.

%-----------------------------------------------%
%
% Predicates to test subsumption.
%

    % list_subsumes(TermsA, TermsB, Subst):
    %
    % Succeeds iff the list TermsA subsumes (is more general than) TermsB,
    % producing a substitution which, when applied to TermsA, will give TermsB.
```

```
    %
:- pred list_subsumes(list(term(T))::in, list(term(T))::in,
    substitution(T)::out) is semidet.

%---------------------------------------------------%
%
% Predicates that list the variables in terms.
%

    % vars(Term, Vars):
    %
    % Vars is the list of variables contained in Term, in the order
    % obtained by traversing the term depth first, left-to-right.
    %
:- func vars(term(T)) = list(var(T)).
:- pred vars(term(T)::in, list(var(T))::out) is det.

    % As above, but with an accumulator.
    %
:- func vars_2(term(T), list(var(T))) = list(var(T)).
:- pred vars_2(term(T)::in, list(var(T))::in, list(var(T))::out) is det.

    % vars_list(TermList, Vars):
    %
    % Vars is the list of variables contained in TermList, in the order
    % obtained by traversing the list of terms depth-first, left-to-right.
    %
:- func vars_list(list(term(T))) = list(var(T)).
:- pred vars_list(list(term(T))::in, list(var(T))::out) is det.

    % contains_var(Term, Var):
    %
    % True if Term contains Var. On backtracking returns all the variables
    % contained in Term.
    %
:- pred contains_var(term(T), var(T)).
:- mode contains_var(in, in) is semidet.
:- mode contains_var(in, out) is nondet.

    % contains_var_list(TermList, Var):
    %
    % True if TermList contains Var. On backtracking returns all the variables
    % contained in Term.
    %
:- pred contains_var_list(list(term(T)), var(T)).
:- mode contains_var_list(in, in) is semidet.
:- mode contains_var_list(in, out) is nondet.
```

```
%-----------------------------------------------%
%
% Predicates that look for variables in terms, possibly after a substitution.
%

    % occurs(Term, Var, Substitution):
    %
    % True iff Var occurs in the term resulting after applying Substitution
    % to Term. Var must not be mapped by Substitution.
    %
:- pred occurs(term(T)::in, var(T)::in, substitution(T)::in) is semidet.

    % As above, except for a list of terms rather than a single term.
    %
:- pred occurs_list(list(term(T))::in, var(T)::in, substitution(T)::in)
    is semidet.

    % is_ground(Term) is true iff Term contains no variables.
    %
:- pred is_ground(term(T)::in) is semidet.

    % is_ground_in_bindings(Term, Bindings) is true iff all variables contained
    % in Term are mapped to ground terms by Bindings.
    %
:- pred is_ground_in_bindings(term(T)::in, substitution(T)::in) is semidet.

%-----------------------------------------------%
%
% Rename predicates that specify the substitution by giving the
% variable/variable pair or pairs directly.
%

    % relabel_variable(Term0, Var, ReplacementVar, Term):
    %
    % Replace all occurrences of Var in Term0 with ReplacementVar and return
    % the result as Term.
    %
:- func relabel_variable(term(T), var(T), var(T)) = term(T).
:- pred relabel_variable(term(T)::in, var(T)::in, var(T)::in, term(T)::out)
    is det.
:- pragma obsolete(relabel_variable/3, [rename_var_in_term/4]).
:- pragma obsolete(relabel_variable/4, [rename_var_in_term/4]).

    % relabel_variables(Terms0, Var, ReplacementVar, Terms):
    %
    % Replace all occurrences of Var in Terms0 with ReplacementVar and return
```

```
    % the result as Terms.
    %
:- func relabel_variables(list(term(T)), var(T), var(T)) = list(term(T)).
:- pred relabel_variables(list(term(T))::in, var(T)::in, var(T)::in,
    list(term(T))::out) is det.
:- pragma obsolete(relabel_variables/3, [rename_vars_in_terms/4]).
:- pragma obsolete(relabel_variables/4, [rename_vars_in_terms/4]).

%-----------------------------------------------%

    % rename(Term0, Var, ReplacementVar, Term):
    %
    % Replace all occurrences of Var in Term0 with ReplacementVar,
    % and return the result in Term.
    %
:- func rename(term(T), var(T), var(T)) = term(T).
:- pred rename(term(T)::in, var(T)::in, var(T)::in, term(T)::out) is det.
:- pragma obsolete(rename/3, [rename_var_in_term/4]).
:- pragma obsolete(rename/4, [rename_var_in_term/4]).

    % rename_list(Terms0, Var, ReplacementVar, Terms):
    %
    % Replace all occurrences of Var in Terms0 with ReplacementVar,
    % and return the result in Terms.
    %
:- func rename_list(list(term(T)), var(T), var(T)) = list(term(T)).
:- pred rename_list(list(term(T))::in, var(T)::in, var(T)::in,
    list(term(T))::out) is det.
:- pragma obsolete(rename_list/3, [rename_var_in_terms/4]).
:- pragma obsolete(rename_list/4, [rename_var_in_terms/4]).

%-----------------------------------------------%

    % rename_var_in_term(Var, ReplacementVar, Term0, Term):
    %
    % Replace all occurrences of Var in Term0 with ReplacementVar,
    % and return the result in Term.
    %
:- pred rename_var_in_term(var(T)::in, var(T)::in,
    term(T)::in, term(T)::out) is det.

    % rename_var_in_terms(Var, ReplacementVar, Terms0, Terms):
    %
    % Replace all occurrences of Var in Terms0 with ReplacementVar,
    % and return the result in Terms.
    %
:- pred rename_var_in_terms(var(T)::in, var(T)::in,
```

```
    list(term(T))::in, list(term(T))::out) is det.

%---------------------------------------------------%
%
% Rename predicates that specify the rename by giving an explicit
% variable to variable map.
%

    % apply_renaming(Term0, Renaming, Term):
    %
    % Apply renaming to Term0 and return the result in Term.
    %
:- func apply_renaming(term(T), renaming(T)) = term(T).
:- pred apply_renaming(term(T)::in, renaming(T)::in, term(T)::out) is det.
:- pragma obsolete(apply_renaming/2, [apply_renaming_in_term/3]).
:- pragma obsolete(apply_renaming/3, [apply_renaming_in_term/3]).

    % As above, except applies to a list of terms rather than a single term.
    %
:- func apply_renaming_to_list(list(term(T)), renaming(T)) = list(term(T)).
:- pred apply_renaming_to_list(list(term(T))::in, renaming(T)::in,
    list(term(T))::out) is det.
:- pragma obsolete(apply_renaming_to_list/2, [apply_renaming_in_terms/3]).
:- pragma obsolete(apply_renaming_to_list/3, [apply_renaming_in_terms/3]).

%---------------------------------------------------%

    % Applies apply_variable_renaming to a var.
    %
:- func apply_variable_renaming_to_var(renaming(T), var(T)) = var(T).
:- pred apply_variable_renaming_to_var(renaming(T)::in,
    var(T)::in, var(T)::out) is det.
:- pragma obsolete(apply_variable_renaming_to_var/2,
    [apply_renaming_in_var/3]).
:- pragma obsolete(apply_variable_renaming_to_var/3,
    [apply_renaming_in_var/3]).

    % Applies apply_variable_renaming to a list of vars.
    %
:- func apply_variable_renaming_to_vars(renaming(T),
    list(var(T))) = list(var(T)).
:- pred apply_variable_renaming_to_vars(renaming(T)::in,
    list(var(T))::in, list(var(T))::out) is det.
:- pragma obsolete(apply_variable_renaming_to_vars/2,
    [apply_renaming_in_vars/3]).
:- pragma obsolete(apply_variable_renaming_to_vars/3,
    [apply_renaming_in_vars/3]).
```

```
    % Same as relabel_variable, except relabels multiple variables.
    % If a variable is not in the map, it is not replaced.
    %
:- func apply_variable_renaming(term(T), renaming(T)) = term(T).
:- pred apply_variable_renaming(term(T)::in, renaming(T)::in,
    term(T)::out) is det.
:- pragma obsolete(apply_variable_renaming/2, [apply_renaming_in_term/3]).
:- pragma obsolete(apply_variable_renaming/3, [apply_renaming_in_term/3]).

    % Applies apply_variable_renaming to a list of terms.
    %
:- func apply_variable_renaming_to_list(list(term(T)), renaming(T)) =
    list(term(T)).
:- pred apply_variable_renaming_to_list(list(term(T))::in, renaming(T)::in,
    list(term(T))::out) is det.
:- pragma obsolete(apply_variable_renaming_to_list/2,
    [apply_renaming_in_terms/3]).
:- pragma obsolete(apply_variable_renaming_to_list/3,
    [apply_renaming_in_terms/3]).

%-----------------------------------------------------%

    % apply_renaming_in_var(Renaming, Var0, Var):
    %
    % Apply Renaming in Var0, and return the result as Var.
    %
:- pred apply_renaming_in_var(renaming(T)::in,
    var(T)::in, var(T)::out) is det.

    % apply_renaming_in_vars(Renaming, Vars0, Vars):
    %
    % Apply Renaming in Vars0, and return the result as Vars.
    %
:- pred apply_renaming_in_vars(renaming(T)::in,
    list(var(T))::in, list(var(T))::out) is det.

    % apply_renaming_in_term(Renaming, Term0, Term):
    %
    % Apply Renaming in Term0, and return the result as Term.
    %
:- pred apply_renaming_in_term(renaming(T)::in,
    term(T)::in, term(T)::out) is det.

    % apply_renaming_in_terms(Renaming, Terms0, Terms):
    %
    % Apply Renaming in Terms0, and return the result as Terms.
```

```
    %
:- pred apply_renaming_in_terms(renaming(T)::in,
    list(term(T))::in, list(term(T))::out) is det.

%-----------------------------------------------------%
%
% Substitution predicates that specify the substitution by giving the
% variable/term pair or pairs directly.
%

    % substitute(Term0, Var, ReplacementTerm, Term):
    %
    % Replace all occurrences of Var in Term0 with ReplacementTerm,
    % and return the result as Term.
    %
:- func substitute(term(T), var(T), term(T)) = term(T).
:- pred substitute(term(T)::in, var(T)::in, term(T)::in, term(T)::out) is det.
:- pragma obsolete(substitute/3, [substitute_var_in_term/4]).
:- pragma obsolete(substitute/4, [substitute_var_in_term/4]).

    % substitute_list(Var, ReplacementTerm, Terms0, Terms):
    %
    % Replace all occurrences of Var in Terms0 with ReplacementTerm,
    % and return the result as Terms.
    %
:- func substitute_list(list(term(T)), var(T), term(T)) = list(term(T)).
:- pred substitute_list(list(term(T))::in, var(T)::in, term(T)::in,
    list(term(T))::out) is det.
:- pragma obsolete(substitute_list/3, [substitute_var_in_terms/4]).
:- pragma obsolete(substitute_list/4, [substitute_var_in_terms/4]).

    % substitute_corresponding(Vars, ReplacementTerms, Term0, Term):
    %
    % Replace all occurrences of variables in Vars in Term0 with
    % the corresponding term in ReplacementTerms, and return the result
    % as Term. If Vars contains duplicates, or if Vars and ReplacementTerms
    % have different lengths, the behaviour is undefined and probably harmful.
    %
:- func substitute_corresponding(list(var(T)), list(term(T)),
    term(T)) = term(T).
:- pred substitute_corresponding(list(var(T))::in, list(term(T))::in,
    term(T)::in, term(T)::out) is det.
:- pragma obsolete(substitute_corresponding/3,
    [substitute_corresponding_in_term/4]).
:- pragma obsolete(substitute_corresponding/4,
    [substitute_corresponding_in_term/4]).
```

```
    % substitute_corresponding_list(Vars, ReplacementTerms, Terms0, Terms):
    %
    % Replace all occurrences of variables in Vars in Terms0 with
    % the corresponding term in ReplacementTerms, and return the result
    % as Terms. If Vars contains duplicates, or if Vars and ReplacementTerms
    % have different lengths, the behaviour is undefined and probably harmful.
    %
:- func substitute_corresponding_list(list(var(T)), list(term(T)),
    list(term(T))) = list(term(T)).
:- pred substitute_corresponding_list(list(var(T))::in, list(term(T))::in,
    list(term(T))::in, list(term(T))::out) is det.
:- pragma obsolete(substitute_corresponding_list/3,
    [substitute_corresponding_in_terms/4]).
:- pragma obsolete(substitute_corresponding_list/4,
    [substitute_corresponding_in_terms/4]).

%-----------------------------------------------%

    % substitute_var_in_term(Var, ReplacementTerm, Term0, Term):
    %
    % Replace all occurrences of Var in Term0 with ReplacementTerm,
    % and return the result in Term.
    %
:- pred substitute_var_in_term(var(T)::in, term(T)::in,
    term(T)::in, term(T)::out) is det.

    % substitute_var_in_terms(Var, ReplacementTerm, Terms0, Terms):
    %
    % Replace all occurrences of Var in Terms0 with ReplacementTerm,
    % and return the result in Terms.
    %
:- pred substitute_var_in_terms(var(T)::in, term(T)::in,
    list(term(T))::in, list(term(T))::out) is det.

    % substitute_corresponding_in_term(Vars, ReplacementTerms, Term0, Term):
    %
    % Replace all occurrences of variables in Vars in Term0 with
    % the corresponding term in ReplacementTerms, and return the result
    % as Term. If Vars contains duplicates, or if Vars and ReplacementTerms
    % have different lengths, the behaviour is undefined and probably harmful.
    %
:- pred substitute_corresponding_in_term(list(var(T))::in, list(term(T))::in,
    term(T)::in, term(T)::out) is det.

    % substitute_corresponding_in_terms(Vars, ReplacementTerms, Terms0, Terms):
    %
    % Replace all occurrences of variables in Vars in Terms0 with
```

```
    % the corresponding term in ReplacementTerms, and return the result
    % as Terms. If Vars contains duplicates, or if Vars and ReplacementTerms
    % have different lengths, the behaviour is undefined and probably harmful.
    %
:- pred substitute_corresponding_in_terms(list(var(T))::in, list(term(T))::in,
    list(term(T))::in, list(term(T))::out) is det.

%-----------------------------------------------------%
%
% Substitution predicates that specify the substitution by giving
% an explicit variable to term map.
%

    % apply_substitution(Term0, Substitution, Term):
    %
    % Apply Substitution to Term0 and return the result as Term.
    %
:- func apply_substitution(term(T), substitution(T)) = term(T).
:- pred apply_substitution(term(T)::in, substitution(T)::in,
    term(T)::out) is det.
:- pragma obsolete(apply_substitution/2,
    [apply_substitution_in_term/3]).
:- pragma obsolete(apply_substitution/3,
    [apply_substitution_in_term/3]).

    % apply_substitution_to_list(Term0, Substitution, Term):
    %
    % Apply Substitution to Term0 and return the result as Term.
    %
:- func apply_substitution_to_list(list(term(T)), substitution(T)) =
    list(term(T)).
:- pred apply_substitution_to_list(list(term(T))::in, substitution(T)::in,
    list(term(T))::out) is det.
:- pragma obsolete(apply_substitution_to_list/2,
    [apply_substitution_in_terms/3]).
:- pragma obsolete(apply_substitution_to_list/3,
    [apply_substitution_in_terms/3]).

    % apply_rec_substitution(Term0, Substitution, Term):
    %
    % Recursively apply Substitution to Term0 until no more substitutions
    % can be applied, and then return the result as Term.
    %
:- func apply_rec_substitution(term(T), substitution(T)) = term(T).
:- pred apply_rec_substitution(term(T)::in, substitution(T)::in,
    term(T)::out) is det.
:- pragma obsolete(apply_rec_substitution/2,
```

```
    [apply_rec_substitution_in_term/3]).
:- pragma obsolete(apply_rec_substitution/3,
    [apply_rec_substitution_in_term/3]).

    % apply_rec_substitution_to_list(Terms0, Substitution, Terms):
    %
    % Recursively apply Substitution to Terms0 until no more substitutions
    % can be applied, and then return the result as Terms.
    %
:- func apply_rec_substitution_to_list(list(term(T)), substitution(T)) =
    list(term(T)).
:- pred apply_rec_substitution_to_list(list(term(T))::in, substitution(T)::in,
    list(term(T))::out) is det.
:- pragma obsolete(apply_rec_substitution_to_list/2,
    [apply_rec_substitution_in_terms/3]).
:- pragma obsolete(apply_rec_substitution_to_list/3,
    [apply_rec_substitution_in_terms/3]).

%-----------------------------------------------%

    % apply_substitution_in_term(Substitution, Term0, Term):
    %
    % Apply Substitution to Term0 and return the result as Term.
    %
:- pred apply_substitution_in_term(substitution(T)::in,
    term(T)::in, term(T)::out) is det.

    % apply_substitution_in_terms(Substitution, Terms0, Terms):
    %
    % Apply Substitution to Terms0 and return the result as Terms.
    %
:- pred apply_substitution_in_terms(substitution(T)::in,
    list(term(T))::in, list(term(T))::out) is det.

    % apply_rec_substitution_in_term(Substitution, Term0, Term):
    %
    % Recursively apply Substitution to Term0 until no more substitutions
    % can be applied, and then return the result as Term.
    %
:- pred apply_rec_substitution_in_term(substitution(T)::in,
    term(T)::in, term(T)::out) is det.

    % apply_rec_substitution_in_terms(Substitution, Terms0, Terms):
    %
    % Recursively apply Substitution to Terms0 until no more substitutions
    % can be applied, and then return the result as Terms.
    %
```

```
:- pred apply_rec_substitution_in_terms(substitution(T)::in,
    list(term(T))::in, list(term(T))::out) is det.

%-----------------------------------------------------%
%
% Conversions between variables and terms.
%

    % Convert a list of terms which are all vars into a list of vars.
    % Throw an exception if the list contains any non-variables.
    %
:- func term_list_to_var_list(list(term(T))) = list(var(T)).

    % Convert a list of terms which are all vars into a list of vars.
    %
:- pred term_list_to_var_list(list(term(T))::in, list(var(T))::out) is semidet.

    % Convert a list of terms which are all vars into a list of vars
    % (or vice versa).
    %
:- func var_list_to_term_list(list(var(T))) = list(term(T)).
:- pred var_list_to_term_list(list(var(T))::in, list(term(T))::out) is det.

%-----------------------------------------------------%

    % generic_term(Term) is true iff 'Term' is a term of type
    % 'term' ie 'term(generic)'. It is useful because in some instances
    % it doesn't matter what the type of a term is, and passing it to this
    % predicate will ground the type avoiding unbound type variable warnings.
    %
:- pred generic_term(term::in) is det.

    % Coerce a term of type 'T' into a term of type 'U'.
    %
:- func coerce(term(T)) = term(U).
:- pred coerce(term(T)::in, term(U)::out) is det.

    % Coerce a var of type 'T' into a var of type 'U'.
    %
:- func coerce_var(var(T)) = var(U).
:- pred coerce_var(var(T)::in, var(U)::out) is det.

    % Coerce a var_supply of type 'T' into a var_supply of type 'U'.
    %
:- func coerce_var_supply(var_supply(T)) = var_supply(U).
:- pred coerce_var_supply(var_supply(T)::in, var_supply(U)::out) is det.
```

```
%-----------------------------------------------%

:- type term.context
    --->    context(string, int).
            % file name, line number.

    % Return the context of a term.
    %
:- func get_term_context(term(T)) = term.context.

    % Initialize the term context when reading in (or otherwise constructing)
    % a term.
    %
:- func context_init(string, int) = context.
:- pred context_init(string::in, int::in, context::out) is det.

    % Return a dummy term context.
    %
:- func dummy_context_init = context.
:- func context_init = context.
:- pred context_init(context::out) is det.

:- pred is_dummy_context(context::in) is semidet.

    % Given a term context, return the source line number.
    %
:- func context_line(context) = int.
:- pred context_line(context::in, int::out) is det.

    % Given a term context, return the source file.
    %
:- func context_file(context) = string.
:- pred context_file(context::in, string::out) is det.

%-----------------------------------------------%
%-----------------------------------------------%
```

88 term_conversion

```
%-----------------------------------------------%
% vim: ts=4 sw=4 et ft=mercury
%-----------------------------------------------%
% Copyright (C) 2015-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------%
```

```
%
% File: term_conversion.m.
% Stability: medium.
%
% This file provides predicates to convert values of arbitrary types to terms,
% and vice versa.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module term_conversion.
:- interface.

:- import_module list.
:- import_module term.
:- import_module type_desc.
:- import_module univ.

%-----------------------------------------------------%
%
% Types that record the results of term to type conversions.
%

:- type term_to_type_result(T, U)
    --->    ok(T)
    ;       error(term_to_type_error(U)).

:- type term_to_type_result(T) == term_to_type_result(T, generic).

:- type term_to_type_error(T)
    --->    type_error(
                term(T),
                type_desc.type_desc,
                context,
                term_to_type_context
            )
    ;       mode_error(
                var(T),
                term_to_type_context
            ).

:- type term_to_type_context == list(term_to_type_arg_context).

:- type term_to_type_arg_context
    --->    arg_context(
                const,      % functor
                int,        % argument number (starting from 1)
```

```
                context     % filename & line number
            ).

%-----------------------------------------------------%
%
% The following predicates can convert values of (almost) any type
% to the type `term' and back again.
%

    % try_term_to_type(Term, Result):
    %
    % Try to convert the given term to a ground value of type T.
    % If successful, return `ok(X)' where X is the converted value.
    % If Term is not ground, return `mode_error(Var, Context)',
    % where Var is a variable occurring in Term.
    % If Term is not a valid term of the specified type, return
    % `type_error(SubTerm, ExpectedType, Context, ArgContexts)',
    % where SubTerm is a sub-term of Term and ExpectedType is the type
    % expected for that part of Term.
    % Context specifies the file and line number where the
    % offending part of the term was read in from, if available.
    % ArgContexts specifies the path from the root of the term
    % to the offending subterm.
    %
:- func try_term_to_type(term(U)) = term_to_type_result(T, U).
:- pred try_term_to_type(term(U)::in, term_to_type_result(T, U)::out) is det.

    % term_to_type(Term, Type) :- try_term_to_type(Term, ok(Type)).
    %
:- pred term_to_type(term(U)::in, T::out) is semidet.

    % Like term_to_type, but calls error/1 rather than failing.
    %
:- func det_term_to_type(term(_)) = T.
:- pred det_term_to_type(term(_)::in, T::out) is det.

    % Converts a value to a term representation of that value.
    %
:- func type_to_term(T) = term(_).
:- pred type_to_term(T::in, term(_)::out) is det.

    % Convert the value stored in the univ (as distinct from the univ itself)
    % to a term.
    %
:- func univ_to_term(univ) = term(_).
:- pred univ_to_term(univ::in, term(_)::out) is det.
```

```
%-----------------------------------------------------%
%-----------------------------------------------------%
```

89 term_io

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 1994-2006, 2009, 2011-2012 The University of Melbourne.
% Copyright (C) 2014-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: term_io.m.
% Main author: fjh.
% Stability: medium to high.
%
% This file encapsulates all the term I/O.
% This exports predicates to read and write terms in the
% nice ground representation provided in term.m.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module term_io.
:- interface.

:- import_module char.
:- import_module io.
:- import_module ops.
:- import_module stream.
:- import_module term.
:- import_module varset.

%-----------------------------------------------------%

:- type read_term(T)
    --->    eof
    ;       error(string, int)
    ;       term(varset(T), term(T)).

:- type read_term == read_term(generic).

    % Read a term from the current input stream or from the given input stream.
    %
```

```
    % Similar to NU-Prolog read_term/2, except that resulting term
    % is in the ground representation.
    %
    % Binds Result to either 'eof', 'term(VarSet, Term)', or
    % 'error(Message, LineNumber)'.
    %
:- pred read_term(read_term(T)::out, io::di, io::uo) is det.
:- pred read_term(io.text_input_stream::in, read_term(T)::out,
    io::di, io::uo) is det.

    % As above, except uses the given operator table instead of
    % the standard Mercury operators.
    %
:- pred read_term_with_op_table(Ops::in,
    read_term(T)::out, io::di, io::uo) is det <= op_table(Ops).
:- pred read_term_with_op_table(io.text_input_stream::in, Ops::in,
    read_term(T)::out, io::di, io::uo) is det <= op_table(Ops).

%-----------------------------------------------%

    % Writes a term to the current output stream or to the specified output
    % stream. Uses the variable names specified by the varset.
    % Writes _N for all unnamed variables, with N starting at 0.
    %
:- pred write_term(varset(T)::in, term(T)::in, io::di, io::uo) is det.
:- pred write_term(io.output_stream::in, varset(T)::in, term(T)::in,
    io::di, io::uo) is det.

    % As above, except uses the given operator table instead of the
    % standard Mercury operators.
    %
:- pred write_term_with_op_table(Ops::in,
    varset(T)::in, term(T)::in, io::di, io::uo) is det <= op_table(Ops).
:- pred write_term_with_op_table(io.text_output_stream::in, Ops::in,
    varset(T)::in, term(T)::in, io::di, io::uo) is det <= op_table(Ops).

    % As above, except it appends a period and new-line.
    %
:- pred write_term_nl(varset(T)::in, term(T)::in, io::di, io::uo) is det.
:- pred write_term_nl(io.text_output_stream::in, varset(T)::in, term(T)::in,
    io::di, io::uo) is det.

    % As above, except it appends a period and new-line.
    %
:- pred write_term_nl_with_op_table(Ops::in,
    varset(T)::in, term(T)::in, io::di, io::uo) is det <= op_table(Ops).
:- pred write_term_nl_with_op_table(io.text_output_stream::in, Ops::in,
```

```
    varset(T)::in, term(T)::in, io::di, io::uo) is det <= op_table(Ops).

%---------------------------------------------------%

    % Writes a constant (integer, float, string, or atom) to
    % the current output stream, or to the specified output stream.
    %
:- pred write_constant(const::in, io::di, io::uo) is det.
:- pred write_constant(io.text_output_stream::in, const::in,
    io::di, io::uo) is det.

    % Like write_constant, but return the result in a string.
    %
:- func format_constant(const) = string.

%---------------------------------------------------%

    % Writes a variable to the current output stream, or to the
    % specified output stream.
    %
:- pred write_variable(var(T)::in, varset(T)::in, io::di, io::uo) is det.
:- pred write_variable(io.text_output_stream::in, var(T)::in, varset(T)::in,
    io::di, io::uo) is det.

    % As above, except uses the given operator table instead of the
    % standard Mercury operators.
    %
:- pred write_variable_with_op_table(Ops::in,
    var(T)::in, varset(T)::in, io::di, io::uo) is det <= op_table(Ops).
:- pred write_variable_with_op_table(io.text_output_stream::in, Ops::in,
    var(T)::in, varset(T)::in, io::di, io::uo) is det <= op_table(Ops).

%---------------------------------------------------%

    % Given a character C, write C in single-quotes,
    % escaped if necessary, to stdout.
    %
:- pred quote_char(char::in, io::di, io::uo) is det.
:- pred quote_char(Stream::in, char::in, State::di, State::uo) is det
    <= (stream.writer(Stream, string, State),
    stream.writer(Stream, char, State)).

    % Like quote_char, but return the result in a string.
    %
:- func quoted_char(char) = string.

    % Given a character C, write C, escaped if necessary, to stdout.
```

```
    % The character is not enclosed in quotes.
    %
:- pred write_escaped_char(char::in, io::di, io::uo) is det.
:- pred write_escaped_char(Stream::in, char::in, State::di, State::uo) is det
    <= (stream.writer(Stream, string, State),
    stream.writer(Stream, char, State)).

    % Like write_escaped_char, but return the result in a string.
    %
:- func escaped_char(char) = string.

    % A reversible version of escaped_char.
    %
:- pred string_is_escaped_char(char, string).
:- mode string_is_escaped_char(in, out) is det.
:- mode string_is_escaped_char(out, in) is semidet.

%-----------------------------------------------%

    % Given a string S, write S in double-quotes, with characters
    % escaped if necessary, to stdout.
    %
:- pred quote_string(string::in, io::di, io::uo) is det.
:- pred quote_string(Stream::in, string::in, State::di, State::uo) is det
    <= (stream.writer(Stream, string, State),
    stream.writer(Stream, char, State)).

    % Like quote_string, but return the result in a string.
    %
:- func quoted_string(string) = string.

    % Given a string S, write S, with characters escaped if necessary,
    % to stdout. The string is not enclosed in quotes.
    %
:- pred write_escaped_string(string::in, io::di, io::uo) is det.
:- pred write_escaped_string(Stream::in, string::in,
    State::di, State::uo) is det
    <= (stream.writer(Stream, string, State),
    stream.writer(Stream, char, State)).

    % Like write_escaped_char, but return the result in a string.
    %
:- func escaped_string(string) = string.

%-----------------------------------------------%

    % Given an atom-name A, write A, enclosed in single-quotes if necessary,
```

```
    % with characters escaped if necessary, to stdout.
    %
:- pred quote_atom(string::in, io::di, io::uo) is det.
:- pred quote_atom(Stream::in, string::in, State::di, State::uo) is det
    <= (stream.writer(Stream, string, State),
    stream.writer(Stream, char, State)).

    % Like quote_atom, but return the result in a string.
    %
:- func quoted_atom(string) = string.

%---------------------------------------------------%
%---------------------------------------------------%
```

90 term_to_xml

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 1993-2007, 2010-2011 The University of Melbourne.
% Copyright (C) 2014-2015, 2017-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: term_to_xml.m.
% Main author: maclarty.
% Stability: low.
%
% This module provides two mechanisms for converting Mercury terms
% to XML documents.
%
% Method 1
% --------
% The first method requires a type to be an instance of the xmlable typeclass
% before values of the type can be written as XML.
% Members of the xmlable typeclass must implement a to_xml method which
% maps values of the type to XML elements.
% The XML elements may contain arbitrary children, comments and data.
%
% Method 2
% --------
% The second method is less flexible than the first, but it allows for the
% automatic generation of a DTD.
% Each functor in a term is given a corresponding well-formed element name
% in the XML document according to a mapping. Some predefined mappings are
```

```
% provided, but user defined mappings may also be used.
%
% Method 1 vs. Method 2
% ---------------------
%
% Method 2 can automatically generate DTDs, while method 1 cannot.
%
% Method 1 allows values of a specific type to be mapped to arbitrary XML
% elements with arbitrary children and arbitrary attributes.
% With method 2, each functor in a term can be mapped to only one XML element.
% Method 2 also only allows a selected set of attributes.
%
% Method 1 is useful for mapping a specific type to XML, for example
% mapping terms which represent mathematical expressions to MathML.
% Method 2 is useful for mapping terms of *any* type to XML.
%
% In both methods, the XML document can be annotated with a stylesheet
% reference.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module term_to_xml.
:- interface.

:- import_module deconstruct.
:- import_module list.
:- import_module maybe.
:- import_module stream.
:- import_module type_desc.

%-----------------------------------------------------%
%
% Method 1 interface.
%

    % Instances of this typeclass can be converted to XML.
    %
:- typeclass xmlable(T) where [
    func to_xml(T::in) = (xml::out(xml_doc)) is det
].

    % Values of this type represent either a full XML document
    % or a portion of one.
    %
:- type xml
    --->    elem(
```

```
                % An XML element with a name, list of attributes
                % and a list of children.
                element_name    :: string,
                attributes      :: list(attr),
                children        :: list(xml)
            )

    ;       data(string)
            % Textual data. '<', '>', '&', ''' and '"' characters
            % will be replaced by '&lt;', '&gt;', '&amp;', '&apos;'
            % and '&quot;' respectively.

    ;       cdata(string)
            % Data to be enclosed in '<![CDATA[' and ']]>' tags.
            % The string may not contain "]]>" as a substring.
            % If it does, then the generated XML will be invalid.

    ;       comment(string)
            % An XML comment. The comment should not include
            % the '<!--' and '-->'. Any occurrences of the substring "-
-"
            % will be replaced by " - ", since "--" is not allowed
            % in XML comments.

    ;       entity(string)
            % An entity reference. The string will have '&' prepended
            % and ';' appended before being output.

    ;       raw(string).
            % Raw XML data. The data will be written out verbatim.

    % An XML document must have an element at the top level.
    % The following inst is used to enforce this restriction.
    %
:- inst xml_doc for xml/0
    --->    elem(
                ground, % element_name
                ground, % attributes
                ground  % children
            ).

    % An element attribute, mapping a name to a value.
    %
:- type attr
    --->    attr(string, string).

    % Values of this type specify the DOCTYPE of an XML document when
```

```
    % the DOCTYPE is defined by an external DTD.
    %
:- type doctype
    --->    public(string)                  % Formal Public Identifier (FPI)
    ;       public_system(string, string)   % FPI, URL
    ;       system(string).                 % URL

    % Values of this type specify whether a DTD should be included in
    % a generated XML document, and if so, how.
    %
:- type maybe_dtd
    --->    embed_dtd
            % Generate and embed the entire DTD in the document
            % (only available for method 2).

    ;       external_dtd(doctype)
            % Included a reference to an external DTD.

    ;       no_dtd.
            % Do not include any DOCTYPE information.

:- inst non_embedded_dtd for maybe_dtd/0
    --->    external_dtd(ground)
    ;       no_dtd.

    % Values of this type indicate whether a stylesheet reference should be
    % included in a generated XML document.
    %
:- type maybe_stylesheet
    --->    with_stylesheet(
                stylesheet_type :: string, % For example "text/xsl"
                stylesheet_href :: string
            )
    ;       no_stylesheet.

    % write_xml_doc(Stream, Term, !State):
    %
    % Output Term as an XML document to the given stream.
    % Term must be an instance of the xmlable typeclass.
    %
:- pred write_xml_doc(Stream::in, T::in, State::di, State::uo)
    is det <= (xmlable(T), stream.writer(Stream, string, State)).

    % write_xml_doc_style_dtd(Stream, Term, MaybeStyleSheet, MaybeDTD, !State):
    %
    % Write Term to the given stream as an XML document.
    % MaybeStyleSheet and MaybeDTD specify whether or not a stylesheet
```

```
    % reference and/or a DTD should be included.
    % Using this predicate, only external DTDs can be included, i.e.
    % a DTD cannot be automatically generated and embedded
    % (that feature is available only for method 2 -- see below).
    %
:- pred write_xml_doc_style_dtd(Stream::in, T::in, maybe_stylesheet::in,
    maybe_dtd::in(non_embedded_dtd), State::di, State::uo) is det
    <= (xmlable(T), stream.writer(Stream, string, State)).

    % write_xml_header(Stream, MaybeEncoding, !State):
    %
    % Write an XML header (i.e. '<?xml version="1.0"?>) to the
    % current file output stream.
    % If MaybeEncoding is yes(Encoding), then include 'encoding="Encoding"'
    % in the header.
    %
:- pred write_xml_header(Stream::in, maybe(string)::in,
    State::di, State::uo) is det <= stream.writer(Stream, string, State).

    % write_xml_element(Stream, Indent, Term, !State):
    %
    % Write Term out as XML to the given stream, using Indent as the
    % indentation level (each indentation level is one tab character).
    % No '<?xml ... ?>' header will be written.
    % This is useful for generating large XML documents piecemeal.
    %
:- pred write_xml_element(Stream::in, int::in, T::in,
    State::di, State::uo) is det
    <= (xmlable(T), stream.writer(Stream, string, State)).

%-----------------------------------------------%
%
% Method 2 interface.
%

    % Values of this type specify which mapping from functors to elements
    % to use when generating XML. The role of a mapping is twofold:
    %   1. To map functors to elements, and
    %   2. To map functors to a set of attributes that should be
    %      generated for the corresponding element.
    %
    % We provide two predefined mappings:
    %
    %   1. simple: The functors '[]', '[|]' and '{}' are mapped to the elements
    %   'List', 'Nil' and 'Tuple' respectively. Arrays are assigned the
    %   'Array' element. The builtin types are assigned the elements 'Int',
    %   'Int8', 'Int16', 'Int32' 'Int64', 'UInt', 'UInt8', 'UInt16, 'UInt32',
```

```
    %   'UInt64', 'String', 'Float' and 'Char'. All other functors are assigned
    %   elements with the same name as the functor provided the functor name is
    %   well formed and does not start with a capital letter. Otherwise, a
    %   mangled version of the functor name is used.
    %
    %   All elements except those corresponding to builtin types will have
    %   their 'functor', 'arity', 'type' and 'field' (if there is a field name)
    %   attributes set. Elements corresponding to builtin types will just have
    %   their 'type' and possibly their 'field' attributes set.
    %
    %   The 'simple' mapping is designed to be easy to read and use, but
    %   may result in the same element being assigned to different functors.
    %
    %   2. unique: Here we use the same mapping as 'simple' except we append
    %   the functor arity for discriminated unions and a mangled version
    %   of the type name for every element. The same attributes as the
    %   'simple' scheme are provided. The advantage of this scheme is that
    %   it maps each functor to a unique element. This means that it will
    %   always be possible to generate a DTD using this mapping so long as
    %   there is only one top level functor and no unsupported types
    %   can appear in terms of the type.
    %
    % A custom mapping can be provided using the 'custom' functor. See the
    % documentation for the element_pred type below for more information.
    %
:- type element_mapping
    --->    simple
    ;       unique
    ;       custom(element_pred).

:- inst element_mapping for element_mapping/0
    --->    simple
    ;       unique
    ;       custom(element_pred).

    % Deterministic procedures with the following signature can be used as
    % custom functor to element mappings. The inputs to the procedure are
    % a type and some information about a functor for that type if the type
    % is a discriminated union. The output should be a well formed XML element
    % name and a list of attributes that should be set for that element.
    % See the types 'maybe_functor_info' and 'attr_from_source' below.
    %
:- type element_pred == (pred(type_desc, maybe_functor_info, string,
    list(attr_from_source))).

:- inst element_pred == (pred(in, in, out, out) is det).
```

```
    % Values of this type are passed to custom functor-to-element mapping
    % predicates to tell the predicate which functor to generate
    % an element name for if the type is a discriminated union.
    % If the type is not a discriminated union, then non_du is passed
    % to the predicate when requesting an element for the type.
    %
:- type maybe_functor_info
    --->    du_functor(
                % The functor's name and arity.
                functor_name    :: string,
                functor_arity   :: int
            )

    ;       non_du.
            % The type is not a discriminated union.

    % Values of this type specify attributes that should be set from
    % a particular source. The attribute_name field specifies the name
    % of the attribute in the generated XML and the attribute_source
    % field indicates where the attribute's value should come from.
    %
:- type attr_from_source
    --->    attr_from_source(
                attr_name   :: string,
                attr_source :: attr_source
            ).

    % Possible attribute sources.
    %
:- type attr_source
    --->    functor
            % The original functor name as returned by
            % deconstruct.deconstruct/5.

    ;       field_name
            % The field name, if the functor appears in a named field.
            % (If the field is not named, this attribute is omitted.)

    ;       type_name
            % The fully qualified type name the functor is for.

    ;       arity.
            % The arity of the functor as returned by
            % deconstruct.deconstruct/5.

    % To support third parties generating XML which is compatible with the
    % XML generated using method 2, a DTD for a Mercury type can also be
```

```
    % generated. A DTD for a given type and functor-to-element mapping may
    % be generated provided the following conditions hold:
    %
    %   1. If the type is a discriminated union, then there must be only one
    %   top-level functor for the type. This is because the top level functor
    %   will be used to generate the document type name.
    %
    %   2. The functor-to-element mapping must map each functor to a
    %   unique element name for every functor that could appear in
    %   terms of the type.
    %
    %   3. Only types whose terms consist of discriminated unions,
    %   arrays and the builtin types 'int', 'string', 'character' and
    %   'float' can be used to automatically generate DTDs.
    %   Existential types are not supported either.
    %
    % The generated DTD is also a good reference when creating a stylesheet
    % as it contains comments describing the mapping from functors to elements.
    %
    % Values of the following type indicate whether a DTD was successfully
    % generated or not.
    %
:- type dtd_generation_result
    --->    ok

    ;       multiple_functors_for_root
            % The root type is a discriminated union with multiple functors.

    ;       duplicate_elements(
                % The functor-to-element mapping maps different functors
                % to the same element. The arguments identify the duplicate
                % element and a list of the types whose functors map
                % to that element.
                duplicate_element   :: string,
                duplicate_types     :: list(type_desc)
            )

    ;       unsupported_dtd_type(type_desc)
            % At the moment we only support generation of DTDs for types
            % made up of discriminated unions, arrays, strings, ints,
            % characters and floats. If a component type is not supported,
            % then it is returned as the argument of this functor.

    ;       type_not_ground(pseudo_type_desc).
            % If one of the arguments of a functor is existentially typed,
            % then the pseudo_type_desc for the existentially quantified
            % argument is returned as the argument of this functor.
```

```
            % Since the values of existentially typed arguments can be of
            % any type (provided any typeclass constraints are satisfied),
            % it is not generally possible to generate DTD rules for functors
            % with existentially typed arguments.

    % write_xml_doc_general(Stream, Term, ElementMapping,
    %   MaybeStyleSheet, MaybeDTD, DTDResult, !State):
    %
    % Write Term to the given stream as an XML document using ElementMapping
    % as the scheme to map functors to elements. MaybeStyleSheet and MaybeDTD
    % specify whether or not a stylesheet reference and/or a DTD should be
    % included. Any non-canonical terms will be canonicalized. If an embedded
    % DTD is requested, but it is not possible to generate a DTD for Term
    % using ElementMapping, then a value other than 'ok' is returned in
    % DTDResult and nothing is written out. See the dtd_generation_result type
    % for a list of the other possible values of DTDResult and their meanings.
    %
:- pred write_xml_doc_general(Stream::in, T::in,
    element_mapping::in(element_mapping), maybe_stylesheet::in,
    maybe_dtd::in, dtd_generation_result::out, State::di, State::uo) is det
    <= stream.writer(Stream, string, State).

    % write_xml_doc_general_cc(Stream, Term, ElementMapping, MaybeStyleSheet,
    %    MaybeDTD, DTDResult, !State):
    %
    % Write Term to the current file output stream as an XML document using
    % ElementMapping as the scheme to map functors to elements.
    % MaybeStyleSheet and MaybeDTD specify whether or not a stylesheet
    % reference and/or a DTD should be included. Any non-canonical terms
    % will be written out in full. If an embedded DTD is requested, but
    % it is not possible to generate a DTD for Term using ElementMapping,
    % then a value other than 'ok' is returned in DTDResult and nothing is
    % written out. See the dtd_generation_result type for a list of the
    % other possible values of DTDResult and their meanings.
    %
:- pred write_xml_doc_general_cc(Stream::in, T::in,
    element_mapping::in(element_mapping), maybe_stylesheet::in,
    maybe_dtd::in, dtd_generation_result::out, State::di, State::uo)
    is cc_multi <= stream.writer(Stream, string, State).

    % write_xml_element_general(Stream, NonCanon, MakeElement, IndentLevel,
    %   Term, !State):
    %
    % Write XML elements for the given term and all its descendents, using
    % IndentLevel as the initial indentation level (each indentation level
    % is one tab character) and using the MakeElement predicate to map
    % functors to elements. No <?xml ... ?> header will be written.
```

```
    % Non-canonical terms will be handled according to the value of NonCanon.
    % See the deconstruct module in the standard library for more information
    % on this argument.
    %
:- pred write_xml_element_general(Stream, deconstruct.noncanon_handling,
    element_mapping, int, T, State, State)
    <= stream.writer(Stream, string, State).
:- mode write_xml_element_general(in, in(do_not_allow), in(element_mapping),
    in, in, di, uo) is det.
:- mode write_xml_element_general(in, in(canonicalize), in(element_mapping),
    in, in, di, uo) is det.
:- mode write_xml_element_general(in, in(include_details_cc),
    in(element_mapping), in, in, di, uo) is cc_multi.
:- mode write_xml_element_general(in, in, in(element_mapping),
    in, in, di, uo) is cc_multi.

%-----------------------------------------------%

    % can_generate_dtd(ElementMapping, Type) = Result:
    %
    % Check if a DTD can be generated for the given Type using the
    % functor-to-element mapping scheme ElementMapping. Return 'ok' if it
    % is possible to generate a DTD. See the documentation of the
    % dtd_generation_result type for the meaning of the return value when
    % it is not 'ok'.
    %
:- func can_generate_dtd(element_mapping::in(element_mapping),
    type_desc::in) = (dtd_generation_result::out) is det.

    % write_dtd(Stream, Term, ElementMapping, DTDResult, !State):
    %
    % Write a DTD for the given term to the current file output stream using
    % ElementMapping to map functors to elements. If a DTD cannot be generated
    % for Term using ElementMapping, then a value other than 'ok' is returned
    % in DTDResult and nothing is written. See the dtd_generation_result type
    % for a list of the other possible values of DTDResult and their meanings.
    %
:- pred write_dtd(Stream::in, T::unused,
    element_mapping::in(element_mapping), dtd_generation_result::out,
    State::di, State::uo) is det
    <= stream.writer(Stream, string, State).

    % write_dtd_for_type(Stream, Type, ElementMapping, DTDResult, !State):
    %
    % Write a DTD for the given type to the given stream. If a DTD cannot
    % be generated for Type using ElementMapping then a value other than 'ok'
    % is returned in DTDResult and nothing is written. See the
```

```
    % dtd_generation_result type for a list of the other possible values
    % of DTDResult and their meanings.
    %
:- pred write_dtd_from_type(Stream::in, type_desc::in,
    element_mapping::in(element_mapping), dtd_generation_result::out,
    State::di, State::uo) is det <= stream.writer(Stream, string, State).

%---------------------------------------------------%
%---------------------------------------------------%
```

91 thread.barrier

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 2005, 2014 Mission Critical IT.
% Copyright (C) 2014-2015, 2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: thread.barrier.m
% Original author: Peter Ross
% Stability: low
%
% This module provides a barrier implementation.
%
% A barrier is a position in a program that any thread (of N threads) must
% be suspended at until all the other threads (of N) reach the same
% position.
%
% Barriers are represented by calls to barrier/3 (defined below). Different
% code locations can belong to the same conceptual barrier using values of
% type barrier. The same code location can also be used by multiple
% barriers by supplying different values.
%
%---------------------------------------------------%
%---------------------------------------------------%

:- module thread.barrier.
:- interface.

:- import_module io.

:- type barrier.
```

```
    % init(N, Barrier, !IO)
    %
    % Create a barrier for N threads.
    %
:- pred init(int::in, barrier::out, io::di, io::uo) is det.

    % wait(Barrier, !IO)
    %
    % Indicate that the current thread has reached the barrier. Throws a
    % software_error/1 exception if this barrier has been used by more than
    % N threads.
    %
:- pred wait(barrier::in, io::di, io::uo) is det.

    % release_barrier(Barrier, !IO)
    %
    % Release all the threads waiting at the barrier regardless of whether
    % or not N threads have arrived at the barrier. This can be called by
    % any thread, it does not have to be a thread that would normally call
    % wait/3.
    %
:- pred release(barrier::in, io::di, io::uo) is det.

%-----------------------------------------------------%
%-----------------------------------------------------%
```

92 thread.channel

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 2000-2001, 2006-2007 The University of Melbourne.
% Copyright (C) 2014-2015, 2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: thread.channel.m.
% Main author: petdr.
% Stability: low.
%
% A mvar can only contain a single value, a channel on the other hand provides
% unbounded buffering.
%
% For example a program could consist of 2 worker threads and one logging
% thread. The worker threads can place messages into the channel, and they
```

```
% will be buffered for processing by the logging thread.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module thread.channel.
:- interface.

:- import_module io.
:- import_module maybe.

%-----------------------------------------------------%

:- type channel(T).

    % Initialise a channel.
    %
:- pred init(channel(T)::out, io::di, io::uo) is det.

    % Put an item at the end of the channel.
    %
:- pred put(channel(T)::in, T::in, io::di, io::uo) is det.

    % Take an item from the start of the channel, block if there is
    % nothing in the channel.
    %
:- pred take(channel(T)::in, T::out, io::di, io::uo) is det.

    % Take an item from the start of the channel.
    % Returns immediately with no if the channel was empty.
    %
:- pred try_take(channel(T)::in, maybe(T)::out, io::di, io::uo) is det.

    % Duplicate a channel. The new channel sees all (and only) the
    % data written to the channel after the 'duplicate'/4 call.
    %
:- pred duplicate(channel(T)::in, channel(T)::out, io::di, io::uo)
    is det.

    % Place an item back at the start of the channel.
    %
    % WARNING: a call to 'channel.untake' will deadlock if a call to
    % 'channel.take' is blocked on the same channel.
    %
:- pragma obsolete(untake/4).
:- pred untake(channel(T)::in, T::in, io::di, io::uo) is det.
```

```
%---------------------------------------------------%
%---------------------------------------------------%
```

93 thread.closeable_channel

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 2020 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: thread.closeable_channel.m.
% Main author: wangp.
% Stability: low.
%
% Unbounded closeable channels.
%
%---------------------------------------------------%
%---------------------------------------------------%

:- module thread.closeable_channel.
:- interface.

:- import_module bool.
:- import_module io.

%---------------------------------------------------%

:- type closeable_channel(T).

    % Initialise a channel.
    %
:- pred init(closeable_channel(T)::out, io::di, io::uo) is det.

    % Put an item at the end of the channel.
    % Returns 'yes' if successful, or 'no' if the channel is closed.
    %
:- pred put(closeable_channel(T)::in, T::in, bool::out, io::di, io::uo)
    is det.

    % Close a channel. Once a channel is closed, no more items can be added
    % to it. Closing a channel that is already closed has no effect.
    %
:- pred close(closeable_channel(T)::in, io::di, io::uo) is det.
```

```
:- type take_result(T)
    --->    ok(T)
    ;       closed.

    % Take an item from the start of the channel, blocking until an item is
    % available or until the channel is closed. Returns ‘ok(Item)’ if ‘Item’
    % was taken, or ‘closed’ if the channel is closed.
    %
:- pred take(closeable_channel(T)::in, take_result(T)::out, io::di, io::uo)
    is det.

:- type try_take_result(T)
    --->    ok(T)
    ;       closed
    ;       would_block.

    % Take an item from the start of the channel, but do not block.
    % Returns ‘ok(Item)’ if ‘Item’ was taken from the channel,
    % ‘closed’ if no item was taken because the channel is closed, or
    % ‘would_block’ if no item could be taken from the channel without
    % blocking. ‘would_block’ may be returned even if the channel is not
    % empty.
    %
:- pred try_take(closeable_channel(T)::in, try_take_result(T)::out,
    io::di, io::uo) is det.

%---------------------------------------------------%
%---------------------------------------------------%
```

94 thread.future

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 2014-2015, 2018 The Mercury Team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: thread.future.m.
% Authors: pbone.
% Stability: low.
%
% This module defines the data types future_io/1 and future/1 which are
% useful for parallel and concurrent programming.
```

```
%
% A future represents a value that might not exist yet.  A value for a
% future may be provided exactly once, but can be read any number of times.
% In these situations futures can be faster than mvars as their
% implementation is simpler: they need only one semaphore and they can avoid
% using it in some cases.
%
% There are two kinds of futures:
%
%   + future(T) is a value that will be evaluated by another thread.  The
%     function future/1 will spawn a new thread to evaluate its argument
%     whose result can be retrieved later by calling the function wait/1.
%     For example:
%
%       Future = future(SomeFunction),
%       ... do something in the meantime ...
%       Value = wait(Future).
%
%   + future_io(T) provides more flexibility, allowing the caller to control
%     the creation of the thread that provides its value.  It can be used
%     as follows:
%
%       First:
%           future(Future, !IO),
%
%       Then in a separate thread:
%           signal(Future, Value0, !IO),
%
%       Finally, in the original thread:
%           wait(Future, Value, !IO),
%
%     This is more flexible because the thread can be used to signal
%     multiple futures or do other things, but it requires the I/O state.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module thread.future.
:- interface.

%-----------------------------------------------------%

    % future/1 represents a value that will be computed by another thread.
    %
:- type future(T).

    % Create a future which has the value that the argument, when evaluated,
```

```
    % will produce.  This function will create a thread to evaluate the
    % argument using spawn/3.
    %
    % If the argument throws an exception, that exception will be rethrown by
    % wait/1.
    %
:- func future((func) = T) = future(T).

    % Return the value of the future, blocking until the value is available.
    %
:- func wait(future(T)) = T.

%---------------------------------------------------%

    % future_io/1 represents a value that may not have been computed yet.
    % Future values are intended to be computed by separate threads (using
    % spawn/3).
    %
    % Generally in computer science and in some other languages this is
    % known as a promise.  We called it future_io because promise is a
    % reserved word in Mercury.
    %
:- type future_io(T).

    % Create a new empty future_io.
    %
:- pred init(future_io(T)::uo, io::di, io::uo) is det.

    % Provide a value for the future_io and signal any waiting threads.  Any
    % further calls to wait will return immediately.
    %
    % Calling signal multiple times will result in undefined behaviour.
    %
:- pred signal(future_io(T)::in, T::in, io::di, io::uo) is det.

    % Return the future_io's value, potentially blocking until it is
    % signaled.
    %
:- pred wait(future_io(T)::in, T::out, io::di, io::uo) is det.

%---------------------------------------------------%
%---------------------------------------------------%
```

95 thread

```
%---------------------------------------------------%
```

```
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 2000-2001, 2003-2004, 2006-2008, 2010-2011 The University
% of Melbourne.
% Copyright (C) 2014-2018 The Mercury Team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: thread.m.
% Authors: conway, wangp.
% Stability: medium.
%
% This module defines the Mercury concurrency interface.
%
% The term ‘concurrency’ refers to threads, not necessarily to parallel
% execution of those threads. (The latter is also possible if you are using
% one of the .par grades or the Java or C# backends.)
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module thread.
:- interface.

:- import_module io.
:- import_module maybe.

:- include_module barrier.
:- include_module channel.
:- include_module closeable_channel.
:- include_module future.
:- include_module mvar.
:- include_module semaphore.

%-----------------------------------------------------%

    % Abstract type representing a thread.
    %
:- type thread.

    % can_spawn succeeds if spawn/4 is supported in the current grade.
    %
:- pred can_spawn is semidet.

    % can_spawn_native succeeds if spawn_native/4 is supported in the current
    % grade.
    %
```

```
:- pred can_spawn_native is semidet.

    % spawn(Closure, IO0, IO) is true iff 'IO0' denotes a list of I/O
    % transactions that is an interleaving of those performed by 'Closure'
    % and those contained in 'IO' - the list of transactions performed by
    % the continuation of spawn/3.
    %
    % Operationally, spawn/3 is like spawn/4 except that Closure does not
    % accept a thread handle argument, and an exception is thrown if the
    % thread cannot be created.
    %
:- pred spawn(pred(io, io), io, io).
:- mode spawn(pred(di, uo) is cc_multi, di, uo) is cc_multi.

    % spawn(Closure, Res, IO0, IO) creates a new thread and performs Clo-
sure in
    % that thread. On success it returns ok(Thread) where Thread is a han-
dle to
    % the new thread. Otherwise it returns an error.
    %
:- pred spawn(pred(thread, io, io), maybe_error(thread), io, io).
:- mode spawn(pred(in, di, uo) is cc_multi, out, di, uo) is cc_multi.

    % spawn_native(Closure, Res, IO0, IO):
    % Like spawn/4, but Closure will be performed in a separate "native thread"
    % of the environment the program is running in (POSIX thread, Windows
    % thread, Java thread, etc.).
    %
    % spawn_native exposes a low-level implementation detail, so it is more
    % likely to change with the implementation.
    %
    % Rationale: on the low-level C backend Mercury threads are multiplexed
    % onto a limited number of OS threads. A call to a blocking procedure
    % prevents that OS thread from making progress on another Mercury thread.
    % Also, some foreign code depends on OS thread-local state so needs to be
    % consistently executed on a dedicated OS thread to be usable.
    %
:- pred spawn_native(pred(thread, io, io), maybe_error(thread), io, io).
:- mode spawn_native(pred(in, di, uo) is cc_multi, out, di, uo) is cc_multi.

    % yield(IO0, IO) is logically equivalent to (IO = IO0) but
    % operationally, yields the Mercury engine to some other thread
    % if one exists.
    %
    % NOTE: this is not yet implemented in the hl*.par.gc grades; currently
    % it is a no-op in those grades.
    %
```

```
:- pred yield(io::di, io::uo) is det.

    % num_processors(Num, !IO)
    %
    % Retrieve the number of processors available to this process for
    % parallel execution, if known.
    %
    % Note that the number of available processors can be different from the
    % actual number of processors/cores:
    %
    %  + It includes hardware threads.
    %  + The Mercury grade may restrict the process to one processor.
    %  + The OS may be configured to restrict the number of processors
    %    available (e.g. cpuset(7) on Linux).
    %
:- pred num_processors(maybe(int)::out, io::di, io::uo) is det.

%---------------------------------------------------%
%---------------------------------------------------%
```

96 thread.mvar

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 2000-2003, 2006-2007, 2011 The University of Melbourne.
% Copyright (C) 2014, 2016, 2018 The Mercury Team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: thread.mvar.m.
% Main author: petdr, fjh.
% Stability: low.
%
% This module provides a Mercury version of Haskell mutable variables.  A
% mutable variable (mvar) is a reference to a mutable location which can
% either contain a value of type T or be empty.
%
% Access to a mvar is thread-safe and can be used to synchronize between
% different threads.
%
%---------------------------------------------------%
%---------------------------------------------------%

:- module thread.mvar.
```

```
:- interface.

:- import_module bool.
:- import_module io.
:- import_module maybe.

%-----------------------------------------------------%

:- type mvar(T).

    % Create an empty mvar.
    %
:- pred init(mvar(T)::out, io::di, io::uo) is det.

    % Create an mvar with the given initial value.
    %
:- pred init(T::in, mvar(T)::out, io::di, io::uo) is det.

    % Create an empty mvar.
    %
:- impure func impure_init = (mvar(T)::uo) is det.

    % Create an mvar with the given initial value.
    %
:- impure func impure_init(T) = mvar(T).

    % Create an empty mvar.
    %
    % This has been renamed to impure_init.
    %
:- impure func init = (mvar(T)::uo) is det.
:- pragma obsolete(init/0).

    % Take the contents of the mvar out, leaving the mvar empty.
    % If the mvar is empty, block until some thread fills the mvar.
    %
:- pred take(mvar(T)::in, T::out, io::di, io::uo) is det.

    % Take the contents of the mvar out, leaving the mvar empty.
    % Returns `yes(X)' if the mvar contained X, or `no' if the operation
    % would block.
    %
    % WARNING: a return value of `no' does not necessarily mean the mvar
    % is or was empty. For example, another thread attempting to read or take
    % an item out of the mvar may also cause `try_take' to return `no'.
    %
:- pred try_take(mvar(T)::in, maybe(T)::out, io::di, io::uo) is det.
```

```
    % Place the value of type T into an empty mvar.
    % If the mvar is full then block until it becomes empty.
    %
:- pred put(mvar(T)::in, T::in, io::di, io::uo) is det.

    % Place the value of type T into an empty mvar, returning yes on success.
    % If the mvar is full then return no immediately without blocking.
    %
:- pred try_put(mvar(T)::in, T::in, bool::out, io::di, io::uo) is det.

    % Read the contents of mvar without taking it out.
    % If the mvar is empty then block until it is full.
    % This is similar to mvar.take followed by mvar.put, but another value
    % cannot be placed into the mvar between the two operations.
    %
:- pred read(mvar(T)::in, T::out, io::di, io::uo) is det.

    % Try to read the contents of mvar without taking it out.
    % Returns 'yes(X)' if the mvar contained X, or 'no' if the operation
    % would block.
    %
    % WARNING: a return value of 'no' does not necessarily mean the mvar
    % is or was empty. For example, another thread attempting to read or take
    % an item out of the mvar may also cause 'try_read' to return 'no'.
    %
:- pred try_read(mvar(T)::in, maybe(T)::out, io::di, io::uo) is det.

%---------------------------------------------------%
%---------------------------------------------------%
```

97 thread.semaphore

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 2000-2001,2003-2004, 2006-2007, 2009-2011 The University of Melbourne.
% Copyright (C) 2014-2018 The Mercury Team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: thread.semaphore.m.
% Main author: conway.
% Stability: medium.
%
```

```
% This module implements a simple semaphore data type for allowing
% threads to synchronise with one another.
%
% The operations in this module are no-ops in the hlc grades that do not
% contain a .par component.
%
%-----------------------------------------------------%

:- module thread.semaphore.
:- interface.

:- import_module bool.
:- import_module io.

%-----------------------------------------------------%

:- type semaphore.

    % init(Count, Sem, !IO) creates a new semaphore ‘Sem’ with its counter
    % initialized to ‘Count’.
    %
:- pred init(int::in, semaphore::uo, io::di, io::uo) is det.

    % init(Sem, !IO) creates a new semaphore ‘Sem’ with its counter
    % initialized to 0.
    %
:- pred init(semaphore::uo, io::di, io::uo) is det.

    % Sem = init(Count) returns a new semaphore ‘Sem’ with its counter
    % initialized to ‘Count’.
    %
:- impure func impure_init(int::in) = (semaphore::uo) is det.

    % Sem = init(Count) returns a new semaphore ‘Sem’ with its counter
    % initialized to ‘Count’.
    %
    % This has been renamed to impure_init.
    %
:- impure func init(int::in) = (semaphore::uo) is det.
:- pragma obsolete(init/1).

    % signal(Sem, !IO) increments the counter associated with ‘Sem’
    % and if the resulting counter has a value greater than 0, it wakes
    % one or more threads that are waiting on this semaphore (if
    % any).
    %
:- pred signal(semaphore::in, io::di, io::uo) is det.
```

```
    % wait(Sem, !IO) blocks until the counter associated with ‘Sem’
    % becomes greater than 0, whereupon it wakes, decrements the
    % counter and returns.
    %
:- pred wait(semaphore::in, io::di, io::uo) is det.

    % try_wait(Sem, Succ, !IO) is the same as wait/3, except that
    % instead of blocking, it binds ‘Succ’ to a boolean indicating
    % whether the call succeeded in obtaining the semaphore or not.
    %
:- pred try_wait(semaphore::in, bool::out, io::di, io::uo) is det.

%---------------------------------------------------%
%---------------------------------------------------%
```

98 time

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Originally written in 1999 by Tomas By <T.By@dcs.shef.ac.uk>
% "Feel free to use this code or parts of it any way you want."
%
% Some portions are Copyright (C) 1999-2007,2009-2012 The University of Melbourne.
% Copyright (C) 2014-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: time.m.
% Main authors: Tomas By <T.By@dcs.shef.ac.uk>, fjh.
% Stability: medium.
%
% Time functions.
%
%---------------------------------------------------%
%---------------------------------------------------%

:- module time.
:- interface.

:- import_module io.
:- import_module maybe.

%---------------------------------------------------%
```

```
    % The 'clock_t' type represents times measured in clock ticks.
    % NOTE: the unit used for a value of this type depends on whether it was
    % returned by 'clock' or 'times'. See the comments on these
    % predicates below.
    %
:- type clock_t == int.

    % The 'tms' type holds information about the amount of processor
    % time that a process and its child processes have consumed.
    %
:- type tms
    --->    tms(
                clock_t,    % tms_utime: user time
                clock_t,    % tms_stime: system time
                clock_t,    % tms_cutime: user time of children
                clock_t     % tms_cstime: system time of children
            ).

    % The 'time_t' type is an abstract type that represents
    % calendar times.
    %
:- type time_t.

    % The 'tm' type is a concrete type that represents calendar
    % times, broken down into their constituent components.
    % Comparison (via compare/3) of 'tm' values whose 'tm_dst'
    % components are identical is equivalent to comparison of
    % the times those 'tm' values represent.
    %
:- type tm
    --->    tm(
                tm_year :: int,         % Year (number since 1900)
                tm_mon  :: int,         % Month (number since January, 0-
11)
                tm_mday :: int,         % MonthDay (1-31)
                tm_hour :: int,         % Hours (after midnight, 0-23)
                tm_min  :: int,         % Minutes (0-59)
                tm_sec  :: int,         % Seconds (0-61)
                                        % (60 and 61 are for leap seconds)
                tm_yday :: int,         % YearDay (number since Jan 1st, 0-
365)
                tm_wday :: int,         % WeekDay (number since Sunday, 0-
6)
                tm_dst  :: maybe(dst)   % IsDST (is DST in effect?)
            ).
```

```
:- type dst
    --->    standard_time   % no, DST is not in effect
    ;       daylight_time.  % yes, DST is in effect

    % Some of the procedures in this module throw this type
    % as an exception if they can't obtain a result.
    %
:- type time_error
    --->    time_error(string). % Error message

%-----------------------------------------------%

    % clock(Result, !IO):
    %
    % Returns the elapsed processor time (number of clock ticks). The base time
    % is arbitrary but doesn't change within a single process. If the time
    % cannot be obtained, this procedure will throw a time_error exception.
    % To obtain a time in seconds, divide Result by `clocks_per_sec'.
    %
    % On Java the elapsed time for the calling thread is returned.
    %
:- pred clock(clock_t::out, io::di, io::uo) is det.

    % clocks_per_sec:
    %
    % Returns the number of "clocks" per second as defined by CLOCKS_PER_SEC.
    % A `clock_t' value returned by `clock' can be divided by this value
    % to obtain a time in seconds. Note that the value of this function does
    % not necessarily reflect the actual clock precision; it just indi-
cates the
    % scaling factor for the results of `clock'.
    %
:- func clocks_per_sec = int.

%-----------------------------------------------%

    % time(Result, !IO):
    %
    % Returns the current (simple) calendar time. If the time cannot be
    % obtained, this procedure will throw a time_error exception.
    %
:- pred time(time_t::out, io::di, io::uo) is det.

%-----------------------------------------------%

    % times(ProcessorTime, ElapsedRealTime, !IO):
    %
```

```
    % (POSIX)
    %
    % Returns the processor time information in the ‘tms’ value, and the
    % elapsed real time relative to an arbitrary base in the ‘clock_t’ value.
    % To obtain a time in seconds, divide the result by ‘clk_tck’.
    % If the time cannot be obtained, this procedure will throw a time_error
    % exception.
    %
    % On non-POSIX systems that do not support this functionality,
    % this procedure may simply always throw an exception.
    %
    % On Java the times for the calling thread are returned.
    % On Win32 and Java the child part of 'tms' is always zero.
    %
:- pred times(tms::out, clock_t::out, io::di, io::uo) is det.

    % clk_tck:
    %
    % Returns the number of "clock ticks" per second as defined by
    % sysconf(_SC_CLK_TCK). A ‘clock_t’ value returned by ‘times’
    % can be divided by this value to obtain a time in seconds.
    %
    % On non-POSIX systems that do not support this functionality,
    % this procedure may simply always throw an exception.
    %
:- func clk_tck = int.

%-----------------------------------------------------%

    % difftime(Time1, Time0) = Diff:
    %
    % Computes the number of seconds elapsed between ‘Time1’ and ‘Time0’.
    %
:- func difftime(time_t, time_t) = float.

    % localtime(Time, TM, !IO):
    %
    % Converts the (simple) calendar time ‘Time’ to a broken-down
    % representation ‘TM’, expressed relative to the current time zone.
    %
:- pred localtime(time_t::in, tm::out, io::di, io::uo) is det.

    % This function is deprecated because the current time zone is not
    % reflected in its arguments.
    %
:- pragma obsolete(localtime/1).
:- func localtime(time_t) = tm.
```

```
    % gmtime(Time) = TM:
    %
    % Converts the (simple) calendar time 'Time' to a broken-down
    % representation 'TM', expressed as UTC (Universal Coordinated Time).
    %
:- func gmtime(time_t) = tm.

    % mktime(TM) = Time:
    %
    % Converts the broken-down time value 'TM' to a (simple) calendar time
    % 'Time'. That is, 'TM' is relative to the current time zone.
    % The 'tm_wday' and 'tm_yday' fields of 'TM' are ignored.
    %
:- pred mktime(tm::in, time_t::out, io::di, io::uo) is det.

    % This function is deprecated because the current time zone is not
    % reflected in its arguments.
    %
:- pragma obsolete(mktime/1).
:- func mktime(tm) = time_t.

%-----------------------------------------------%

    % asctime(TM) = String:
    %
    % Converts the broken-down time value 'TM' to a string in a standard
    % format.
    %
:- func asctime(tm) = string.

    % ctime(Time) = String:
    %
    % Converts the calendar time value 'Time' to a string in a standard format
    % (i.e. same as "asctime (localtime (<time>))").
    %
    % This function is deprecated because the current time zone is not
    % reflected in its arguments. New code should write:
    %
    %   localtime(Time, TM, !IO),
    %   String = asctime(TM)
    %
:- pragma obsolete(ctime/1).
:- func ctime(time_t) = string.

%-----------------------------------------------%
%-----------------------------------------------%
```

99 tree234

```
%----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%----------------------------------------------------%
% Copyright (C) 1994-1997,1999-2000,2002-2012 The University of Melbourne.
% Copyright (C) 2013-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%----------------------------------------------------%
%
% File: tree234.m.
% Main author: conway.
% Stability: medium.
%
% This module implements a map (dictionary) using 2-3-4 trees - see
% map.m for further documentation.
%
%----------------------------------------------------%
%----------------------------------------------------%

:- module tree234.
:- interface.

:- import_module assoc_list.
:- import_module list.
:- import_module pretty_printer.
:- import_module term.

%----------------------------------------------------%

:- type tree234(K, V).

%----------------------------------------------------%

:- func init = tree234(K, V).
:- pred init(tree234(K, V)::uo) is det.

:- func singleton(K, V) = tree234(K, V).

:- pred is_empty(tree234(K, V)::in) is semidet.

    % True if both trees have the same set of key-value pairs, regardless of
    % how the trees were constructed.
    %
```

```
    % Unifying trees does not work as one might expect because the internal
    % structures of two trees that contain the same set of key-value pairs
    % may be different.
    %
:- pred equal(tree234(K, V)::in, tree234(K, V)::in) is semidet.

%---------------------------------------------------%

:- pred member(tree234(K, V)::in, K::out, V::out) is nondet.

:- pred search(tree234(K, V)::in, K::in, V::out) is semidet.

:- func lookup(tree234(K, V), K) = V.
:- pred lookup(tree234(K, V)::in, K::in, V::out) is det.

    % Search for a key-value pair using the key. If there is no entry
    % for the given key, returns the pair for the next lower key instead.
    % Fails if there is no key with the given or lower value.
    %
:- pred lower_bound_search(tree234(K, V)::in, K::in, K::out, V::out)
    is semidet.

    % Search for a key-value pair using the key. If there is no entry
    % for the given key, returns the pair for the next lower key instead.
    % Throws an exception if there is no key with the given or lower value.
    %
:- pred lower_bound_lookup(tree234(K, V)::in, K::in, K::out, V::out)
    is det.

    % Search for a key-value pair using the key. If there is no entry
    % for the given key, returns the pair for the next higher key instead.
    % Fails if there is no key with the given or higher value.
    %
:- pred upper_bound_search(tree234(K, V)::in, K::in, K::out, V::out)
    is semidet.

    % Search for a key-value pair using the key. If there is no entry
    % for the given key, returns the pair for the next higher key instead.
    % Throws an exception if there is no key with the given or higher value.
    %
:- pred upper_bound_lookup(tree234(K, V)::in, K::in, K::out, V::out)
    is det.

:- func max_key(tree234(K, V)) = K is semidet.

:- func min_key(tree234(K, V)) = K is semidet.
```

```
%---------------------------------------------------%

    % Insert the given key/value pair into the tree. If the key is already
    % in the tree, fail.
    %
:- pred insert(K::in, V::in, tree234(K, V)::in, tree234(K, V)::out)
    is semidet.

    % search_insert(K, V, MaybeOldV, !Tree):
    %
    % Search for the key K in the tree. If the key is already in the tree,
    % with corresponding value OldV, set MaybeOldV to yes(OldV). If it is
    % not in the tree, then insert it into the tree with value V, and set
    % MaybeOldV to no.
    %
:- pred search_insert(K::in, V::in, maybe(V)::out,
    tree234(K, V)::in, tree234(K, V)::out) is det.

    % Update the value corresponding to the given key in the tree.
    % If the key is not already in the tree, fail.
    %
:- pred update(K::in, V::in, tree234(K, V)::in, tree234(K, V)::out)
    is semidet.

    % set(K, V, !Tree):
    %
    % Set the value corresponding to K to V, regardless of whether K is
    % already in the tree or not.
    %
:- func set(tree234(K, V), K, V) = tree234(K, V).
:- pred set(K::in, V::in, tree234(K, V)::in, tree234(K, V)::out) is det.

%---------------------------------------------------%

    % Update the value at the given key by applying the supplied
    % transformation to it. This is faster than first searching for
    % the value and then updating it.
    %
:- pred transform_value(pred(V, V)::in(pred(in, out) is det), K::in,
    tree234(K, V)::in, tree234(K, V)::out) is semidet.

%---------------------------------------------------%

    % Delete the given key from the tree if it is there.
    %
:- func delete(tree234(K, V), K) = tree234(K, V).
:- pred delete(K::in, tree234(K, V)::in, tree234(K, V)::out) is det.
```

```
    % If the given key exists in the tree, return it and then delete the pair.
    % Otherwise, fail.
    %
:- pred remove(K, V, tree234(K, V), tree234(K, V)).
:- mode remove(in, out, in, out) is semidet.

    % Remove the smallest key from the tree, and return both it and the value
    % corresponding to it. If the tree is empty, fail.
    %
:- pred remove_smallest(K, V, tree234(K, V), tree234(K, V)).
:- mode remove_smallest(out, out, in, out) is semidet.

%-----------------------------------------------%

    % Given a tree234, return a list of all the keys in the tree.
    % The list that is returned is in sorted order (ascending on keys).
    %
:- func keys(tree234(K, V)) = list(K).
:- pred keys(tree234(K, V)::in, list(K)::out) is det.

    % Given a tree234, return a list of all the values in the tree.
    % The list that is returned is in sorted order (ascending on the original
    % keys, but not sorted on the values).
    %
:- func values(tree234(K, V)) = list(V).
:- pred values(tree234(K, V)::in, list(V)::out) is det.

    % Given a tree234, return lists of all the keys and values in the tree.
    % The key list is in sorted order (ascending on keys).
    % The values list is in sorted order (ascending on their keys,
    % but not on the values themselves).
    %
:- pred keys_and_values(tree234(K, V)::in, list(K)::out, list(V)::out) is det.

%-----------------------------------------------%

    % Count the number of elements in a tree.
    %
:- func count(tree234(K, V)) = int.
:- pred count(tree234(K, V)::in, int::out) is det.

    % Given a tree234, return an association list of all the keys and values
    % in the tree. The association list that is returned is sorted on the keys.
    %
:- func tree234_to_assoc_list(tree234(K, V)) = assoc_list(K, V).
:- pred tree234_to_assoc_list(tree234(K, V)::in,
```

```
    assoc_list(K, V)::out) is det.

:- func assoc_list_to_tree234(assoc_list(K, V)) = tree234(K, V).
:- pred assoc_list_to_tree234(assoc_list(K, V)::in,
    tree234(K, V)::out) is det.

    % Given an assoc list of keys and values that are sorted on the keys
    % in ascending order (with no duplicate keys), convert it directly
    % to a tree.
    %
:- pred from_sorted_assoc_list(assoc_list(K, V)::in,
    tree234(K, V)::out) is det.

    % Given an assoc list of keys and values that are sorted on the keys
    % in descending order (with no duplicate keys), convert it directly
    % to a tree.
    %
:- pred from_rev_sorted_assoc_list(assoc_list(K, V)::in,
    tree234(K, V)::out) is det.

%-----------------------------------------------%

:- func foldl(func(K, V, A) = A, tree234(K, V), A) = A.

:- pred foldl(pred(K, V, A, A), tree234(K, V), A, A).
:- mode foldl(pred(in, in, in, out) is det, in, in, out) is det.
:- mode foldl(pred(in, in, mdi, muo) is det, in, mdi, muo) is det.
:- mode foldl(pred(in, in, di, uo) is det, in, di, uo) is det.
:- mode foldl(pred(in, in, in, out) is semidet, in, in, out) is semidet.
:- mode foldl(pred(in, in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode foldl(pred(in, in, di, uo) is semidet, in, di, uo) is semidet.
:- mode foldl(pred(in, in, in, out) is cc_multi, in, in, out) is cc_multi.
:- mode foldl(pred(in, in, di, uo) is cc_multi, in, di, uo) is cc_multi.
:- mode foldl(pred(in, in, mdi, muo) is cc_multi, in, mdi, muo) is cc_multi.

:- pred foldl2(pred(K, V, A, A, B, B), tree234(K, V), A, A, B, B).
:- mode foldl2(pred(in, in, in, out, in, out) is det,
    in, in, out, in, out) is det.
:- mode foldl2(pred(in, in, in, out, mdi, muo) is det,
    in, in, out, mdi, muo) is det.
:- mode foldl2(pred(in, in, in, out, di, uo) is det,
    in, in, out, di, uo) is det.
:- mode foldl2(pred(in, in, di, uo, di, uo) is det,
    in, di, uo, di, uo) is det.
:- mode foldl2(pred(in, in, in, out, in, out) is semidet,
    in, in, out, in, out) is semidet.
:- mode foldl2(pred(in, in, in, out, mdi, muo) is semidet,
```

```
    in, in, out, mdi, muo) is semidet.
:- mode foldl2(pred(in, in, in, out, di, uo) is semidet,
    in, in, out, di, uo) is semidet.
:- mode foldl2(pred(in, in, in, out, in, out) is cc_multi,
    in, in, out, in, out) is cc_multi.
:- mode foldl2(pred(in, in, in, out, mdi, muo) is cc_multi,
    in, in, out, mdi, muo) is cc_multi.
:- mode foldl2(pred(in, in, in, out, di, uo) is cc_multi,
    in, in, out, di, uo) is cc_multi.
:- mode foldl2(pred(in, in, di, uo, di, uo) is cc_multi,
    in, di, uo, di, uo) is cc_multi.

:- pred foldl3(pred(K, V, A, A, B, B, C, C), tree234(K, V),
    A, A, B, B, C, C).
:- mode foldl3(pred(in, in, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out) is det.
:- mode foldl3(pred(in, in, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, mdi, muo) is det.
:- mode foldl3(pred(in, in, in, out, in, out, di, uo) is det,
    in, in, out, in, out, di, uo) is det.
:- mode foldl3(pred(in, in, in, out, di, uo, di, uo) is det,
    in, in, out, di, uo, di, uo) is det.
:- mode foldl3(pred(in, in, di, uo, di, uo, di, uo) is det,
    in, di, uo, di, uo, di, uo) is det.
:- mode foldl3(pred(in, in, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out) is semidet.
:- mode foldl3(pred(in, in, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, mdi, muo) is semidet.
:- mode foldl3(pred(in, in, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, di, uo) is semidet.

:- pred foldl4(pred(K, V, A, A, B, B, C, C, D, D), tree234(K, V),
    A, A, B, B, C, C, D, D).
:- mode foldl4(pred(in, in, in, out, in, out, in, out, in, out)
    is det,
    in, in, out, in, out, in, out, in, out) is det.
:- mode foldl4(pred(in, in, in, out, in, out, in, out, mdi, muo)
    is det,
    in, in, out, in, out, in, out, mdi, muo) is det.
:- mode foldl4(pred(in, in, in, out, in, out, in, out, di, uo) is det,
    in, in, out, in, out, in, out, di, uo) is det.
:- mode foldl4(pred(in, in, in, out, in, out, di, uo, di, uo) is det,
    in, in, out, in, out, di, uo, di, uo) is det.
:- mode foldl4(pred(in, in, in, out, di, uo, di, uo, di, uo) is det,
    in, in, out, di, uo, di, uo, di, uo) is det.
:- mode foldl4(pred(in, in, di, uo, di, uo, di, uo, di, uo) is det,
    in, di, uo, di, uo, di, uo, di, uo) is det.
```

```
:- mode foldl4(pred(in, in, in, out, in, out, in, out, in, out)
    is semidet,
    in, in, out, in, out, in, out, in, out) is semidet.
:- mode foldl4(pred(in, in, in, out, in, out, in, out, mdi, muo)
    is semidet,
    in, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode foldl4(pred(in, in, in, out, in, out, in, out, di, uo)
    is semidet,
    in, in, out, in, out, in, out, di, uo) is semidet.

:- pred foldl5(pred(K, V, A, A, B, B, C, C, D, D, E, E), tree234(K, V),
    A, A, B, B, C, C, D, D, E, E).
:- mode foldl5(pred(in, in, in, out, in, out, in, out, in, out, in, out)
    is det,
    in, in, out, in, out, in, out, in, out, in, out) is det.
:- mode foldl5(pred(in, in, in, out, in, out, in, out, in, out, mdi, muo)
    is det,
    in, in, out, in, out, in, out, in, out, mdi, muo) is det.
:- mode foldl5(pred(in, in, in, out, in, out, in, out, in, out, di, uo)
    is det,
    in, in, out, in, out, in, out, in, out, di, uo) is det.
:- mode foldl5(pred(in, in, in, out, in, out, in, out, in, out, in, out)
    is semidet,
    in, in, out, in, out, in, out, in, out, in, out) is semidet.
:- mode foldl5(pred(in, in, in, out, in, out, in, out, in, out, mdi, muo)
    is semidet,
    in, in, out, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode foldl5(pred(in, in, in, out, in, out, in, out, in, out, di, uo)
    is semidet,
    in, in, out, in, out, in, out, in, out, di, uo) is semidet.

:- pred foldl_values(pred(V, A, A), tree234(K, V), A, A).
:- mode foldl_values(pred(in, in, out) is det, in, in, out) is det.
:- mode foldl_values(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode foldl_values(pred(in, di, uo) is det, in, di, uo) is det.
:- mode foldl_values(pred(in, in, out) is semidet, in, in, out)
    is semidet.
:- mode foldl_values(pred(in, mdi, muo) is semidet, in, mdi, muo)
    is semidet.
:- mode foldl_values(pred(in, di, uo) is semidet, in, di, uo)
    is semidet.
:- mode foldl_values(pred(in, in, out) is cc_multi, in, in, out)
    is cc_multi.
:- mode foldl_values(pred(in, di, uo) is cc_multi, in, di, uo)
    is cc_multi.
:- mode foldl_values(pred(in, mdi, muo) is cc_multi, in, mdi, muo)
    is cc_multi.
```

```
:- pred foldl2_values(pred(V, A, A, B, B), tree234(K, V), A, A, B, B).
:- mode foldl2_values(pred(in, in, out, in, out) is det, in,
    in, out, in, out) is det.
:- mode foldl2_values(pred(in, in, out, mdi, muo) is det, in,
    in, out, mdi, muo) is det.
:- mode foldl2_values(pred(in, in, out, di, uo) is det, in,
    in, out, di, uo) is det.
:- mode foldl2_values(pred(in, in, out, in, out) is semidet, in,
    in, out, in, out) is semidet.
:- mode foldl2_values(pred(in, in, out, mdi, muo) is semidet, in,
    in, out, mdi, muo) is semidet.
:- mode foldl2_values(pred(in, in, out, di, uo) is semidet, in,
    in, out, di, uo) is semidet.
:- mode foldl2_values(pred(in, in, out, in, out) is cc_multi, in,
    in, out, in, out) is cc_multi.
:- mode foldl2_values(pred(in, in, out, mdi, muo) is cc_multi, in,
    in, out, mdi, muo) is cc_multi.
:- mode foldl2_values(pred(in, in, out, di, uo) is cc_multi, in,
    in, out, di, uo) is cc_multi.

:- pred foldl3_values(pred(V, A, A, B, B, C, C), tree234(K, V),
    A, A, B, B, C, C).
:- mode foldl3_values(pred(in, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out) is det.
:- mode foldl3_values(pred(in, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, mdi, muo) is det.
:- mode foldl3_values(pred(in, in, out, in, out, di, uo) is det,
    in, in, out, in, out, di, uo) is det.
:- mode foldl3_values(pred(in, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out) is semidet.
:- mode foldl3_values(pred(in, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, mdi, muo) is semidet.
:- mode foldl3_values(pred(in, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, di, uo) is semidet.
:- mode foldl3_values(pred(in, in, out, in, out, in, out) is cc_multi,
    in, in, out, in, out, in, out) is cc_multi.
:- mode foldl3_values(pred(in, in, out, in, out, mdi, muo) is cc_multi,
    in, in, out, in, out, mdi, muo) is cc_multi.
:- mode foldl3_values(pred(in, in, out, in, out, di, uo) is cc_multi,
    in, in, out, in, out, di, uo) is cc_multi.

:- pred foldl4_values(pred(V, A, A, B, B, C, C, D, D), tree234(K, V),
    A, A, B, B, C, C, D, D).
:- mode foldl4_values(pred(in, in, out, in, out, in, out, in, out)
    is det,
    in, in, out, in, out, in, out, in, out) is det.
```

```
:- mode foldl4_values(pred(in, in, out, in, out, in, out, mdi, muo)
    is det,
    in, in, out, in, out, in, out, mdi, muo) is det.
:- mode foldl4_values(pred(in, in, out, in, out, in, out, di, uo) is det,
    in, in, out, in, out, in, out, di, uo) is det.
:- mode foldl4_values(pred(in, in, out, in, out, di, uo, di, uo) is det,
    in, in, out, in, out, di, uo, di, uo) is det.
:- mode foldl4_values(pred(in, in, out, di, uo, di, uo, di, uo) is det,
    in, in, out, di, uo, di, uo, di, uo) is det.
:- mode foldl4_values(pred(in, di, uo, di, uo, di, uo, di, uo) is det,
    in, di, uo, di, uo, di, uo, di, uo) is det.
:- mode foldl4_values(pred(in, in, out, in, out, in, out, in, out)
    is semidet,
    in, in, out, in, out, in, out, in, out) is semidet.
:- mode foldl4_values(pred(in, in, out, in, out, in, out, mdi, muo)
    is semidet,
    in, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode foldl4_values(pred(in, in, out, in, out, in, out, di, uo)
    is semidet,
    in, in, out, in, out, in, out, di, uo) is semidet.

:- pred foldl5_values(pred(V, A, A, B, B, C, C, D, D, E, E), tree234(K, V),
    A, A, B, B, C, C, D, D, E, E).
:- mode foldl5_values(pred(in, in, out, in, out, in, out, in, out, in, out)
    is det,
    in, in, out, in, out, in, out, in, out, in, out) is det.
:- mode foldl5_values(pred(in, in, out, in, out, in, out, in, out, mdi, muo)
    is det,
    in, in, out, in, out, in, out, in, out, mdi, muo) is det.
:- mode foldl5_values(pred(in, in, out, in, out, in, out, in, out, di, uo)
    is det,
    in, in, out, in, out, in, out, in, out, di, uo) is det.
:- mode foldl5_values(pred(in, in, out, in, out, in, out, in, out, in, out)
    is semidet,
    in, in, out, in, out, in, out, in, out, in, out) is semidet.
:- mode foldl5_values(pred(in, in, out, in, out, in, out, in, out, mdi, muo)
    is semidet,
    in, in, out, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode foldl5_values(pred(in, in, out, in, out, in, out, in, out, di, uo)
    is semidet,
    in, in, out, in, out, in, out, in, out, di, uo) is semidet.

:- func foldr(func(K, V, A) = A, tree234(K, V), A) = A.

:- pred foldr(pred(K, V, A, A), tree234(K, V), A, A).
:- mode foldr(pred(in, in, in, out) is det, in, in, out) is det.
:- mode foldr(pred(in, in, mdi, muo) is det, in, mdi, muo) is det.
```

```
:- mode foldr(pred(in, in, di, uo) is det, in, di, uo) is det.
:- mode foldr(pred(in, in, in, out) is semidet, in, in, out) is semidet.
:- mode foldr(pred(in, in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode foldr(pred(in, in, di, uo) is semidet, in, di, uo) is semidet.
:- mode foldr(pred(in, in, in, out) is cc_multi, in, in, out) is cc_multi.
:- mode foldr(pred(in, in, di, uo) is cc_multi, in, di, uo) is cc_multi.
:- mode foldr(pred(in, in, mdi, muo) is cc_multi, in, mdi, muo) is cc_multi.

:- pred foldr2(pred(K, V, A, A, B, B), tree234(K, V), A, A, B, B).
:- mode foldr2(pred(in, in, in, out, in, out) is det,
    in, in, out, in, out) is det.
:- mode foldr2(pred(in, in, in, out, mdi, muo) is det,
    in, in, out, mdi, muo) is det.
:- mode foldr2(pred(in, in, in, out, di, uo) is det,
    in, in, out, di, uo) is det.
:- mode foldr2(pred(in, in, di, uo, di, uo) is det,
    in, di, uo, di, uo) is det.
:- mode foldr2(pred(in, in, in, out, in, out) is semidet,
    in, in, out, in, out) is semidet.
:- mode foldr2(pred(in, in, in, out, mdi, muo) is semidet,
    in, in, out, mdi, muo) is semidet.
:- mode foldr2(pred(in, in, in, out, di, uo) is semidet,
    in, in, out, di, uo) is semidet.

:- pred foldr3(pred(K, V, A, A, B, B, C, C), tree234(K, V),
    A, A, B, B, C, C).
:- mode foldr3(pred(in, in, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out) is det.
:- mode foldr3(pred(in, in, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, mdi, muo) is det.
:- mode foldr3(pred(in, in, in, out, in, out, di, uo) is det,
    in, in, out, in, out, di, uo) is det.
:- mode foldr3(pred(in, in, in, out, di, uo, di, uo) is det,
    in, in, out, di, uo, di, uo) is det.
:- mode foldr3(pred(in, in, di, uo, di, uo, di, uo) is det,
    in, di, uo, di, uo, di, uo) is det.
:- mode foldr3(pred(in, in, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out) is semidet.
:- mode foldr3(pred(in, in, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, mdi, muo) is semidet.
:- mode foldr3(pred(in, in, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, di, uo) is semidet.

:- pred foldr4(pred(K, V, A, A, B, B, C, C, D, D), tree234(K, V),
    A, A, B, B, C, C, D, D).
:- mode foldr4(pred(in, in, in, out, in, out, in, out, in, out)
    is det,
```

```
    in, in, out, in, out, in, out, in, out) is det.
:- mode foldr4(pred(in, in, in, out, in, out, in, out, mdi, muo)
    is det,
    in, in, out, in, out, in, out, mdi, muo) is det.
:- mode foldr4(pred(in, in, in, out, in, out, in, out, di, uo) is det,
    in, in, out, in, out, in, out, di, uo) is det.
:- mode foldr4(pred(in, in, in, out, in, out, di, uo, di, uo) is det,
    in, in, out, in, out, di, uo, di, uo) is det.
:- mode foldr4(pred(in, in, in, out, di, uo, di, uo, di, uo) is det,
    in, in, out, di, uo, di, uo, di, uo) is det.
:- mode foldr4(pred(in, in, di, uo, di, uo, di, uo, di, uo) is det,
    in, di, uo, di, uo, di, uo, di, uo) is det.
:- mode foldr4(pred(in, in, in, out, in, out, in, out, in, out)
    is semidet,
    in, in, out, in, out, in, out, in, out) is semidet.
:- mode foldr4(pred(in, in, in, out, in, out, in, out, mdi, muo)
    is semidet,
    in, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode foldr4(pred(in, in, in, out, in, out, in, out, di, uo)
    is semidet,
    in, in, out, in, out, in, out, di, uo) is semidet.

:- pred foldr5(pred(K, V, A, A, B, B, C, C, D, D, E, E), tree234(K, V),
    A, A, B, B, C, C, D, D, E, E).
:- mode foldr5(pred(in, in, in, out, in, out, in, out, in, out, in, out)
    is det,
    in, in, out, in, out, in, out, in, out, in, out) is det.
:- mode foldr5(pred(in, in, in, out, in, out, in, out, in, out, mdi, muo)
    is det,
    in, in, out, in, out, in, out, in, out, mdi, muo) is det.
:- mode foldr5(pred(in, in, in, out, in, out, in, out, in, out, di, uo)
    is det,
    in, in, out, in, out, in, out, in, out, di, uo) is det.
:- mode foldr5(pred(in, in, in, out, in, out, in, out, in, out, in, out)
    is semidet,
    in, in, out, in, out, in, out, in, out, in, out) is semidet.
:- mode foldr5(pred(in, in, in, out, in, out, in, out, in, out, mdi, muo)
    is semidet,
    in, in, out, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode foldr5(pred(in, in, in, out, in, out, in, out, in, out, di, uo)
    is semidet,
    in, in, out, in, out, in, out, in, out, di, uo) is semidet.

%---------------------------------------------------%

:- func map_values(func(K, V) = W, tree234(K, V)) = tree234(K, W).
```

```
:- pred map_values(pred(K, V, W), tree234(K, V), tree234(K, W)).
:- mode map_values(pred(in, in, out) is det, in, out) is det.
:- mode map_values(pred(in, in, out) is semidet, in, out) is semidet.

:- func map_values_only(func(V) = W, tree234(K, V)) = tree234(K, W).

:- pred map_values_only(pred(V, W), tree234(K, V), tree234(K, W)).
:- mode map_values_only(pred(in, out) is det, in, out) is det.
:- mode map_values_only(pred(in, out) is semidet, in, out) is semidet.

:- pred filter_map_values(pred(K, V, W)::in(pred(in, in, out) is semidet),
    tree234(K, V)::in, tree234(K, W)::out) is det.

:- pred filter_map_values_only(pred(V, W)::in(pred(in, out) is semidet),
    tree234(K, V)::in, tree234(K, W)::out) is det.

%-----------------------------------------------%

:- pred map_foldl(pred(K, V, W, A, A),
    tree234(K, V), tree234(K, W), A, A).
:- mode map_foldl(pred(in, in, out, in, out) is det,
    in, out, in, out) is det.
:- mode map_foldl(pred(in, in, out, mdi, muo) is det,
    in, out, mdi, muo) is det.
:- mode map_foldl(pred(in, in, out, di, uo) is det,
    in, out, di, uo) is det.
:- mode map_foldl(pred(in, in, out, in, out) is semidet,
    in, out, in, out) is semidet.
:- mode map_foldl(pred(in, in, out, mdi, muo) is semidet,
    in, out, mdi, muo) is semidet.
:- mode map_foldl(pred(in, in, out, di, uo) is semidet,
    in, out, di, uo) is semidet.

:- pred map_foldl2(pred(K, V, W, A, A, B, B),
    tree234(K, V), tree234(K, W), A, A, B, B).
:- mode map_foldl2(pred(in, in, out, in, out, in, out) is det,
    in, out, in, out, in, out) is det.
:- mode map_foldl2(pred(in, in, out, in, out, mdi, muo) is det,
    in, out, in, out, mdi, muo) is det.
:- mode map_foldl2(pred(in, in, out, di, uo, di, uo) is det,
    in, out, di, uo, di, uo) is det.
:- mode map_foldl2(pred(in, in, out, in, out, di, uo) is det,
    in, out, in, out, di, uo) is det.
:- mode map_foldl2(pred(in, in, out, in, out, in, out) is semidet,
    in, out, in, out, in, out) is semidet.
:- mode map_foldl2(pred(in, in, out, in, out, mdi, muo) is semidet,
    in, out, in, out, mdi, muo) is semidet.
```

```
:- mode map_foldl2(pred(in, in, out, in, out, di, uo) is semidet,
    in, out, in, out, di, uo) is semidet.

:- pred map_foldl3(pred(K, V, W, A, A, B, B, C, C),
    tree234(K, V), tree234(K, W), A, A, B, B, C, C).
:- mode map_foldl3(pred(in, in, out, in, out, in, out, in, out) is det,
    in, out, in, out, in, out, in, out) is det.
:- mode map_foldl3(pred(in, in, out, in, out, in, out, mdi, muo) is det,
    in, out, in, out, in, out, mdi, muo) is det.
:- mode map_foldl3(pred(in, in, out, di, uo, di, uo, di, uo) is det,
    in, out, di, uo, di, uo, di, uo) is det.
:- mode map_foldl3(pred(in, in, out, in, out, di, uo, di, uo) is det,
    in, out, in, out, di, uo, di, uo) is det.
:- mode map_foldl3(pred(in, in, out, in, out, in, out, di, uo) is det,
    in, out, in, out, in, out, di, uo) is det.
:- mode map_foldl3(pred(in, in, out, in, out, in, out, in, out)
    is semidet,
    in, out, in, out, in, out, in, out) is semidet.
:- mode map_foldl3(pred(in, in, out, in, out, in, out, mdi, muo)
    is semidet,
    in, out, in, out, in, out, mdi, muo) is semidet.
:- mode map_foldl3(pred(in, in, out, in, out, in, out, di, uo)
    is semidet,
    in, out, in, out, in, out, di, uo) is semidet.

:- pred map_foldl4(pred(K, V, W, A, A, B, B, C, C, D, D),
    tree234(K, V), tree234(K, W), A, A, B, B, C, C, D, D).
:- mode map_foldl4(pred(in, in, out, in, out, in, out, in, out, in, out)
    is det,
    in, out, in, out, in, out, in, out, in, out) is det.
:- mode map_foldl4(pred(in, in, out, in, out, in, out, in, out, mdi, muo)
    is det,
    in, out, in, out, in, out, in, out, mdi, muo) is det.
:- mode map_foldl4(pred(in, in, out, in, out, di, uo, di, uo, di, uo) is det,
    in, out, in, out, di, uo, di, uo, di, uo) is det.
:- mode map_foldl4(pred(in, in, out, in, out, in, out, di, uo, di, uo) is det,
    in, out, in, out, in, out, di, uo, di, uo) is det.
:- mode map_foldl4(pred(in, in, out, in, out, in, out, in, out, di, uo) is det,
    in, out, in, out, in, out, in, out, di, uo) is det.
:- mode map_foldl4(pred(in, in, out, in, out, in, out, in, out, in, out)
    is semidet,
    in, out, in, out, in, out, in, out, in, out) is semidet.
:- mode map_foldl4(pred(in, in, out, in, out, in, out, in, out, mdi, muo)
    is semidet,
    in, out, in, out, in, out, in, out, mdi, muo) is semidet.
:- mode map_foldl4(pred(in, in, out, in, out, in, out, in, out, di, uo)
    is semidet,
```

```
    in, out, in, out, in, out, in, out, di, uo) is semidet.

:- pred map_values_foldl(pred(V, W, A, A),
    tree234(K, V), tree234(K, W), A, A).
:- mode map_values_foldl(pred(in, out, di, uo) is det,
    in, out, di, uo) is det.
:- mode map_values_foldl(pred(in, out, in, out) is det,
    in, out, in, out) is det.
:- mode map_values_foldl(pred(in, out, in, out) is semidet,
    in, out, in, out) is semidet.

:- pred map_values_foldl2(pred(V, W, A, A, B, B),
    tree234(K, V), tree234(K, W), A, A, B, B).
:- mode map_values_foldl2(pred(in, out, di, uo, di, uo) is det,
    in, out, di, uo, di, uo) is det.
:- mode map_values_foldl2(pred(in, out, in, out, di, uo) is det,
    in, out, in, out, di, uo) is det.
:- mode map_values_foldl2(pred(in, out, in, out, in, out) is det,
    in, out, in, out, in, out) is det.
:- mode map_values_foldl2(pred(in, out, in, out, in, out) is semidet,
    in, out, in, out, in, out) is semidet.

:- pred map_values_foldl3(pred(V, W, A, A, B, B, C, C),
    tree234(K, V), tree234(K, W), A, A, B, B, C, C).
:- mode map_values_foldl3(
    pred(in, out, di, uo, di, uo, di, uo) is det,
    in, out, di, uo, di, uo, di, uo) is det.
:- mode map_values_foldl3(
    pred(in, out, in, out, di, uo, di, uo) is det,
    in, out, in, out, di, uo, di, uo) is det.
:- mode map_values_foldl3(
    pred(in, out, in, out, in, out, di, uo) is det,
    in, out, in, out, in, out, di, uo) is det.
:- mode map_values_foldl3(
    pred(in, out, in, out, in, out, in, out) is det,
    in, out, in, out, in, out, in, out) is det.
:- mode map_values_foldl3(
    pred(in, out, in, out, in, out, in, out) is semidet,
    in, out, in, out, in, out, in, out) is semidet.

%---------------------------------------------------%

    % Convert a tree234 into a pretty_printer.doc. A tree mapping
    % K1 to V1, K2 to V2, ... is formatted as
    % "map([K1 -> V1, K2 -> V2, ...])". The functor "map" is used
    % because tree234 values are almost exclusively maps.
    %
```

```
:- func tree234_to_doc(tree234(K, V)) = pretty_printer.doc.

%---------------------------------------------------%
%---------------------------------------------------%
```

100 tree_bitset

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 2006, 2009-2012 The University of Melbourne.
% Copyright (C) 2014-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: tree_bitset.m.
% Author: zs, based on sparse_bitset.m by stayl.
% Stability: medium.
%
% This module provides an ADT for storing sets of non-negative integers.
% If the integers stored are closely grouped, a tree_bitset is more compact
% than the representation provided by set.m, and the operations will be much
% faster. Compared to sparse_bitset.m, the operations provided by this module
% for contains, union, intersection and difference can be expected to have
% lower asymptotic complexity (often logarithmic in the number of elements in
% the sets, rather than linear). The price for this is a representation that
% requires more memory, higher constant factors, and an additional factor
% representing the tree in the complexity of the operations that construct
% tree_bitsets. However, since the depth of the tree has a small upper bound
% for all sets of a practical size, we will fold this into the "higher
% constant factors" in the descriptions of the complexity of the individual
% operations below.
%
% All this means that using a tree_bitset in preference to a sparse_bitset
% is likely to be a good idea only when the sizes of the sets to be manipulated
% are quite big, or when worst-case performance is important.
%
% For the time being, this module can only handle items that map to nonnegative
% integers. This may change once unsigned integer operations are available.
%
%---------------------------------------------------%

:- module tree_bitset.
:- interface.
```

```
:- import_module enum.
:- import_module list.
:- import_module term.

:- use_module set.

%-----------------------------------------------------%

:- type tree_bitset(T). % <= enum(T).

%-----------------------------------------------------%
%
% Initial creation of sets.
%

    % Return an empty set.
    %
:- func init = tree_bitset(T).

    % 'make_singleton_set(Elem)' returns a set containing just the single
    % element 'Elem'.
    %
:- func make_singleton_set(T) = tree_bitset(T) <= enum(T).

%-----------------------------------------------------%
%
% Emptiness and singleton-ness tests.
%

:- pred empty(tree_bitset(T)).
:- mode empty(in) is semidet.
:- mode empty(out) is det.
:- pragma obsolete(empty/1, [init/0, is_empty/1]).

:- pred is_empty(tree_bitset(T)::in) is semidet.

:- pred is_non_empty(tree_bitset(T)::in) is semidet.

    % Is the given set a singleton, and if yes, what is the element?
    %
:- pred is_singleton(tree_bitset(T)::in, T::out) is semidet <= enum(T).

%-----------------------------------------------------%
%
% Membership tests.
%
```

```
    % 'member(X, Set)' is true iff 'X' is a member of 'Set'.
    % Takes O(card(Set)) time for the semidet mode.
    %
:- pred member(T, tree_bitset(T)) <= enum(T).
:- mode member(in, in) is semidet.
:- mode member(out, in) is nondet.

    % 'contains(Set, X)' is true iff 'X' is a member of 'Set'.
    % Takes O(log(card(Set))) time.
    %
:- pred contains(tree_bitset(T)::in, T::in) is semidet <= enum(T).

%-----------------------------------------------%
%
% Insertions and deletions.
%

    % 'insert(Set, X)' returns the union of 'Set' and the set containing
    % only 'X'. Takes O(log(card(Set))) time and space.
    %
:- func insert(tree_bitset(T), T) = tree_bitset(T) <= enum(T).
:- pred insert(T::in, tree_bitset(T)::in, tree_bitset(T)::out)
    is det <= enum(T).

    % 'insert_new(X, Set0, Set)' returns the union of 'Set' and the set
    % containing only 'X' is 'Set0' does not contain 'X'; if it does, it fails.
    % Takes O(log(card(Set))) time and space.
    %
:- pred insert_new(T::in, tree_bitset(T)::in, tree_bitset(T)::out)
    is semidet <= enum(T).

    % 'insert_list(Set, X)' returns the union of 'Set' and the set containing
    % only the members of 'X'. Same as 'union(Set, list_to_set(X))', but may be
    % more efficient.
    %
:- func insert_list(tree_bitset(T), list(T)) = tree_bitset(T) <= enum(T).
:- pred insert_list(list(T)::in, tree_bitset(T)::in, tree_bitset(T)::out)
    is det <= enum(T).

%-----------------------------------------------%

    % 'delete(Set, X)' returns the difference of 'Set' and the set containing
    % only 'X'. Takes O(card(Set)) time and space.
    %
:- func delete(tree_bitset(T), T) = tree_bitset(T) <= enum(T).
:- pred delete(T::in, tree_bitset(T)::in, tree_bitset(T)::out)
    is det <= enum(T).
```

```
    % 'delete_list(Set, X)' returns the difference of 'Set' and the set
    % containing only the members of 'X'. Same as
    % 'difference(Set, list_to_set(X))', but may be more efficient.
    %
:- func delete_list(tree_bitset(T), list(T)) = tree_bitset(T) <= enum(T).
:- pred delete_list(list(T)::in, tree_bitset(T)::in, tree_bitset(T)::out)
    is det <= enum(T).

    % 'remove(X, Set0, Set)' returns in 'Set' the difference of 'Set0'
    % and the set containing only 'X', failing if 'Set0' does not contain 'X'.
    % Takes O(log(card(Set))) time and space.
    %
:- pred remove(T::in, tree_bitset(T)::in, tree_bitset(T)::out)
    is semidet <= enum(T).

    % 'remove_list(X, Set0, Set)' returns in 'Set' the difference of 'Set0'
    % and the set containing all the elements of 'X', failing if any element
    % of 'X' is not in 'Set0'. Same as 'subset(list_to_set(X), Set0),
    % difference(Set0, list_to_set(X), Set)', but may be more efficient.
    %
:- pred remove_list(list(T)::in, tree_bitset(T)::in, tree_bitset(T)::out)
    is semidet <= enum(T).

    % 'remove_leq(Set, X)' returns 'Set' with all elements less than or equal
    % to 'X' removed. In other words, it returns the set containing all the
    % elements of 'Set' which are greater than 'X'. Takes O(log(card(Set)))
    % time and space.
    %
:- func remove_leq(tree_bitset(T), T) = tree_bitset(T) <= enum(T).

    % 'remove_gt(Set, X)' returns 'Set' with all elements greater than 'X'
    % removed. In other words, it returns the set containing all the elements
    % of 'Set' which are less than or equal to 'X'. Takes O(log(card(Set)))
    % time and space.
    %
:- func remove_gt(tree_bitset(T), T) = tree_bitset(T) <= enum(T).

    % 'remove_least(Set0, X, Set)' is true iff 'X' is the least element in
    % 'Set0', and 'Set' is the set which contains all the elements of 'Set0'
    % except 'X'. Takes O(1) time and space.
    %
:- pred remove_least(T::out, tree_bitset(T)::in, tree_bitset(T)::out)
    is semidet <= enum(T).

%---------------------------------------------------%
%
```

```
% Comparisons between sets.
%

    % 'equal(SetA, SetB)' is true iff 'SetA' and 'SetB' contain the same
    % elements. Takes O(min(card(SetA), card(SetB))) time.
    %
:- pred equal(tree_bitset(T)::in, tree_bitset(T)::in) is semidet <= enum(T).

    % 'subset(Subset, Set)' is true iff 'Subset' is a subset of 'Set'.
    % Same as 'intersect(Set, Subset, Subset)', but may be more efficient.
    %
:- pred subset(tree_bitset(T)::in, tree_bitset(T)::in) is semidet.

    % 'superset(Superset, Set)' is true iff 'Superset' is a superset of 'Set'.
    % Same as 'intersect(Superset, Set, Set)', but may be more efficient.
    %
:- pred superset(tree_bitset(T)::in, tree_bitset(T)::in) is semidet.

%-----------------------------------------------%
%
% Operations on two or more sets.
%

    % 'union(SetA, SetB)' returns the union of 'SetA' and 'SetB'. The
    % efficiency of the union operation is not sensitive to the argument
    % ordering. Takes somewhere between O(log(card(SetA)) + log(card(SetB)))
    % and O(card(SetA) + card(SetB)) time and space.
    %
:- func union(tree_bitset(T), tree_bitset(T)) = tree_bitset(T).
:- pred union(tree_bitset(T)::in, tree_bitset(T)::in, tree_bitset(T)::out)
    is det.

    % 'union_list(Sets, Set)' returns the union of all the sets in Sets.
    %
:- func union_list(list(tree_bitset(T))) = tree_bitset(T).
:- pred union_list(list(tree_bitset(T))::in, tree_bitset(T)::out) is det.

    % 'intersect(SetA, SetB)' returns the intersection of 'SetA' and 'SetB'.
    % The efficiency of the intersection operation is not sensitive to the
    % argument ordering. Takes somewhere between
    % O(log(card(SetA)) + log(card(SetB))) and O(card(SetA) + card(SetB)) time,
    % and O(min(card(SetA)), card(SetB)) space.
    %
:- func intersect(tree_bitset(T), tree_bitset(T)) = tree_bitset(T).
:- pred intersect(tree_bitset(T)::in, tree_bitset(T)::in, tree_bitset(T)::out)
    is det.
```

```
    % ‘intersect_list(Sets, Set)’ returns the intersection of all the sets
    % in Sets.
    %
:- func intersect_list(list(tree_bitset(T))) = tree_bitset(T).
:- pred intersect_list(list(tree_bitset(T))::in, tree_bitset(T)::out) is det.

    % ‘difference(SetA, SetB)’ returns the set containing all the elements
    % of ‘SetA’ except those that occur in ‘SetB’. Takes somewhere between
    % O(log(card(SetA)) + log(card(SetB))) and O(card(SetA) + card(SetB)) time,
    % and O(card(SetA)) space.
    %
:- func difference(tree_bitset(T), tree_bitset(T)) = tree_bitset(T).
:- pred difference(tree_bitset(T)::in, tree_bitset(T)::in, tree_bitset(T)::out)
    is det.

%-----------------------------------------------------%
%
% Operations that divide a set into two parts.
%

    % divide(Pred, Set, InPart, OutPart):
    % InPart consists of those elements of Set for which Pred succeeds;
    % OutPart consists of those elements of Set for which Pred fails.
    %
:- pred divide(pred(T)::in(pred(in) is semidet), tree_bitset(T)::in,
    tree_bitset(T)::out, tree_bitset(T)::out) is det <= enum(T).

    % divide_by_set(DivideBySet, Set, InPart, OutPart):
    % InPart consists of those elements of Set which are also in DivideBySet;
    % OutPart consists of those elements of Set which are not in DivideBySet.
    %
:- pred divide_by_set(tree_bitset(T)::in, tree_bitset(T)::in,
    tree_bitset(T)::out, tree_bitset(T)::out) is det <= enum(T).

%-----------------------------------------------------%
%
% Converting lists to sets.
%

    % ‘list_to_set(List)’ returns a set containing only the members of ‘List’.
    % Takes O(length(List)) time and space.
    %
:- func list_to_set(list(T)) = tree_bitset(T) <= enum(T).
:- pred list_to_set(list(T)::in, tree_bitset(T)::out) is det <= enum(T).

    % ‘sorted_list_to_set(List)’ returns a set containing only the members
    % of ‘List’. ‘List’ must be sorted. Takes O(length(List)) time and space.
```

```
    %
:- func sorted_list_to_set(list(T)) = tree_bitset(T) <= enum(T).
:- pred sorted_list_to_set(list(T)::in, tree_bitset(T)::out) is det <= enum(T).

%-----------------------------------------------------%
%
% Converting sets to lists.
%

    % 'to_sorted_list(Set)' returns a list containing all the members of 'Set',
    % in sorted order. Takes O(card(Set)) time and space.
    %
:- func to_sorted_list(tree_bitset(T)) = list(T) <= enum(T).
:- pred to_sorted_list(tree_bitset(T)::in, list(T)::out) is det <= enum(T).

%-----------------------------------------------------%
%
% Converting between different kinds of sets.
%

    % 'from_set(Set)' returns a bitset containing only the members of 'Set'.
    % Takes O(card(Set)) time and space.
    %
:- func from_set(set.set(T)) = tree_bitset(T) <= enum(T).

    % 'to_sorted_list(Set)' returns a set.set containing all the members
    % of 'Set', in sorted order. Takes O(card(Set)) time and space.
    %
:- func to_set(tree_bitset(T)) = set.set(T) <= enum(T).

%-----------------------------------------------------%
%
% Counting.
%

    % 'count(Set)' returns the number of elements in 'Set'.
    % Takes O(card(Set)) time.
    %
:- func count(tree_bitset(T)) = int <= enum(T).

%-----------------------------------------------------%
%
% Standard higher order functions on collections.
%

    % all_true(Pred, Set) succeeds iff Pred(Element) succeeds
    % for all the elements of Set.
```

```
    %
:- pred all_true(pred(T)::in(pred(in) is semidet), tree_bitset(T)::in)
    is semidet <= enum(T).

    % 'filter(Pred, Set) = TrueSet' returns the elements of Set for which
    % Pred succeeds.
    %
:- func filter(pred(T), tree_bitset(T)) = tree_bitset(T) <= enum(T).
:- mode filter(pred(in) is semidet, in) = out is det.

    % 'filter(Pred, Set, TrueSet, FalseSet)' returns the elements of Set
    % for which Pred succeeds, and those for which it fails.
    %
:- pred filter(pred(T), tree_bitset(T), tree_bitset(T), tree_bitset(T))
    <= enum(T).
:- mode filter(pred(in) is semidet, in, out, out) is det.

    % 'foldl(Func, Set, Start)' calls Func with each element of 'Set'
    % (in sorted order) and an accumulator (with the initial value of 'Start'),
    % and returns the final value. Takes O(card(Set)) time.
    %
:- func foldl(func(T, U) = U, tree_bitset(T), U) = U <= enum(T).

:- pred foldl(pred(T, U, U), tree_bitset(T), U, U) <= enum(T).
:- mode foldl(pred(in, in, out) is det, in, in, out) is det.
:- mode foldl(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode foldl(pred(in, di, uo) is det, in, di, uo) is det.
:- mode foldl(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode foldl(pred(in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode foldl(pred(in, di, uo) is semidet, in, di, uo) is semidet.
:- mode foldl(pred(in, in, out) is nondet, in, in, out) is nondet.
:- mode foldl(pred(in, mdi, muo) is nondet, in, mdi, muo) is nondet.
:- mode foldl(pred(in, di, uo) is cc_multi, in, di, uo) is cc_multi.
:- mode foldl(pred(in, in, out) is cc_multi, in, in, out) is cc_multi.

:- pred foldl2(pred(T, U, U, V, V), tree_bitset(T), U, U, V, V) <= enum(T).
:- mode foldl2(pred(in, di, uo, di, uo) is det, in, di, uo, di, uo) is det.
:- mode foldl2(pred(in, in, out, di, uo) is det, in, in, out, di, uo) is det.
:- mode foldl2(pred(in, in, out, in, out) is det, in, in, out, in, out) is det.
:- mode foldl2(pred(in, in, out, in, out) is semidet, in, in, out, in, out)
    is semidet.
:- mode foldl2(pred(in, in, out, in, out) is nondet, in, in, out, in, out)
    is nondet.
:- mode foldl2(pred(in, di, uo, di, uo) is cc_multi, in, di, uo, di, uo)
    is cc_multi.
:- mode foldl2(pred(in, in, out, di, uo) is cc_multi, in, in, out, di, uo)
    is cc_multi.
```

```
:- mode foldl2(pred(in, in, out, in, out) is cc_multi, in, in, out, in, out)
    is cc_multi.

    % 'foldr(Func, Set, Start)' calls Func with each element of 'Set'
    % (in reverse sorted order) and an accumulator (with the initial value
    % of 'Start'), and returns the final value. Takes O(card(Set)) time.
    %
:- func foldr(func(T, U) = U, tree_bitset(T), U) = U <= enum(T).

:- pred foldr(pred(T, U, U), tree_bitset(T), U, U) <= enum(T).
:- mode foldr(pred(in, di, uo) is det, in, di, uo) is det.
:- mode foldr(pred(in, in, out) is det, in, in, out) is det.
:- mode foldr(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode foldr(pred(in, in, out) is nondet, in, in, out) is nondet.
:- mode foldr(pred(in, di, uo) is cc_multi, in, di, uo) is cc_multi.
:- mode foldr(pred(in, in, out) is cc_multi, in, in, out) is cc_multi.

:- pred foldr2(pred(T, U, U, V, V), tree_bitset(T), U, U, V, V) <= enum(T).
:- mode foldr2(pred(in, di, uo, di, uo) is det, in, di, uo, di, uo) is det.
:- mode foldr2(pred(in, in, out, di, uo) is det, in, in, out, di, uo) is det.
:- mode foldr2(pred(in, in, out, in, out) is det, in, in, out, in, out) is det.
:- mode foldr2(pred(in, in, out, in, out) is semidet, in, in, out, in, out)
    is semidet.
:- mode foldr2(pred(in, in, out, in, out) is nondet, in, in, out, in, out)
    is nondet.
:- mode foldr2(pred(in, di, uo, di, uo) is cc_multi, in, di, uo, di, uo)
    is cc_multi.
:- mode foldr2(pred(in, in, out, di, uo) is cc_multi, in, in, out, di, uo)
    is cc_multi.
:- mode foldr2(pred(in, in, out, in, out) is cc_multi, in, in, out, in, out)
    is cc_multi.

%---------------------------------------------------%
```

101 type_desc

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 2002-2007, 2009-2012 The University of Melbourne.
% Copyright (C) 2013-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: type_desc.m.
```

```
% Main author: fjh, zs.
% Stability: low.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module type_desc.
:- interface.

:- import_module list.

%-----------------------------------------------------%

    % The 'type_desc', 'pseudo_type_desc' and 'type_ctor_desc' types
    % provide access to type information.
    % A type_desc represents a type, e.g. 'list(int)'.
    % A pseudo_type_desc represents a type that possibly contains type
    % variables, e.g. 'list(T)'.
    % A type_ctor_desc represents a type constructor, e.g. 'list/1'.
    %
:- type type_desc.
:- type pseudo_type_desc.
:- type type_ctor_desc.

    % The possibly nonground type represented by a pseudo_type_desc
    % is either a type constructor applied to zero or more
    % pseudo_type_descs, or a type variable. If the latter, the
    % type variable may be either universally or existentially quantified.
    % In either case, the type is identified by an integer, which has no
    % meaning beyond the fact that two type variables will be represented
    % by identical integers if and only if they are the same type variable.
    % Existentially quantified type variables may have type class
    % constraints placed on them, but for now we can't return these.
    %
:- type pseudo_type_rep
    --->    bound(type_ctor_desc, list(pseudo_type_desc))
    ;       univ_tvar(int)
    ;       exist_tvar(int).

:- pred pseudo_type_desc_is_ground(pseudo_type_desc::in) is semidet.

    % This function allows the caller to look into the structure
    % of the given pseudo_type_desc.
    %
:- func pseudo_type_desc_to_rep(pseudo_type_desc) = pseudo_type_rep.

    % Convert a type_desc, which by definition describes a ground type,
```

```
    % to a pseudo_type_desc.
    %
:- func type_desc_to_pseudo_type_desc(type_desc) = pseudo_type_desc.

    % Convert a pseudo_type_desc describing a ground type to a type_desc.
    % If the pseudo_type_desc describes a non-ground type, fail.
    %
:- func ground_pseudo_type_desc_to_type_desc(pseudo_type_desc) = type_desc
    is semidet.
:- pred ground_pseudo_type_desc_to_type_desc(pseudo_type_desc::in,
    type_desc::out) is semidet.

    % Convert a pseudo_type_desc describing a ground type to a type_desc.
    % Throw an exception if the pseudo_type_desc describes a non-ground type.
    %
:- func det_ground_pseudo_type_desc_to_type_desc(pseudo_type_desc) = type_desc.

%-----------------------------------------------------%

    % The function type_of/1 returns a representation of the type
    % of its argument.
    %
    % (Note: it is not possible for the type of a variable to be an unbound
    % type variable; if there are no constraints on a type variable, then the
    % typechecker will use the type `void'. `void' is a special (builtin) type
    % that has no constructors. There is no way of creating an object of
    % type `void'. `void' is not considered to be a discriminated union, so
    % get_functor/5 and construct/3 will fail if used upon a value of
    % this type.)
    %
:- func type_of(T::unused) = (type_desc::out) is det.

    % The predicate has_type/2 is basically an existentially typed inverse
    % to the function type_of/1. It constrains the type of the first argument
    % to be the type represented by the second argument.
    %
:- some [T] pred has_type(T::unused, type_desc::in) is det.

    % The predicate same_type/2 ensures type identity of the two arguments.
    %
:- pred same_type(T::unused, T::unused) is det.

    % type_name(Type) returns the name of the specified type
    % (e.g. type_name(type_of([2,3])) = "list.list(int)").
    % Any equivalence types will be fully expanded.
    % Builtin types (those defined in builtin.m) will not have
    % a module qualifier.
```

```
    %
:- func type_name(type_desc) = string.

    % type_ctor_and_args(Type, TypeCtor, TypeArgs):
    %
    % True iff 'TypeCtor' is a representation of the top-level type constructor
    % for 'Type', and 'TypeArgs' is a list of the corresponding type arguments
    % to 'TypeCtor', and 'TypeCtor' is not an equivalence type.
    %
    % For example, type_ctor_and_args(type_of([2,3]), TypeCtor, TypeArgs)
    % will bind 'TypeCtor' to a representation of the type constructor list/1,
    % and will bind 'TypeArgs' to the list '[Int]', where 'Int' is a
    % representation of the type 'int'.
    %
    % Note that the requirement that 'TypeCtor' not be an equivalence type
    % is fulfilled by fully expanding any equivalence types. For example,
    % if you have a declaration ':- type foo == bar.', then
    % type_ctor_and_args/3 will always return a representation of type
    % constructor 'bar/0', not 'foo/0'. (If you don't want them expanded,
    % you can use the reverse mode of make_type/2 instead.)
    %
:- pred type_ctor_and_args(type_desc::in,
    type_ctor_desc::out, list(type_desc)::out) is det.

    % pseudo_type_ctor_and_args(Type, TypeCtor, TypeArgs):
    %
    % True iff 'TypeCtor' is a representation of the top-level type constructor
    % for 'Type', and 'TypeArgs' is a list of the corresponding type arguments
    % to 'TypeCtor', and 'TypeCtor' is not an equivalence type.
    %
    % Similar to type_ctor_and_args, but works on pseudo_type_infos.
    % Fails if the input pseudo_type_info is a variable.
    %
:- pred pseudo_type_ctor_and_args(pseudo_type_desc::in,
    type_ctor_desc::out, list(pseudo_type_desc)::out) is semidet.

    % type_ctor(Type) = TypeCtor :-
    %   type_ctor_and_args(Type, TypeCtor, _).
    %
:- func type_ctor(type_desc) = type_ctor_desc.

    % pseudo_type_ctor(Type) = TypeCtor :-
    %   pseudo_type_ctor_and_args(Type, TypeCtor, _).
    %
:- func pseudo_type_ctor(pseudo_type_desc) = type_ctor_desc is semidet.

    % type_args(Type) = TypeArgs :-
```

```
    %   type_ctor_and_args(Type, _, TypeArgs).
    %
:- func type_args(type_desc) = list(type_desc).

    % pseudo_type_args(Type) = TypeArgs :-
    %   pseudo_type_ctor_and_args(Type, _, TypeArgs).
    %
:- func pseudo_type_args(pseudo_type_desc) = list(pseudo_type_desc) is semidet.

    % type_ctor_name(TypeCtor) returns the name of specified type constructor.
    % (e.g. type_ctor_name(type_ctor(type_of([2,3]))) = "list").
    %
:- func type_ctor_name(type_ctor_desc) = string.

    % type_ctor_module_name(TypeCtor) returns the module name of specified
    % type constructor.
    % (e.g. type_ctor_module_name(type_ctor(type_of(2))) = "builtin").
    %
:- func type_ctor_module_name(type_ctor_desc) = string.

    % type_ctor_arity(TypeCtor) returns the arity of specified
    % type constructor.
    % (e.g. type_ctor_arity(type_ctor(type_of([2,3]))) = 1).
    %
:- func type_ctor_arity(type_ctor_desc) = int.

    % type_ctor_name_and_arity(TypeCtor, ModuleName, TypeName, Arity) :-

    %   Name = type_ctor_name(TypeCtor),
    %   ModuleName = type_ctor_module_name(TypeCtor),
    %   Arity = type_ctor_arity(TypeCtor).
    %
:- pred type_ctor_name_and_arity(type_ctor_desc::in,
    string::out, string::out, int::out) is det.

    % make_type(TypeCtor, TypeArgs) = Type:
    %
    % True iff 'Type' is a type constructed by applying the type constructor
    % 'TypeCtor' to the type arguments 'TypeArgs'.
    %
    % Operationally, the forwards mode returns the type formed by applying
    % the specified type constructor to the specified argument types, or fails
    % if the length of TypeArgs is not the same as the arity of TypeCtor.
    % The reverse mode returns a type constructor and its argument types,
    % given a type_desc; the type constructor returned may be an equivalence
    % type (and hence this reverse mode of make_type/2 may be more useful
    % for some purposes than the type_ctor/1 function).
```

```
    %
:- func make_type(type_ctor_desc, list(type_desc)) = type_desc.
:- mode make_type(in, in) = out is semidet.
:- mode make_type(out, out) = in is cc_multi.

    % det_make_type(TypeCtor, TypeArgs):
    %
    % Returns the type formed by applying the specified type constructor
    % to the specified argument types. Throws an exception if the length of
    % ‘TypeArgs’ is not the same as the arity of ‘TypeCtor’.
    %
:- func det_make_type(type_ctor_desc, list(type_desc)) = type_desc.

%---------------------------------------------------%
%---------------------------------------------------%
```

102 uint

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 2016-2020 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: uint.m
% Main author: juliensf
% Stability: low.
%
% Predicates and functions for dealing with unsigned machine sized integer
% numbers.
%
%---------------------------------------------------%

:- module uint.
:- interface.

:- import_module pretty_printer.

%---------------------------------------------------%

    % Convert an int to a uint.
    % Fails if the int is less than zero.
    %
:- pred from_int(int::in, uint::out) is semidet.
```

```
    % As above, but throw an exception instead of failing.
    %
:- func det_from_int(int) = uint.

:- func cast_from_int(int) = uint.

:- func cast_to_int(uint) = int.

    % Less than.
    %
:- pred (uint::in) < (uint::in) is semidet.

    % Greater than.
    %
:- pred (uint::in) > (uint::in) is semidet.

    % Less than or equal.
    %
:- pred (uint::in) =< (uint::in) is semidet.

    % Greater than or equal.
    %
:- pred (uint::in) >= (uint::in) is semidet.

    % Maximum.
    %
:- func max(uint, uint) = uint.

    % Minimum.
    %
:- func min(uint, uint) = uint.

    % Addition.
    %
:- func uint + uint = uint.
:- mode in   + in  = uo is det.
:- mode uo   + in  = in is det.
:- mode in   + uo  = in is det.

:- func plus(uint, uint) = uint.

    % Subtraction.
    %
:- func uint - uint = uint.
:- mode in   - in   = uo is det.
:- mode uo   - in   = in is det.
```

```
:- mode in   - uo   = in is det.

:- func minus(uint, uint) = uint.

    % Multiplication.
    %
:- func (uint::in) * (uint::in) = (uint::uo) is det.
:- func times(uint, uint) = uint.

    % Truncating integer division.
    %
    % Throws a ‘domain_error’ exception if the right operand is zero.
    %
:- func (uint::in) div (uint::in) = (uint::uo) is det.

    % Truncating integer division.
    %
    % Throws a ‘domain_error’ exception if the right operand is zero.
    %
:- func (uint::in) // (uint::in) = (uint::uo) is det.

    % (/)/2 is a synonym for (//)/2.
    %
:- func (uint::in) / (uint::in) = (uint::uo) is det.

    % unchecked_quotient(X, Y) is the same as X // Y, but the behaviour
    % is undefined if the right operand is zero.
    %
:- func unchecked_quotient(uint::in, uint::in) = (uint::uo) is det.

    % Modulus.
    % X mod Y = X - (X div Y) * Y
    %
    % Throws a ‘domain_error’ exception if the right operand is zero.
    %
:- func (uint::in) mod (uint::in) = (uint::uo) is det.

    % Remainder.
    % X rem Y = X - (X // Y) * Y.
    %
    % Throws a ‘domain_error/‘ exception if the right operand is zero.
    %
:- func (uint::in) rem (uint::in) = (uint::uo) is det.

    % unchecked_rem(X, Y) is the same as X rem Y, but the behaviour is
    % undefined if the right operand is zero.
    %
```

```
:- func unchecked_rem(uint::in, uint::in) = (uint::uo) is det.

    % Left shift.
    % X << Y returns X "left shifted" by Y bits.
    % The bit positions vacated by the shift are filled by zeros.
    % Throws an exception if Y is not in [0, bits_per_uint).
    %
:- func (uint::in) << (int::in) = (uint::uo) is det.

    % unchecked_left_shift(X, Y) is the same as X << Y except that the
    % behaviour is undefined if Y is not in [0, bits_per_uint).
    % It will typically be implemented more efficiently than X << Y.
    %
:- func unchecked_left_shift(uint::in, int::in) = (uint::uo) is det.

    % Right shift.
    % X >> Y returns X "right shifted" by Y bits.
    % The bit positions vacated by the shift are filled by zeros.
    % Throws an exception if Y is not in [0, bits_per_uint).
    %
:- func (uint::in) >> (int::in) = (uint::uo) is det.

    % unchecked_right_shift(X, Y) is the same as X >> Y except that the
    % behaviour is undefined if Y is not in [0, bits_per_uint).
    % It will typically be implemented more efficiently than X >> Y.
    %
:- func unchecked_right_shift(uint::in, int::in) = (uint::uo) is det.

    % even(X) is equivalent to (X mod 2 = 0).
    %
:- pred even(uint::in) is semidet.

    % odd(X) is equivalent to (not even(X)), i.e. (X mod 2 = 1).
    %
:- pred odd(uint::in) is semidet.

    % Bitwise and.
    %
:- func (uint::in) /\ (uint::in) = (uint::uo) is det.

    % Bitwise or.
    %
:- func (uint::in) \/ (uint::in) = (uint::uo) is det.

    % Bitwise exclusive or (xor).
    %
:- func xor(uint, uint) = uint.
```

```
:- mode xor(in, in) = uo is det.
:- mode xor(in, uo) = in is det.
:- mode xor(uo, in) = in is det.

    % Bitwise complement.
    %
:- func \ (uint::in) = (uint::uo) is det.

    % max_uint is the maximum value of a uint on this machine.
    %
:- func max_uint = uint.

    % bits_per_uint is the number of bits in a uint on this machine.
    %
:- func bits_per_uint = int.

    % Convert a uint to a pretty_printer.doc for formatting.
    %
:- func uint_to_doc(uint) = pretty_printer.doc.

%---------------------------------------------------%
%
% Computing hashes of uints.
%

    % Compute a hash value for a uint.
    %
:- func hash(uint) = int.
:- pred hash(uint::in, int::out) is det.

%---------------------------------------------------%
%---------------------------------------------------%
```

103 uint8

```
%---------------------------------------------------%
% vim: ts=4 sw=4 et ft=mercury
%---------------------------------------------------%
% Copyright (C) 2017-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: uint8.m
% Main author: juliensf
% Stability: low.
```

```
%
% Predicates and functions for dealing with unsigned 8-bit integer numbers.
%
%-----------------------------------------------------%

:- module uint8.
:- interface.

:- import_module pretty_printer.

%-----------------------------------------------------%
%
% Conversion from int.
%

    % from_int(I, U8):
    %
    % Convert an int to a uint8.
    % Fails if I is not in [0, 2^8 - 1].
    %
:- pred from_int(int::in, uint8::out) is semidet.

    % det_from_int(I) = U8:
    %
    % Convert an int to a uint8.
    % Throws an exception if I is not in [0, 2^8 - 1].
    %
:- func det_from_int(int) = uint8.

    % cast_from_int(I) = U8:
    %
    % Convert an int to a uint8.
    % Always succeeds, but will yield a result that is mathematically equal
    % to I only if I is in [0, 2^8 - 1].
    %
:- func cast_from_int(int) = uint8.

%-----------------------------------------------------%
%
% Conversion to int.
%

    % to_int(U8) = I:
    %
    % Convert a uint8 to an int.
    % Always succeeds, and yields a result that is mathematically equal
    % to U8.
```

```
    %
:- func to_int(uint8) = int.

    % cast_to_int(U8) = I:
    %
    % Convert a uint8 to an int.
    % Always succeeds, and yields a result that is mathematically equal
    % to U8.
    %
:- func cast_to_int(uint8) = int.

%---------------------------------------------------%
%
% Conversion to uint.
%

    % cast_to_uint(U8) = U:
    %
    % Convert a uint8 to a uint.
    % Always succeeds, and yields a result that is mathematically equal
    % to U8.
    %
:- func cast_to_uint(uint8) = uint.

%---------------------------------------------------%
%
% Change of signedness.
%

    % cast_from_int8(I8) = U8:
    %
    % Convert an int8 to a uint8. This will yield a result that is
    % mathematically equal to I8 only if I8 is in [0, 2^7 - 1].
    %
:- func cast_from_int8(int8) = uint8.

%---------------------------------------------------%
%
% Comparisons and related operations.
%

    % Less than.
    %
:- pred (uint8::in) < (uint8::in) is semidet.

    % Greater than.
    %
```

```
:- pred (uint8::in) > (uint8::in) is semidet.

    % Less than or equal.
    %
:- pred (uint8::in) =< (uint8::in) is semidet.

    % Greater than or equal.
    %
:- pred (uint8::in) >= (uint8::in) is semidet.

    % Maximum.
    %
:- func max(uint8, uint8) = uint8.

    % Minimum.
    %
:- func min(uint8, uint8) = uint8.

%-----------------------------------------------%
%
% Arithmetic operations.
%

    % Addition.
    %
:- func uint8 + uint8 = uint8.
:- mode in + in = uo is det.
:- mode uo + in = in is det.
:- mode in + uo = in is det.

:- func plus(uint8, uint8) = uint8.

    % Subtraction.
    %
:- func uint8 - uint8 = uint8.
:- mode in - in = uo is det.
:- mode uo - in = in is det.
:- mode in - uo = in is det.

:- func minus(uint8, uint8) = uint8.

    % Multiplication.
    %
:- func (uint8::in) * (uint8::in) = (uint8::uo) is det.
:- func times(uint8, uint8) = uint8.

    % Truncating integer division.
```

```
    %
    % Throws a ‘domain_error’ exception if the right operand is zero.
    %
:- func (uint8::in) div (uint8::in) = (uint8::uo) is det.

    % Truncating integer division.
    %
    % Throws a ‘domain_error’ exception if the right operand is zero.
    %
:- func (uint8::in) // (uint8::in) = (uint8::uo) is det.

    % (/)/2 is a synonym for (//)/2.
    %
:- func (uint8::in) / (uint8::in) = (uint8::uo) is det.

    % unchecked_quotient(X, Y) is the same as X // Y, but the behaviour
    % is undefined if the right operand is zero.
    %
:- func unchecked_quotient(uint8::in, uint8::in) = (uint8::uo) is det.

    % Modulus.
    % X mod Y = X - (X div Y) * Y
    %
    % Throws a ‘domain_error’ exception if the right operand is zero.
    %
:- func (uint8::in) mod (uint8::in) = (uint8::uo) is det.

    % Remainder.
    % X rem Y = X - (X // Y) * Y.
    %
    % Throws a ‘domain_error/‘ exception if the right operand is zero.
    %
:- func (uint8::in) rem (uint8::in) = (uint8::uo) is det.

    % unchecked_rem(X, Y) is the same as X rem Y, but the behaviour is
    % undefined if the right operand is zero.
    %
:- func unchecked_rem(uint8::in, uint8::in) = (uint8::uo) is det.

    % even(X) is equivalent to (X mod 2u8 = 0u8).
    %
:- pred even(uint8::in) is semidet.

    % odd(X) is equivalent to (not even(X)), i.e. (X mod 2u8 = 1u8).
    %
:- pred odd(uint8::in) is semidet.
```

```
%-----------------------------------------------%
%
% Shift operations.
%

    % Left shift.
    % X << Y returns X "left shifted" by Y bits.
    % The bit positions vacated by the shift are filled by zeros.
    % Throws an exception if Y is not in [0, 8).
    %
:- func (uint8::in) << (int::in) = (uint8::uo) is det.

    % unchecked_left_shift(X, Y) is the same as X << Y except that the
    % behaviour is undefined if Y is not in [0, 8).
    % It will typically be implemented more efficiently than X << Y.
    %
:- func unchecked_left_shift(uint8::in, int::in) = (uint8::uo) is det.

    % Right shift.
    % X >> Y returns X "right shifted" by Y bits.
    % The bit positions vacated by the shift are filled by zeros.
    % Throws an exception if Y is not in [0, 8).
    %
:- func (uint8::in) >> (int::in) = (uint8::uo) is det.

    % unchecked_right_shift(X, Y) is the same as X >> Y except that the
    % behaviour is undefined if Y is not in [0, 8).
    % It will typically be implemented more efficiently than X >> Y.
    %
:- func unchecked_right_shift(uint8::in, int::in) = (uint8::uo) is det.

%-----------------------------------------------%
%
% Logical operations.
%

    % Bitwise and.
    %
:- func (uint8::in) /\ (uint8::in) = (uint8::uo) is det.

    % Bitwise or.
    %
:- func (uint8::in) \/ (uint8::in) = (uint8::uo) is det.

    % Bitwise exclusive or (xor).
    %
:- func xor(uint8, uint8) = uint8.
```

```
:- mode xor(in, in) = uo is det.
:- mode xor(in, uo) = in is det.
:- mode xor(uo, in) = in is det.

    % Bitwise complement.
    %
:- func \ (uint8::in) = (uint8::uo) is det.

%---------------------------------------------------%
%
% Operations on bits and bytes.
%

    % num_zeros(U) = N:
    %
    % N is the number of zeros in the binary representation of U.
    %
:- func num_zeros(uint8) = int.

    % num_ones(U) = N:
    %
    % N is the number of ones in the binary representation of U.
    %
:- func num_ones(uint8) = int.

    % num_leading_zeros(U) = N:
    %
    % N is the number of leading zeros in the binary representation of U,
    % starting at the most significant bit position.
    % Note that num_leading_zeros(0u8) = 8.
    %
:- func num_leading_zeros(uint8) = int.

    % num_trailing_zeros(U) = N:
    %
    % N is the number of trailing zeros in the binary representation of U,
    % starting at the least significant bit position.
    % Note that num_trailing_zeros(0u8) = 8.
    %
:- func num_trailing_zeros(uint8) = int.

    % reverse_bits(A) = B:
    %
    % B is the is value that results from reversing the bits in the binary
    % representation of A.
    %
:- func reverse_bits(uint8) = uint8.
```

```
%-----------------------------------------------------%
%
% Limits.
%

:- func max_uint8 = uint8.

%-----------------------------------------------------%
%
% Prettyprinting.
%

    % Convert an uint8 to a pretty_printer.doc for formatting.
    %
:- func uint8_to_doc(uint8) = pretty_printer.doc.

%-----------------------------------------------------%
%-----------------------------------------------------%
```

104 uint16

```
%-----------------------------------------------------%
% vim: ts=4 sw=4 et ft=mercury
%-----------------------------------------------------%
% Copyright (C) 2017-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: uint16.m
% Main author: juliensf
% Stability: low.
%
% Predicates and functions for dealing with unsigned 16-bit integer numbers.
%
%-----------------------------------------------------%

:- module uint16.
:- interface.

:- import_module pretty_printer.

%-----------------------------------------------------%
%
% Conversion from int.
```

```
%

    % from_int(I, U16):
    %
    % Convert an int into a uint16.
    % Fails if I is not in [0, 2^16 - 1].
    %
:- pred from_int(int::in, uint16::out) is semidet.

    % det_from_int(I) = U16:
    %
    % Convert an int into a uint16.
    % Throws an exception if I is not in [0, 2^16 - 1].
    %
:- func det_from_int(int) = uint16.

    % cast_from_int(I) = U16:
    %
    % Convert an int to a uint16.
    % Always succeeds, but will yield a result that is mathematically equal
    % to I only if I is in [0, 2^16 - 1].
    %
:- func cast_from_int(int) = uint16.

%-----------------------------------------------------%
%
% Conversion to int.
%

    % to_int(U16) = I:
    %
    % Convert a uint16 to an int.
    % Always succeeds, and yields a result that is mathematically equal
    % to U16.
    %
:- func to_int(uint16) = int.

    % cast_to_int(U16) = I:
    %
    % Convert a uint16 to an int.
    % Always succeeds, and yields a result that is mathematically equal
    % to U16.
    %
:- func cast_to_int(uint16) = int.

%-----------------------------------------------------%
%
```

```
% Conversion to uint.
%

    % cast_to_uint(U16) = U:
    %
    % Convert a uint16 to a uint.
    % Always succeeds, and yields a result that is mathematically equal
    % to U16.
    %
:- func cast_to_uint(uint16) = uint.

%---------------------------------------------------%
%
% Conversion to/from uint64.
%

    % cast_to_uint64(U16) = U64:
    %
    % Convert a uint16 to a uint64.
    % Always succeeds, and yields a result that is mathematically equal
    % to U16.
    %
:- func cast_to_uint64(uint16) = uint64.

    % cast_from_uint64(U64) = U16:
    %
    % Convert a uint64 to a uint16.
    % Always succeeds, but will yield a result that is mathematically equal
    % to I only if I is in [0, 2^16 - 1].
    %
:- func cast_from_uint64(uint64) = uint16.

%---------------------------------------------------%
%
% Change of signedness.
%

    % cast_from_int16(I16) = U16:
    %
    % Convert an int16 to a uint16. This will yield a result that is
    % mathematically equal to I16 only if I16 is in [0, 2^15 - 1].
    %
:- func cast_from_int16(int16) = uint16.

%---------------------------------------------------%
%
% Conversion from byte sequence.
```

```
%

    % from_bytes_le(LSB, MSB) = U16:
    %
    % U16 is the uint16 whose least and most significant bytes are given by the
    % uint8s LSB and MSB respectively.
    %
:- func from_bytes_le(uint8, uint8) = uint16.

    % from_bytes_be(MSB, LSB) = U16:
    %
    % U16 is the uint16 whose least and most significant bytes are given by the
    % uint8s LSB and MSB respectively.
    %
:- func from_bytes_be(uint8, uint8) = uint16.

%---------------------------------------------------%
%
% Comparisons and related operations.
%

    % Less than.
    %
:- pred (uint16::in) < (uint16::in) is semidet.

    % Greater than.
    %
:- pred (uint16::in) > (uint16::in) is semidet.

    % Less than or equal.
    %
:- pred (uint16::in) =< (uint16::in) is semidet.

    % Greater than or equal.
    %
:- pred (uint16::in) >= (uint16::in) is semidet.

    % Maximum.
    %
:- func max(uint16, uint16) = uint16.

    % Minimum.
    %
:- func min(uint16, uint16) = uint16.

%---------------------------------------------------%
%
```

```
% Arithmetic operations.
%

    % Addition.
    %
:- func uint16 + uint16 = uint16.
:- mode in + in = uo is det.
:- mode uo + in = in is det.
:- mode in + uo = in is det.

:- func plus(uint16, uint16) = uint16.

    % Subtraction.
    %
:- func uint16 - uint16 = uint16.
:- mode in - in = uo is det.
:- mode uo - in = in is det.
:- mode in - uo = in is det.

:- func minus(uint16, uint16) = uint16.

    % Multiplication.
    %
:- func (uint16::in) * (uint16::in) = (uint16::uo) is det.
:- func times(uint16, uint16) = uint16.

    % Truncating integer division.
    %
    % Throws a 'domain_error' exception if the right operand is zero.
    %
:- func (uint16::in) div (uint16::in) = (uint16::uo) is det.

    % Truncating integer division.
    %
    % Throws a 'domain_error' exception if the right operand is zero.
    %
:- func (uint16::in) // (uint16::in) = (uint16::uo) is det.

    % (/)/2 is a synonym for (//)/2.
    %
:- func (uint16::in) / (uint16::in) = (uint16::uo) is det.

    % unchecked_quotient(X, Y) is the same as X // Y, but the behaviour
    % is undefined if the right operand is zero.
    %
:- func unchecked_quotient(uint16::in, uint16::in) = (uint16::uo) is det.
```

```
    % Modulus.
    % X mod Y = X - (X div Y) * Y
    %
    % Throws a 'domain_error' exception if the right operand is zero.
    %
:- func (uint16::in) mod (uint16::in) = (uint16::uo) is det.

    % Remainder.
    % X rem Y = X - (X // Y) * Y.
    %
    % Throws a 'domain_error/' exception if the right operand is zero.
    %
:- func (uint16::in) rem (uint16::in) = (uint16::uo) is det.

    % unchecked_rem(X, Y) is the same as X rem Y, but the behaviour is
    % undefined if the right operand is zero.
    %
:- func unchecked_rem(uint16::in, uint16::in) = (uint16::uo) is det.

    % even(X) is equivalent to (X mod 2 = 0).
    %
:- pred even(uint16::in) is semidet.

    % odd(X) is equivalent to (not even(X)), i.e. (X mod 2 = 1).
    %
:- pred odd(uint16::in) is semidet.

%-----------------------------------------------------%
%
% Shift operations.
%

    % Left shift.
    % X << Y returns X "left shifted" by Y bits.
    % The bit positions vacated by the shift are filled by zeros.
    % Throws an exception if Y is not in [0, 16).
    %
:- func (uint16::in) << (int::in) = (uint16::uo) is det.

    % unchecked_left_shift(X, Y) is the same as X << Y except that the
    % behaviour is undefined if Y is not in [0, 16).
    % It will typically be implemented more efficiently than X << Y.
    %
:- func unchecked_left_shift(uint16::in, int::in) = (uint16::uo) is det.

    % Right shift.
    % X >> Y returns X "right shifted" by Y bits.
```

```
    % The bit positions vacated by the shift are filled by zeros.
    % Throws an exception if Y is not in [0, 16).
    %
:- func (uint16::in) >> (int::in) = (uint16::uo) is det.

    % unchecked_right_shift(X, Y) is the same as X >> Y except that the
    % behaviour is undefined if Y is not in [0, 16).
    % It will typically be implemented more efficiently than X >> Y.
    %
:- func unchecked_right_shift(uint16::in, int::in) = (uint16::uo) is det.

%-----------------------------------------------------%
%
% Logical operations.
%

    % Bitwise and.
    %
:- func (uint16::in) /\ (uint16::in) = (uint16::uo) is det.

    % Bitwise or.
    %
:- func (uint16::in) \/ (uint16::in) = (uint16::uo) is det.

    % Bitwise exclusive or (xor).
    %
:- func xor(uint16, uint16) = uint16.
:- mode xor(in, in) = uo is det.
:- mode xor(in, uo) = in is det.
:- mode xor(uo, in) = in is det.

    % Bitwise complement.
    %
:- func \ (uint16::in) = (uint16::uo) is det.

%-----------------------------------------------------%
%
% Operations on bits and bytes.
%

    % num_zeros(U) = N:
    %
    % N is the number of zeros in the binary representation of U.
    %
:- func num_zeros(uint16) = int.

    % num_ones(U) = N:
```

```
    %
    % N is the number of ones in the binary representation of U.
    %
:- func num_ones(uint16) = int.

    % num_leading_zeros(U) = N:
    %
    % N is the number of leading zeros in the binary representation of U,
    % starting at the most significant bit position.
    % Note that num_leading_zeros(0u16) = 16.
    %
:- func num_leading_zeros(uint16) = int.

    % num_trailing_zeros(U) = N:
    %
    % N is the number of trailing zeros in the binary representation of U,
    % starting at the least significant bit position.
    % Note that num_trailing_zeros(0u16) = 16.
    %
:- func num_trailing_zeros(uint16) = int.

    % reverse_bytes(A) = B:
    %
    % B is the value that results from reversing the bytes in the binary
    % representation of A.
    %
:- func reverse_bytes(uint16) = uint16.

    % reverse_bits(A) = B:
    %
    % B is the is value that results from reversing the bits in the binary
    % representation of A.
    %
:- func reverse_bits(uint16) = uint16.

%---------------------------------------------------%
%
% Limits.
%

:- func max_uint16 = uint16.

%---------------------------------------------------%
%
% Prettyprinting.
%
```

```
    % Convert a uint16 to a pretty_printer.doc for formatting.
    %
:- func uint16_to_doc(uint16) = pretty_printer.doc.

%---------------------------------------------------%
%---------------------------------------------------%
```

105 uint32

```
%---------------------------------------------------%
% vim: ts=4 sw=4 et ft=mercury
%---------------------------------------------------%
% Copyright (C) 2017-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%---------------------------------------------------%
%
% File: uint32.m
% Main author: juliensf
% Stability: low.
%
% Predicates and functions for dealing with unsigned 32-bit integer numbers.
%
%---------------------------------------------------%

:- module uint32.
:- interface.

:- import_module pretty_printer.

%---------------------------------------------------%
%
% Conversion from int.
%

    % from_int(I, U32):
    %
    % Convert an int into a uint32.
    % Fails if I is not in [0, 2^32 - 1].
    %
:- pred from_int(int::in, uint32::out) is semidet.

    % det_from_int(I) = U32:
    %
    % Convert an int into a uint32.
    % Throws an exception if I is not in [0, 2^32 - 1].
```

```
    %
:- func det_from_int(int) = uint32.

    % cast_from_int(I) = U32:
    %
    % Convert an int to a uint32.
    % Always succeeds, but will yield a result that is mathematically equal
    % to I only if I is in [0, 2^32 - 1].
    %
:- func cast_from_int(int) = uint32.

%-----------------------------------------------%
%
% Conversion to int.
%

    % cast_to_int(U32) = I:
    %
    % Convert a uint32 to an int.
    % Always succeeds. If ints are 64 bits, I will always be
    % mathematically equal to U32. However, if ints are 32 bits,
    % then I will be mathematically equal to U32 only if
    % U32 is in [0, 2^31 - 1].
    %
:- func cast_to_int(uint32) = int.

%-----------------------------------------------%
%
% Conversion to uint.
%

    % cast_to_uint(U32) = U:
    %
    % Convert a uint32 to a uint.
    % Always succeeds, and always yields a result that is
    % mathematically equal to U32.
    %
:- func cast_to_uint(uint32) = uint.

    % cast_from_uint(U) = U32:
    %
    % Convert a uint to a uint32.
    % Always succeeds, but will yield a result that is mathematically equal
    % to I only if I is in [0, 2^32 - 1].
    %
:- func cast_from_uint(uint) = uint32.
```

```
%---------------------------------------------------%
%
% Conversion to/from uint64.
%

    % cast_to_uint64(U32) = U64:
    %
    % Convert a uint32 to a uint64.
    % Always succeeds, and always yields a result that is
    % mathematically equal to U32.
    %
:- func cast_to_uint64(uint32) = uint64.

    % cast_from_uint64(U64) = U32:
    %
    % Convert a uint64 to a uint32.
    % Always succeeds, but will yield a result that is mathematically equal
    % to I only if I is in [0, 2^32 - 1].
    %
:- func cast_from_uint64(uint64) = uint32.

%---------------------------------------------------%
%
% Change of signedness.
%

    % cast_from_int32(I32) = U32:
    %
    % Convert an int32 to a uint32. This will yield a result that is
    % mathematically equal to I32 only if I32 is in [0, 2^31 - 1].
    %
:- func cast_from_int32(int32) = uint32.

%---------------------------------------------------%
%
% Conversion from byte sequence.
%

    % from_bytes_le(Byte0, Byte1, Byte2, Byte3) = U32:
    %
    % U32 is the uint32 whose bytes are given in little-endian order by the
    % arguments from left-to-right (i.e. Byte0 is the least significant byte
    % and Byte3 is the most significant byte).
    %
:- func from_bytes_le(uint8, uint8, uint8, uint8) = uint32.

    % from_bytes_be(Byte0, Byte1, Byte2, Byte3) = U32:
```

```
    %
    % U32 is the uint32 whose bytes are given in big-endian order by the
    % arguments in left-to-right order (i.e. Byte0 is the most significant
    % byte and Byte3 is the least significant byte).
    %
:- func from_bytes_be(uint8, uint8, uint8, uint8) = uint32.

%---------------------------------------------------%
%
% Comparisons and related operations.
%

    % Less than.
    %
:- pred (uint32::in) < (uint32::in) is semidet.

    % Greater than.
    %
:- pred (uint32::in) > (uint32::in) is semidet.

    % Less than or equal.
    %
:- pred (uint32::in) =< (uint32::in) is semidet.

    % Greater than or equal.
    %
:- pred (uint32::in) >= (uint32::in) is semidet.

    % Maximum.
    %
:- func max(uint32, uint32) = uint32.

    % Minimum.
    %
:- func min(uint32, uint32) = uint32.

%---------------------------------------------------%
%
% Arithmetic operations.
%

    % Addition.
    %
:- func uint32 + uint32 = uint32.
:- mode in   + in  = uo is det.
:- mode uo   + in  = in is det.
:- mode in   + uo  = in is det.
```

```
:- func plus(uint32, uint32) = uint32.

    % Subtraction.
    %
:- func uint32 - uint32 = uint32.
:- mode in   - in   = uo is det.
:- mode uo   - in   = in is det.
:- mode in   - uo   = in is det.

:- func minus(uint32, uint32) = uint32.

    % Multiplication.
    %
:- func (uint32::in) * (uint32::in) = (uint32::uo) is det.
:- func times(uint32, uint32) = uint32.

    % Truncating integer division.
    %
    % Throws a 'domain_error' exception if the right operand is zero.
    %
:- func (uint32::in) div (uint32::in) = (uint32::uo) is det.

    % Truncating integer division.
    %
    % Throws a 'domain_error' exception if the right operand is zero.
    %
:- func (uint32::in) // (uint32::in) = (uint32::uo) is det.

    % (/)/2 is a synonym for (//)/2.
    %
:- func (uint32::in) / (uint32::in) = (uint32::uo) is det.

    % unchecked_quotient(X, Y) is the same as X // Y, but the behaviour
    % is undefined if the right operand is zero.
    %
:- func unchecked_quotient(uint32::in, uint32::in) = (uint32::uo) is det.

    % Modulus.
    % X mod Y = X - (X div Y) * Y
    %
    % Throws a 'domain_error' exception if the right operand is zero.
    %
:- func (uint32::in) mod (uint32::in) = (uint32::uo) is det.

    % Remainder.
    % X rem Y = X - (X // Y) * Y.
```

```
    %
    % Throws a `domain_error/` exception if the right operand is zero.
    %
:- func (uint32::in) rem (uint32::in) = (uint32::uo) is det.

    % unchecked_rem(X, Y) is the same as X rem Y, but the behaviour is
    % undefined if the right operand is zero.
    %
:- func unchecked_rem(uint32::in, uint32::in) = (uint32::uo) is det.

    % even(X) is equivalent to (X mod 2 = 0).
    %
:- pred even(uint32::in) is semidet.

    % odd(X) is equivalent to (not even(X)), i.e. (X mod 2 = 1).
    %
:- pred odd(uint32::in) is semidet.

%-----------------------------------------------%
%
% Shift operations.
%

    % Left shift.
    % X << Y returns X "left shifted" by Y bits.
    % The bit positions vacated by the shift are filled by zeros.
    % Throws an exception if Y is not in [0, 32).
    %
:- func (uint32::in) << (int::in) = (uint32::uo) is det.

    % unchecked_left_shift(X, Y) is the same as X << Y except that the
    % behaviour is undefined if Y is not in [0, 32).
    % It will typically be implemented more efficiently than X << Y.
    %
:- func unchecked_left_shift(uint32::in, int::in) = (uint32::uo) is det.

    % Right shift.
    % X >> Y returns X "right shifted" by Y bits.
    % The bit positions vacated by the shift are filled by zeros.
    % Throws an exception if Y is not in [0, 32).
    %
:- func (uint32::in) >> (int::in) = (uint32::uo) is det.

    % unchecked_right_shift(X, Y) is the same as X >> Y except that the
    % behaviour is undefined if Y is not in [0, 32).
    % It will typically be implemented more efficiently than X >> Y.
    %
```

```
:- func unchecked_right_shift(uint32::in, int::in) = (uint32::uo) is det.

%-----------------------------------------------%
%
% Logical operations.
%

    % Bitwise and.
    %
:- func (uint32::in) /\ (uint32::in) = (uint32::uo) is det.

    % Bitwise or.
    %
:- func (uint32::in) \/ (uint32::in) = (uint32::uo) is det.

    % Bitwise exclusive or (xor).
    %
:- func xor(uint32, uint32) = uint32.
:- mode xor(in, in) = uo is det.
:- mode xor(in, uo) = in is det.
:- mode xor(uo, in) = in is det.

    % Bitwise complement.
    %
:- func \ (uint32::in) = (uint32::uo) is det.

%-----------------------------------------------%
%
% Operations on bits and bytes.
%

    % num_zeros(U) = N:
    %
    % N is the number of zeros in the binary representation of U.
    %
:- func num_zeros(uint32) = int.

    % num_ones(U) = N:
    %
    % N is the number of ones in the binary representation of U.
    %
:- func num_ones(uint32) = int.

    % num_leading_zeros(U) = N:
    %
    % N is the number of leading zeros in the binary representation of U,
    % starting at the most significant bit position.
```

```
    % Note that num_leading_zeros(0u32) = 32.
    %
:- func num_leading_zeros(uint32) = int.

    % num_trailing_zeros(U) = N:
    %
    % N is the number of trailing zeros in the binary representation of U,
    % starting at the least significant bit position.
    % Note that num_trailing_zeros(0u32) = 32.
    %
:- func num_trailing_zeros(uint32) = int.

    % reverse_bytes(A) = B:
    %
    % B is the value that results from reversing the bytes in the binary
    % representation of A.
    %
:- func reverse_bytes(uint32) = uint32.

    % reverse_bits(A) = B:
    %
    % B is the is value that results from reversing the bits in the binary
    % representation of A.
    %
:- func reverse_bits(uint32) = uint32.

%---------------------------------------------------%
%
% Limits.
%

:- func max_uint32 = uint32.

%---------------------------------------------------%
%
% Prettyprinting.
%

    % Convert a uint32 to a pretty_printer.doc for formatting.
    %
:- func uint32_to_doc(uint32) = pretty_printer.doc.

%---------------------------------------------------%
%---------------------------------------------------%
```

106 uint64

```
%-----------------------------------------------------%
% vim: ts=4 sw=4 et ft=mercury
%-----------------------------------------------------%
% Copyright (C) 2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: uint64.m
% Main author: juliensf
% Stability: low.
%
% Predicates and functions for dealing with unsigned 64-bit integer numbers.
%
%-----------------------------------------------------%

:- module uint64.
:- interface.

:- import_module pretty_printer.

%-----------------------------------------------------%
%
% Conversion from int.
%

    % from_int(I, U64):
    %
    % Convert an int into a uint64.
    % Fails if I is not in [0, 2^64 - 1].
    %
:- pred from_int(int::in, uint64::out) is semidet.

    % det_from_int(I) = U64:
    %
    % Convert an int into a uint64.
    % Throws an exception if I is not in [0, 2^64 - 1].
    %
:- func det_from_int(int) = uint64.

    % cast_from_int(I) = U64:
    %
    % Convert an int to a uint64.
    % Always succeeds, but will yield a result that is mathematically equal
    % to I only if I is in [0, 2^64 - 1].
    %
```

```
:- func cast_from_int(int) = uint64.

%-----------------------------------------------------%
%
% Conversion to int.
%

    % cast_to_int(U64) = I:
    %
    % Convert a uint64 to an int.
    % Always succeeds. If ints are 64 bits, I will be mathematically
    % equal to U64 only if U64 is in [0, 2^63 - 1]. If ints are 32
    % bits, I will be mathematically equal to U64 only if U64 is in
    % [0, 2^31 - 1].
    %
:- func cast_to_int(uint64) = int.

%-----------------------------------------------------%
%
% Conversion to uint.
%

    % cast_to_uint(U64) = U:
    %
    % Convert a uint64 to a uint.
    % Always succeeds, but will yield a result that is mathematically equal
    % to U64 only if uints are 64 bits.
    %
:- func cast_to_uint(uint64) = uint.

%-----------------------------------------------------%
%
% Change of signedness.
%

    % cast_from_int64(I64) = U64:
    %
    % Convert an int64 to a uint64. This will yield a result that is
    % mathematically equal to I64 only if I64 is in [0, 2^63 - 1].
    %
:- func cast_from_int64(int64) = uint64.

%-----------------------------------------------------%
%
% Conversion from byte sequence.
%
```

```
    % from_bytes_le(Byte0, Byte1, ..., Byte7) = U64:
    %
    % U64 is the uint64 whose bytes are given in little-endian order by the
    % arguments from left-to-right (i.e. Byte0 is the least significant byte
    % and Byte7 is the most significant byte).
    %
:- func from_bytes_le(uint8, uint8, uint8, uint8, uint8, uint8, uint8, uint8)
    = uint64.

    % from_bytes_be(Byte0, Byte1, ..., Byte7) = U64:
    %
    % U64 is the uint64 whose bytes are given in big-endian order by the
    % arguments in left-to-right order (i.e. Byte0 is the most significant
    % byte and Byte7 is the least significant byte).
    %
:- func from_bytes_be(uint8, uint8, uint8, uint8, uint8, uint8, uint8, uint8)
    = uint64.

%-----------------------------------------------%
%
% Comparisons and related operations.
%

    % Less than.
    %
:- pred (uint64::in) < (uint64::in) is semidet.

    % Greater than.
    %
:- pred (uint64::in) > (uint64::in) is semidet.

    % Less than or equal.
    %
:- pred (uint64::in) =< (uint64::in) is semidet.

    % Greater than or equal.
    %
:- pred (uint64::in) >= (uint64::in) is semidet.

    % Maximum.
    %
:- func max(uint64, uint64) = uint64.

    % Minimum.
    %
:- func min(uint64, uint64) = uint64.
```

```
%-----------------------------------------------%
%
% Arithmetic operations.
%

    % Addition.
    %
:- func uint64 + uint64 = uint64.
:- mode in + in = uo is det.
:- mode uo + in = in is det.
:- mode in + uo = in is det.

:- func plus(uint64, uint64) = uint64.

    % Subtraction.
    %
:- func uint64 - uint64 = uint64.
:- mode in - in = uo is det.
:- mode uo - in = in is det.
:- mode in - uo = in is det.

:- func minus(uint64, uint64) = uint64.

    % Multiplication.
    %
:- func (uint64::in) * (uint64::in) = (uint64::uo) is det.
:- func times(uint64, uint64) = uint64.

    % Truncating integer division.
    %
    % Throws a 'domain_error' exception if the right operand is zero.
    %
:- func (uint64::in) div (uint64::in) = (uint64::uo) is det.

    % Truncating integer division.
    %
    % Throws a 'domain_error' exception if the right operand is zero.
    %
:- func (uint64::in) // (uint64::in) = (uint64::uo) is det.

    % (/)/2 is a synonym for (//)/2.
    %
:- func (uint64::in) / (uint64::in) = (uint64::uo) is det.

    % unchecked_quotient(X, Y) is the same as X // Y, but the behaviour
    % is undefined if the right operand is zero.
    %
```

```
:- func unchecked_quotient(uint64::in, uint64::in) = (uint64::uo) is det.

    % Modulus.
    % X mod Y = X - (X div Y) * Y
    %
    % Throws a ‘domain_error’ exception if the right operand is zero.
    %
:- func (uint64::in) mod (uint64::in) = (uint64::uo) is det.

    % Remainder.
    % X rem Y = X - (X // Y) * Y.
    %
    % Throws a ‘domain_error/‘ exception if the right operand is zero.
    %
:- func (uint64::in) rem (uint64::in) = (uint64::uo) is det.

    % unchecked_rem(X, Y) is the same as X rem Y, but the behaviour is
    % undefined if the right operand is zero.
    %
:- func unchecked_rem(uint64::in, uint64::in) = (uint64::uo) is det.

    % even(X) is equivalent to (X mod 2 = 0).
    %
:- pred even(uint64::in) is semidet.

    % odd(X) is equivalent to (not even(X)), i.e. (X mod 2 = 1).
    %
:- pred odd(uint64::in) is semidet.

%-----------------------------------------------------%
%
% Shift operations.
%

    % Left shift.
    % X << Y returns X "left shifted" by Y bits.
    % The bit positions vacated by the shift are filled by zeros.
    % Throws an exception if Y is not in [0, 64).
    %
:- func (uint64::in) << (int::in) = (uint64::uo) is det.

    % unchecked_left_shift(X, Y) is the same as X << Y except that
    % the behaviour is undefined if Y is not in [0, 64).
    % It will typically be implemented more efficiently than X << Y.
    %
:- func unchecked_left_shift(uint64::in, int::in) = (uint64::uo) is det.
```

```
    % Right shift.
    % X >> Y returns X "right shifted" by Y bits.
    % The bit positions vacated by the shift are filled by zeros.
    % Throws an exception if Y is not in [0, 64).
    %
:- func (uint64::in) >> (int::in) = (uint64::uo) is det.

    % unchecked_right_shift(X, Y) is the same as X >> Y except that
    % the behaviour is undefined if Y is not in [0, 64).
    % It will typically be implemented more efficiently than X >> Y.
    %
:- func unchecked_right_shift(uint64::in, int::in) = (uint64::uo) is det.

%-----------------------------------------------------%
%
% Logical operations.
%

    % Bitwise and.
    %
:- func (uint64::in) /\ (uint64::in) = (uint64::uo) is det.

    % Bitwise or.
    %
:- func (uint64::in) \/ (uint64::in) = (uint64::uo) is det.

    % Bitwise exclusive or (xor).
    %
:- func xor(uint64, uint64) = uint64.
:- mode xor(in, in) = uo is det.
:- mode xor(in, uo) = in is det.
:- mode xor(uo, in) = in is det.

    % Bitwise complement.
    %
:- func \ (uint64::in) = (uint64::uo) is det.

%-----------------------------------------------------%
%
% Operations on bits and bytes.
%

    % num_zeros(U) = N:
    %
    % N is the number of zeros in the binary representation of U.
    %
:- func num_zeros(uint64) = int.
```

```
    % num_ones(U) = N:
    %
    % N is the number of ones in the binary representation of U.
    %
:- func num_ones(uint64) = int.

    % num_leading_zeros(U) = N:
    %
    % N is the number of leading zeros in the binary representation of U.
    %
:- func num_leading_zeros(uint64) = int.

    % num_trailing_zeros(U) = N:
    % N is the number of trailing zeros in the binary representation of U.
    %
:- func num_trailing_zeros(uint64) = int.

    % reverse_bytes(A) = B:
    %
    % B is the value that results from reversing the bytes in the binary
    % representation of A.
    %
:- func reverse_bytes(uint64) = uint64.

    % reverse_bits(A) = B:
    %
    % B is the is value that results from reversing the bits
    % in the binary representation of A.
    %
:- func reverse_bits(uint64) = uint64.

%---------------------------------------------------%
%
% Limits.
%

:- func max_uint64 = uint64.

%---------------------------------------------------%
%
% Prettyprinting.
%

    % Convert a uint64 to a pretty_printer.doc for formatting.
    %
:- func uint64_to_doc(uint64) = pretty_printer.doc.
```

```
%-----------------------------------------------------%
%-----------------------------------------------------%
```

107 unit

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 1994-2006 The University of Melbourne.
% Copyright (C) 2014-2015, 2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: unit.m.
% Main author: fjh.
% Stability: high.
%
% The "unit" type -  stores no information at all.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module unit.
:- interface.

%-----------------------------------------------------%

:- type unit ---> unit.

:- type unit(T) ---> unit1.

%-----------------------------------------------------%
:- end_module unit.
%-----------------------------------------------------%
```

108 univ

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 1994-2010 The University of Melbourne.
% Copyright (C) 2014-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
```

```
%
% File: univ.m.
% Main author: fjh.
% Stability: medium.
%
% The universal type ‘univ’
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module univ.
:- interface.

:- import_module type_desc.

%-----------------------------------------------------%

    % An object of type ‘univ’ can hold the type and value of an object of any
    % other type.
    %
:- type univ.

    % type_to_univ(Object, Univ).
    %
    % True iff the type stored in ‘Univ’ is the same as the type of ‘Object’,
    % and the value stored in ‘Univ’ is equal to the value of ‘Object’.
    %
    % Operational, the forwards mode converts an object to type ‘univ’,
    % while the reverse mode converts the value stored in ‘Univ’
    % to the type of ‘Object’, but fails if the type stored in ‘Univ’
    % does not match the type of ‘Object’.
    %
:- pred type_to_univ(T, univ).
:- mode type_to_univ(di, uo) is det.
:- mode type_to_univ(in, out) is det.
:- mode type_to_univ(out, in) is semidet.

    % univ_to_type(Univ, Object) :- type_to_univ(Object, Univ).
    %
:- pred univ_to_type(univ, T).
:- mode univ_to_type(in, out) is semidet.
:- mode univ_to_type(out, in) is det.
:- mode univ_to_type(uo, di) is det.

    % The function univ/1 provides the same functionality as type_to_univ/2.
    % univ(Object) = Univ :- type_to_univ(Object, Univ).
    %
```

```
:- func univ(T) = univ.
:- mode univ(in) = out is det.
:- mode univ(di) = uo is det.
:- mode univ(out) = in is semidet.

    % det_univ_to_type(Univ, Object).
    %
    % The same as the forwards mode of univ_to_type, but throws an exception
    % if univ_to_type fails.
    %
:- pred det_univ_to_type(univ::in, T::out) is det.

    % univ_type(Univ).
    %
    % Returns the type_desc for the type stored in ‘Univ’.
    %
:- func univ_type(univ) = type_desc.

    % univ_value(Univ).
    %
    % Returns the value of the object stored in Univ.
    %
:- some [T] func univ_value(univ) = T.

%-----------------------------------------------------%
%-----------------------------------------------------%
```

109 varset

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 1993-2000,2002-2007, 2009-2011 The University of Melbourne.
% Copyright (C) 2014-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: varset.m.
% Main author: fjh.
% Stability: low.
%
% This file provides facilities for manipulating collections of variables.
% through the 'varset' ADT. These variables are object-level variables,
% and are represented as ground terms, so it might help to think of them
% as "variable ids" rather than Prolog-style variables.
```

```
%
% A varset may record a name and/or a value (binding) with each variable.
%
% Many situations require dealing with several distinct sets of variables
% that should never be mixed together. For example, a compiler may handle
% both program variables and type variables, and it does not make sense
% to a have a single varset containing both program variables and type
% variables. The varsets provided by this module are thus parameterized;
% the compiler can use e.g. varset(prog_var_type) to hold program variables
% and varset(tvar_type) to hold type variables. Since all operations on
% two or more varsets require agreement on the argument of the varset/1
% type constructor, any accidental mixup of different instances of varset/1
% is guaranteed to be caught by the compiler.
%
% In situations in which this is not a concern, programmers may use
% the standard generic varset instance.
%
% Note that there are some design flaws in the relationship between
% varset.m and term.m. There is too much coupling between the two,
% which may and should be fixed later, e.g. by merging the two modules.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module varset.
:- interface.

:- import_module assoc_list.
:- import_module list.
:- import_module map.
:- import_module maybe.
:- import_module set.
:- import_module term.

%-----------------------------------------------------%

:- type varset(T).
:- type varset == varset(generic).

%-----------------------------------------------------%

    % Construct an empty varset.
    %
:- func init = varset(T).
:- pred init(varset(T)::out) is det.

    % Check whether a varset is empty.
```

```
    %
:- pred is_empty(varset(T)::in) is semidet.

%-----------------------------------------------%

    % Create a new variable.
    %
:- pred new_var(var(T)::out, varset(T)::in, varset(T)::out) is det.

    % Create a new named variable.
    %
:- pred new_named_var(string::in, var(T)::out,
    varset(T)::in, varset(T)::out) is det.

    % Create a new named variable with a unique (w.r.t. the varset) number
    % appended to the name.
    %
:- pred new_uniquely_named_var(string::in, var(T)::out,
    varset(T)::in, varset(T)::out) is det.

    % Create a new variable, and maybe give it a name.
    %
:- pred new_maybe_named_var(maybe(string)::in, var(T)::out,
    varset(T)::in, varset(T)::out) is det.

    % Create multiple new variables.
    % Throws an exception if a negative number of new variables
    % is requested.
    %
:- pred new_vars(int::in, list(var(T))::out,
    varset(T)::in, varset(T)::out) is det.

%-----------------------------------------------%

    % Delete the name and value for a variable.
    %
:- func delete_var(varset(T), var(T)) = varset(T).
:- pred delete_var(var(T)::in, varset(T)::in, varset(T)::out) is det.

    % Delete the names and values for a list of variables.
    %
:- func delete_vars(varset(T), list(var(T))) = varset(T).
:- pred delete_vars(list(var(T))::in, varset(T)::in, varset(T)::out)
    is det.

    % Delete the names and values for a sorted list of variables.
    % (If the list is not sorted, the predicate or function will
```

```
    % either throw an exception or return incorrect output.)
    %
:- func delete_sorted_vars(varset(T), list(var(T))) = varset(T).
:- pred delete_sorted_vars(list(var(T))::in,
    varset(T)::in, varset(T)::out) is det.

%---------------------------------------------------%

    % Return a list of all the variables in a varset.
    %
:- func vars(varset(T)) = list(var(T)).
:- pred vars(varset(T)::in, list(var(T))::out) is det.

%---------------------------------------------------%

    % Set the name of a variable.
    %
:- func name_var(varset(T), var(T), string) = varset(T).
:- pred name_var(var(T)::in, string::in,
    varset(T)::in, varset(T)::out) is det.

    % Lookup the name of a variable;
    % If it doesn't have one, create on using V_ as a prefix.
    %
:- func lookup_name(varset(T), var(T)) = string.
:- pred lookup_name(varset(T)::in, var(T)::in, string::out) is det.

    % Lookup the name of a variable;
    % if it doesn't have one, create one using the specified prefix.
    %
:- func lookup_name(varset(T), var(T), string) = string.
:- pred lookup_name(varset(T)::in, var(T)::in, string::in, string::out)
    is det.

    % Lookup the name of a variable;
    % fail if it doesn't have one.
    %
:- pred search_name(varset(T)::in, var(T)::in, string::out) is semidet.

%---------------------------------------------------%

    % Bind a value to a variable.
    % This will overwrite any existing binding.
    %
:- func bind_var(varset(T), var(T), term(T)) = varset(T).
:- pred bind_var(var(T)::in, term(T)::in,
    varset(T)::in, varset(T)::out) is det.
```

```
    % Bind a set of terms to a set of variables.
    %
:- func bind_vars(varset(T), substitution(T)) = varset(T).
:- pred bind_vars(substitution(T)::in,
    varset(T)::in, varset(T)::out) is det.

    % Lookup the value of a variable.
    %
:- pred search_var(varset(T)::in, var(T)::in, term(T)::out) is semidet.

%-----------------------------------------------------%

    % Get the bindings for all the bound variables.
    %
:- func lookup_vars(varset(T)) = substitution(T).
:- pred lookup_vars(varset(T)::in, substitution(T)::out) is det.

    % Get the bindings for all the bound variables.
    %
:- func get_bindings(varset(T)) = substitution(T).
:- pred get_bindings(varset(T)::in, substitution(T)::out) is det.

    % Set the bindings for all the bound variables.
    %
:- func set_bindings(varset(T), substitution(T)) = varset(T).
:- pred set_bindings(varset(T)::in, substitution(T)::in,
    varset(T)::out) is det.

%-----------------------------------------------------%

    % Combine two different varsets, renaming apart:
    % merge_renaming(VarSet0, NewVarSet, VarSet, Renaming) is true
    % iff VarSet is the varset that results from joining a suitably renamed
    % version of NewVarSet to VarSet0. (Any bindings in NewVarSet are ignored.)
    % Renaming will map each variable in NewVarSet to the corresponding
    % fresh variable in VarSet.
    %
:- pred merge_renaming(varset(T)::in, varset(T)::in, varset(T)::out,
    renaming(T)::out) is det.

    % Same as merge_renaming, except that the names of variables
    % in NewVarSet are not included in the final varset.
    % This is useful if create_name_var_map needs to be used
    % on the resulting varset.
    %
:- pred merge_renaming_without_names(varset(T)::in,
```

```
    varset(T)::in, varset(T)::out, renaming(T)::out) is det.

    % Does the same job as merge_renaming, but returns the renaming
    % as a general substitution in which all the terms in the range happen
    % to be variables.
    %
    % Consider using merge_renaming instead.
    %
:- pred merge_subst(varset(T)::in, varset(T)::in, varset(T)::out,
    substitution(T)::out) is det.
:- pragma obsolete(merge_subst/4).

    % Same as merge_subst, except that the names of variables
    % in NewVarSet are not included in the final varset.
    % This is useful if create_name_var_map needs to be used
    % on the resulting varset.
    %
    % Consider using merge_renaming_without_names instead.
    %
:- pred merge_subst_without_names(varset(T)::in,
    varset(T)::in, varset(T)::out, substitution(T)::out) is det.
:- pragma obsolete(merge_subst_without_names/4).

    % merge(VarSet0, NewVarSet, Terms0, VarSet, Terms):
    %
    % As merge_renaming, except instead of returning the renaming,
    % this predicate applies it to the given list of terms.
    %
:- pred merge(varset(T)::in, varset(T)::in, list(term(T))::in,
    varset(T)::out, list(term(T))::out) is det.

    % Same as merge, except that the names of variables
    % in NewVarSet are not included in the final varset.
    % This is useful if create_name_var_map needs to be used
    % on the resulting varset.
    %
:- pred merge_without_names(varset(T)::in, varset(T)::in,
    list(term(T))::in, varset(T)::out, list(term(T))::out) is det.

%-----------------------------------------------------%

    % Create a map from names to variables.
    % Each name is mapped to only one variable, even if a name is
    % shared by more than one variable. Therefore this predicate
    % is only really useful if it is already known that no two
    % variables share the same name.
    %
```

```
:- func create_name_var_map(varset(T)) = map(string, var(T)).
:- pred create_name_var_map(varset(T)::in, map(string, var(T))::out)
    is det.

    % Return an association list giving the name of each variable.
    % Every variable has an entry in the returned association list,
    % even if it shares its name with another variable.
    %
:- func var_name_list(varset(T)) = assoc_list(var(T), string).
:- pred var_name_list(varset(T)::in, assoc_list(var(T), string)::out)
    is det.

    % Given a list of variable and varset in which some variables have
    % no name but some other variables may have the same name,
    % return another varset in which every variable has a unique name.
    % If necessary, names will have suffixes added on the end;
    % the second argument gives the suffix to use.
    %
:- func ensure_unique_names(list(var(T)), string, varset(T))
    = varset(T).
:- pred ensure_unique_names(list(var(T))::in,
    string::in, varset(T)::in, varset(T)::out) is det.

%-----------------------------------------------------%

    % Given a varset and a set of variables, remove the names
    % and values of any other variables stored in the varset.
    %
:- func select(varset(T), set(var(T))) = varset(T).
:- pred select(set(var(T))::in, varset(T)::in, varset(T)::out) is det.

    % Given a varset and a list of variables, construct a new varset
    % containing one variable for each one in the list (and no others).
    % Also return a substitution mapping the selected variables in the
    % original varset into variables in the new varset. The relative
    % ordering of variables in the original varset is maintained.
    %
:- pred squash(varset(T)::in, list(var(T))::in,
    varset(T)::out, renaming(T)::out) is det.

    % Coerce the types of the variables in a varset.
    %
:- func coerce(varset(T)) = varset(U).
:- pred coerce(varset(T)::in, varset(U)::out) is det.

%-----------------------------------------------------%
%-----------------------------------------------------%
```

110 version_array

```
%-----------------------------------------------------%
% vim: ts=4 sw=4 et ft=mercury
%-----------------------------------------------------%
% Copyright (C) 2004-2012 The University of Melbourne.
% Copyright (C) 2014-2020 The Mercury Team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: version_array.m.
% Author: Ralph Becket <rafe@cs.mu.oz.au>.
% Stability: low.
%
% Version types are efficient pure implementations of typically imperative
% structures, subject to the following caveat: efficient access is only
% guaranteed for the "latest" version of a given structure. An older version
% incurs an access cost proportional to the number of its descendants.
%
% For example, if A0 is a version array, and A1 is created by updating A0,
% and A2 is created by updating A1, ..., and An is created by updating An-
1,
% then accesses to An cost O(1) (assuming no further versions of the array
% have been created from An), but accesses to A0 cost O(n).
%
% Updates to older versions of the structure (for example A(n-1)) may have
% additional costs, for arrays this cost is O(m) where m is the size of the
% array, as the whole array is copied to make a new version array.
%
% Most version data structures come with impure, unsafe means to "rewind"
% to an earlier version, restoring that version's O(1) access times, but
% leaving later versions undefined (i.e. only do this if you are discarding
% all later versions of the structure.)
%
% The motivation for using version types is that they are ordinary ground
% structures and do not depend upon uniqueness, while in many circumstances
% offering similar levels of performance.
%
% This module implements version arrays. A version array provides O(1)
% access and update for the "latest" version of the array. "Older"
% versions of the array incur an O(k) penalty on accesses where k is
% the number of updates that have been made since.
%
% The advantage of version arrays is that in the common, singly threaded,
```

```
% case, they are almost as fast as unique arrays, but can be treated as
% ordinary ground values rather than unique values.
%
% Version arrays are zero based.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module version_array.
:- interface.

:- import_module list.
:- import_module pretty_printer.

%-----------------------------------------------------%

:- type version_array(T).

    % An `version_array.index_out_of_bounds' is the exception thrown
    % on out-of-bounds array accesses. The string describes
    % the predicate or function reporting the error.
    %
:- type version_array.index_out_of_bounds
    --->    version_array.index_out_of_bounds(string).

%-----------------------------------------------------%

    % empty_array returns the empty array.
    %
:- func empty = version_array(T).

    % init(N, X) returns an array of size N with each item initialised to X.
    %
:- func init(int, T) = version_array(T).

    % Same as empty/0 except the resulting version_array is not thread safe.
    %
    % That is your program can crash or behave strangely if you attempt to
    % concurrently access or update the array from different threads, or any
    % two arrays produced from operations on the same original array.
    % However this version is much quicker if you guarantee that you never
    % concurrently access the version array.
    %
:- func unsafe_empty = version_array(T).

    % Same as init(N, X) except the resulting version_array is not thread safe.
    %
```

```
    % That is your program can crash or behave strangely if you attempt to
    % concurrently access or update the array from different threads, or any
    % two arrays produced from operations on the same original array.
    % However this version is much quicker if you guarantee that you never
    % concurrently access the version array.
    %
:- func unsafe_init(int, T) = version_array(T).

    % version_array(Xs) returns an array constructed from the items in the list
    % Xs.
    %
:- func version_array(list(T)) = version_array(T).

    % A synonym for the above.
    %
:- func from_list(list(T)) = version_array(T).

    % from_reverse_list(Xs) returns an array constructed from the items in the
    % list Xs in reverse order.
    %
:- func from_reverse_list(list(T)) = version_array(T).

    % lookup(A, I) = X iff the I'th member of A is X.
    % (The first item has index 0).
    %
:- func lookup(version_array(T), int) = T.

    % A ^ elem(I) = lookup(A, I)
    %
:- func version_array(T) ^ elem(int) = T.

    % set(I, X, A0, A): A is a copy of array A0 with item I updated to be X.
    % An exception is thrown if I is out of bounds.
    %
:- pred set(int::in, T::in, version_array(T)::in, version_array(T)::out)
    is det.

    % (A0 ^ elem(I) := X) = A is equivalent to set(I, X, A0, A).
    %
:- func (version_array(T) ^ elem(int) := T) = version_array(T).

    % size(A) = N if A contains N items (i.e. the valid indices for A
    % range from 0 to N - 1).
    %
:- func size(version_array(T)) = int.

    % max(Z) = size(A) - 1.
```

```
    % Returns -1 for an empty array.
    %
:- func max(version_array(T)) = int.

    % is_empty(Array) is true iff Array is the empty array.
    %
:- pred is_empty(version_array(T)::in) is semidet.

    % resize(A, N, X) returns a new array whose items from
    % 0..min(size(A), N - 1) are taken from A and whose items
    % from min(size(A), N - 1)..(N - 1) (if any) are initialised to X.
    % A predicate version is also provided.
    %
:- func resize(version_array(T), int, T) = version_array(T).
:- pred resize(int::in, T::in, version_array(T)::in, version_array(T)::out)
    is det.

    % copy(A) is a copy of array A. Access to the copy is O(1).
    %
:- func copy(version_array(T)) = version_array(T).

    % list(A) = Xs where Xs is the list of items in A
    % (i.e. A = version_array(Xs)).
    %
:- func list(version_array(T)) = list(T).

    % A synonym for the above.
    %
:- func to_list(version_array(T)) = list(T).

    % foldl(F, A, X) is equivalent to list.foldl(F, list(A), X).
    %
:- func foldl(func(T1, T2) = T2, version_array(T1), T2) = T2.

    % foldl(P, A, !X) is equivalent to list.foldl(P, list(A), !X).
    %
:- pred foldl(pred(T1, T2, T2), version_array(T1), T2, T2).
:- mode foldl(pred(in, in, out) is det, in, in, out) is det.
:- mode foldl(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode foldl(pred(in, di, uo) is det, in, di, uo) is det.
:- mode foldl(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode foldl(pred(in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode foldl(pred(in, di, uo) is semidet, in, di, uo) is semidet.

    % foldl2(P, A, !Acc1, !Acc2) is equivalent to
    % list.foldl2(P, list(A), !Acc1, !Acc2) but more efficient.
    %
```

```
:- pred foldl2(pred(T1, T2, T2, T3, T3), version_array(T1), T2, T2, T3, T3).
:- mode foldl2(pred(in, in, out, in, out) is det, in, in, out, in, out)
    is det.
:- mode foldl2(pred(in, in, out, mdi, muo) is det, in, in, out, mdi, muo)
    is det.
:- mode foldl2(pred(in, in, out, di, uo) is det, in, in, out, di, uo)
    is det.
:- mode foldl2(pred(in, in, out, in, out) is semidet, in,
    in, out, in, out) is semidet.
:- mode foldl2(pred(in, in, out, mdi, muo) is semidet, in,
    in, out, mdi, muo) is semidet.
:- mode foldl2(pred(in, in, out, di, uo) is semidet, in,
    in, out, di, uo) is semidet.

    % foldr(F, A, X) is equivalent to list.foldr(F, list(A), Xs).
    %
:- func foldr(func(T1, T2) = T2, version_array(T1), T2) = T2.

:- pred foldr(pred(T1, T2, T2), version_array(T1), T2, T2).
:- mode foldr(pred(in, in, out) is det, in, in, out) is det.
:- mode foldr(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode foldr(pred(in, di, uo) is det, in, di, uo) is det.
:- mode foldr(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode foldr(pred(in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode foldr(pred(in, di, uo) is semidet, in, di, uo) is semidet.

:- pred foldr2(pred(T1, T2, T2, T3, T3), version_array(T1), T2, T2, T3, T3).
:- mode foldr2(pred(in, in, out, in, out) is det, in, in, out, in, out)
    is det.
:- mode foldr2(pred(in, in, out, mdi, muo) is det, in, in, out, mdi, muo)
    is det.
:- mode foldr2(pred(in, in, out, di, uo) is det, in, in, out, di, uo)
    is det.
:- mode foldr2(pred(in, in, out, in, out) is semidet, in,
    in, out, in, out) is semidet.
:- mode foldr2(pred(in, in, out, mdi, muo) is semidet, in,
    in, out, mdi, muo) is semidet.
:- mode foldr2(pred(in, in, out, di, uo) is semidet, in,
    in, out, di, uo) is semidet.

    % version_array.all_true(Pred, Array):
    % True iff Pred is true for every element of Array.
    %
:- pred all_true(pred(T)::in(pred(in) is semidet), version_array(T)::in)
    is semidet.

    % version_array.all_false(Pred, Array):
```

```
    % True iff Pred is false for every element of Array.
    %
:- pred all_false(pred(T)::in(pred(in) is semidet), version_array(T)::in)
    is semidet.

    % unsafe_rewind(A) produces a version of A for which all accesses are O(1).
    % Invoking this predicate renders A and all later versions undefined that
    % were derived by performing individual updates. Only use this when you are
    % absolutely certain there are no live references to A or later versions
    % of A. (A predicate version is also provided.)
    %
:- func unsafe_rewind(version_array(T)) = version_array(T).
:- pred unsafe_rewind(version_array(T)::in, version_array(T)::out) is det.

    % Convert a version_array to a pretty_printer.doc for formatting.
    %
:- func version_array_to_doc(version_array(T)) = pretty_printer.doc.

%-----------------------------------------------------%
%-----------------------------------------------------%

% The first implementation of version arrays used nb_references.
% This incurred three memory allocations for every update. This version
% works at a lower level, but only performs one allocation per update.

%-----------------------------------------------------%
```

111 version_array2d

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 2004-2006, 2011 The University of Melbourne.
% Copyright (C) 2013-2015, 2017-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: version_array2d.m.
% Author: Ralph Becket <rafe@cs.mu.oz.au>.
% Stability: medium-low.
%
% Two-dimensional rectangular (i.e. not ragged) version arrays.
%
% See the header comments in version_array.m for more details about version
% structures.
```

```
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module version_array2d.
:- interface.

:- import_module list.

%-----------------------------------------------------%

    % A version_array2d is a two-dimensional version array stored in row-
major
    % order (that is, the elements of the first row in left-to-right order,
    % followed by the elements of the second row, and so on.)
    %
:- type version_array2d(T).

    % version_array2d([[X11, ..., X1N], ..., [XM1, ..., XMN]]) constructs
    % a 2d version array of size M * N, with the special case that
    % bounds(version_array2d([]), 0, 0).
    %
    % An exception is thrown if the sublists are not all the same length.
    %
:- func version_array2d(list(list(T))) = version_array2d(T).

    % init(M, N, X) = version_array2d([[X11, ..., X1N], ..., [XM1, ..., XMN]])
    % where each XIJ = X.
    %
    % An exception is thrown if M < 0 or N < 0.
    %
:- func init(int, int, T) = version_array2d(T).

    % bounds(version_array2d([[X11, ..., X1N], ..., [XM1, ..., XMN]), M, N)
    %
:- pred bounds(version_array2d(T)::in, int::out, int::out) is det.

    % in_bounds(version_array2d([[X11, ..., X1N], ..., [XM1, ..., XMN]), I, J)
    % succeeds iff 0 =< I < M, 0 =< J < N.
    %
:- pred in_bounds(version_array2d(T)::in, int::in, int::in) is semidet.

    % version_array2d([[X11, ..., X1N], ..., [XM1, ..., XMN]]) ^ elem(I, J) = X
    % where X is the J+1th element of the I+1th row (i.e. indices start from
    % zero.)
    %
    % An exception is thrown unless 0 =< I < M, 0 =< J < N.
```

```
    %
:- func version_array2d(T) ^ elem(int, int) = T.

    % ( VA2D0 ^ elem(I, J) := X ) = VA2D
    % where VA2D ^ elem(II, JJ) = X                    if I = II, J = JJ
    % and   VA2D ^ elem(II, JJ) = VA2D0 ^ elem(II, JJ) otherwise.
    %
    % An exception is thrown unless 0 =< I < M, 0 =< J < N.
    %
    % A predicate version is also provided.
    %
:- func ( version_array2d(T) ^ elem(int, int) := T  ) = version_array2d(T).
:- pred set(int::in, int::in, T::in,
    version_array2d(T)::in, version_array2d(T)::out) is det.

    % lists(version_array2d([[X11, ..., X1N], ..., [XM1, ..., XMN])) =
    %     [[X11, ..., X1N], ..., [XM1, ..., XMN]]
    %
:- func lists(version_array2d(T)) = list(list(T)).

    % copy(VA2D) returns a copy of VA2D with O(1) access times.
    %
:- func copy(version_array2d(T)) = version_array2d(T).

    % resize(VA2D, M, N, X) returns a copy of VA2D resized to M * N.
    % Items with coordinates in common are copied from VA2D; other
    % items are initialised to X.
    %
    % An exception is thrown if M < 0 or N < 0.
    %
:- func resize(version_array2d(T), int, int, T) = version_array2d(T).

    % unsafe_rewind(VA2D) returns a new 2d version array with O(1) access
    % times, at the cost of rendering VA2D and its descendants undefined.
    % Only call this function if you are absolutely certain there are no
    % remaining live references to VA2D or any descendent of VA2D.
    %
:- func unsafe_rewind(version_array2d(T)) = version_array2d(T).

%-----------------------------------------------%
%-----------------------------------------------%
```

112 version_bitmap

```
%-----------------------------------------------%
```

```
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 2004-2007, 2010-2011 The University of Melbourne
% Copyright (C) 2013-2015, 2017-2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: version_bitmap.m.
% Author: Ralph Becket <rafe@cs.mu.oz.au>.
% Stability: low.
%
% (See the header comments in version_array.m for an explanation of version
% types.)
%
% Version bitmaps: an implementation of bitmaps using version arrays.
%
% The advantage of version bitmaps is that in the common, singly threaded,
% case, they are almost as fast as unique bitmaps, but can be treated as
% ordinary ground values rather than unique values.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module version_bitmap.
:- interface.

:- import_module bool.

%-----------------------------------------------------%

:- type version_bitmap.

    % init(N, B) creates a version_bitmap of size N (indexed 0 .. N-1)
    % setting each bit if B = yes and clearing each bit if B = no.
    % An exception is thrown if N is negative.
    %
:- func init(int, bool) = version_bitmap.

    % resize(BM, N, B) resizes version_bitmap BM to have N bits;
    % if N is smaller than the current number of bits in BM, then
    % the excess are discarded. If N is larger than the current number
    % of bits in BM then the new bits are set if B = yes and cleared if
    % B = no.
    %
:- func resize(version_bitmap, int, bool) = version_bitmap.

    % Version of the above suitable for use with state variables.
```

```
    %
:- pred resize(int::in, bool::in, version_bitmap::in, version_bitmap::out)
    is det.

    % Returns the number of bits in a version_bitmap.
    %
:- func num_bits(version_bitmap) = int.

    % Get the given bit.
    %
:- func version_bitmap ^ bit(int) = bool.

    % Set the given bit.
    %
:- func (version_bitmap ^ bit(int) := bool) = version_bitmap.

    % set(BM, I), clear(BM, I) and flip(BM, I) set, clear and flip
    % bit I in BM respectively. An exception is thrown if I is out
    % of range. Predicate versions are also provided.
    %
:- func set(version_bitmap, int) = version_bitmap.
:- pred set(int::in, version_bitmap::in, version_bitmap::out) is det.

:- func clear(version_bitmap, int) = version_bitmap.
:- pred clear(int::in, version_bitmap::in, version_bitmap::out) is det.

:- func flip(version_bitmap, int) = version_bitmap.
:- pred flip(int::in, version_bitmap::in, version_bitmap::out) is det.

    % is_set(BM, I) and is_clear(BM, I) succeed iff bit I in BM
    % is set or clear respectively.
    %
:- pred is_set(version_bitmap::in, int::in) is semidet.
:- pred is_clear(version_bitmap::in, int::in) is semidet.

    % Create a new copy of a version_bitmap.
    %
:- func copy(version_bitmap) = version_bitmap.

    % Set operations; the second argument is altered in all cases.
    %
:- func complement(version_bitmap) = version_bitmap.

:- func union(version_bitmap, version_bitmap) = version_bitmap.

:- func intersect(version_bitmap, version_bitmap) = version_bitmap.
```

```
:- func difference(version_bitmap, version_bitmap) = version_bitmap.

:- func xor(version_bitmap, version_bitmap) = version_bitmap.

    % unsafe_rewind(B) produces a version of B for which all accesses are
    % O(1). Invoking this predicate renders B and all later versions undefined
    % that were derived by performing individual updates. Only use this when
    % you are absolutely certain there are no live references to B or later
    % versions of B.
    %
:- func unsafe_rewind(version_bitmap) = version_bitmap.

    % A version of the above suitable for use with state variables.
    %
:- pred unsafe_rewind(version_bitmap::in, version_bitmap::out) is det.

%-----------------------------------------------------%
%-----------------------------------------------------%
```

113 version_hash_table

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 2004-2006, 2010-2012 The University of Melbourne.
% Copyright (C) 2013-2015, 2017-2020 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: version_hash_table.m.
% Main author: rafe, wangp.
% Stability: low.
%
% (See the header comments in version_array.m for an explanation of version
% types.)
%
% Version hash tables. The "latest" version of the hash table provides
% roughly the same performance as the unique hash table implementation.
% "Older" versions of the hash table are still accessible, but will incur
% a performance penalty that grows as more updates are made to the hash table.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module version_hash_table.
```

```
:- interface.

:- import_module assoc_list.
:- import_module char.

%---------------------------------------------------%

:- type version_hash_table(K, V).

:- type hash_pred(K) == (pred(K,  int)).
:- inst hash_pred    == (pred(in, out) is det).

    % init(HashPred, N, MaxOccupancy)
    % constructs a new hash table with initial size 2 ^ N that is
    % doubled whenever MaxOccupancy is achieved; elements are indexed
    % using HashPred.
    %
    % HashPred must compute a hash for a given key.
    % N must be greater than 0.
    % MaxOccupancy must be in (0.0, 1.0).
    %
    % XXX Values too close to the limits may cause bad things
    % to happen.
    %
:- func init(hash_pred(K)::in(hash_pred), int::in, float::in) =
    (version_hash_table(K, V)::out) is det.

    % unsafe_init(HashPred, N, MaxOccupancy)
    %
    % Like init/3, but the constructed hash table is backed by a
    % non-thread-safe version array. It is unsafe to concurrently access
    % or update the hash table from different threads, or any two hash tables
    % which were produced from operations on the same original hash table.
    % However, if the hash table or its descendants will not be used in such a
    % manner, a non-thread-safe hash table can be much faster than a thread
    % safe one.
    %
:- func unsafe_init(hash_pred(K)::in(hash_pred), int::in, float::in) =
   (version_hash_table(K, V)::out) is det.

    % init_default(HashFn) constructs a hash table with default size and
    % occupancy arguments.
    %
:- func init_default(hash_pred(K)::in(hash_pred)) =
   (version_hash_table(K, V)::out) is det.

    % unsafe_init_default(HashFn)
```

```
    %
    % Like init_default/3 but the constructed hash table is backed by a
    % non-thread-safe version array. See the description of unsafe_init/3
    % above.
    %
:- func unsafe_init_default(hash_pred(K)::in(hash_pred)) =
   (version_hash_table(K, V)::out) is det.

    % Retrieve the hash_pred associated with a hash table.
    %
% :- func hash_pred(version_hash_table(K, V)) = hash_pred(K).

    % Return the number of buckets in a hash table.
    %
:- func num_buckets(version_hash_table(K, V)) = int.

    % Return the number of occupants in a hash table.
    %
:- func num_occupants(version_hash_table(K, V)) = int.

    % Default hash_preds for ints and strings and everything.
    % They are very simple and almost certainly not very good
    % for your purpose, whatever your purpose is.
    %
:- pragma obsolete(int_hash/2, [int.hash/2]).
:- pred int_hash(int::in, int::out) is det.
:- pragma obsolete(uint_hash/2, [uint.hash/2]).
:- pred uint_hash(uint::in, int::out) is det.
:- pragma obsolete(char_hash/2, [char.hash/2]).
:- pred char_hash(char::in, int::out) is det.
:- pragma obsolete(string_hash/2, [string.hash/2]).
:- pred string_hash(string::in, int::out) is det.
:- pragma obsolete(float_hash/2, [float.hash/2]).
:- pred float_hash(float::in, int::out) is det.
:- pragma obsolete(generic_hash/2).
:- pred generic_hash(T::in, int::out) is det.

    % Copy the hash table explicitly.
    %
    % An explicit copy allows programmers to control the cost of copying
    % the table. For more information see the comments at the top of the
    % version_array module.
    %
    % This is not a deep copy: it copies only the structure.
    %
:- func copy(version_hash_table(K, V)) = version_hash_table(K, V).
```

```
    % Search for the value associated with the given key. Fail
    % if there is no entry for the key.
    %
:- func search(version_hash_table(K, V), K) = V is semidet.
:- pred search(version_hash_table(K, V)::in, K::in, V::out) is semidet.

    % Lookup the value associated with the given key. Throw an exception
    % if there is no entry for the key.
    %
:- func lookup(version_hash_table(K, V), K) = V.

    % Field access for hash tables.
    % `HT ^ elem(K)' is equivalent to `lookup(HT, K)'.
    %
:- func version_hash_table(K, V) ^ elem(K) = V.

    % Insert key-value binding into a hash table. If one is already there,
    % then the previous value is overwritten.
    %
:- func set(version_hash_table(K, V), K, V) = version_hash_table(K, V).
:- pred set(K::in, V::in,
    version_hash_table(K, V)::in, version_hash_table(K, V)::out) is det.

    % Field update for hash tables.
    % `HT ^ elem(K) := V' is equivalent to `set(HT, K, V)'.
    %
:- func 'elem :='(K, version_hash_table(K, V), V) = version_hash_table(K, V).

    % Insert a key-value binding into a hash table. An exception is thrown
    % if a binding for the key is already present.
    %
:- func det_insert(version_hash_table(K, V), K, V) = version_hash_table(K, V).
:- pred det_insert(K::in, V::in,
    version_hash_table(K, V)::in, version_hash_table(K, V)::out) is det.

    % Change a key-value binding in a hash table. Throw exception
    % if a binding for the key does not already exist.
    %
:- func det_update(version_hash_table(K, V), K, V) = version_hash_table(K, V).
:- pred det_update(K::in, V::in,
    version_hash_table(K, V)::in, version_hash_table(K, V)::out) is det.

    % Delete the entry for the given key, leaving the hash table
    % unchanged if there is no such entry.
    %
:- func delete(version_hash_table(K, V), K) = version_hash_table(K, V).
:- pred delete(K::in,
```

```
    version_hash_table(K, V)::in, version_hash_table(K, V)::out) is det.

    % Convert a hash table into an association list.
    %
:- func to_assoc_list(version_hash_table(K, V)) = assoc_list(K, V).

    % from_assoc_list(HashPred, N, MaxOccupancy, AssocList) = Table:
    %
    % Convert an association list into a hash table. The first three parameters
    % are the same as for init/3 above.
    %
:- func from_assoc_list(hash_pred(K)::in(hash_pred), int::in, float::in,
        assoc_list(K, V)::in) =
    (version_hash_table(K, V)::out) is det.

    % A simpler version of from_assoc_list/4, the values for N and
    % MaxOccupancy are configured with defaults such as in init_default/1
    %
:- func from_assoc_list(hash_pred(K)::in(hash_pred), assoc_list(K, V)::in) =
    (version_hash_table(K, V)::out) is det.

    % Fold a function over the key-value bindings in a hash table.
    %
:- func fold(func(K, V, T) = T, version_hash_table(K, V), T) = T.

    % Fold a predicate over the key-value bindings in a hash table.
    %
:- pred fold(pred(K, V, T, T), version_hash_table(K, V), T, T).
:- mode fold(in(pred(in, in, in, out) is det), in, in, out) is det.
:- mode fold(in(pred(in, in, mdi, muo) is det), in, mdi, muo) is det.
:- mode fold(in(pred(in, in, di, uo) is det), in, di, uo) is det.
:- mode fold(in(pred(in, in, in, out) is semidet), in, in, out) is semidet.
:- mode fold(in(pred(in, in, mdi, muo) is semidet), in, mdi, muo) is semidet.
:- mode fold(in(pred(in, in, di, uo) is semidet), in, di, uo) is semidet.

%---------------------------------------------------%

    % Test if two version_hash_tables are equal. This predicate is used by
    % unifications on the version_hash_table type.
    %
:- pred equal(version_hash_table(K, V)::in, version_hash_table(K, V)::in)
    is semidet.
% This pragma is required because termination analysis can't analyse the use
% of higher order code.
:- pragma terminates(equal/2).

%---------------------------------------------------%
```

```
%-----------------------------------------------------%
```

114 version_store

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 2004-2006, 2011 The University of Melbourne.
% Copyright (C) 2013-2016, 2018 The Mercury team.
% This file is distributed under the terms specified in COPYING.LIB.
%-----------------------------------------------------%
%
% File: version_store.m.
% Author: Ralph Becket <rafe@cs.mu.oz.au>
% Stability: low.
%
% See the header comments in version_array.m for an explanation of version
% types.
%
% A version_store is similar to, albeit slightly slower than, an ordinary
% store, but does not depend upon uniqueness.
%
% Note that, unlike ordinary stores, liveness of data is via the version store
% rather than the mutvars. This means that dead data (i.e. data whose mutvar
% is out of scope) in a version_store may not be garbage collected.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module version_store.
:- interface.

%-----------------------------------------------------%

:- type version_store(S).

:- type mutvar(T, S).

    % Construct a new version store. This is distinguished from other
    % version stores by its existentially quantified type. This means
    % the compiler can automatically detect any attempt to use a mutvar
    % with the wrong version store.
    %
:- some [S] func init = version_store(S).
```

```
    % new_mutvar(X, Mutvar, VS0, VS) adds a new mutvar with value refer-
ence X
    % to the version store.
    %
:- pred new_mutvar(T::in, mutvar(T, S)::out,
    version_store(S)::in, version_store(S)::out) is det.

    % new_cyclic_mutvar(F, Mutvar, VS0, VS) adds a new mutvar with value
    % reference F(Mutvar) to the version store. This can be used to
    % construct cyclic terms.
    %
:- pred new_cyclic_mutvar((func(mutvar(T, S)) = T)::in, mutvar(T, S)::out,
    version_store(S)::in, version_store(S)::out) is det.

    % copy_mutvar(Mutvar, NewMutvar, VS0, VS) constructs NewMutvar
    % with the same value reference as Mutvar.
    %
:- pred copy_mutvar(mutvar(T, S)::in, mutvar(T, S)::out,
    version_store(S)::in, version_store(S)::out) is det.

    % VS ^ elem(Mutvar) returns the element referenced by Mutvar in
    % the version store.
    %
:- func version_store(S) ^ elem(mutvar(T, S)) = T.

    % lookup(VS, Mutvar) = VS ^ elem(Mutvar).
    %
    % A predicate version is also provided.
    %
:- func lookup(version_store(S), mutvar(T, S)) = T.
:- pred get_mutvar(mutvar(T, S)::in, T::out,
    version_store(S)::in, version_store(S)::out) is det.

    % ( VS ^ elem(Mutvar) := X ) updates the version store so that
    % Mutvar now refers to value X.
    %
:- func ( version_store(S) ^ elem(mutvar(T, S)) := T ) = version_store(S).

    % set(VS, Mutvar, X) = ( VS ^ elem(Mutvar) := X ).
    %
    % A predicate version is also provided.
    %
:- func set(version_store(S), mutvar(T, S), T) = version_store(S).
:- pred set_mutvar(mutvar(T, S)::in, T::in,
    version_store(S)::in, version_store(S)::out) is det.

    % unsafe_rewind(VS) produces a version of VS for which all accesses
```

```
    % are O(1). Invoking this predicate renders undefined VS and all later
    % versions undefined that were derived by performing individual updates.
    % Only use this when you are absolutely certain there are no live
    % references to VS or later versions of VS.
    %
    % A predicate version is also provided.
    %
:- func unsafe_rewind(version_store(T)) = version_store(T).
:- pred unsafe_rewind(version_store(T)::in, version_store(T)::out) is det.

%---------------------------------------------------%
%---------------------------------------------------%
```