The Mercury Library Reference Manual

Version 14.01.1

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.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% 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 '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 array.index_out_of_bounds
    --->    array.index_out_of_bounds(string).

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

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

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

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

:- func array.init(int, T) = array(T).
:- mode array.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 array.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.

    % array.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).
```

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

    % array.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 array.generate_foldl(int, pred(int, T, A, A), array(T), A, A).
:- mode array.generate_foldl(in, in(pred(in, out, in, out) is det),
    array_uo, in, out) is det.
:- mode array.generate_foldl(in, in(pred(in, out, mdi, muo) is det),
    array_uo, mdi, muo) is det.
:- mode array.generate_foldl(in, in(pred(in, out, di, uo) is det),
    array_uo, di, uo) is det.
:- mode array.generate_foldl(in, in(pred(in, out, in, out) is semidet),
    array_uo, in, out) is semidet.
:- mode array.generate_foldl(in, in(pred(in, out, mdi, muo) is semidet),
    array_uo, mdi, muo) is semidet.
:- mode array.generate_foldl(in, in(pred(in, out, di, uo) is semidet),
    array_uo, di, uo) is semidet.

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

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

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

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

    % array.max returns the upper bound of the array.
    %
:- pred array.max(array(_T), int).
%:- mode array.max(array_ui, out) is det.
:- mode array.max(in, out) is det.

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

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

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

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

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

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

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

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

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

    % array.lookup returns the Nth element of an array.
    % Throws an exception if the index is out of bounds.
    %
:- pred array.lookup(array(T), int, T).
%:- mode array.lookup(array_ui, in, out) is det.
:- mode array.lookup(in, in, out) is det.

:- func array.lookup(array(T), int) = T.
```

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

    % array.semidet_lookup returns the Nth element of an array.
    % It fails if the index is out of bounds.
    %
:- pred array.semidet_lookup(array(T), int, T).
%:- mode array.semidet_lookup(array_ui, in, out) is semidet.
:- mode array.semidet_lookup(in, in, out) is semidet.

    % array.unsafe_lookup returns the Nth element of an array.
    % It is an error if the index is out of bounds.
    %
:- pred array.unsafe_lookup(array(T), int, T).
%:- mode array.unsafe_lookup(array_ui, in, out) is det.
:- mode array.unsafe_lookup(in, in, out) is det.

    % array.set sets the nth 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 array.set(int, T, array(T), array(T)).
:- mode array.set(in, in, array_di, array_uo) is det.

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

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

    % array.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 array.unsafe_set(int, T, array(T), array(T)).
:- mode array.unsafe_set(in, in, array_di, array_uo) is det.

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

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

    % array.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 array.semidet_slow_set(int, T, array(T), array(T)).
%:- mode array.semidet_slow_set(in, in, array_ui, array_uo) is semidet.
:- mode array.semidet_slow_set(in, in, in, array_uo) is semidet.

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

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

    % Field update for arrays.
    % (Array ^ elem(Index) := Value) = array.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.

    % Returns every element of the array, one by one.
    %
:- pred array.member(array(T)::in, T::out) is nondet.

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

    % array.copy(Array0, Array):
    % Makes a new unique copy of an array.
    %
```

```
:- pred array.copy(array(T), array(T)).
%:- mode array.copy(array_ui, array_uo) is det.
:- mode array.copy(in, array_uo) is det.

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

    % array.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’.
    %
:- pred array.resize(int, T, array(T), array(T)).
:- mode array.resize(in, in, array_di, array_uo) is det.

    % array.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’.
    %
:- func array.resize(array(T), int, T) = array(T).
:- mode array.resize(array_di, in, in) = array_uo is det.

    % array.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’.
    %
:- pred array.shrink(int, array(T), array(T)).
:- mode array.shrink(in, array_di, array_uo) is det.

    % array.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’.
    %
:- func array.shrink(array(T), int) = array(T).
:- mode array.shrink(array_di, in) = array_uo is det.

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

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

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

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

    % array.fetch_items takes an array and a lower and upper index,
    % and places those items in the array between these indices into a list.
    % It is an error if either index is out of bounds.
    %
:- pred array.fetch_items(array(T), int, int, list(T)).
:- mode array.fetch_items(in, in, in, out) is det.

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

    % XXX We prefer users to call the new array.binary_search predicate
    % instead of array.bsearch, which may be deprecated in later releases.
    %
    % array.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 array.bsearch(array(T), T, comparison_pred(T), maybe(int)).
%:- mode array.bsearch(array_ui, in, in(comparison_pred), out) is det.
:- mode array.bsearch(in, in, in(comparison_pred), out) is det.

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

    % array.approx_binary_search(A, X, I) performs a binary search for an
    % approximate match for X in array A, computing I as the result.  More
    % specifically, if the call succeeds, then either A ^ elem(I) = X or
    % A ^ elem(I) @< X and either X @< A ^ elem(I + 1) or I is the last index
    % in A.
    %
    % array.binary_search(A, X, I) performs a binary search for an
    % exact match for X in array A (i.e., it succeeds iff X = A ^ elem(I)).
    %
```

```
    % A must be sorted into ascending order, but may contain duplicates
    % (the ordering must be with respect to the supplied comparison predicate
    % if one is supplied, otherwise with respect to the Mercury standard
    % ordering).
    %
:- pred array.approx_binary_search(array(T)::array_ui,
    T::in, int::out) is semidet.
:- pred array.approx_binary_search(comparison_func(T)::in, array(T)::array_ui,
    T::in, int::out) is semidet.
:- pred array.binary_search(array(T)::array_ui,
    T::in, int::out) is semidet.
:- pred array.binary_search(comparison_func(T)::in, array(T)::array_ui,
    T::in, int::out) is semidet.

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

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

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

    % array.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 array.sort(array(T)) = array(T).
:- mode array.sort(array_di) = array_uo is det.

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

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

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

    % As above, but with three accumulators.
    %
:- pred array.foldl3(pred(T1, T2, T2, T3, T3, T4, T4), array(T1),
    T2, T2, T3, T3, T4, T4).
:- mode array.foldl3(pred(in, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out) is det.
:- mode array.foldl3(pred(in, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, mdi, muo) is det.
:- mode array.foldl3(pred(in, in, out, in, out, di, uo) is det,
    in, in, out, in, out, di, uo) is det.
:- mode array.foldl3(pred(in, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out) is semidet.
:- mode array.foldl3(pred(in, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, mdi, muo) is semidet.
:- mode array.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 array.foldl4(pred(T1, T2, T2, T3, T3, T4, T4, T5, T5), array(T1),
    T2, T2, T3, T3, T4, T4, T5, T5).
:- mode array.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 array.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 array.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 array.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 array.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 array.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 array.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 array.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 array.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 array.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 array.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 array.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 array.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.

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

```
:- mode array.foldr(func(in, in) = out is det, in, in) = out is det.
%:- mode array.foldr(func(in, di) = uo is det, array_ui, di) = uo is det.
:- mode array.foldr(func(in, di) = uo is det, in, di) = uo is det.

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

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

    % As above, but with three accumulators.
    %
:- pred array.foldr3(pred(T1, T2, T2, T3, T3, T4, T4), array(T1),
    T2, T2, T3, T3, T4, T4).
:- mode array.foldr3(pred(in, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out) is det.
:- mode array.foldr3(pred(in, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, mdi, muo) is det.
:- mode array.foldr3(pred(in, in, out, in, out, di, uo) is det, in,
    in, out, in, out, di, uo) is det.
:- mode array.foldr3(pred(in, in, out, in, out, in, out) is semidet, in,
    in, out, in, out, in, out) is semidet.
:- mode array.foldr3(pred(in, in, out, in, out, mdi, muo) is semidet, in,
    in, out, in, out, mdi, muo) is semidet.
:- mode array.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 array.foldr4(pred(T1, T2, T2, T3, T3, T4, T4, T5, T5), array(T1),
    T2, T2, T3, T3, T4, T4, T5, T5).
:- mode array.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 array.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 array.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 array.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 array.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 array.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 array.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 array.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 array.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 array.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 array.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 array.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 array.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.

    % array.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.

    % array.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.
    %
    % C will have as many elements as A does. In most uses, B will also have
    % this many elements, but may have more; it may NOT have fewer.
    %
:- pred array.map_corresponding_foldl(pred(T1, T2, T3, T4, T4),
    array(T1), array(T2), array(T3), T4, T4).
:- mode array.map_corresponding_foldl(
    in(pred(in, in, out, in, out) is det),
    in, in, array_uo, in, out) is det.
:- mode array.map_corresponding_foldl(
    in(pred(in, in, out, mdi, muo) is det),
    in, in, array_uo, mdi, muo) is det.
:- mode array.map_corresponding_foldl(
    in(pred(in, in, out, di, uo) is det),
    in, in, array_uo, di, uo) is det.
:- mode array.map_corresponding_foldl(
    in(pred(in, in, out, in, out) is semidet),
    in, in, array_uo, in, out) is semidet.

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

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

```
:- mode array.all_false(in(pred(in) is semidet), in) is semidet.

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

    % array.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 array.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.array_to_doc(array(T)) = pretty_printer.doc.
:- mode array.array_to_doc(array_ui) = out is det.

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

2 array2d

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%-----------------------------------------------------%
% Copyright (C) 2003, 2005-2007, 2011-2012 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% 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.

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

    % A 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 ---> 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 a 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.

    % array2d([[X11, ..., X1N], ..., [XM1, ..., XMN]]) ^ elem(I, J) = X
    % where X is the J+1th element of the I+1th 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.

    % 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.
```

```
    % 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.

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

3 assoc_list

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%---------------------------------------------------%
% Copyright (C) 1995-1997, 1999-2001, 2004-2006, 2010-2011 The University of Melbourne
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%---------------------------------------------------%
%
% 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)).

    % Swap the two sides of the pairs in each member of the list.
    %
```

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

    % Zip together two lists; abort if they are of different lengths.
    %
:- func assoc_list.from_corresponding_lists(list(K), list(V))
    = assoc_list(K, V).
:- pred assoc_list.from_corresponding_lists(list(K)::in, list(V)::in,
    assoc_list(K, V)::out) is det.

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

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

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

    % Find the first element of the association list that matches
    % the given key, and return the associated value.
    %
:- pred assoc_list.search(assoc_list(K, V)::in, K::in, V::out) is semidet.

    % An alternative version of assoc_list.search.
    %
:- func assoc_list(K, V) ^ elem(K) = V is semidet.

    % An alternative version of assoc_list.search that throws an
    % exception if the key in question does not appear in the assoc_list.
    %
:- func assoc_list(K, V) ^ det_elem(K) = V is det.

    % 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 assoc_list.remove(assoc_list(K, V)::in, K::in, V::out,
    assoc_list(K, V)::out) is semidet.
```

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

:- func assoc_list.map_keys_only(func(K) = L, assoc_list(K, V))
    = assoc_list(L, V).

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

:- func assoc_list.map_values_only(func(V) = W, assoc_list(K, V))
    = assoc_list(K, W).

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

:- func assoc_list.map_values(func(K, V) = W, assoc_list(K, V))
    = assoc_list(K, W).

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

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

    % assoc_list.filter(Pred, List, TrueList, FalseList) takes a closure with
    % one input argument and for each member K - V of List X, calls the closure
    % on the key. K - V is included in TrueList iff Pred(K) is true.
    % K - V is included in FalseList iff Pred(K) is false.
    %
:- pred assoc_list.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.
```

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

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

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

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

    % As above, but with two accumulators.
    %
:- pred assoc_list.foldl2_values(pred(V, A, A, B, B), assoc_list(K, V),
    A, A, B, B).
:- mode assoc_list.foldl2_values(pred(in, in, out, in, out) is det, in,
    in, out, in, out) is det.
:- mode assoc_list.foldl2_values(pred(in, in, out, mdi, muo) is det, in,
    in, out, mdi, muo) is det.
:- mode assoc_list.foldl2_values(pred(in, in, out, di, uo) is det, in,
    in, out, di, uo) is det.
:- mode assoc_list.foldl2_values(pred(in, in, out, in, out) is semidet, in,
    in, out, in, out) is semidet.
:- mode assoc_list.foldl2_values(pred(in, in, out, mdi, muo) is semidet, in,
    in, out, mdi, muo) is semidet.
:- mode assoc_list.foldl2_values(pred(in, in, out, di, uo) is semidet, in,
    in, out, di, uo) is semidet.
:- mode assoc_list.foldl2_values(pred(in, in, out, in, out) is multi, in,
    in, out, in, out) is multi.
:- mode assoc_list.foldl2_values(pred(in, in, out, in, out) is nondet, in,
    in, out, in, out) is nondet.

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

```
:- mode assoc_list.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 wm=0 tw=0
%---------------------------------------------------%
% Copyright (C) 1994-1999, 2003-2007, 2011 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%---------------------------------------------------%
%
% 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 bag.init = bag(T).
:- pred bag.init(bag(T)::out) is det.

    % Return the number of values in a bag (including duplicate values).
    %
:- func bag.count(bag(T)) = int.
```

```
    % Return the number of unique values in a bag, duplicate values are counted
    % only once.
    %
:- func bag.count_unique(bag(T)) = int.

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

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

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

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

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

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

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

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

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

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

    % Remove one occurrence of a particular value from a bag.
    % Abort if the item does not exist in the bag.
    %
:- func bag.det_remove(bag(T), T) = bag(T).
:- pred bag.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:
    %
    %   bag.remove_list(Bag0, RemoveList, Bag) :-
    %       bag.from_list(RemoveList, RemoveBag),
    %       bag.is_subbag(RemoveBag, Bag0),
```

```
    %       bag.subtract(Bag0, RemoveBag, Bag).
    %
:- pred bag.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.  Abort if any of the items in the list
    % do not exist in the bag.
    %
:- func bag.det_remove_list(bag(T), list(T)) = bag(T).
:- pred bag.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 bag.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.
    % Abort if any of the items in the set do not exist in the bag.
    %
:- func bag.det_remove_set(bag(T), set(T)) = bag(T).
:- pred bag.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 bag.delete(bag(T), T) = bag(T).
:- pred bag.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 bag.remove_all(T::in, bag(T)::in, bag(T)::out) is semidet.

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

    % Check whether a bag contains a particular value.
    %
:- pred bag.contains(bag(T)::in, T::in) is semidet.

    % Count how many occurrences of the value the bag contains.
    %
:- func bag.count_value(bag(T), T) = int.
:- pred bag.count_value(bag(T)::in, T::in, int::out) is det.
```

```
    % bag.subtract(Bag0, SubBag, Bag):
    %
    % Subtracts SubBag from Bag0 to produce Bag. Each element in SubBag is
    % removed from Bag0 to produce Bag. If an element exists in SubBag,
    % but not in Bag, then that element is not removed. An example:
    % bag.subtract({1, 1, 2, 2, 3 }, {1, 1, 2, 3, 3, 3}, {2}).
    %
:- func bag.subtract(bag(T), bag(T)) = bag(T).
:- pred bag.subtract(bag(T)::in, bag(T)::in, bag(T)::out) is det.

    % The third bag is the union of the first 2 bags,
    % e.g. {1, 1, 2, 2} U {2, 2, 3, 3} = {1, 1, 2, 2, 2, 2, 3, 3}.
    % If the two input bags are known to be unequal in size, then making
    % the first bag the larger bag will usually be more efficient.
    %
:- func bag.union(bag(T), bag(T)) = bag(T).
:- pred bag.union(bag(T)::in, bag(T)::in, bag(T)::out) is det.

    % The third bag is the intersection of the first 2 bags. Every element
    % in the third bag exists in both of the first 2 bags, e.g.
    % bag.intersect({1, 2, 2, 3, 3}, {2, 2, 3, 4}, {2, 2, 3}).
    %
:- func bag.intersect(bag(T), bag(T)) = bag(T).
:- pred bag.intersect(bag(T)::in, bag(T)::in, bag(T)::out) is det.

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

    % The third bag is the smallest bag that has both the first two bags
    % as subbags. If an element X is present N times in one of the first
    % two bags, X will be present at least N times in the third bag.
    % E.g. {1, 1, 2} upper_bound {2, 2, 3} = {1, 1, 2, 2, 3}.
    % If the two input bags are known to be unequal in size, then making
    % the first bag the larger bag will usually be more efficient.
    %
:- func bag.least_upper_bound(bag(T), bag(T)) = bag(T).
:- pred bag.least_upper_bound(bag(T)::in, bag(T)::in, bag(T)::out) is det.

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

```
:- pred bag.is_subbag(bag(T)::in, bag(T)::in) is semidet.

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

    % Fails if the bag is empty.
    %
:- pred bag.remove_smallest(T::out, bag(T)::in, bag(T)::out) is semidet.

    % Compares the two bags, and returns whether the first bag is a
    % subset (<), is equal (=), or is a superset (>) of the second.
    % bag.subset_compare(<, {apple, orange}, {apple, apple, orange}).
    % bag.subset_compare(=, {apple, orange}, {apple, orange}).
    % bag.subset_compare(>, {apple, apple, orange}, {apple, orange}).
    % bag.subset_compare(_, {apple, apple}, {orange, orange}) :- fail.
    %
:- pred bag.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 bag.foldl(pred(T, int, A, A), bag(T), A, A).
:- mode bag.foldl(pred(in, in, in, out) is det, in, in, out) is det.
:- mode bag.foldl(pred(in, in, mdi, muo) is det, in, mdi, muo) is det.
:- mode bag.foldl(pred(in, in, di, uo) is det, in, di, uo) is det.
:- mode bag.foldl(pred(in, in, in, out) is semidet, in, in, out) is semidet.
:- mode bag.foldl(pred(in, in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode bag.foldl(pred(in, in, di, uo) is semidet, in, di, uo) is semidet.

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

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

5 benchmarking

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%-----------------------------------------------------%
% Copyright (C) 1994-2011 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% 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 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, !IO) is a procedure intended for use in
    % profiling the memory usage by a program. In 'memprof.gc' grades it has
    % the side-effect of forcing a garbage collection and reporting 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 la-
bel is
    % for your reference.
    %
    % On other grades this procedure does nothing.
    %
:- 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 tw=0 wm=0 ft=mercury
%-----------------------------------------------------%
% Copyright (C) 1994-1995,1997,1999,2004-2006,2008,2011-2012 The Univer-
sity of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% 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 bimap.init = bimap(K, V).
:- pred bimap.init(bimap(K, V)::out) is det.

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

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

    % Search the bimap for the value corresponding to a given key.
    %
```

```
:- func bimap.forward_search(bimap(K, V), K) = V is semidet.
:- pred bimap.forward_search(bimap(K, V)::in, K::in, V::out) is semidet.

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

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

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

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

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

    % bimap.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. The value of V
    % should be guaranteed to be different to all the values already
    % in !.Bimap. If it isn't, this predicate will abort.
    %
:- pred bimap.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 bimap.set(bimap(K, V), K, V) = bimap(K, V).
:- pred bimap.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 bimap.insert_from_assoc_list(assoc_list(K, V), bimap(K, V)) =
    bimap(K, V) is semidet.
:- pred bimap.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 bimap.det_insert_from_assoc_list(assoc_list(K, V), bimap(K, V))
    = bimap(K, V).
:- pred bimap.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 bimap.det_insert_from_corresponding_lists(list(K), list(V),
```

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

    % Apply bimap.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 bimap.set_from_assoc_list(assoc_list(K, V), bimap(K, V))
    = bimap(K, V).
:- pred bimap.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 bimap.set_from_corresponding_lists(list(K), list(V),
    bimap(K, V)) = bimap(K, V).
:- pred bimap.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 bimap.delete_key(bimap(K, V), K) = bimap(K, V).
:- pred bimap.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 bimap.delete_value(bimap(K, V), V) = bimap(K, V).
:- pred bimap.delete_value(V::in, bimap(K, V)::in, bimap(K, V)::out) is det.

    % Apply bimap.delete_key to a list of keys.
    %
:- func bimap.delete_keys(bimap(K, V), list(K)) = bimap(K, V).
:- pred bimap.delete_keys(list(K)::in, bimap(K, V)::in, bimap(K, V)::out)
    is det.

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

    % bimap.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 bimap.overlay(bimap(K, V), bimap(K, V)) = bimap(K, V).
:- pred bimap.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 bimap.count(bimap(K, V)) = int.

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

:- func bimap.apply_forward_map_to_list(bimap(K, V), list(K)) = list(V).
```

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

:- func bimap.apply_reverse_map_to_list(bimap(K, V), list(V)) = list(K).
:- pred bimap.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 bimap.map_keys(func(V, K) = L, bimap(K, V)) = bimap(L, V).
:- pred bimap.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 bimap.map_values(func(K, V) = W, bimap(K, V)) = bimap(K, W).
:- pred bimap.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 bimap.foldl(func(K, V, A) = A, bimap(K, V), A) = A.
:- pred bimap.foldl(pred(K, V, A, A), bimap(K, V), A, A).
:- mode bimap.foldl(pred(in, in, in, out) is det, in, in, out) is det.
:- mode bimap.foldl(pred(in, in, mdi, muo) is det, in, mdi, muo) is det.
:- mode bimap.foldl(pred(in, in, di, uo) is det, in, di, uo) is det.
:- mode bimap.foldl(pred(in, in, in, out) is semidet, in, in, out) is semidet.
:- mode bimap.foldl(pred(in, in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode bimap.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 bimap.foldl, this is often a more convenient format,
    % and a little more efficient).
    %
:- pred bimap.foldl2(pred(K, V, A, A, B, B), bimap(K, V), A, A, B, B).
:- mode bimap.foldl2(pred(in, in, in, out, in, out) is det,
    in, in, out, in, out) is det.
:- mode bimap.foldl2(pred(in, in, in, out, mdi, muo) is det,
    in, in, out, mdi, muo) is det.
:- mode bimap.foldl2(pred(in, in, in, out, di, uo) is det,
    in, in, out, di, uo) is det.
:- mode bimap.foldl2(pred(in, in, di, uo, di, uo) is det,
    in, di, uo, di, uo) is det.
```

```
:- mode bimap.foldl2(pred(in, in, in, out, in, out) is semidet,
    in, in, out, in, out) is semidet.
:- mode bimap.foldl2(pred(in, in, in, out, mdi, muo) is semidet,
    in, in, out, mdi, muo) is semidet.
:- mode bimap.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 bimap.foldl, this is often a more convenient format,
    % and a little more efficient).
    %
:- pred bimap.foldl3(pred(K, V, A, A, B, B, C, C), bimap(K, V),
    A, A, B, B, C, C).
:- mode bimap.foldl3(pred(in, in, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out) is det.
:- mode bimap.foldl3(pred(in, in, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, mdi, muo) is det.
:- mode bimap.foldl3(pred(in, in, in, out, in, out, di, uo) is det,
    in, in, out, in, out, di, uo) is det.
:- mode bimap.foldl3(pred(in, in, in, out, di, uo, di, uo) is det,
    in, in, out, di, uo, di, uo) is det.
:- mode bimap.foldl3(pred(in, in, di, uo, di, uo, di, uo) is det,
    in, di, uo, di, uo, di, uo) is det.
:- mode bimap.foldl3(pred(in, in, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out) is semidet.
:- mode bimap.foldl3(pred(in, in, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, mdi, muo) is semidet.
:- mode bimap.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 bimap.forward_map(bimap(K, V)) = map(K, V).

    % Extract the reverse map from the bimap, the map from value to key.
    %
:- func bimap.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
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
% 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 tw=0 wm=0 ft=mercury
%-----------------------------------------------------%
% Copyright (C) 2007, 2010-2011 The University of Melbourne
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
% 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 tw=0 wm=0 ft=mercury
%---------------------------------------------------%
% Copyright (C) 2007, 2011 The University of Melbourne
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%---------------------------------------------------%
% 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 tw=0 wm=0 ft=mercury
%-----------------------------------------------------%
% Copyright (C) 2001-2002, 2004-2007, 2009-2011 The University of Melbourne
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% 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 are 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.

    % 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.

    % 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.

    % Returns 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.
```

```
    % Returns 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.

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

    %
    % 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 the integer.
    % 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.

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

    % Slice = bitmap.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 bitmap.slice.
:- func bitmap.slice(bitmap, bit_index, num_bits) = bitmap.slice.

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

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

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

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

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

    %
    % 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.
    %
:- func from_string(string) = bitmap.
:- mode from_string(in) = bitmap_uo is semidet.

    % 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.

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

    %
    % 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.

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

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

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

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

:- pred unsafe_flip(bit_index, bitmap, bitmap).
:- mode unsafe_flip(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.

    %
    % Predicate versions, for use with state variables.
    %
```

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

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

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

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

11 bool

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%-----------------------------------------------------%
% Copyright (C) 1996-1997,2000,2002-2007,2009-2010 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% 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).

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

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

    % not(A) = yes iff A = no.
    %
:- func bool.not(bool) = bool.
:- pred bool.not(bool::in, bool::out) is det.

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

    % 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 wm=0 tw=0
%-----------------------------------------------------%
% Copyright (C) 1997, 1999-2000, 2002-2003, 2005-2006 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% 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 bt_array.resize and bt_array.shrink for more details.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

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

:- type bt_array(T).

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

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

    % bt_array.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 bt_array.init(int::in, int::in, T::in, bt_array(T)::out) is det.
:- func bt_array.init(int, int, T) = bt_array(T).

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

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

    % array.max returns the upper bound of the array.
    %
:- pred bt_array.max(bt_array(_T)::in, int::out) is det.
:- func bt_array.max(bt_array(_T)) = int.

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

    % bt_array.bounds returns the upper and lower bounds of a bt_array.
    %
:- pred bt_array.bounds(bt_array(_T)::in, int::out, int::out) is det.

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

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

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

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

    % bt_array.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 bt_array.set(bt_array(T)::in, int::in, T::in, bt_array(T)::out)
    is det.
:- func bt_array.set(bt_array(T), int, T) = bt_array(T).

    % bt_array.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 bt_array.semidet_set(bt_array(T)::in, int::in, T::in,
    bt_array(T)::out) is semidet.

    % 'bt_array.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 bt_array.resize(bt_array(T)::in, int::in, int::in, T::in,
    bt_array(T)::out) is det.
:- func bt_array.resize(bt_array(T), int, int, T) = bt_array(T).

    % bt_array.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 bt_array.shrink(bt_array(T)::in, int::in, int::in, bt_array(T)::out)
    is det.
:- func bt_array.shrink(bt_array(T), int, int) = bt_array(T).

    % bt_array.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 bt_array.from_list(int::in, list(T)::in, bt_array(T)::out) is det.
:- func bt_array.from_list(int, list(T)) = bt_array(T).
```

```
    % bt_array.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 bt_array.to_list(bt_array(T)::in, list(T)::out) is det.
:- func bt_array.to_list(bt_array(T)) = list(T).

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

    % bt_array.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 bt_array.bsearch(bt_array(T)::in, T::in,
    comparison_pred(T)::in(comparison_pred), int::out) is semidet.

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

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

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

13 builtin

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%---------------------------------------------------%
% Copyright (C) 1994-2007, 2010-2012 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
```

```
%-----------------------------------------------------%
%
% 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', `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.  (XXX TODO: report an error for attempts to redefine
% these types.)

    % 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 functions insts ‘func(<modes>) = <mode> is det’
% are also builtin.

    % 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.

    % The ‘any’ inst used for the constraint solver interface is also
    % builtin.  The insts ‘new’ and ‘old’ are allowed as synonyms for
    % ‘free’ and ‘any’, respectively, since some of the literature uses
    % this terminology.
    %
:- inst old == any.
:- inst new == free.

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

:- mode unused == free >> free.
:- mode output == free >> ground.
:- 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.

    % The modes ‘no’ and ‘oo’ are allowed as synonyms, since some of the
    % literature uses this terminology.
    %
:- mode no == new >> old.
:- mode oo == old >> old.

%---------------------------------------------------%
%
% 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: we recommend using the a 'promise_equivalent_solutions' goal
    %       instead of this function.
    %
:- 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: we recommend using a 'promise_equivalent_solutions' goal
    %       instead of this predicate.
    %
:- 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).
    % Note to implementors: the modes must appear in this order:
    % compiler/higher_order.m depends on it, as does
    % compiler/simplify.m (for the inequality simplification.)
:- 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 wm=0 tw=0
%-----------------------------------------------------%
% Copyright (C) 2009-2010 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% 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.

    % 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.

    % 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 aborts 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 aborts 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.

    % 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 aborts 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 wm=0 tw=0
%---------------------------------------------------%
% Copyright (C) 1994-2008, 2011 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%---------------------------------------------------%
%
% File: char.m.
% Main author: fjh.
% Stability: high.
%
% This module defines some predicates that manipulate characters.
%
% The set of characters which are supported and the mapping from
% characters to integer values are both implementation-dependent.
%
% 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.
%
%---------------------------------------------------%
%---------------------------------------------------%

:- module char.
:- interface.

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

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

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

:- instance enum(character).

    % ‘char.to_int’/1 and ‘char.to_int(in, out)’ convert a character to its
    % corresponding numerical code (integer value).
    %
```

```
    % ‘char.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 ‘char.det_from_int(0)’.
    % Null characters are not allowed in Mercury strings in C grades.
    %
:- func char.to_int(char) = int.
:- pred char.to_int(char, int).
:- mode char.to_int(in, out) is det.
:- mode char.to_int(in, in) is semidet.    % implied
:- mode char.to_int(out, in) is semidet.

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

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

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

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

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

```
:- pred char.to_upper(char::in, char::out) is det.

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

    % char.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 char.lower_upper(char, char).
:- mode char.lower_upper(in, out) is semidet.
:- mode char.lower_upper(out, in) is semidet.

    % True iff the character is a whitespace character in the ASCII range,
    % i.e. a space, tab, newline, carriage return, form-feed, or verti-
cal tab.
    %
:- pred char.is_whitespace(char::in) is semidet.

    % True iff the character is an uppercase letter (A-Z) in the ASCII range.
    %
:- pred char.is_upper(char::in) is semidet.

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

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

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

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

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

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

:- pred char.is_hex_digit(char, int).
:- mode char.is_hex_digit(in, out) is semidet.

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

    % Succeeds if char 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).
    %
:- pred char.digit_to_int(char::in, int::out) is semidet.

    % char.int_to_uppercase_digit(Int, DigitChar):
    %
    % True iff `Int' is an integer in the range 0-35 and
    % `DigitChar' is a decimal digit or uppercase letter
    % whose value as a digit is `Int'.
    %
:- pred char.int_to_digit(int, char).
:- mode char.int_to_digit(in, out) is semidet.
:- mode char.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.
    %
:- func char.det_int_to_digit(int) = char.
```

```
:- pred char.det_int_to_digit(int::in, char::out) is det.

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

    % Encode a Unicode code point in UTF-8.
    % Fails for surrogate code points.
    %
:- pred char.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 char.to_utf16(char::in, list(int)::out) is semidet.

    % Succeed if 'Char' is a Unicode surrogate code point.
    % In UTF-16, a code point with a scalar value greater than 0xffff
    % is encoded with a pair of surrogate code points.
    %
:- pred char.is_surrogate(char::in) is semidet.

    % Succeed if 'Char' is a 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 char.is_noncharacter(char::in) is semidet.

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

16 construct

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 2002-2009, 2011 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% 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.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%---------------------------------------------------%
%
% 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).

    % 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).

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

    % 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).

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

    % 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).

    % Append together a list of cords, and return the result as a list.
    %
:- func 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,
    % - 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,
    % - Pred(X) is true, then X is included in TrueCord.
    % - 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.

    % 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 wm=0 tw=0
%---------------------------------------------------%
% Copyright (C) 2000, 2005-2006, 2011 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%---------------------------------------------------%
%
% 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.

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

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

    % 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 counter.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.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%---------------------------------------------------%
%
% 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 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 ---> do_not_allow.
:- inst canonicalize ---> canonicalize.
:- inst include_details_cc ---> include_details_cc.
:- inst canonicalize_or_do_not_allow
    --->    do_not_allow
    ;       canonicalize.
:- inst do_not_allow_or_include_details_cc
    --->    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 floats, the string is a floating point, base 10 number;
%     positive floating point numbers have no sign.
%   - for strings, the string, inside double quotation marks
%   - for characters, the character inside single quotation marks
%   - 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 ptr(0xXXXX) where XXXX is the
%     hexadecimal representation of the pointer.
%   - for bitmaps, the bitmap converted to a 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 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.
    % 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.
    % Don't 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 abort.
    %
:- 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 abort.
    %
:- 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 digraph

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 1995-1999,2002-2007,2010-2012 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%---------------------------------------------------%
%
% 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)).

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

    % digraph.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 digraph.add_vertex(T::in, digraph_key(T)::out,
    digraph(T)::in, digraph(T)::out) is det.

    % digraph.search_key returns the key associated with a vertex.
    % Fails if the vertex is not in the graph.
    %
:- pred digraph.search_key(digraph(T)::in, T::in, digraph_key(T)::out)
    is semidet.

    % digraph.lookup_key returns the key associated with a vertex.
    % Aborts if the vertex is not in the graph.
    %
:- func digraph.lookup_key(digraph(T), T) = digraph_key(T).
:- pred digraph.lookup_key(digraph(T)::in, T::in, digraph_key(T)::out)
    is det.

    % digraph.lookup_vertex returns the vertex associated with a key.
    %
:- func digraph.lookup_vertex(digraph(T), digraph_key(T)) = T.
```

```
:- pred digraph.lookup_vertex(digraph(T)::in, digraph_key(T)::in, T::out)
    is det.

    % digraph.add_edge adds an edge to the digraph if it doesn't already
    % exist, and leaves the digraph unchanged otherwise.
    %
:- func digraph.add_edge(digraph_key(T), digraph_key(T), digraph(T)) =
    digraph(T).
:- pred digraph.add_edge(digraph_key(T)::in, digraph_key(T)::in,
    digraph(T)::in, digraph(T)::out) is det.

    % digraph.add_vertices_and_edge adds a pair of vertices and an edge
    % between them to the digraph.
    %
    % digraph.add_vertices_and_edge(X, Y, !G) :-
    %    digraph.add_vertex(X, XKey, !G),
    %    digraph.add_vertex(Y, YKey, !G),
    %    digraph.add_edge(XKey, YKey, !G).
    %
:- func digraph.add_vertices_and_edge(T, T, digraph(T)) = digraph(T).
:- pred digraph.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 digraph.add_vertex_pair(pair(T), digraph(T)) = digraph(T).
:- pred digraph.add_vertex_pair(pair(T)::in,
    digraph(T)::in, digraph(T)::out) is det.

    % digraph.add_assoc_list adds a list of edges to a digraph.
    %
:- func digraph.add_assoc_list(assoc_list(digraph_key(T), digraph_key(T)),
    digraph(T)) = digraph(T).
:- pred digraph.add_assoc_list(assoc_list(digraph_key(T), digraph_key(T))::in,
    digraph(T)::in, digraph(T)::out) is det.

    % digraph.delete_edge deletes an edge from the digraph if it exists,
    % and leaves the digraph unchanged otherwise.
    %
:- func digraph.delete_edge(digraph_key(T), digraph_key(T), digraph(T)) =
    digraph(T).
:- pred digraph.delete_edge(digraph_key(T)::in, digraph_key(T)::in,
    digraph(T)::in, digraph(T)::out) is det.

    % digraph.delete_assoc_list deletes a list of edges from a digraph.
    %
:- func digraph.delete_assoc_list(assoc_list(digraph_key(T), digraph_key(T)),
```

```
    digraph(T)) = digraph(T).
:- pred digraph.delete_assoc_list(
    assoc_list(digraph_key(T), digraph_key(T))::in,
    digraph(T)::in, digraph(T)::out) is det.

    % digraph.is_edge checks to see if an edge is in the digraph.
    %
:- pred digraph.is_edge(digraph(T), digraph_key(T), digraph_key(T)).
:- mode digraph.is_edge(in, in, out) is nondet.
:- mode digraph.is_edge(in, in, in) is semidet.

    % digraph.is_edge_rev is equivalent to digraph.is_edge, except that
    % the nondet mode works in the reverse direction.
    %
:- pred digraph.is_edge_rev(digraph(T), digraph_key(T), digraph_key(T)).
:- mode digraph.is_edge_rev(in, out, in) is nondet.
:- mode digraph.is_edge_rev(in, in, in) is semidet.

    % Given key x, digraph.lookup_from returns the set of keys y such that
    % there is an edge (x,y) in the digraph.
    %
:- func digraph.lookup_from(digraph(T), digraph_key(T)) = set(digraph_key(T)).
:- pred digraph.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 digraph.lookup_key_set_from(digraph(T), digraph_key(T)) =
    digraph_key_set(T).
:- pred digraph.lookup_key_set_from(digraph(T)::in, digraph_key(T)::in,
    digraph_key_set(T)::out) is det.

    % Given a key y, digraph.lookup_to returns the set of keys x such that
    % there is an edge (x,y) in the digraph.
    %
:- func digraph.lookup_to(digraph(T), digraph_key(T)) = set(digraph_key(T)).
:- pred digraph.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 digraph.lookup_key_set_to(digraph(T), digraph_key(T)) =
    digraph_key_set(T).
:- pred digraph.lookup_key_set_to(digraph(T)::in, digraph_key(T)::in,
    digraph_key_set(T)::out) is det.

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

```
    % digraph.to_assoc_list turns a digraph into a list of pairs of vertices,
    % one for each edge.
    %
:- func digraph.to_assoc_list(digraph(T)) = assoc_list(T, T).
:- pred digraph.to_assoc_list(digraph(T)::in, assoc_list(T, T)::out) is det.

    % digraph.to_key_assoc_list turns a digraph into a list of pairs of keys,
    % one for each edge.
    %
:- func digraph.to_key_assoc_list(digraph(T)) =
    assoc_list(digraph_key(T), digraph_key(T)).
:- pred digraph.to_key_assoc_list(digraph(T)::in,
    assoc_list(digraph_key(T), digraph_key(T))::out) is det.

    % digraph.from_assoc_list turns a list of pairs of vertices into a digraph.
    %
:- func digraph.from_assoc_list(assoc_list(T, T)) = digraph(T).
:- pred digraph.from_assoc_list(assoc_list(T, T)::in, digraph(T)::out) is det.

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

    % digraph.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 digraph.dfs(digraph(T), digraph_key(T)) = list(digraph_key(T)).
:- pred digraph.dfs(digraph(T)::in, digraph_key(T)::in,
    list(digraph_key(T))::out) is det.

    % digraph.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 digraph.dfsrev(digraph(T), digraph_key(T)) = list(digraph_key(T)).
:- pred digraph.dfsrev(digraph(T)::in, digraph_key(T)::in,
    list(digraph_key(T))::out) is det.

    % digraph.dfs(G, Dfs) is true if Dfs is a depth-first sorting of G,
    % i.e. a list of all the keys in G such that all keys for 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 digraph.dfs(digraph(T)) = list(digraph_key(T)).
:- pred digraph.dfs(digraph(T)::in, list(digraph_key(T))::out) is det.
```

```
    % digraph.dfsrev(G, DfsRev) is true if DfsRev is a reverse depth-first
    % sorting of G. That is, DfsRev is the reverse of Dfs from digraph.dfs/2.
    %
:- func digraph.dfsrev(digraph(T)) = list(digraph_key(T)).
:- pred digraph.dfsrev(digraph(T)::in, list(digraph_key(T))::out) is det.

    % digraph.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 digraph.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.

    % digraph.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 digraph.dfs/5.
    % !:Visit is !.Visit + DfsRev.
    %
:- pred digraph.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.

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

    % digraph.vertices returns the set of vertices in a digraph.
    %
:- func digraph.vertices(digraph(T)) = set(T).
:- pred digraph.vertices(digraph(T)::in, set(T)::out) is det.

    % digraph.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 digraph.inverse(digraph(T)) = digraph(T).
:- pred digraph.inverse(digraph(T)::in, digraph(T)::out) is det.

    % digraph.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 digraph.compose(digraph(T), digraph(T)) = digraph(T).
:- pred digraph.compose(digraph(T)::in, digraph(T)::in, digraph(T)::out)
    is det.
```

```
    % digraph.is_dag(G) is true iff G is a directed acyclic graph.
    %
:- pred digraph.is_dag(digraph(T)::in) is semidet.

    % digraph.components(G, Comp) is true if Comp is the set of the
    % connected components of G.
    %
:- func digraph.components(digraph(T)) = set(set(digraph_key(T))).
:- pred digraph.components(digraph(T)::in, set(set(digraph_key(T)))::out)
    is det.

    % digraph.cliques(G, Cliques) is true if Cliques is the set of the
    % cliques (strongly connected components) of G.
    %
:- func digraph.cliques(digraph(T)) = set(set(digraph_key(T))).
:- pred digraph.cliques(digraph(T)::in, set(set(digraph_key(T)))::out) is det.

    % digraph.reduced(G, R) is true if R is the reduced digraph (digraph of
    % cliques) obtained from G.
    %
:- func digraph.reduced(digraph(T)) = digraph(set(T)).
:- pred digraph.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 digraph.reduced(digraph(T)::in, digraph(set(T))::out,
    map(digraph_key(T), digraph_key(set(T)))::out) is det.

    % digraph.tsort(G, TS) is true if TS is a topological sorting of G.
    % It fails if G is cyclic.
    %
:- pred digraph.tsort(digraph(T)::in, list(T)::out) is semidet.

    % digraph.atsort(G, ATS) is true if ATS is a topological sorting
    % of the cliques in G.
    %
:- func digraph.atsort(digraph(T)) = list(set(T)).
:- pred digraph.atsort(digraph(T)::in, list(set(T))::out) is det.

    % digraph.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 digraph.sc(digraph(T)) = digraph(T).
:- pred digraph.sc(digraph(T)::in, digraph(T)::out) is det.

    % digraph.tc(G, TC) is true if TC is the transitive closure of G.
```

```
    %
:- func digraph.tc(digraph(T)) = digraph(T).
:- pred digraph.tc(digraph(T)::in, digraph(T)::out) is det.

    % digraph.rtc(G, RTC) is true if RTC is the reflexive transitive closure
    % of G.
    %
:- func digraph.rtc(digraph(T)) = digraph(T).
:- pred digraph.rtc(digraph(T)::in, digraph(T)::out) is det.

    % digraph.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 digraph.traverse(digraph(T), pred(T, A, A), pred(T, T, A, A), A, A).
:- mode digraph.traverse(in, pred(in, di, uo) is det,
    pred(in, in, di, uo) is det, di, uo) is det.
:- mode digraph.traverse(in, pred(in, in, out) is det,
    pred(in, in, in, out) is det, in, out) is det.

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

21 dir

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 1994-1995,1997,1999-2000,2002-2012 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%---------------------------------------------------%
%
% 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 dir.directory_separator = character.
:- pred dir.directory_separator(character::out) is det.

    % Is the character a directory separator.
    % On Microsoft Windows systems this will succeed for '/'
    % as well as '\\'.
    %
:- pred dir.is_directory_separator(character).
:- mode dir.is_directory_separator(in) is semidet.
:- mode dir.is_directory_separator(out) is multi.

    % Returns ".".
    %
:- func dir.this_directory = string.
:- pred dir.this_directory(string::out) is det.

    % Returns "..".
    %
:- func dir.parent_directory = string.
:- pred dir.parent_directory(string::out) is det.

    % dir.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 `dir.split_name("C:foo", "C:", "foo")'.
    % (`X:' is the current directory on drive `X').
    % Note that Cygwin doesn't support drive current directories,
    % so `dir.split_name("C:foo, _, _)' will fail when running under Cygwin.
    %
:- pred dir.split_name(string::in, string::out, string::out) is semidet.

    % dir.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 dir.basename(string) = string is semidet.
:- pred dir.basename(string::in, string::out) is semidet.

    % As above, but throws an exception instead of failing.
    %
:- func dir.det_basename(string) = string.

    % dir.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 `dir.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 dir.dirname(string) = string.
:- pred dir.dirname(string::in, string::out) is det.

    % dir.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 dir.path_name_is_root_directory).
    %
:- pred dir.path_name_is_absolute(string::in) is semidet.

    % dir.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 dir.path_name_is_root_directory(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 dir.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 dir.relative_path_name_from_components(list(string)) = string.

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

    % dir.current_directory(Result)
    % Return the current working directory.
    %
:- pred dir.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 dir.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 dir.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 dir.foldl2 to process each entry in a directory.
    % Processing will stop if Continue is bound to 'no'.
    %
:- type dir.foldl_pred(T) ==
    pred(string, string, io.file_type, bool, T, T, io, io).
:- inst dir.foldl_pred == (pred(in, in, in, out, in, out, di, uo) is det).

    % dir.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 dir.foldl2(dir.foldl_pred(T)::in(dir.foldl_pred), string::in,
    T::in, io.maybe_partial_res(T)::out, io::di, io::uo) is det.

    % dir.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 dir.recursive_foldl2(dir.foldl_pred(T)::in(dir.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).

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

22 enum

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

```
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%-----------------------------------------------------%
% Copyright (C) 2000, 2005-2006 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% 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.

% For all instances the following must hold:
% all [X] (X = from_int(to_int(X)))
% all [Int] (some [Y] Int = to_int(Y) => from_int(Int) = Y)
:- typeclass enum(T) where [
func to_int(T) = int,
func from_int(int) = T is semidet
].

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

23 eqvclass

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 1995-1997, 1999, 2003-2006, 2011-2012 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% 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 equivalance class.
    %
:- func eqvclass.init = eqvclass(T).
:- pred eqvclass.init(eqvclass(T)::out) is det.

    % Is this item known to the equivalence class?
    %
:- pred eqvclass.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 eqvclass.same_eqvclass is more expressive than calling
    % eqvclass.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 eqvclass.partition_id on X just once and
    % comparing this with the partition_ids of the Yi will be more efficient.
    %
:- pred eqvclass.partition_id(eqvclass(T)::in, T::in, partition_id::out)
    is semidet.

    % 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 eqvclass.partition_id.
    %
:- pred eqvclass.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 may already be known to the class;
    % if it isn't, it is created without any equivalence relationships.
    %
:- func eqvclass.ensure_element(eqvclass(T), T) = eqvclass(T).
:- pred eqvclass.ensure_element(T::in, 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 eqvclass.new_element(eqvclass(T), T) = eqvclass(T).
:- pred eqvclass.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 eqvclass.ensure_equivalence(eqvclass(T), T, T) = eqvclass(T).
:- pred eqvclass.ensure_equivalence(T::in, T::in,
    eqvclass(T)::in, eqvclass(T)::out) is det.

:- func eqvclass.ensure_corresponding_equivalences(list(T), list(T),
    eqvclass(T)) = eqvclass(T).
:- pred eqvclass.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 eqvclass.new_equivalence(eqvclass(T), T, T) = eqvclass(T).
:- pred eqvclass.new_equivalence(T::in, T::in,
    eqvclass(T)::in, eqvclass(T)::out) is det.

    % Test if two elements are equivalent.
    %
:- pred eqvclass.same_eqvclass(eqvclass(T)::in, T::in, T::in) is semidet.

    % Test if a list of elements are equivalent.
    %
:- pred eqvclass.same_eqvclass_list(eqvclass(T)::in, list(T)::in) is semidet.

    % Return the set of the partitions of the equivalence class.
    %
:- func eqvclass.partition_set(eqvclass(T)) = set(set(T)).
:- pred eqvclass.partition_set(eqvclass(T)::in, set(set(T))::out) is det.
```

```
    % Return a list of the partitions of the equivalence class.
    %
:- func eqvclass.partition_list(eqvclass(T)) = list(set(T)).
:- pred eqvclass.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 eqvclass.partition_set_to_eqvclass(set(set(T))) = eqvclass(T).
:- pred eqvclass.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 eqvclass.partition_list_to_eqvclass(list(set(T))) = eqvclass(T).
:- pred eqvclass.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 eqvclass.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 eqvclass.get_minimum_element(eqvclass(T), T) = T.

    % Remove the given element and all other elements equivalent to it
    % from the given equivalence class.
    %
:- func eqvclass.remove_equivalent_elements(eqvclass(T), T) = eqvclass(T).
:- pred eqvclass.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 eqvclass.divide_equivalence_classes(func(T) = U, eqvclass(T))
    = eqvclass(T).

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

24 erlang_builtin

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%-----------------------------------------------------%
% Copyright (C) 2007, 2011 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% File: erlang_builtin.m.
% Main author: wangp.
% Stability: low.
%
% This file is intended to hold things related to Erlang for the Erlang grade.
% In non-Erlang grades this file should do nothing.
%
% Currently it contains a server that is started at program initialisation
% to emulate global variables.  Lookups and updates of global mutables work by
% sending and receiving messages to this server.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module erlang_builtin.
:- interface.

% This module exports nothing yet for public consumption; all exports
% are via foreign_export.

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

25 exception

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%-----------------------------------------------------%
% Copyright (C) 1997-2008, 2010-2011 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
```

```
% 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).

    % throw(Exception):
    %   Throw the specified exception.
    %
:- func throw(T) = _ is erroneous.
:- pred throw(T::in) 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
    --->    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 MaybeException = yes(E), otherwise
    %   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 [] ---> [].
:- inst nil_or_singleton_list ---> [] ; [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.

    % 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.

    % finally(P, PRes, Cleanup, CleanupRes, IO0, 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.

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

26 fat_sparse_bitset

```
%-----------------------------------------------------%
% vim: ts=4 sw=4 et ft=mercury
%-----------------------------------------------------%
% Copyright (C) 2011-2012 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% 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).

    % Return an empty set.
    %
:- func init = fat_sparse_bitset(T).
:- pred init(fat_sparse_bitset(T)::out) is det.

:- pred empty(fat_sparse_bitset(T)).
:- mode empty(in) is semidet.
:- mode empty(out) is det.

:- pred is_empty(fat_sparse_bitset(T)::in) is semidet.

:- pred is_non_empty(fat_sparse_bitset(T)::in) is semidet.

    % '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.

    % '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).

    % '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 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).
```

```
    % '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).

    % 'make_singleton_set(Elem)' returns a set containing just the single
    % element 'Elem'.
    %
:- func make_singleton_set(T) = fat_sparse_bitset(T) <= enum(T).

    % 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).

    % 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).

    % '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.

    % '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).

    % '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.

    % '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).

    % '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.

    % 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).

    % 'count(Set)' returns the number of elements in 'Set'.
    % Takes O(card(Set)) time.
    %
:- func count(fat_sparse_bitset(T)) = int <= enum(T).

    % '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, 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, 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), fat_sparse_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, 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, 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), fat_sparse_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.

    % 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.

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

27 float

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%---------------------------------------------------%
% Copyright (C) 1994-1998,2001-2008,2010, 2012 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%---------------------------------------------------%
%
% 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 'math.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, greater than, less than or equal, greater than or equal.
%
:- pred (float::in)  < (float::in) is semidet.
:- pred (float::in) =< (float::in) is semidet.
:- pred (float::in) >= (float::in) is semidet.
:- pred (float::in) >  (float::in) is semidet.

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

```
% Conversion functions
%

% Convert int to float
%
:- func float(int) = float.

% 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 float.pow(B, 0) = 1.0 for all B, even B=0.0.
% Only pow(0, <negative>) throws a ‘math.domain_error’ exception.
%
```

```
:- func pow(float, int) = float.

% Compute a non-negative integer hash value for a float.
%
:- func hash(float) = int.

% Is the floating point number not a number or infinite?
%
:- pred is_nan_or_inf(float::in) is semidet.

% Is the floating point number not a number?
%
:- pred is_nan(float::in) is semidet.

% Is the floating point number infinite?
%
:- pred is_inf(float::in) is semidet.

%-----------------------------------------------%
%
% System constants
%

% Maximum finite floating-point number
%
% max = (1 - radix ** mantissa_digits) * radix ** max_exponent
%
:- func float.max = float.

% Minimum normalised positive floating-point number
%
% min = radix ** (min_exponent - 1)
%
:- func float.min = 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 float.epsilon = float.

% Radix of the floating-point representation.
% In the literature, this is sometimes referred to as `b'.
%
:- func float.radix = int.
```

```
% The number of base-radix digits in the mantissa.  In the
% literature, this is sometimes referred to as ‘p’ or ‘t’.
%
:- func float.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 float.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 float.max_exponent = int.

    % Convert a float to a pretty_printer.doc for formatting.
    %
:- func float.float_to_doc(float) = doc.

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

28 gc

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%-----------------------------------------------------%
% Copyright (C) 1999,2001-2007 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% 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.

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

29 getopt

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 1994-1999,2001-2007, 2011 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING in the Mercury distribution.
%-----------------------------------------------------%
%
% File: getopt.m.
% Authors: fjh, zs.
% Stability: medium.
%
% This module exports the predicate getopt.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.

% getopt.process_options(OptionOps, Args, NonOptionArgs, Result)
% getopt.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 getopt.process_options(option_ops(OptionType)::in(option_ops),
    list(string)::in, list(string)::out,
    maybe_option_table(OptionType)::out) is det.

:- pred getopt.process_options(option_ops(OptionType)::in(option_ops),
    list(string)::in, list(string)::out, list(string)::out,
    maybe_option_table(OptionType)::out) is det.

% getopt.process_options_track(OptionOps, Args, OptionArgs,
%       NonOptionArgs, OptionTable0, Result, OptionsSet)

:- pred getopt.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.

:- 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))
            ).

:- 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 ==
    bound((
        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
        )
    )).

:- inst option_ops_track ==
    bound((
        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).

    % 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 getopt.lookup_bool_option(option_table(Option)::in, Option::in,
    bool::out) is det.
:- func getopt.lookup_bool_option(option_table(Option), Option) = bool.

:- pred getopt.lookup_int_option(option_table(Option)::in, Option::in,
    int::out) is det.
:- func getopt.lookup_int_option(option_table(Option), Option) = int.

:- pred getopt.lookup_string_option(option_table(Option)::in, Option::in,
    string::out) is det.
:- func getopt.lookup_string_option(option_table(Option), Option) = string.

:- pred getopt.lookup_maybe_int_option(option_table(Option)::in, Option::in,
    maybe(int)::out) is det.
:- func getopt.lookup_maybe_int_option(option_table(Option), Option) =
    maybe(int).

:- pred getopt.lookup_maybe_string_option(option_table(Option)::in,
    Option::in, maybe(string)::out) is det.
:- func getopt.lookup_maybe_string_option(option_table(Option), Option) =
    maybe(string).

:- pred getopt.lookup_accumulating_option(option_table(Option)::in,
    Option::in, list(string)::out) is det.
:- func getopt.lookup_accumulating_option(option_table(Option), Option) =
    list(string).

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

30 getopt_io

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 2005-2007, 2011 The University of Melbourne.
```

```
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING in the Mercury distribution.
%-----------------------------------------------------%
%
% File: getopt_io.m
% Authors: fjh, zs
% Stability: medium
%
% This module exports the predicate getopt_io.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.

% getopt_io.process_options(OptionOps, Args, NonOptionArgs, Result)
% getopt_io.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 getopt_io.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 getopt_io.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.

% getopt_io.process_options_track(OptionOps, Args, OptionArgs,
%       NonOptionArgs, OptionTable0, Result, OptionsSet)

:- pred getopt_io.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.

:- 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))
            ).
```

```
:- 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 ==
    bound((
        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
        )
    )).

:- inst option_ops_track ==
    bound((
        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).

    % 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 getopt_io.lookup_bool_option(option_table(Option)::in, Option::in,
    bool::out) is det.
:- func getopt_io.lookup_bool_option(option_table(Option), Option) = bool.

:- pred getopt_io.lookup_int_option(option_table(Option)::in, Option::in,
    int::out) is det.
:- func getopt_io.lookup_int_option(option_table(Option), Option) = int.

:- pred getopt_io.lookup_string_option(option_table(Option)::in, Option::in,
    string::out) is det.
:- func getopt_io.lookup_string_option(option_table(Option), Option) = string.

:- pred getopt_io.lookup_maybe_int_option(option_table(Option)::in, Option::in,
    maybe(int)::out) is det.
:- func getopt_io.lookup_maybe_int_option(option_table(Option), Option) =
    maybe(int).
```

```
:- pred getopt_io.lookup_maybe_string_option(option_table(Option)::in,
    Option::in, maybe(string)::out) is det.
:- func getopt_io.lookup_maybe_string_option(option_table(Option), Option) =
    maybe(string).

:- pred getopt_io.lookup_accumulating_option(option_table(Option)::in,
    Option::in, list(string)::out) is det.
:- func getopt_io.lookup_accumulating_option(option_table(Option), Option) =
    list(string).

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

31 hash_table

```
%-----------------------------------------------------%
% vim: ts=4 sw=4 et tw=0 wm=0 ft=mercury
%-----------------------------------------------------%
% Copyright (C) 2001, 2003-2006, 2010-2012 The University of Melbourne
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% File: hash_table.m.
% Main author: rafe, wangp.
% Stability: low.
%
% Hash table implementation.
%
% This implementation requires the user to supply a predicate that
% will compute 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 a user set occupancy level is achieved, the number of buckets
% in the table is doubled and the previous contents reinserted into
% the new hash table.
%
% CAVEAT: the user is referred to the warning at the head of array.m
% with regard to the current use of unique objects.  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 == 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 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), int, float) = hash_table(K, V).
:- mode init(in(hash_pred), in, in) = hash_table_uo is det.

    % A synonym for the above.
    %
:- pragma obsolete(new/3).
:- func new(hash_pred(K), int, float) = hash_table(K, V).
```

```
:- mode new(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.

    % A synonym for the above.
    %
:- pragma obsolete(new_default/1).
:- func new_default(hash_pred(K)) = hash_table(K, V).
:- mode new_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.

    % Default hash_preds for ints and strings and everything (buwahahaha!)
    %
:- pred int_hash(int::in, int::out) is det.
:- pred string_hash(string::in, int::out) is det.
:- pred char_hash(char::in, int::out) is det.
:- pred float_hash(float::in, int::out) is det.
:- pred generic_hash(T::in, int::out) 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 the previous value is overwritten.
    % A predicate version is also provided.
    %
:- 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.  An
    % exception is thrown if a binding for the key is already
    % present.  A predicate version is also provided.
    %
:- 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.  An
    % exception is thrown if a binding for the key does not
    % already exist.  A predicate version is also provided.
    %
:- 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.  A predicate version is also
    % provided.
    %
:- 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.  An exception
```

```
    % is raised 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.

    % Like lookup, but just fails 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.

    % 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, T, T), hash_table(K, V), T, T).
:- 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.

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

32 injection

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%-----------------------------------------------------%
% Copyright (C) 2005-2006, 2010-2011 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% 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 injection.init = injection(K, V).
:- pred injection.init(injection(K, V)::out) is det.

    % Intialise an injection wit the given key-value pair.
    %
:- func injection.singleton(K, V) = injection(K, V).

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

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

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

```
    % Combined forward/reverse search.
    % (Declaratively equivalent to reverse_search.)
    %
:- pred injection.search(injection(K, V), K, V).
:- mode injection.search(in, in, out) is cc_nondet.
:- mode injection.search(in, out, in) is semidet.

    % Look up the value for a given key, but throw an exception if it
    % is not present.
    %
:- func injection.lookup(injection(K, V), K) = V.
:- pred injection.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 injection.reverse_lookup(injection(K, V), V) = K.
:- pred injection.reverse_lookup(injection(K, V)::in, K::out, V::in) is det.

    % Return the list of all keys in the injection.
    %
:- func injection.keys(injection(K, V)) = list(K).
:- pred injection.keys(injection(K, V)::in, list(K)::out) is det.

    % Return the list of all values in the injection.
    %
:- func injection.values(injection(K, V)) = list(V).
:- pred injection.values(injection(K, V)::in, list(V)::out) is det.

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

    % Succeeds if the injection contains the given value.
    %
:- pred injection.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 injection.insert(injection(K, V), K, V) = injection(K, V) is semidet.
:- pred injection.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 injection.det_insert(injection(K, V), K, V) = injection(K, V).
:- pred injection.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 injection.update(injection(K, V), K, V) = injection(K, V) is semidet.
:- pred injection.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 injection.det_update(injection(K, V), K, V) = injection(K, V).
:- pred injection.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 injection.set(injection(K, V), K, V) = injection(K, V) is semidet.
:- pred injection.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 injection.det_set(injection(K, V), K, V) = injection(K, V).
:- pred injection.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 injection.insert_from_assoc_list(assoc_list(K, V), injection(K, V)) =
    injection(K, V) is semidet.
:- pred injection.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 injection.det_insert_from_assoc_list(assoc_list(K, V),
```

```
    injection(K, V)) = injection(K, V).
:- pred injection.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 injection.set_from_assoc_list(assoc_list(K, V), injection(K, V)) =
    injection(K, V) is semidet.
:- pred injection.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 injection.det_set_from_assoc_list(assoc_list(K, V), injection(K, V)) =
    injection(K, V).
:- pred injection.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 injection.insert_from_corresponding_lists(list(K), list(V),
    injection(K, V)) = injection(K, V) is semidet.
:- pred injection.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 injection.det_insert_from_corresponding_lists(list(K), list(V),
    injection(K, V)) = injection(K, V).
:- pred injection.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 injection.set_from_corresponding_lists(list(K), list(V),
    injection(K, V)) = injection(K, V) is semidet.
:- pred injection.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 injection.det_set_from_corresponding_lists(list(K), list(V),
    injection(K, V)) = injection(K, V).
:- pred injection.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 injection.delete_key(injection(K, V), K) = injection(K, V).
:- pred injection.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 injection.delete_value(injection(K, V), V) = injection(K, V).
:- pred injection.delete_value(V::in, injection(K, V)::in,
    injection(K, V)::out) is det.

    % Apply injection.delete_key to a list of keys.
    %
:- func injection.delete_keys(injection(K, V), list(K)) = injection(K, V).
:- pred injection.delete_keys(list(K)::in, injection(K, V)::in,
    injection(K, V)::out) is det.

    % Apply injection.delete_value to a list of values.
    %
:- func injection.delete_values(injection(K, V), list(V)) = injection(K, V).
:- pred injection.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 injection.merge(injection(K, V), injection(K, V)) = injection(K, V).
:- pred injection.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 injection.overlay(injection(K, V), injection(K, V)) = injection(K, V).
```

```
:- pred injection.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 injection.apply_forward_map_to_list(injection(K, V), list(K)) =
    list(V).
:- pred injection.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 injection.apply_reverse_map_to_list(injection(K, V), list(V)) =
    list(K).
:- pred injection.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 injection.map_keys(func(V, K) = L, injection(K, V)) = injection(L, V).
:- pred injection.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 injection.map_keys, but deletes any keys for which the
    % transformation fails.
    %
:- pred injection.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 injection.map_values(func(K, V) = W, injection(K, V)) =
    injection(K, W).
:- pred injection.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 injection.forward_map(injection(K, V)) = map(K, V).
:- pred injection.forward_map(injection(K, V)::in, map(K, V)::out) is det.

    % Extract the reverse map from an injection.
    %
:- func injection.reverse_map(injection(K, V)) = map(V, K).
:- pred injection.reverse_map(injection(K, V)::in, map(V, K)::out) is det.

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

33 int

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%---------------------------------------------------%
% Copyright (C) 1994-2011 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%---------------------------------------------------%
%
% 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.

    % Absolute value.
    %
:- func int.abs(int) = int.
:- pred int.abs(int::in, int::out) is det.

    % Maximum.
    %
:- func int.max(int, int) = int.
:- pred int.max(int::in, int::in, int::out) is det.

    % Minimum.
    %
:- func int.min(int, int) = int.
:- pred int.min(int::in, int::in, int::out) is det.

    % Exponentiation.
    % int.pow(X, Y, Z): Z is X raised to the Yth power.
    % Throws a 'math.domain_error' exception if Y is negative.
    %
:- func int.pow(int, int) = int.
:- pred int.pow(int::in, int::in, int::out) is det.

    % Base 2 logarithm.
    % int.log2(X) = N is the least integer such that 2 to the power N
    % is greater than or equal to X.
    % Throws a 'math.domain_error' exception if X is not positive.
    %
:- func int.log2(int) = int.
:- pred int.log2(int::in, int::out) 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 int.plus(int, int) = int.

    % Multiplication.
    %
:- func (int::in) * (int::in) = (int::uo) is det.
:- func int.times(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 int.minus(int, int) = int.

    % Flooring integer division.
    % Truncates towards minus infinity, e.g. (-10) div 3 = (-4).
    %
    % Throws a `math.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 `math.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 'math.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.

    % 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).
    %
:- func (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 'int.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 "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).
    %
```

```
:- func (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 ‘int.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.

    % 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.

    % 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 int.xor(int, int) = int.
:- mode int.xor(in, in) = uo is det.
:- mode int.xor(in, uo) = in is det.
:- mode int.xor(uo, in) = in is det.

    % Bitwise complement.
    %
:- func \ (int::in) = (int::uo) is det.

    % Unary plus.
    %
:- func + (int::in) = (int::uo) is det.

    % Unary minus.
    %
:- func - (int::in) = (int::uo) 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.

    % int.max_int is the maximum value of an int on this machine.
    %
:- func int.max_int = int.
:- pred int.max_int(int::out) is det.

    % int.min_int is the minimum value of an int on this machine.
    %
:- func int.min_int = int.
:- pred int.min_int(int::out) is det.

    % int.bits_per_int is the number of bits in an int on this machine.
    %
:- func int.bits_per_int = int.
:- pred int.bits_per_int(int::out) is det.

    % fold_up(F, Low, High, !Acc) <=> list.foldl(F, Low .. High, !Acc)
    %
    % NOTE: fold_up/5 is undefined if High = int.max_int.
    %
:- pred int.fold_up(pred(int, T, T), int, int, T, T).
:- mode int.fold_up(pred(in, in, out) is det, in, in, in, out) is det.
:- mode int.fold_up(pred(in, mdi, muo) is det, in, in, mdi, muo) is det.
:- mode int.fold_up(pred(in, di, uo) is det, in, in, di, uo) is det.
:- mode int.fold_up(pred(in, array_di, array_uo) is det, in, in,
    array_di, array_uo) is det.
:- mode int.fold_up(pred(in, in, out) is semidet, in, in, in, out)
    is semidet.
:- mode int.fold_up(pred(in, mdi, muo) is semidet, in, in, mdi, muo)
    is semidet.
:- mode int.fold_up(pred(in, di, uo) is semidet, in, in, di, uo)
    is semidet.
:- mode int.fold_up(pred(in, in, out) is nondet, in, in, in, out)
    is nondet.
:- mode int.fold_up(pred(in, mdi, muo) is nondet, in, in, mdi, muo)
    is nondet.
:- mode int.fold_up(pred(in, di, uo) is cc_multi, in, in, di, uo)
    is cc_multi.
:- mode int.fold_up(pred(in, in, out) is cc_multi, in, in, in, out)
    is cc_multi.

    % fold_up(F, Low, High, Acc) <=> list.foldl(F, Low .. High, Acc)
    %
    % NOTE: fold_up/4 is undefined if High = int.max_int.
    %
```

```
:- func int.fold_up(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 int.min_int.
    %
:- pred int.fold_down(pred(int, T, T), int, int, T, T).
:- mode int.fold_down(pred(in, in, out) is det, in, in, in, out) is det.
:- mode int.fold_down(pred(in, mdi, muo) is det, in, in, mdi, muo) is det.
:- mode int.fold_down(pred(in, di, uo) is det, in, in, di, uo) is det.
:- mode int.fold_down(pred(in, array_di, array_uo) is det, in, in,
    array_di, array_uo) is det.
:- mode int.fold_down(pred(in, in, out) is semidet, in, in, in, out)
    is semidet.
:- mode int.fold_down(pred(in, mdi, muo) is semidet, in, in, mdi, muo)
    is semidet.
:- mode int.fold_down(pred(in, di, uo) is semidet, in, in, di, uo)
    is semidet.
:- mode int.fold_down(pred(in, in, out) is nondet, in, in, in, out)
    is nondet.
:- mode int.fold_down(pred(in, mdi, muo) is nondet, in, in, mdi, muo)
    is nondet.
:- mode int.fold_down(pred(in, in, out) is cc_multi, in, in, in, out)
    is cc_multi.
:- mode int.fold_down(pred(in, di, uo) is cc_multi, in, in, di, uo)
    is cc_multi.

    % fold_down(F, Low, High, Acc) <=> list.foldr(F, Low .. High, Acc)
    %
    % NOTE: fold_down/4 is undefined if Low = int.min_int.
    %
:- func int.fold_down(func(int, T) = T, int, int, T) = T.

    % fold_up2(F, Low, High, !Acc1, Acc2) <=>
    %   list.foldl2(F, Low .. High, !Acc1, !Acc2)
    %
    % NOTE: fold_up2/7 is undefined if High = int.max_int.
    %
:- pred int.fold_up2(pred(int, T, T, U, U), int, int, T, T, U, U).
:- mode int.fold_up2(pred(in, in, out, in, out) is det, in, in, in, out,
    in, out) is det.
:- mode int.fold_up2(pred(in, in, out, mdi, muo) is det, in, in, in, out,
    mdi, muo) is det.
:- mode int.fold_up2(pred(in, in, out, di, uo) is det, in, in, in, out,
    di, uo) is det.
:- mode int.fold_up2(pred(in, di, uo, di, uo) is det, in, in, di, uo,
    di, uo) is det.
```

```
:- mode int.fold_up2(pred(in, in, out, array_di, array_uo) is det, in, in,
    in, out, array_di, array_uo) is det.
:- mode int.fold_up2(pred(in, in, out, in, out) is semidet, in, in,
    in, out, in, out) is semidet.
:- mode int.fold_up2(pred(in, in, out, mdi, muo) is semidet, in, in,
    in, out, mdi, muo) is semidet.
:- mode int.fold_up2(pred(in, in, out, di, uo) is semidet, in, in,
    in, out, di, uo) is semidet.
:- mode int.fold_up2(pred(in, in, out, in, out) is nondet, in, in,
    in, out, in, out) is nondet.
:- mode int.fold_up2(pred(in, in, out, mdi, muo) is nondet, in, in,
    in, out, mdi, muo) is nondet.

    % fold_down2(F, Low, High, !Acc1, !Acc2) <=>
    %   list.foldr2(F, Low .. High, !Acc1, Acc2).
    %
    % NOTE: fold_down2/7 is undefined if Low = int.min_int.
    %
:- pred int.fold_down2(pred(int, T, T, U, U), int, int, T, T, U, U).
:- mode int.fold_down2(pred(in, in, out, in, out) is det, in, in, in, out,
    in, out) is det.
:- mode int.fold_down2(pred(in, in, out, mdi, muo) is det, in, in, in, out,
    mdi, muo) is det.
:- mode int.fold_down2(pred(in, in, out, di, uo) is det, in, in, in, out,
    di, uo) is det.
:- mode int.fold_down2(pred(in, di, uo, di, uo) is det, in, in, di, uo,
    di, uo) is det.
:- mode int.fold_down2(pred(in, in, out, array_di, array_uo) is det, in, in,
    in, out, array_di, array_uo) is det.
:- mode int.fold_down2(pred(in, in, out, in, out) is semidet, in, in,
    in, out, in, out) is semidet.
:- mode int.fold_down2(pred(in, in, out, di, uo) is semidet, in, in,
    in, out, di, uo) is semidet.
:- mode int.fold_down2(pred(in, in, out, in, out) is nondet, in, in,
    in, out, in, out) is nondet.
:- mode int.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.foldl3(F, Low .. High, !Acc1, !Acc2, !Acc3)
    %
    % NOTE: fold_up3/9 is undefined if High = int.max_int.
    %
:- pred int.fold_up3(pred(int, T, T, U, U, V, V), int, int, T, T, U, U, V, V).
:- mode int.fold_up3(pred(in, in, out, in, out, in, out) is det,
    in, in, in, out, in, out, in, out) is det.
:- mode int.fold_up3(pred(in, in, out, in, out, mdi, muo) is det,
```

```
    in, in, in, out, in, out, mdi, muo) is det.
:- mode int.fold_up3(pred(in, in, out, in, out, di, uo) is det,
    in, in, in, out, in, out, di, uo) is det.
:- mode int.fold_up3(pred(in, in, out, di, uo, di, uo) is det,
    in, in, in, out, di, uo, di, uo) is det.
:- mode int.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 int.fold_up3(pred(in, in, out, in, out, in, out) is semidet,
    in, in, in, out, in, out, in, out) is semidet.
:- mode int.fold_up3(pred(in, in, out, in, out, mdi, muo) is semidet,
    in, in, in, out, in, out, mdi, muo) is semidet.
:- mode int.fold_up3(pred(in, in, out, in, out, di, uo) is semidet,
    in, in, in, out, in, out, di, uo) is semidet.
:- mode int.fold_up3(pred(in, in, out, in, out, in, out) is nondet,
    in, in, in, out, in, out, in, out) is nondet.
:- mode int.fold_up3(pred(in, in, out, in, out, mdi, muo) is nondet,
    in, in, in, out, 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 = int.min_int.
    %
:- pred int.fold_down3(pred(int, T, T, U, U, V, V), int, int, T, T, U, U, V, V).
:- mode int.fold_down3(pred(in, in, out, in, out, in, out) is det,
    in, in, in, out, in, out, in, out) is det.
:- mode int.fold_down3(pred(in, in, out, in, out, mdi, muo) is det,
    in, in, in, out, in, out, mdi, muo) is det.
:- mode int.fold_down3(pred(in, in, out, in, out, di, uo) is det,
    in, in, in, out, in, out, di, uo) is det.
:- mode int.fold_down3(pred(in, in, out, di, uo, di, uo) is det,
    in, in, in, out, di, uo, di, uo) is det.
:- mode int.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 int.fold_down3(pred(in, in, out, in, out, in, out) is semidet,
    in, in, in, out, in, out, in, out) is semidet.
:- mode int.fold_down3(pred(in, in, out, in, out, mdi, muo) is semidet,
    in, in, in, out, in, out, mdi, muo) is semidet.
:- mode int.fold_down3(pred(in, in, out, in, out, di, uo) is semidet,
    in, in, in, out, in, out, di, uo) is semidet.
:- mode int.fold_down3(pred(in, in, out, in, out, in, out) is nondet,
    in, in, in, out, in, out, in, out) is nondet.
:- mode int.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(Lo, Hi, I):
    %
```

```
    % On successive successes, set I to every integer from Lo to Hi.
    %
:- pred nondet_int_in_range(int::in, int::in, int::out) is nondet.

    % Convert an int to a pretty_printer.doc for formatting.
    %
:- func int.int_to_doc(int) = pretty_printer.doc.

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

34 integer

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%---------------------------------------------------%
% Copyright (C) 1997-2000, 2003-2007, 2011-2012 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%---------------------------------------------------%
%
% File: integer.m.
% Main authors: aet, Dan Hazel <odin@svrc.uq.edu.au>.
% Stability: high.
%
% Implements an arbitrary precision integer type and basic
% operations on it. (An arbitrary precision integer may have
% any number of digits, unlike an int, which is limited to the
% precision of the machine's int type, which is typically 32 bits.)
%
% NOTE: All operators behave as the equivalent operators on ints do.
% This includes the division operators: / // rem div mod.
%
%---------------------------------------------------%
%---------------------------------------------------%

:- module integer.
:- interface.

:- type integer.

:- pred '<'(integer::in, integer::in) is semidet.

:- pred '>'(integer::in, integer::in) is semidet.
```

```
:- pred '=<'(integer::in, integer::in) is semidet.

:- pred '>='(integer::in, integer::in) is semidet.

:- func integer.integer(int) = integer.

:- func integer.to_string(integer) = string.

:- func integer.from_string(string::in) = (integer::out) is semidet.

:- func integer.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 function fails.
    %
:- func integer.from_base_string(int, string) = integer is semidet.

    % As above but throws an exception rather than failing.
    %
:- func integer.det_from_base_string(int, string) = integer.

:- func '+'(integer) = integer.

:- func '-'(integer) = integer.

:- func integer + integer = integer.

:- func integer - integer = integer.

:- func integer * integer = integer.

:- func integer // integer = integer.

:- func integer div integer = integer.

:- func integer rem integer = integer.

:- func integer mod 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.
```

```
:- func integer << int = integer.

:- func integer >> int = integer.

:- func integer /\ integer = integer.

:- func integer \/ integer = integer.

:- func integer `xor` integer = integer.

:- func \ integer = integer.

:- func integer.abs(integer) = integer.

:- func integer.pow(integer, integer) = integer.

:- func integer.float(integer) = float.
:- func integer.int(integer) = int.

:- func integer.zero = integer.

:- func integer.one = integer.

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

35 io

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%---------------------------------------------------%
% Copyright (C) 1993-2012 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%---------------------------------------------------%
%
% 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 you don’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 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 == io.state.

    % Opaque handles for text I/O streams.
    %
:- type io.input_stream.
:- type io.output_stream.
```

```
    % Alternative names for the above.
    %
:- type io.text_input_stream == io.input_stream.
:- type io.text_output_stream == io.output_stream.

    % Opaque handles for binary I/O streams.
    %
:- type io.binary_input_stream.
:- type io.binary_output_stream.

    % A unique identifier for an I/O stream.
    %
:- type io.stream_id.

    % Various types used for the result from the access predicates.
    %
:- type io.res
    --->    ok
    ;       error(io.error).

:- type io.res(T)
    --->    ok(T)
    ;       error(io.error).

    % io.maybe_partial_res is used where it is possible to return
    % a partial result when an error occurs.
    %
:- type io.maybe_partial_res(T)
    --->    ok(T)
    ;       error(T, io.error).

:- inst io.maybe_partial_res(T)
    --->    ok(T)
    ;       error(T, ground).

:- type io.result
    --->    ok
    ;       eof
    ;       error(io.error).

:- type io.result(T)
    --->    ok(T)
    ;       eof
    ;       error(io.error).

:- type io.read_result(T)
```

```
        --->    ok(T)
        ;       eof
        ;       error(string, int). % error message, line number

:- type io.error.   % Use io.error_message to decode it.

    % Poly-type is used for io.write_many and io.format,
    % which do printf-like formatting.
    %
:- type io.poly_type == string.poly_type.

    % io.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 io.whence
    --->    set
    ;       cur
    ;       end.

%---------------------------------------------------%
%
% Text input predicates
%

    % Reads a character (code point) from the current input stream.
    %
:- pred io.read_char(io.result(char)::out, io::di, io::uo) is det.

    % Reads a whitespace delimited word from the current input stream.
    %
:- pred io.read_word(io.result(list(char))::out, io::di, io::uo) is det.

    % Reads a line from the current input stream, returns the result
    % as a list of characters (code points).
    %
:- pred io.read_line(io.result(list(char))::out, io::di, io::uo) is det.

    % Reads a line from the current input stream, returns the result
    % as a string. See the documentation for 'string.line' for the
    % definition of a line.
    %
:- pred io.read_line_as_string(io.result(string)::out, io::di, io::uo) is det.

    % Reads all the characters (code points) from the current input stream
    % until eof or error.
```

```
    %
:- pred io.read_file(io.maybe_partial_res(list(char))::out, io::di, io::uo)
    is det.

    % Reads all the characters (code points) from the current input 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.
    %
:- pred io.read_file_as_string(io.maybe_partial_res(string)::out,
    io::di, io::uo) is det.

    % Applies the given closure to each character (code point) read from
    % the input stream in turn, until eof or error.
    %
:- pred io.input_stream_foldl(pred(char, T, T), T, io.maybe_partial_res(T),
    io, io).
:- mode io.input_stream_foldl((pred(in, in, out) is det), in, out,
    di, uo) is det.
:- mode io.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 io.input_stream_foldl_io(pred(char, io, io), io.res, io, io).
:- mode io.input_stream_foldl_io((pred(in, di, uo) is det), out, di, uo)
    is det.
:- mode io.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 io.input_stream_foldl2_io(pred(char, T, T, io, io),
    T, io.maybe_partial_res(T), io, io).
:- mode io.input_stream_foldl2_io((pred(in, in, out, di, uo) is det),
    in, out, di, uo) is det.
:- mode io.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, or the closure returns 'no' as
    % its second argument.
    %
```

```
:- pred io.input_stream_foldl2_io_maybe_stop(
    pred(char, bool, T, T, io, io),
    T, io.maybe_partial_res(T), io, io).
:- mode io.input_stream_foldl2_io_maybe_stop(
    (pred(in, out, in, out, di, uo) is det),
    in, out, di, uo) is det.
:- mode io.input_stream_foldl2_io_maybe_stop(
    (pred(in, out, in, out, di, uo) is cc_multi),
    in, out, di, uo) is cc_multi.

    % Un-reads a character (code point) from the current input stream.
    % You can put back as many characters as you like.
    % You can even put back something that you didn't actually read.
    % Note: 'io.putback_char' uses the C library function ungetc().
    % On some systems only one byte of pushback is guaranteed.
    % 'io.putback_char' will throw an io.error exception if ungetc() fails.
    %
:- pred io.putback_char(char::in, io::di, io::uo) is det.

    % Reads a character (code point) from specified stream.
    %
:- pred io.read_char(io.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 io.read_char_unboxed(io.input_stream::in, io.result::out, char::out,
    io::di, io::uo) is det.

    % Reads a whitespace delimited word from specified stream.
    %
:- pred io.read_word(io.input_stream::in, io.result(list(char))::out,
    io::di, io::uo) is det.

    % Reads a line from specified stream, returning the result
    % as a list of characters (code point).
    %
:- pred io.read_line(io.input_stream::in, io.result(list(char))::out,
    io::di, io::uo) is det.

    % Reads a line from specified stream, returning the
    % result as a string. See the documentation for 'string.line' for
    % the definition of a line.
    %
:- pred io.read_line_as_string(io.input_stream::in, io.result(string)::out,
    io::di, io::uo) is det.
```

```
    % Reads all the characters (code points) from the given input stream until
    % eof or error.
    %
:- pred io.read_file(io.input_stream::in,
    io.maybe_partial_res(list(char))::out, io::di, io::uo) is det.

    % Reads all the characters from the given input 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.
    %
:- pred io.read_file_as_string(io.input_stream::in,
    io.maybe_partial_res(string)::out, io::di, io::uo) is det.

    % Applies the given closure to each character (code point) read from
    % the input stream in turn, until eof or error.
    %
:- pred io.input_stream_foldl(io.input_stream, pred(char, T, T),
    T, io.maybe_partial_res(T), io, io).
:- mode io.input_stream_foldl(in, in(pred(in, in, out) is det),
    in, out, di, uo) is det.
:- mode io.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 io.input_stream_foldl_io(io.input_stream, pred(char, io, io),
    io.res, io, io).
:- mode io.input_stream_foldl_io(in, in(pred(in, di, uo) is det),
    out, di, uo) is det.
:- mode io.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 io.input_stream_foldl2_io(io.input_stream,
    pred(char, T, T, io, io),
    T, io.maybe_partial_res(T), io, io).
:- mode io.input_stream_foldl2_io(in,
    in(pred(in, in, out, di, uo) is det),
    in, out, di, uo) is det.
:- mode io.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 io.input_stream_foldl2_io_maybe_stop(io.input_stream,
    pred(char, bool, T, T, io, io),
    T, io.maybe_partial_res(T), io, io).
:- mode io.input_stream_foldl2_io_maybe_stop(in,
    (pred(in, out, in, out, di, uo) is det),
    in, out, di, uo) is det.
:- mode io.input_stream_foldl2_io_maybe_stop(in,
    (pred(in, out, in, out, di, uo) is cc_multi),
    in, out, di, uo) is cc_multi.

    % Un-reads a character from specified stream.
    % You can put back as many characters as you like.
    % You can even put back something that you didn’t actually read.
    % Note: ‘io.putback_char’ uses the C library function ungetc().
    % On some systems only one byte of pushback is guaranteed.
    % ‘io.putback_char’ will throw an io.error exception if ungetc() fails.
    %
:- pred io.putback_char(io.input_stream::in, char::in, io::di, io::uo) is det.

    % Reads a ground term of any type, written using standard Mercury syntax,
    % from the current or specified input stream. The type of the term read
    % is determined by the context in which ‘io.read’ is used.
    %
    % First, the input stream is read until an end-of-term token, end-of-
file,
    % or I/O error is reached.  (An end-of-term token consists of a ‘.’
    % followed by whitespace. The trailing whitespace is left in the input
    % stream.)
    %
    % Then, the result is determined according to the tokens read. If there
    % were no non-whitespace characters before the end of file, then ‘io.read’
    % returns ‘eof’. If the tokens read formed a syntactically correct ground
    % term of the correct type, followed by an end-of-term token, then it
    % returns ‘ok(Term)’. If characters read from the input stream did not form
    % a syntactically correct term, or if the term read is not a ground term,
    % or if the term is not a valid term of the appropriate type, or if an
    % I/O error is encountered, then it returns ‘error(Message, LineNumber)’.
    %
:- pred io.read(io.read_result(T)::out, io::di, io::uo) is det.
:- pred io.read(io.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(int, int, int).
            % line number, offset of start of line, current offset (the first
            % two are used only for the purposes of computing term_contexts,
            % for use e.g. in error messages). Offsets start at zero.

    % io.read_from_string(FileName, String, MaxPos, Result, Posn0, Posn):
    % Same as io.read/4 except that it reads from a string rather than
    % 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 io.read_from_string(string::in, string::in, int::in,
    io.read_result(T)::out, posn::in, posn::out) is det.

    % Discards all the whitespace from the current stream.
    %
:- pred io.ignore_whitespace(io.result::out, io::di, io::uo) is det.

    % Discards all the whitespace from the specified stream.
    %
:- pred io.ignore_whitespace(io.input_stream::in, io.result::out,
    io::di, io::uo) is det.

%---------------------------------------------------%
%
% Text output predicates
%

% These will all throw an io.error exception if an I/O error occurs.

    % io.print/3 writes its argument to the standard output stream.
    % io.print/4 writes its second argument to the output stream speci-
fied 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 univ, then it
    % will print out the value stored in the univ, but not the type.
    %
    % io.print/5 is the same as io.print/4 except that it allows the caller to
    % specify how non-canonical types should be handled. io.print/3 and
    % io.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.
    %
    % io.print_cc/3 is the same as io.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 io.print(T::in, io::di, io::uo) is det.

:- pred io.print(io.output_stream::in, T::in, io::di, io::uo) is det.

:- pred io.print(io.output_stream, deconstruct.noncanon_handling, T, io, io).
:- mode io.print(in, in(do_not_allow), in, di, uo) is det.
:- mode io.print(in, in(canonicalize), in, di, uo) is det.
:- mode io.print(in, in(include_details_cc), in, di, uo) is cc_multi.
:- mode io.print(in, in, in, di, uo) is cc_multi.

:- pred io.print_cc(T::in, io::di, io::uo) is cc_multi.

    % io.write/3 writes its argument to the current output stream.
    % io.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.  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 'io.read'.  For the types containing
    % existential quantifiers, the type 'type_desc' and closure types, the
    % result may not be parsable by 'io.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
```

```
    % 'io.read'.
    %
    % io.write/5 is the same as io.write/4 except that it allows the caller
    % to specify how non-canonical types should be handled.  io.write_cc/3
    % is the same as io.write/3 except that it specifies 'include_details_cc'
    % rather than 'canonicalize'.
    %
:- pred io.write(T::in, io::di, io::uo) is det.

:- pred io.write(io.output_stream::in, T::in, io::di, io::uo) is det.

:- pred io.write(io.output_stream, deconstruct.noncanon_handling, T, io, io).
:- mode io.write(in, in(do_not_allow), in, di, uo) is det.
:- mode io.write(in, in(canonicalize), in, di, uo) is det.
:- mode io.write(in, in(include_details_cc), in, di, uo) is cc_multi.
:- mode io.write(in, in, in, di, uo) is cc_multi.

:- pred io.write_cc(T::in, io::di, io::uo) is cc_multi.

    % Writes a newline character to the current output stream.
    %
:- pred io.nl(io::di, io::uo) is det.

    % Writes a newline character to the specified output stream.
    %
:- pred io.nl(io.output_stream::in, io::di, io::uo) is det.

    % Writes a string to the current output stream.
    %
:- pred io.write_string(string::in, io::di, io::uo) is det.

    % Writes a string to the specified output stream.
    %
:- pred io.write_string(io.output_stream::in, string::in, io::di, io::uo)
    is det.

    % Writes a list of strings to the current output stream.
    %
:- pred io.write_strings(list(string)::in, io::di, io::uo) is det.

    % Writes a list of strings to the specified output stream.
    %
:- pred io.write_strings(io.output_stream::in, list(string)::in,
    io::di, io::uo) is det.

    % Writes a character to the current output stream.
    %
```

```
:- pred io.write_char(char::in, io::di, io::uo) is det.

    % Writes a character to the specified output stream.
    %
:- pred io.write_char(io.output_stream::in, char::in, io::di, io::uo) is det.

    % Writes an integer to the current output stream.
    %
:- pred io.write_int(int::in, io::di, io::uo) is det.

    % Writes an integer to the specified output stream.
    %
:- pred io.write_int(io.output_stream::in, int::in, io::di, io::uo) is det.

    % Writes a floating point number to the current output stream.
    %
:- pred io.write_float(float::in, io::di, io::uo) is det.

    % Writes a floating point number to the specified output stream.
    %
:- pred io.write_float(io.output_stream::in, float::in, io::di, io::uo)
    is det.

    % Formats the specified arguments according to the format string,
    % using string.format, and then writes the result to the current
    % output stream. (See the documentation of string.format for details.)
    %
:- pred io.format(string::in, list(io.poly_type)::in, io::di, io::uo) is det.

    % Formats the specified argument list according to the format string,
    % using string.format, and then writes the result to the specified
    % output stream. (See the documentation of string.format for details.)
    %
:- pred io.format(io.output_stream::in, string::in, list(io.poly_type)::in,
    io::di, io::uo) is det.

    % Writes the specified arguments to the current output stream.
    %
:- pred io.write_many(list(io.poly_type)::in, io::di, io::uo) is det.

    % Writes the specified arguments to the specified output stream.
    %
:- pred io.write_many(io.output_stream::in, list(io.poly_type)::in,
    io::di, io::uo) is det.

    % io.write_list(List, Separator, OutputPred, !IO):
    % applies OutputPred to each element of List, printing Separator
```

```
    % between each element. Outputs to the current output stream.
    %
:- pred io.write_list(list(T), string, pred(T, io, io), io, io).
:- mode io.write_list(in, in, pred(in, di, uo) is det, di, uo) is det.
:- mode io.write_list(in, in, pred(in, di, uo) is cc_multi, di, uo)
    is cc_multi.

    % io.write_list(Stream, List, Separator, OutputPred, !IO):
    % applies OutputPred to each element of List, printing Separator
    % between each element. Outputs to Stream.
    %
:- pred io.write_list(io.output_stream, list(T), string,
    pred(T, io, io), io, io).
:- mode io.write_list(in, in, in, pred(in, di, uo) is det, di, uo) is det.
:- mode io.write_list(in, in, in, pred(in, di, uo) is cc_multi, di, uo)
    is cc_multi.

    % Flush the output buffer of the current output stream.
    %
:- pred io.flush_output(io::di, io::uo) is det.

    % Flush the output buffer of the specified output stream.
    %
:- pred io.flush_output(io.output_stream::in, io::di, io::uo) is det.

%---------------------------------------------------%
%
% Input text stream predicates
%

    % io.see(File, Result, !IO).
    % Attempts to open a file for input, and if successful,
    % sets the current input stream to the newly opened stream.
    % Result is either 'ok' or 'error(ErrorCode)'.
    %
:- pred io.see(string::in, io.res::out, 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 io.seen(io::di, io::uo) is det.

    % Attempts to open a file for input.
    % Result is either 'ok(Stream)' or 'error(ErrorCode)'.
    %
:- pred io.open_input(string::in, io.res(io.input_stream)::out,
```

```
    io::di, io::uo) is det.

    % Closes an open input stream.
    % Throw an io.error exception if an I/O error occurs.
    %
:- pred io.close_input(io.input_stream::in, io::di, io::uo) is det.

    % Retrieves the current input stream.
    % Does not modify the I/O state.
    %
:- pred io.input_stream(io.input_stream::out, io::di, io::uo) is det.

    % io.set_input_stream(NewStream, OldStream, !IO):
    % Changes the current input stream to the stream specified.
    % Returns the previous stream.
    %
:- pred io.set_input_stream(io.input_stream::in, io.input_stream::out,
    io::di, io::uo) is det.

    % Retrieves the standard input stream.
    %
:- func io.stdin_stream = io.input_stream.

    % Retrieves the standard input stream.
    % Does not modify the I/O state.
    %
:- pred io.stdin_stream(io.input_stream::out, io::di, io::uo) is det.

    % Retrieves the human-readable name associated with the current input
    % stream. For file streams, this is the filename. For stdin,
    % this is the string "<standard input>".
    %
:- pred io.input_stream_name(string::out, io::di, io::uo) is det.

    % Retrieves the human-readable name associated with the specified input
    % stream. For file streams, this is the filename. For stdin,
    % this is the string "<standard input>".
    %
:- pred io.input_stream_name(io.input_stream::in, string::out,
    io::di, io::uo) is det.

    % Return the line number of the current input stream. Lines are normally
    % numbered starting at 1, but this can be overridden by calling
    % io.set_line_number.
    %
:- pred io.get_line_number(int::out, io::di, io::uo) is det.
```

```
    % Return the line number of the specified input stream. Lines are normally
    % numbered starting at 1, but this can be overridden by calling
    % io.set_line_number.
    %
:- pred io.get_line_number(io.input_stream::in, int::out, io::di, io::uo)
    is det.

    % Set the line number of the current input stream.
    %
:- pred io.set_line_number(int::in, io::di, io::uo) is det.

    % Set the line number of the specified input stream.
    %
:- pred io.set_line_number(io.input_stream::in, int::in, io::di, io::uo)
    is det.

%---------------------------------------------------%
%
% Output text stream predicates
%

    % Attempts to open a file for output, and if successful sets the current
    % output stream to the newly opened stream. As per Prolog tell/1.
    % Result is either 'ok' or 'error(ErrCode)'.
    %
:- pred io.tell(string::in, io.res::out, io::di, io::uo) is det.

    % Closes the current output stream; the default 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 io.told(io::di, io::uo) is det.

    % Attempts to open a file for output.
    % Result is either 'ok(Stream)' or 'error(ErrorCode)'.
    %
:- pred io.open_output(string::in, io.res(io.output_stream)::out,
    io::di, io::uo) is det.

    % Attempts to open a file for appending.
    % Result is either 'ok(Stream)' or 'error(ErrorCode)'.
    %
:- pred io.open_append(string::in, io.res(io.output_stream)::out,
    io::di, io::uo) is det.

    % Closes an open output stream.
    % This will throw an io.error exception if an I/O error occurs.
```

```
    %
:- pred io.close_output(io.output_stream::in, io::di, io::uo) is det.

    % Retrieves the current output stream.
    % Does not modify the I/O state.
    %
:- pred io.output_stream(io.output_stream::out, io::di, io::uo) is det.

    % Changes the current output stream to the stream specified.
    % Returns the previous stream.
    %
:- pred io.set_output_stream(io.output_stream::in, io.output_stream::out,
    io::di, io::uo) is det.

    % Retrieves the standard output stream.
    %
:- func io.stdout_stream = io.output_stream.

    % Retrieves the standard output stream.
    % Does not modify the I/O state.
    %
:- pred io.stdout_stream(io.output_stream::out, io::di, io::uo) is det.

    % Retrieves the standard error stream.
    %
:- func io.stderr_stream = io.output_stream.

    % Retrieves the standard error stream.
    % Does not modify the I/O state.
    %
:- pred io.stderr_stream(io.output_stream::out, io::di, io::uo) is det.

    % Retrieves the human-readable name associated with the current
    % 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 io.output_stream_name(string::out, io::di, io::uo) is det.

    % Retrieves the human-readable name associated with the specified 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 io.output_stream_name(io.output_stream::in, string::out,
    io::di, io::uo) is det.
```

```
    % Return the line number of the current output stream. Lines are normally
    % numbered starting at 1, but this can be overridden by calling
    % io.set_output_line_number.
    %
:- pred io.get_output_line_number(int::out, io::di, io::uo) is det.

    % Return the line number of the specified output stream. Lines are normally
    % numbered starting at 1, but this can be overridden by calling
    % io.set_output_line_number.
    %
:- pred io.get_output_line_number(io.output_stream::in, int::out,
    io::di, io::uo) is det.

    % Set the line number of the current output stream.
    %
:- pred io.set_output_line_number(int::in, io::di, io::uo) is det.

    % Set the line number of the specified output stream.
    %
:- pred io.set_output_line_number(io.output_stream::in, int::in,
    io::di, io::uo) is det.

%---------------------------------------------------%
%
% Binary input predicates
%

    % Reads a binary representation of a term of type T from the current
    % binary input stream.
    %
:- pred io.read_binary(io.result(T)::out, io::di, io::uo) is det.

    % Reads a binary representation of a term of type T 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 due to the current implementation,
    % io.read_binary will not work for the Java back-end.
    %
:- pred io.read_binary(io.binary_input_stream::in, io.result(T)::out,
    io::di, io::uo) is det.
```

```
    % Reads a single 8-bit byte from the current binary input stream.
    %
:- pred io.read_byte(io.result(int)::out, io::di, io::uo) is det.

    % Reads a single 8-bit byte from the specified binary input stream.
    %
:- pred io.read_byte(io.binary_input_stream::in, io.result(int)::out,
    io::di, io::uo) is det.

    % Fill a bitmap from the current binary input stream.
    % Returns 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'.
    %
:- pred io.read_bitmap(bitmap::bitmap_di, bitmap::bitmap_uo,
    int::out, io.res::out, io::di, io::uo) is det.

    % Fill a bitmap from the specified binary input stream.
    % Returns 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'.
    %
:- pred io.read_bitmap(io.binary_input_stream::in,
    bitmap::bitmap_di, bitmap::bitmap_uo, int::out, io.res::out,
    io::di, io::uo) is det.

    % io.read_bitmap(StartByte, NumBytes, !Bitmap, BytesRead, Result, !IO)
    %
    % Read NumBytes bytes into a bitmap starting at StartByte
    % from the current binary input stream.
    % Returns 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 io.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.

    % io.read_bitmap(Stream, !Bitmap, StartByte, NumBytes,
    %       BytesRead, Result, !IO)
    %
    % Read NumBytes bytes into a bitmap starting at StartByte
    % from the specified binary input stream.
    % Returns 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 io.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.

    % Reads all the bytes from the current binary input stream
    % until eof or error into a bitmap.
    %
:- pred io.read_binary_file_as_bitmap(io.res(bitmap)::out,
    io::di, io::uo) is det.

    % Reads all the bytes from the given binary input stream into a bitmap
    % until eof or error.
    %
:- pred io.read_binary_file_as_bitmap(io.binary_input_stream::in,
    io.res(bitmap)::out, io::di, io::uo) is det.

    % Reads all the bytes from the current binary input stream
    % until eof or error.
    %
:- pred io.read_binary_file(io.result(list(int))::out, io::di, io::uo) is det.

    % Reads all the bytes from the given binary input stream until
    % eof or error.
    %
:- pred io.read_binary_file(io.binary_input_stream::in,
    io.result(list(int))::out, io::di, io::uo) is det.

    % Applies the given closure to each byte read from the current binary
    % input stream in turn, until eof or error.
    %
:- pred io.binary_input_stream_foldl(pred(int, T, T),
    T, io.maybe_partial_res(T), io, io).
:- mode io.binary_input_stream_foldl((pred(in, in, out) is det),
    in, out, di, uo) is det.
:- mode io.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 current binary
    % input stream in turn, until eof or error.
    %
:- pred io.binary_input_stream_foldl_io(pred(int, io, io),
    io.res, io, io).
:- mode io.binary_input_stream_foldl_io((pred(in, di, uo) is det),
    out, di, uo) is det.
:- mode io.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 current binary
    % input stream in turn, until eof or error.
    %
:- pred io.binary_input_stream_foldl2_io(
    pred(int, T, T, io, io), T, io.maybe_partial_res(T), io, io).
:- mode io.binary_input_stream_foldl2_io(
    in(pred(in, in, out, di, uo) is det), in, out, di, uo) is det.
:- mode io.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 io.binary_input_stream_foldl2_io_maybe_stop(
    pred(int, bool, T, T, io, io), T, io.maybe_partial_res(T), io, io).
:- mode io.binary_input_stream_foldl2_io_maybe_stop(
    (pred(in, out, in, out, di, uo) is det), in, out, di, uo) is det.
:- mode io.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.
    %
:- pred io.binary_input_stream_foldl(io.binary_input_stream,
    pred(int, T, T), T, io.maybe_partial_res(T), io, io).
:- mode io.binary_input_stream_foldl(in, in(pred(in, in, out) is det),
    in, out, di, uo) is det.
:- mode io.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 given binary
    % input stream in turn, until eof or error.
    %
:- pred io.binary_input_stream_foldl_io(io.binary_input_stream,
    pred(int, io, io), io.res, io, io).
:- mode io.binary_input_stream_foldl_io(in, in(pred(in, di, uo) is det),
    out, di, uo) is det.
:- mode io.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 given binary
    % input stream in turn, until eof or error.
    %
:- pred io.binary_input_stream_foldl2_io(io.binary_input_stream,
    pred(int, T, T, io, io), T, io.maybe_partial_res(T), io, io).
```

```
:- mode io.binary_input_stream_foldl2_io(in,
    (pred(in, in, out, di, uo) is det), in, out, di, uo) is det.
:- mode io.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,
    % or the closure returns `no' as its second argument.
    %
:- pred io.binary_input_stream_foldl2_io_maybe_stop(io.binary_input_stream,
    pred(int, bool, T, T, io, io), T, io.maybe_partial_res(T), io, io).
:- mode io.binary_input_stream_foldl2_io_maybe_stop(in,
    (pred(in, out, in, out, di, uo) is det), in, out, di, uo) is det.
:- mode io.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.

    % Un-reads a byte from the current binary input stream.
    % You can put back as many bytes as you like.
    % You can even put back something that you didn't actually read.
    % The byte is taken from the bottom 8 bits of an integer.
    % Note: `io.putback_byte' uses the C library function ungetc().
    % On some systems only one byte of pushback is guaranteed.
    % `io.putback_byte' will throw an io.error exception if ungetc() fails.
    %
    % 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 io.putback_byte(int::in, io::di, io::uo) is det.

    % Un-reads a byte from specified binary input stream.
    % You can put back as many bytes as you like.
    % You can even put back something that you didn't actually read.
    % The byte is returned in the bottom 8 bits of an integer.
    % Note: `io.putback_byte' uses the C library function ungetc().
    % On some systems only one byte of pushback is guaranteed.
    % `io.putback_byte' will throw an io.error exception if ungetc() fails.
    %
    % 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 io.putback_byte(io.binary_input_stream::in, int::in,
    io::di, io::uo) is det.

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

```
% Binary output predicates
%

% These will all throw an io.error exception if an I/O error occurs.
% XXX what about wide characters?

    % Writes a binary representation of a term to the current binary output
    % stream, in a format suitable for reading in again with io.read_binary.
    %
:- pred io.write_binary(T::in, io::di, io::uo) is det.

    % Writes a binary representation of a term to the specified binary output
    % stream, in a format suitable for reading in again with io.read_binary.
    %
    % XXX Note that due to the current implementation, io.write_binary
    % will not work for the Java back-end.
    %
:- pred io.write_binary(io.binary_output_stream::in, T::in, io::di, io::uo)
    is det.

    % Writes a single byte to the current binary output stream.
    % The byte is taken from the bottom 8 bits of an int.
    %
:- pred io.write_byte(int::in, io::di, io::uo) is det.

    % Writes a single byte to the specified binary output stream.
    % The byte is taken from the bottom 8 bits of an int.
    %
:- pred io.write_byte(io.binary_output_stream::in, int::in, io::di, io::uo)
    is det.

    % Write a bitmap to the current binary output stream.
    % The bitmap must not contain a partial final byte.
    %
:- pred io.write_bitmap(bitmap, io, io).
%:- mode io.write_bitmap(bitmap_ui, di, uo) is det.
:- mode io.write_bitmap(in, di, uo) is det.

    % io.write_bitmap(BM, StartByte, NumBytes, !IO).
    % Write part of a bitmap to the current binary output stream.
    %
:- pred io.write_bitmap(bitmap, int, int, io, io).
%:- mode io.write_bitmap(bitmap_ui, in, in, di, uo) is det.
:- mode io.write_bitmap(in, in, in, di, uo) is det.

    % Write a bitmap to the specified binary output stream.
    % The bitmap must not contain a partial final byte.
```

```
    %
:- pred io.write_bitmap(io.binary_output_stream, bitmap, io, io).
%:- mode io.write_bitmap(in, bitmap_ui, di, uo) is det.
:- mode io.write_bitmap(in, in, di, uo) is det.

    % io.write_bitmap(Stream, BM, StartByte, NumBytes, !IO).
    % Write part of a bitmap to the specified binary output stream.
    %
:- pred io.write_bitmap(io.binary_output_stream, bitmap, int, int, io, io).
%:- mode io.write_bitmap(in, bitmap_ui, in, in, di, uo) is det.
:- mode io.write_bitmap(in, in, in, in, di, uo) is det.

    % Flush the output buffer of the current binary output stream.
    %
:- pred io.flush_binary_output(io::di, io::uo) is det.

    % Flush the output buffer of the specified binary output stream.
    %
:- pred io.flush_binary_output(io.binary_output_stream::in,
    io::di, io::uo) is det.

    % 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 io.putback_byte on the stream.
    %
:- pred io.seek_binary_input(io.binary_input_stream::in, io.whence::in,
    int::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 io.seek_binary_output(io.binary_output_stream::in, io.whence::in,
    int::in, io::di, io::uo) is det.

    % Returns the offset (in bytes) into the specified binary input stream.
    %
:- pred io.binary_input_stream_offset(io.binary_input_stream::in, int::out,
    io::di, io::uo) is det.

    % Returns the offset (in bytes) into the specified binary output stream.
    %
:- pred io.binary_output_stream_offset(io.binary_output_stream::in, int::out,
    io::di, io::uo) is det.
```

```
%-----------------------------------------------%
%
% Binary input stream predicates
%

    % 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 io.see_binary(string::in, io.res::out, 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 io.seen_binary(io::di, io::uo) is det.

    % Attempts to open a binary file for input.
    % Result is either 'ok(Stream)' or 'error(ErrorCode)'.
    %
:- pred io.open_binary_input(string::in,
    io.res(io.binary_input_stream)::out, 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 io.close_binary_input(io.binary_input_stream::in,
    io::di, io::uo) is det.

    % Retrieves the current binary input stream.
    % Does not modify the I/O state.
    %
:- pred io.binary_input_stream(io.binary_input_stream::out,
    io::di, io::uo) is det.

    % Changes the current input stream to the stream specified.
    % Returns the previous stream.
    %
:- pred io.set_binary_input_stream(io.binary_input_stream::in,
    io.binary_input_stream::out, io::di, io::uo) is det.

    % Retrieves the standard binary input stream.
    % Does not modify the I/O state.
    %
:- pred io.stdin_binary_stream(io.binary_input_stream::out,
    io::di, io::uo) is det.
```

```
    % Retrieves the human-readable name associated with the current binary
    % input stream. For file streams, this is the filename.
    %
:- pred io.binary_input_stream_name(string::out, io::di, io::uo) is det.

    % Retrieves the human-readable name associated with the specified
    % binary input stream. For file streams, this is the filename.
    %
:- pred io.binary_input_stream_name(io.binary_input_stream::in, string::out,
    io::di, io::uo) is det.

%---------------------------------------------------%
%
% Binary output stream predicates
%

    % 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 io.tell_binary(string::in, io.res::out, 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 io.told_binary(io::di, io::uo) is det.

    % Attempts to open a file for binary output.
    % Result is either 'ok(Stream)' or 'error(ErrorCode)'.
    %
:- pred io.open_binary_output(string::in,
    io.res(io.binary_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 io.open_binary_append(string::in,
    io.res(io.binary_output_stream)::out, 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 io.close_binary_output(io.binary_output_stream::in,
    io::di, io::uo) is det.

    % Retrieves the current binary output stream.
```

```
    % Does not modify the I/O state.
    %
:- pred io.binary_output_stream(io.binary_output_stream::out,
    io::di, io::uo) is det.

    % Retrieves the standard binary output stream.
    % Does not modify the I/O state.
    %
:- pred io.stdout_binary_stream(io.binary_output_stream::out,
    io::di, io::uo) is det.

    % Changes the current binary output stream to the stream specified.
    % Returns the previous stream.
    %
:- pred io.set_binary_output_stream(io.binary_output_stream::in,
    io.binary_output_stream::out, io::di, io::uo) is det.

    % Retrieves the human-readable name associated with the current
    % binary output stream. For file streams, this is the filename.
    %
:- pred io.binary_output_stream_name(string::out, io::di, io::uo) is det.

    % Retrieves the human-readable name associated with the specified
    % output stream. For file streams, this is the filename.
    %
:- pred io.binary_output_stream_name(io.binary_output_stream::in,
    string::out, io::di, io::uo) is det.

%---------------------------------------------------%
%
% Global state predicates
%

    % io.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 io.progname(string::in, string::out, io::di, io::uo) is det.

    % io.progname_base(DefaultProgname, Progname).
    %
    % Like 'io.progname', except that it strips off any path name
    % preceding the program name.  Useful for error messages.
    %
:- pred io.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 io.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 io.get_exit_status(int::out, io::di, io::uo) is det.
:- pred io.set_exit_status(int::in, io::di, io::uo) is det.

    % 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 io.get_globals(univ::out, io::di, io::uo) is det.
:- pred io.set_globals(univ::in, io::di, io::uo) is det.

    % io.update_globals(UpdatePred, !IO).
    % Update the 'globals' field in the I/O state based upon its current
    % value.  This is equivalent to the following:
    %
    %   io.get_globals(Globals0, !IO),
    %   UpdatePred(Globals0, Globals),
    %   io.set_globals(Globals, !IO)
    %
    % In parallel grades calls to io.update_globals/3 are atomic.
    % If 'UpdatePred' throws an exception then the 'globals' field is
    % left unchanged.
    %
:- pred io.update_globals(pred(univ, univ)::in(pred(in, out) is det),
    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 io.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. Will throw an exception
    % if the system runs out of environment space.
    %
    % Note: this predicate is not supported on Java.
    %
:- pred io.set_environment_var(string::in, string::in, io::di, io::uo) is det.

%---------------------------------------------------%
%
% File handling predicates
%

    % io.make_temp(Name, !IO) creates an empty file whose name is different
    % to the name of any existing file. Name is bound to the name of the file.
    % It is the responsibility of the program to delete the file when it is
    % no longer needed.
    %
    % The file will reside in an implementation-dependent directory.
    % 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 io.make_temp(string::out, io::di, io::uo) is det.

    % io.make_temp(Dir, Prefix, Name, !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 using up to
    % the first 5 characters of 'Prefix'. Name is bound to the name of the
    % file. It is the responsibility of the program to delete the file
    % when it is no longer needed.
    %
:- pred io.make_temp(string::in, string::in, string::out, io::di, io::uo)
    is det.
```

```
    % io.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 io.remove_file(string::in, io.res::out, io::di, io::uo) is det.

    % io.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 ‘io.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.

    % io.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 io.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 io.have_symlinks is semidet.

    % io.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 io.make_symlink(string::in, string::in, io.res::out, io::di, io::uo)
    is det.
```

```
    % io.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 io.read_symlink(string::in, io.res(string)::out, io::di, io::uo)
    is det.

:- type io.access_type
    --->    read
    ;       write
    ;       execute.

    % io.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 io.check_file_accessibility(string::in, list(access_type)::in,
    io.res::out, io::di, io::uo) is det.

:- type io.file_type
    --->    regular_file
    ;       directory
    ;       symbolic_link
    ;       named_pipe
    ;       socket
    ;       character_device
    ;       block_device
    ;       message_queue
    ;       semaphore
    ;       shared_memory
    ;       unknown.

    % io.file_type(FollowSymLinks, FileName, TypeResult)
    % finds the type of the given file.
    %
:- pred io.file_type(bool::in, string::in, io.res(file_type)::out,
    io::di, io::uo) is det.

    % io.file_modification_time(FileName, TimeResult)
    % finds the last modification time of the given file.
```

```
    %
:- pred io.file_modification_time(string::in, io.res(time_t)::out,
    io::di, io::uo) is det.

%---------------------------------------------------%
%
% Memory management predicates
%

    % Write memory/time usage statistics to stderr.
    %
:- pred io.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 io.report_stats(string::in, io::di, io::uo) is det.

%---------------------------------------------------%
%
% Miscellaneous 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 io.call_system(string::in, io.res(int)::out, io::di, io::uo) is det.

:- type io.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 io.call_system_return_signal(string::in,
    io.res(io.system_result)::out, io::di, io::uo) is det.

    % Construct an error code including the specified error message.
    %
:- func io.make_io_error(string) = io.error.

    % Look up the error message corresponding to a particular error code.
    %
:- func io.error_message(io.error) = string.
:- pred io.error_message(io.error::in, string::out) is det.

%---------------------------------------------------%
%
% Instances of the stream typeclass
%

:- instance stream.error(io.error).

:- instance stream.stream(io.output_stream, io).
:- instance stream.output(io.output_stream, io).
:- instance stream.writer(io.output_stream, char,   io).
:- instance stream.writer(io.output_stream, float,  io).
:- instance stream.writer(io.output_stream, int,    io).
:- instance stream.writer(io.output_stream, string, io).
:- instance stream.writer(io.output_stream, univ,   io).
:- instance stream.line_oriented(io.output_stream, io).

:- instance stream.stream(io.input_stream, io).
:- instance stream.input(io.input_stream, io).
```

```
:- instance stream.reader(io.input_stream, char, io, io.error).
:- instance stream.reader(io.input_stream, line, io, io.error).
:- instance stream.reader(io.input_stream, text_file, io, io.error).

:- instance stream.line_oriented(io.input_stream, io).
:- instance stream.putback(io.input_stream, char, io, io.error).

:- instance stream.stream(io.binary_output_stream, io).
:- instance stream.output(io.binary_output_stream, io).
:- instance stream.writer(io.binary_output_stream, byte, io).
:- instance stream.writer(io.binary_output_stream, bitmap.slice, io).
:- instance stream.seekable(io.binary_output_stream, io).

:- instance stream.stream(io.binary_input_stream,  io).
:- instance stream.input(io.binary_input_stream, io).
:- instance stream.reader(io.binary_input_stream, int, io, io.error).
:- instance stream.bulk_reader(io.binary_input_stream, int,
        bitmap, io, io.error).
:- instance stream.putback(io.binary_input_stream, int, io, io.error).
:- instance stream.seekable(io.binary_input_stream, io).

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

36 lazy

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%---------------------------------------------------%
% Copyright (C) 1999, 2006, 2009-2010 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%---------------------------------------------------%
%
% lazy.m - provides support for optional explicit lazy evaluation.
%
% Author: fjh, pbone.
% Stability: medium.
%
% 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 contains subterms of type lazy(T),
    % as is the case when T is a recursive type like the lazy_list(T) type
    % defined in lazy_list.m, 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).
%
%-----------------------------------------------------%
%-----------------------------------------------------%
```

37 lexer

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 1993-2000, 2003-2008, 2011-2012 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% 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.

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

:- type token
    --->    name(string)
    ;       variable(string)
    ;       integer(int)
    ;       big_integer(string) % does not fit in int
    ;       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(int).   % the lexer will never return this.
                                % The integer_dot/1 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.

    % 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.

    % Read a list of tokens from the current input stream.
    % Keep reading until we encounter either an 'end' token
    % (i.e. a full stop followed by whitespace) or the end-of-file.
    %
:- pred get_token_list(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.
    %
:- pred string_get_token_list_max(string::in, offset::in, token_list::out,
    posn::in, 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.

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

38 library

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%-----------------------------------------------------%
% Copyright (C) 1993-2007, 2009-2012 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% 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.

:- pred library.version(string::out) is det.
```

39 list

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%-----------------------------------------------------%
% Copyright (C) 1993-2012 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
```

```
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% 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) ---> [] ; [I | list_skel(I)].
:- inst list(I) == list_skel(I).

:- inst non_empty_list ---> [ground | ground].

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

:- pred list.is_empty(list(T)::in) is semidet.
```

```
:- pred list.is_not_empty(list(T)::in) is semidet.

    % list.cons(X, Y, Z) <=> Z = [X | Y].
    %
:- pred list.cons(T::in, list(T)::in, list(T)::out) is det.
:- func list.cons(T, list(T)) = list(T).

    % Standard append predicate:
    % list.append(Start, End, List) is true iff
    % 'List' is the result of concatenating 'Start' and 'End'.
    %
:- pred list.append(list(T), list(T), list(T)).
:- mode list.append(di, di, uo) is det.
:- mode list.append(in, in, out) is det.
:- mode list.append(in, in, in) is semidet.    % implied
:- mode list.append(in, out, in) is semidet.
:- mode list.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 list.remove_suffix instead.
% :- mode list.append(out, in, in) is semidet.

:- func list.append(list(T), list(T)) = list(T).

    % 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.
    % XXX when we implement rewrite rules, we should change this law
    % to a rewrite rule.
:- promise all [L, H, T] ( append([H], T, L) <=> L = [H | T] ).

    % L1 ++ L2 = L :- list.append(L1, L2, L).
    %
:- func list(T) ++ list(T) = list(T).

    % list.remove_suffix(List, Suffix, Prefix):
    %
    % The same as list.append(Prefix, Suffix, List) except that
    % this is semidet whereas list.append(out, in, in) is nondet.
    %
:- pred list.remove_suffix(list(T)::in, list(T)::in, list(T)::out) is semidet.
```

```
    % list.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.
    %
:- pred list.merge(list(T)::in, list(T)::in, list(T)::out) is det.
:- func list.merge(list(T), list(T)) = list(T).

    % list.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.
    %
:- pred list.merge_and_remove_dups(list(T)::in, list(T)::in, list(T)::out)
    is det.
:- func list.merge_and_remove_dups(list(T), list(T)) = list(T).

    % list.remove_adjacent_dups(L0, L):
    %
    % L is the result of replacing every sequence of duplicate elements in L0
    % with a single such element.
    %
:- pred list.remove_adjacent_dups(list(T)::in, list(T)::out) is det.
:- func list.remove_adjacent_dups(list(T)) = list(T).

    % list.remove_dups(L0, L):
    %
    % L is the result of deleting the second and subsequent occurrences
    % of every element that occurs twice in L0.
    %
:- pred list.remove_dups(list(T)::in, list(T)::out) is det.
:- func list.remove_dups(list(T)) = list(T).

    % list.member(Elem, List):
    %
    % True iff ‘List’ contains ‘Elem’.
    %
:- pred list.member(T, list(T)).
:- mode list.member(in, in) is semidet.
:- mode list.member(out, in) is nondet.

    % list.member(Elem, List, SubList):
    %
    % True iff ‘List’ contains ‘Elem’, and ‘SubList’ is a suffix of ‘List’
    % beginning with ‘Elem’.
```

```
    % Same as ‘SubList = [Elem | _], list.append(_, SubList, List)’.
    %
:- pred list.member(T::out, list(T)::in, list(T)::out) is nondet.

    % list.member_index0(Elem, List, Index).
    %
    % True iff ‘List’ contains ‘Elem’ at the zero-based index ‘Index’.
    %
:- pred list.member_index0(T, list(T), int).
:- mode list.member_index0(in, in, in) is semidet.
:- mode list.member_index0(in, in, out) is nondet.
:- mode list.member_index0(out, in, out) is nondet.

    % list.contains(List, Elem) iff list.member(Elem, List).
    % Sometimes you need the arguments in this order, because you want to
    % construct a closure with only the list.
    %
:- pred list.contains(list(T)::in, T::in) is semidet.

    % list.length(List, Length):
    %
    % True iff ‘Length’ is the length of ‘List’, i.e. if ‘List’ contains
    % ‘Length’ elements.
    %
:- pred list.length(list(_T), int).
:- mode list.length(in, out) is det.
    % XXX The current mode checker can’t handle this mode
% :- mode list.length(input_list_skel, out) is det.

:- func list.length(list(T)) = int.

    % list.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 list.same_length(list(T1), list(T2)).
    % XXX The current mode checker can’t handle these modes.
% :- mode list.same_length(in, output_list_skel) is det.
% :- mode list.same_length(output_list_skel, in) is det.
:- mode list.same_length(in, in) is semidet.
% XXX The current mode checker can’t handle these modes
% :- mode list.same_length(input_list_skel, output_list_skel) is det.
% :- mode list.same_length(output_list_skel, input_list_skel) is det.

    % As above, but for three lists.
    %
```

```
:- pred list.same_length3(list(T1)::in, list(T2)::in, list(T3)::in)
    is semidet.

    % list.split_list(Len, List, Start, End):
    %
    % splits ‘List’ into a prefix ‘Start’ of length ‘Len’, and a remainder
    % ‘End’. See also: list.take, list.drop and list.split_upto.
    %
:- pred list.split_list(int::in, list(T)::in, list(T)::out, list(T)::out)
    is semidet.

    % list.det_split_list(Len, List, Start, End):
    %
    % A deterministic version of list.split_list, which aborts instead
    % of failing if Len > list.length(List).
    %
:- pred list.det_split_list(int::in, list(T)::in, list(T)::out, list(T)::out)
    is det.

    % list.split_upto(Len, List, Start, End):
    %
    % splits ‘List’ into a prefix ‘Start’ of length ‘min(Len, length(List))’,
    % and a remainder ‘End’. See also: list.split_list, list.take, list.drop.
    %
:- pred list.split_upto(int::in, list(T)::in, list(T)::out, list(T)::out)
    is det.

    % list.take(Len, List, Start):
    %
    % ‘Start’ is the first ‘Len’ elements of ‘List’. Fails if ‘List’ has
    % less than ‘Len’ elements. See also: list.split_list.
    %
:- pred list.take(int::in, list(T)::in, list(T)::out) is semidet.

    % list.det_take(Len, List, Start):
    %
    % As above, but throw an exception instead of failing.
    %
:- pred list.det_take(int::in, list(T)::in, list(T)::out) is det.

    % list.take_upto(Len, List, Start):
    %
    % ‘Start’ is the first ‘Len’ elements of ‘List’. If ‘List’ has less than
    % ‘Len’ elements, return the entire list.
    %
:- pred list.take_upto(int::in, list(T)::in, list(T)::out) is det.
:- func list.take_upto(int, list(T)) = list(T).
```

```
    % list.drop(Len, List, End):
    %
    % 'End' is the remainder of 'List' after removing the first 'Len' elements.
    % Fails if 'List' does not have at least 'Len' elements.
    % See also: list.split_list.
    %
:- pred list.drop(int::in, list(T)::in, list(T)::out) is semidet.

    % list.det_drop(Len, List, End):
    %
    % 'End' is the remainder of 'List' after removing the first 'Len' elements.
    % Aborts if 'List' does not have at least 'Len' elements.
    % See also: list.split_list.
    %
:- pred list.det_drop(int::in, list(T)::in, list(T)::out) is det.

    % list.insert(Elem, List0, List):
    %
    % 'List' is the result of inserting 'Elem' somewhere in 'List0'.
    % Same as 'list.delete(List, Elem, List0)'.
    %
:- pred list.insert(T, list(T), list(T)).
:- mode list.insert(in, in, in) is semidet.
:- mode list.insert(in, out, in) is nondet.
:- mode list.insert(out, out, in) is nondet.
:- mode list.insert(in, in, out) is multi.

    % list.delete(List, Elem, Remainder):
    %
    % True iff 'Elem' occurs in 'List', and 'Remainder' is the result of
    % deleting one occurrence of 'Elem' from 'List'.
    %
:- pred list.delete(list(T), T, list(T)).
:- mode list.delete(in, in, in) is semidet.
:- mode list.delete(in, in, out) is nondet.
:- mode list.delete(in, out, out) is nondet.
:- mode list.delete(out, in, in) is multi.

:- func list.delete_all(list(T), T) = list(T).

    % list.delete_first(List0, Elem, List) is true iff Elem occurs in List0
    % and List is List0 with the first occurrence of Elem removed.
    %
:- pred list.delete_first(list(T)::in, T::in, list(T)::out) is semidet.

    % list.delete_all(List0, Elem, List) is true iff List is List0 with
```

```
    % all occurrences of Elem removed.
    %
:- pred list.delete_all(list(T), T, list(T)).
:- mode list.delete_all(di, in, uo) is det.
:- mode list.delete_all(in, in, out) is det.

    % list.delete_elems(List0, Elems, List) is true iff List is List0 with
    % all occurrences of all elements of Elems removed.
    %
:- pred list.delete_elems(list(T)::in, list(T)::in, list(T)::out) is det.
:- func list.delete_elems(list(T), list(T)) = list(T).

    % list.replace(List0, D, R, List) is true iff List is List0
    % with an occurrence of D replaced with R.
    %
:- pred list.replace(list(T), T, T, list(T)).
:- mode list.replace(in, in, in, in) is semidet.
:- mode list.replace(in, in, in, out) is nondet.

    % list.replace_first(List0, D, R, List) is true iff List is List0
    % with the first occurrence of D replaced with R.
    %
:- pred list.replace_first(list(T)::in, T::in, T::in, list(T)::out)
    is semidet.

    % list.replace_all(List0, D, R, List) is true iff List is List0
    % with all occurrences of D replaced with R.
    %
:- pred list.replace_all(list(T)::in, T::in, T::in, list(T)::out) is det.
:- func list.replace_all(list(T), T, T) = list(T).

    % list.replace_nth(List0, N, R, List) is true iff List is List0
    % with Nth element replaced with R.
    % Fails if N < 1 or if length of List0 < N.
    % (Position numbers start from 1.)
    %
:- pred list.replace_nth(list(T)::in, int::in, T::in, list(T)::out)
    is semidet.

    % list.det_replace_nth(List0, N, R, List) is true iff List is List0
    % with Nth element replaced with R.
    % Aborts if N < 1 or if length of List0 < N.
    % (Position numbers start from 1.)
    %
:- pred list.det_replace_nth(list(T)::in, int::in, T::in, list(T)::out) is det.
:- func list.det_replace_nth(list(T), int, T) = list(T).
```

```
    % list.sort_and_remove_dups(List0, List):
    %
    % List is List0 sorted with the second and subsequent occurrence of
    % any duplicates removed.
    %
:- pred list.sort_and_remove_dups(list(T)::in, list(T)::out) is det.
:- func list.sort_and_remove_dups(list(T)) = list(T).

    % list.sort(List0, List):
    %
    % List is List0 sorted.
    %
:- pred list.sort(list(T)::in, list(T)::out) is det.
:- func list.sort(list(T)) = list(T).

    % list.reverse(List, Reverse):
    %
    % 'Reverse' is a list containing the same elements as 'List'
    % but in reverse order.
    %
:- pred list.reverse(list(T), list(T)).
:- mode list.reverse(in, out) is det.
:- mode list.reverse(out, in) is det.

:- func list.reverse(list(T)) = list(T).

    % list.perm(List0, List):
    %
    % True iff 'List' is a permutation of 'List0'.
    %
:- pred list.perm(list(T)::in, list(T)::out) is multi.

    % list.nth_member_search(List, Elem, Position):
    %
    % Elem is the Position'th member of List.
    % (Position numbers start from 1.)
    %
:- pred list.nth_member_search(list(T)::in, T::in, int::out) is semidet.

    % A deterministic version of list.nth_member_search, which aborts
    % instead of failing if the element is not found in the list.
    %
:- pred list.nth_member_lookup(list(T)::in, T::in, int::out) is det.

    % list.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 list.index0(list(T)::in, int::in, T::out) is semidet.
:- pred list.index1(list(T)::in, int::in, T::out) is semidet.
:- pred list.det_index0(list(T)::in, int::in, T::out) is det.
:- pred list.det_index1(list(T)::in, int::in, T::out) is det.

:- func list.det_index0(list(T), int) = T.
:- func list.det_index1(list(T), int) = T.

    % list.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 list.index0_of_first_occurrence(list(T)::in, T::in, int::out)
    is semidet.
:- pred list.index1_of_first_occurrence(list(T)::in, T::in, int::out)
    is semidet.
:- func list.det_index0_of_first_occurrence(list(T), T) = int.
:- func list.det_index1_of_first_occurrence(list(T), T) = int.

    % list.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.
    %
:- pred list.zip(list(T)::in, list(T)::in, list(T)::out) is det.
:- func list.zip(list(T), list(T)) = list(T).

    % list.duplicate(Count, Elem, List) is true iff List is a list
    % containing Count duplicate copies of Elem.
    %
:- pred list.duplicate(int::in, T::in, list(T)::out) is det.
:- func list.duplicate(int, T) = list(T).

    % list.condense(ListOfLists, List):
    %
    % 'List' is the result of concatenating all the elements of 'ListOfLists'.
```

```
    %
:- pred list.condense(list(list(T))::in, list(T)::out) is det.
:- func list.condense(list(list(T))) = list(T).

    % list.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’.)
    %
:- pred list.chunk(list(T)::in, int::in, list(list(T))::out) is det.
:- func list.chunk(list(T), int) = list(list(T)).

    % list.sublist(SubList, FullList) is true if one can obtain SubList
    % by starting with FullList and deleting some of its elements.
    %
:- pred list.sublist(list(T)::in, list(T)::in) is semidet.

    % list.all_same(List) is true if all elements of the list are the same.
    %
:- pred list.all_same(list(T)::in) is semidet.

    % list.last(List, Last) is true if Last is the last element of List.
    %
:- pred list.last(list(T)::in, T::out) is semidet.

    % A deterministic version of list.last, which aborts instead of
    % failing if the input list is empty.
    %
:- pred list.det_last(list(T)::in, T::out) is det.
:- func list.det_last(list(T)) = T.

    % list.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 list.split_last(list(T)::in, list(T)::out, T::out) is semidet.

    % A deterministic version of list.split_last, which aborts instead of
    % failing if the input list is empty.
    %
:- pred list.det_split_last(list(T)::in, list(T)::out, T::out) is det.

%-----------------------------------------------%
%
% 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.
%
%-----------------------------------------------------%

    % list.map(T, L, M) uses the closure T
    % to transform the elements of L into the elements of M.
    %
:- pred list.map(pred(X, Y), list(X), list(Y)).
:- mode list.map(pred(in, out) is det, in, out) is det.
:- mode list.map(pred(in, out) is cc_multi, in, out) is cc_multi.
:- mode list.map(pred(in, out) is semidet, in, out) is semidet.
:- mode list.map(pred(in, out) is multi, in, out) is multi.
:- mode list.map(pred(in, out) is nondet, in, out) is nondet.
:- mode list.map(pred(in, in) is semidet, in, in) is semidet.

:- func list.map(func(X) = Y, list(X)) = list(Y).

    % list.map2(T, L, M1, M2) uses the closure T
    % to transform the elements of L into the elements of M1 and M2.
    %
:- pred list.map2(pred(A, B, C), list(A), list(B), list(C)).
:- mode list.map2(pred(in, out, out) is det, in, out, out) is det.
:- mode list.map2(pred(in, out, out) is cc_multi, in, out, out) is cc_multi.
:- mode list.map2(pred(in, out, out) is semidet, in, out, out) is semidet.
:- mode list.map2(pred(in, out, out) is multi, in, out, out) is multi.
:- mode list.map2(pred(in, out, out) is nondet, in, out, out) is nondet.
:- mode list.map2(pred(in, in, in) is semidet, in, in, in) is semidet.

    % list.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 list.map3(pred(A, B, C, D), list(A), list(B), list(C), list(D)).
:- mode list.map3(pred(in, out, out, out) is det, in, out, out, out) is det.
:- mode list.map3(pred(in, out, out, out) is cc_multi, in, out, out, out)
    is cc_multi.
:- mode list.map3(pred(in, out, out, out) is semidet, in, out, out, out)
    is semidet.
:- mode list.map3(pred(in, out, out, out) is multi, in, out, out, out)
    is multi.
:- mode list.map3(pred(in, out, out, out) is nondet, in, out, out, out)
    is nondet.
:- mode list.map3(pred(in, in, in, in) is semidet, in, in, in, in) is semidet.

    % list.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 list.map4(pred(A, B, C, D, E), list(A), list(B), list(C), list(D),
    list(E)).
:- mode list.map4(pred(in, out, out, out, out) is det, in, out, out, out, out)
    is det.
:- mode list.map4(pred(in, out, out, out, out) is cc_multi, in, out, out, out,
    out) is cc_multi.
:- mode list.map4(pred(in, out, out, out, out) is semidet, in, out, out, out,
    out) is semidet.
:- mode list.map4(pred(in, out, out, out, out) is multi, in, out, out, out,
    out) is multi.
:- mode list.map4(pred(in, out, out, out, out) is nondet, in, out, out, out,
    out) is nondet.
:- mode list.map4(pred(in, in, in, in, in) is semidet, in, in, in, in, in)
    is semidet.

    % list.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 list.map5(pred(A, B, C, D, E, F), list(A), list(B), list(C), list(D),
    list(E), list(F)).
:- mode list.map5(pred(in, out, out, out, out, out) is det, in, out, out, out,
    out, out) is det.
:- mode list.map5(pred(in, out, out, out, out, out) is cc_multi, in, out, out,
    out, out, out) is cc_multi.
:- mode list.map5(pred(in, out, out, out, out, out) is semidet, in, out, out,
    out, out, out) is semidet.
:- mode list.map5(pred(in, out, out, out, out, out) is multi, in, out, out,
    out, out, out) is multi.
:- mode list.map5(pred(in, out, out, out, out, out) is nondet, in, out, out,
    out, out, out) is nondet.
:- mode list.map5(pred(in, in, in, in, in, in) is semidet, in, in, in, in, in,
    in) is semidet.

    % list.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 list.map6(pred(A, B, C, D, E, F, G), list(A), list(B), list(C),
    list(D), list(E), list(F), list(G)).
:- mode list.map6(pred(in, out, out, out, out, out, out) is det, in, out, out,
    out, out, out, out) is det.
:- mode list.map6(pred(in, out, out, out, out, out, out) is cc_multi, in, out,
    out, out, out, out, out) is cc_multi.
:- mode list.map6(pred(in, out, out, out, out, out, out) is semidet, in, out,
    out, out, out, out, out) is semidet.
:- mode list.map6(pred(in, out, out, out, out, out, out) is multi, in, out,
```

```
    out, out, out, out, out) is multi.
:- mode list.map6(pred(in, out, out, out, out, out, out) is nondet, in, out,
    out, out, out, out, out) is nondet.
:- mode list.map6(pred(in, in, in, in, in, in, in) is semidet, in, in, in, in,
    in, in, in) is semidet.

    % list.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 list.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 list.map7(pred(in, out, out, out, out, out, out, out) is det,
    in, out, out, out, out, out, out, out) is det.
:- mode list.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 list.map7(pred(in, out, out, out, out, out, out, out) is semidet,
    in, out, out, out, out, out, out, out) is semidet.
:- mode list.map7(pred(in, out, out, out, out, out, out, out) is multi,
    in, out, out, out, out, out, out, out) is multi.
:- mode list.map7(pred(in, out, out, out, out, out, out, out) is nondet,
    in, out, out, out, out, out, out, out) is nondet.
:- mode list.map7(pred(in, in, in, in, in, in, in, in) is semidet,
    in, in, in, in, in, in, in, in) is semidet.

    % list.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 list.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 list.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 list.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 list.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 list.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 list.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 list.map8(pred(in, in, in, in, in, in, in, in, in) is semidet,
    in, in, in, in, in, in, in, in, in) is semidet.

    % list.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 list.map_corresponding(func(A, B) = C, list(A), list(B)) = list(C).
:- pred list.map_corresponding(pred(A, B, C), list(A), list(B), list(C)).
:- mode list.map_corresponding(in(pred(in, in, out) is det), in, in, out)
    is det.
:- mode list.map_corresponding(in(pred(in, in, out) is semidet), in, in, out)
    is semidet.

    % list.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 list.map_corresponding3(func(A, B, C) = D, list(A), list(B), list(C))
    = list(D).

    % list.filter_map_corresponding/3 is like list.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 list.filter_map_corresponding(func(A, B) = C, list(A), list(B))
    = list(C).
:- mode list.filter_map_corresponding(func(in, in) = out is semidet, in, in)
    = out is det.

    % list.filter_map_corresponding3/4 is like list.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 list.filter_map_corresponding3(func(A, B, C) = D,
    list(A), list(B), list(C)) = list(D).
:- mode list.filter_map_corresponding3(func(in, in, in) = out is semidet,
    in, in, in) = out is det.

    % list.map_corresponding_foldl/6 is like list.map_corresponding except
    % that it has an accumulator threaded through it.
    %
:- pred list.map_corresponding_foldl(pred(A, B, C, D, D),
    list(A), list(B), list(C), D, D).
:- mode list.map_corresponding_foldl(pred(in, in, out, in, out) is det,
    in, in, out, in, out) is det.
:- mode list.map_corresponding_foldl(pred(in, in, out, mdi, muo) is det,
    in, in, out, mdi, muo) is det.
:- mode list.map_corresponding_foldl(pred(in, in, out, di, uo) is det,
```

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

    % Like list.map_corresponding_foldl/6 except that it has two
    % accumulators.
    %
:- pred list.map_corresponding_foldl2(pred(A, B, C, D, D, E, E),
    list(A), list(B), list(C), D, D, E, E).
:- mode list.map_corresponding_foldl2(
    pred(in, in, out, in, out, in, out) is det, in, in, out, in, out,
    in, out) is det.
:- mode list.map_corresponding_foldl2(
    pred(in, in, out, in, out, mdi, muo) is det, in, in, out, in, out,
    mdi, muo) is det.
:- mode list.map_corresponding_foldl2(
    pred(in, in, out, in, out, di, uo) is det, in, in, out, in, out,
    di, uo) is det.
:- mode list.map_corresponding_foldl2(
    pred(in, in, out, in, out, in, out) is semidet, in, in, out, in, out,
    in, out) is semidet.
:- mode list.map_corresponding_foldl2(
    pred(in, in, out, in, out, mdi, muo) is semidet, in, in, out, in, out,
    mdi, muo) is semidet.
:- mode list.map_corresponding_foldl2(
    pred(in, in, out, in, out, di, uo) is semidet, in, in, out, in, out,
    di, uo) is semidet.

    % Like list.map_corresponding_foldl/6 except that it has three
    % accumulators.
    %
:- pred list.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 list.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 list.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 list.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 list.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 list.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 list.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.

    % list.map_corresponding3_foldl/7 is like list.map_corresponding3 except
    % that it has an accumulator threaded through it.
    %
:- pred list.map_corresponding3_foldl(pred(A, B, C, D, E, E),
    list(A), list(B), list(C), list(D), E, E).
:- mode list.map_corresponding3_foldl(pred(in, in, in, out, in, out) is det,
    in, in, in, out, in, out) is det.
:- mode list.map_corresponding3_foldl(pred(in, in, in, out, mdi, muo) is det,
    in, in, in, out, mdi, muo) is det.
:- mode list.map_corresponding3_foldl(pred(in, in, in, out, di, uo) is det,
    in, in, in, out, di, uo) is det.
:- mode list.map_corresponding3_foldl(
    pred(in, in, in, out, in, out) is semidet,
    in, in, in, out, in, out) is semidet.
:- mode list.map_corresponding3_foldl(
    pred(in, in, in, out, mdi, muo) is semidet,
    in, in, in, out, mdi, muo) is semidet.
:- mode list.map_corresponding3_foldl(
    pred(in, in, in, out, di, uo) is semidet,
    in, in, in, out, di, uo) is semidet.

    % list.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 list.foldl(pred(L, A, A), list(L), A, A).
:- mode list.foldl(pred(in, in, out) is det, in, in, out) is det.
:- mode list.foldl(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode list.foldl(pred(in, di, uo) is det, in, di, uo) is det.
:- mode list.foldl(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode list.foldl(pred(in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode list.foldl(pred(in, di, uo) is semidet, in, di, uo) is semidet.
:- mode list.foldl(pred(in, in, out) is multi, in, in, out) is multi.
:- mode list.foldl(pred(in, in, out) is nondet, in, in, out) is nondet.
:- mode list.foldl(pred(in, mdi, muo) is nondet, in, mdi, muo) is nondet.
:- mode list.foldl(pred(in, in, out) is cc_multi, in, in, out) is cc_multi.
:- mode list.foldl(pred(in, di, uo) is cc_multi, in, di, uo) is cc_multi.
```

```
:- func list.foldl(func(L, A) = A, list(L), A) = A.

    % list.foldl2(Pred, List, !Acc1, !Acc2):
    % Does the same job as list.foldl, but with two accumulators.
    % (Although no more expressive than list.foldl, this is often
    % a more convenient format, and a little more efficient).
    %
:- pred list.foldl2(pred(L, A, A, Z, Z), list(L), A, A, Z, Z).
:- mode list.foldl2(pred(in, in, out, in, out) is det,
    in, in, out, in, out) is det.
:- mode list.foldl2(pred(in, in, out, mdi, muo) is det,
    in, in, out, mdi, muo) is det.
:- mode list.foldl2(pred(in, in, out, di, uo) is det,
    in, in, out, di, uo) is det.
:- mode list.foldl2(pred(in, di, uo, di, uo) is det,
    in, di, uo, di, uo) is det.
:- mode list.foldl2(pred(in, in, out, in, out) is semidet,
    in, in, out, in, out) is semidet.
:- mode list.foldl2(pred(in, in, out, mdi, muo) is semidet,
    in, in, out, mdi, muo) is semidet.
:- mode list.foldl2(pred(in, in, out, di, uo) is semidet,
    in, in, out, di, uo) is semidet.
:- mode list.foldl2(pred(in, in, out, in, out) is nondet,
    in, in, out, in, out) is nondet.
:- mode list.foldl2(pred(in, in, out, mdi, muo) is nondet,
    in, in, out, mdi, muo) is nondet.
:- mode list.foldl2(pred(in, in, out, in, out) is cc_multi,
    in, in, out, in, out) is cc_multi.
:- mode list.foldl2(pred(in, in, out, mdi, muo) is cc_multi,
    in, in, out, mdi, muo) is cc_multi.
:- mode list.foldl2(pred(in, in, out, di, uo) is cc_multi,
    in, in, out, di, uo) is cc_multi.
:- mode list.foldl2(pred(in, di, uo, di, uo) is cc_multi,
    in, di, uo, di, uo) is cc_multi.

    % list.foldl3(Pred, List, !Acc1, !Acc2, !Acc3):
    % Does the same job as list.foldl, but with three accumulators.
    % (Although no more expressive than list.foldl, this is often
    % a more convenient format, and a little more efficient).
    %
:- pred list.foldl3(pred(L, A, A, B, B, C, C), list(L),
    A, A, B, B, C, C).
:- mode list.foldl3(pred(in, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out) is det.
:- mode list.foldl3(pred(in, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, mdi, muo) is det.
```

```
:- mode list.foldl3(pred(in, in, out, in, out, di, uo) is det,
    in, in, out, in, out, di, uo) is det.
:- mode list.foldl3(pred(in, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out) is semidet.
:- mode list.foldl3(pred(in, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, mdi, muo) is semidet.
:- mode list.foldl3(pred(in, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, di, uo) is semidet.
:- mode list.foldl3(pred(in, in, out, in, out, in, out) is nondet,
    in, in, out, in, out, in, out) is nondet.
:- mode list.foldl3(pred(in, in, out, in, out, mdi, muo) is nondet,
    in, in, out, in, out, mdi, muo) is nondet.
:- mode list.foldl3(pred(in, in, out, in, out, in, out) is cc_multi,
    in, in, out, in, out, in, out) is cc_multi.
:- mode list.foldl3(pred(in, in, out, in, out, di, uo) is cc_multi,
    in, in, out, in, out, di, uo) is cc_multi.

    % list.foldl4(Pred, List, !Acc1, !Acc2, !Acc3, !Acc4):
    % Does the same job as list.foldl, but with four accumulators.
    % (Although no more expressive than list.foldl, this is often
    % a more convenient format, and a little more efficient).
    %
:- pred list.foldl4(pred(L, A, A, B, B, C, C, D, D), list(L),
    A, A, B, B, C, C, D, D).
:- mode list.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 list.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 list.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 list.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 list.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 list.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 list.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 list.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 list.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 list.foldl4(pred(in, in, out, in, out, in, out, mdi, muo) is nondet,
    in, in, out, in, out, in, out, mdi, muo) is nondet.

    % list.foldl5(Pred, List, !Acc1, !Acc2, !Acc3, !Acc4, !Acc5):
    % Does the same job as list.foldl, but with five accumulators.
```

```
    % (Although no more expressive than list.foldl, this is often
    % a more convenient format, and a little more efficient).
    %
:- pred list.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 list.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 list.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 list.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 list.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 list.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 list.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 list.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 list.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 list.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 list.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.

    % list.foldl6(Pred, List, !Acc1, !Acc2, !Acc3, !Acc4, !Acc5, !Acc6):
    % Does the same job as list.foldl, but with six accumulators.
    % (Although no more expressive than list.foldl, this is often
    % a more convenient format, and a little more efficient).
    %
:- pred list.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 list.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 list.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 list.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 list.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 list.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 list.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 list.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 list.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 list.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.

    % list.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 list.foldr(pred(L, A, A), list(L), A, A).
:- mode list.foldr(pred(in, in, out) is det, in, in, out) is det.
:- mode list.foldr(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode list.foldr(pred(in, di, uo) is det, in, di, uo) is det.
:- mode list.foldr(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode list.foldr(pred(in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode list.foldr(pred(in, di, uo) is semidet, in, di, uo) is semidet.
:- mode list.foldr(pred(in, in, out) is multi, in, in, out) is multi.
:- mode list.foldr(pred(in, in, out) is nondet, in, in, out) is nondet.
:- mode list.foldr(pred(in, mdi, muo) is nondet, in, mdi, muo) is nondet.
:- mode list.foldr(pred(in, di, uo) is cc_multi, in, di, uo) is cc_multi.
:- mode list.foldr(pred(in, in, out) is cc_multi, in, in, out) is cc_multi.

:- func list.foldr(func(L, A) = A, list(L), A) = A.

    % list.foldr2(Pred, List, !Acc1, !Acc2):
    % Does the same job as list.foldr, but with two accumulators.
    % (Although no more expressive than list.foldl, this is often
```

```
    % a more convenient format, and a little more efficient).
    %
:- pred list.foldr2(pred(L, A, A, B, B), list(L), A, A, B, B).
:- mode list.foldr2(pred(in, in, out, in, out) is det, in, in, out,
    in, out) is det.
:- mode list.foldr2(pred(in, in, out, mdi, muo) is det, in, in, out,
    mdi, muo) is det.
:- mode list.foldr2(pred(in, in, out, di, uo) is det, in, in, out,
    di, uo) is det.
:- mode list.foldr2(pred(in, in, out, in, out) is semidet, in, in, out,
    in, out) is semidet.
:- mode list.foldr2(pred(in, in, out, mdi, muo) is semidet, in, in, out,
    mdi, muo) is semidet.
:- mode list.foldr2(pred(in, in, out, di, uo) is semidet, in, in, out,
    di, uo) is semidet.
:- mode list.foldr2(pred(in, in, out, in, out) is nondet, in, in, out,
    in, out) is nondet.
:- mode list.foldr2(pred(in, in, out, mdi, muo) is nondet, in, in, out,
    mdi, muo) is nondet.

    % list.foldr3(Pred, List, !Acc1, !Acc2, !Acc3):
    % Does the same job as list.foldr, but with two accumulators.
    % (Although no more expressive than list.foldl, this is often
    % a more convenient format, and a little more efficient).
    %
:- pred list.foldr3(pred(L, A, A, B, B, C, C), list(L), A, A, B, B, C, C).
:- mode list.foldr3(pred(in, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out) is det.
:- mode list.foldr3(pred(in, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, mdi, muo) is det.
:- mode list.foldr3(pred(in, in, out, in, out, di, uo) is det, in,
    in, out, in, out, di, uo) is det.
:- mode list.foldr3(pred(in, in, out, in, out, in, out) is semidet, in,
    in, out, in, out, in, out) is semidet.
:- mode list.foldr3(pred(in, in, out, in, out, mdi, muo) is semidet, in,
    in, out, in, out, mdi, muo) is semidet.
:- mode list.foldr3(pred(in, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, di, uo) is semidet.
:- mode list.foldr3(pred(in, in, out, in, out, in, out) is nondet, in,
    in, out, in, out, in, out) is nondet.
:- mode list.foldr3(pred(in, in, out, in, out, mdi, muo) is nondet, in,
    in, out, in, out, mdi, muo) is nondet.

    % list.foldl_corresponding(P, As, Bs, !Acc):
    % Does the same job as list.foldl, but works on two lists in
    % parallel.  An exception is raised if the list arguments differ
    % in length.
```

```
    %
:- pred list.foldl_corresponding(pred(A, B, C, C), list(A), list(B), C, C).
:- mode list.foldl_corresponding(pred(in, in, in, out) is det,
    in, in, in, out) is det.
:- mode list.foldl_corresponding(pred(in, in, mdi, muo) is det,
    in, in, mdi, muo) is det.
:- mode list.foldl_corresponding(pred(in, in, di, uo) is det,
    in, in, di, uo) is det.
:- mode list.foldl_corresponding(pred(in, in, in, out) is semidet,
    in, in, in, out) is semidet.
:- mode list.foldl_corresponding(pred(in, in, mdi, muo) is semidet,
    in, in, mdi, muo) is semidet.
:- mode list.foldl_corresponding(pred(in, in, di, uo) is semidet,
    in, in, di, uo) is semidet.
:- mode list.foldl_corresponding(pred(in, in, in, out) is nondet,
    in, in, in, out) is nondet.
:- mode list.foldl_corresponding(pred(in, in, mdi, muo) is nondet,
    in, in, mdi, muo) is nondet.
:- mode list.foldl_corresponding(pred(in, in, in, out) is cc_multi,
    in, in, in, out) is cc_multi.
:- mode list.foldl_corresponding(pred(in, in, di, uo) is cc_multi,
    in, in, di, uo) is cc_multi.

:- func list.foldl_corresponding(func(A, B, C) = C, list(A), list(B), C) = C.

    % list.foldl2_corresponding(F, As, Bs, !Acc1, !Acc2):
    % Does the same job as list.foldl_corresponding, but has two
    % accumulators.
    %
:- pred list.foldl2_corresponding(pred(A, B, C, C, D, D), list(A), list(B),
    C, C, D, D).
:- mode list.foldl2_corresponding(pred(in, in, in, out, in, out) is det,
    in, in, in, out, in, out) is det.
:- mode list.foldl2_corresponding(pred(in, in, in, out, mdi, muo) is det,
    in, in, in, out, mdi, muo) is det.
:- mode list.foldl2_corresponding(pred(in, in, in, out, di, uo) is det,
    in, in, in, out, di, uo) is det.
:- mode list.foldl2_corresponding(pred(in, in, in, out, in, out) is semidet,
    in, in, in, out, in, out) is semidet.
:- mode list.foldl2_corresponding(pred(in, in, in, out, mdi, muo) is semidet,
    in, in, in, out, mdi, muo) is semidet.
:- mode list.foldl2_corresponding(pred(in, in, in, out, di, uo) is semidet,
    in, in, in, out, di, uo) is semidet.
:- mode list.foldl2_corresponding(pred(in, in, in, out, in, out) is nondet,
    in, in, in, out, in, out) is nondet.
:- mode list.foldl2_corresponding(pred(in, in, in, out, mdi, muo) is nondet,
    in, in, in, out, mdi, muo) is nondet.
```

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

    % list.foldl3_corresponding(F, As, Bs, !Acc1, !Acc2, !Acc3):
    % Does the same job as list.foldl_corresponding, but has three
    % accumulators.
    %
:- pred list.foldl3_corresponding(pred(A, B, C, C, D, D, E, E),
    list(A), list(B), C, C, D, D, E, E).
:- mode list.foldl3_corresponding(
    pred(in, in, in, out, in, out, in, out) is det, in, in, in, out,
    in, out, in, out) is det.
:- mode list.foldl3_corresponding(
    pred(in, in, in, out, in, out, mdi, muo) is det, in, in, in, out,
    in, out, mdi, muo) is det.
:- mode list.foldl3_corresponding(
    pred(in, in, in, out, in, out, di, uo) is det, in, in, in, out,
    in, out, di, uo) is det.
:- mode list.foldl3_corresponding(
    pred(in, in, in, out, in, out, in, out) is semidet, in, in, in, out,
    in, out, in, out) is semidet.
:- mode list.foldl3_corresponding(
    pred(in, in, in, out, in, out, mdi, muo) is semidet, in, in, in, out,
    in, out, mdi, muo) is semidet.
:- mode list.foldl3_corresponding(
    pred(in, in, in, out, in, out, di, uo) is semidet, in, in, in, out,
    in, out, di, uo) is semidet.

    % list.foldl_corresponding3(P, As, Bs, Cs, !Acc):
    % Like list.foldl_corresponding but folds over three corresponding
    % lists.
    %
:- pred list.foldl_corresponding3(pred(A, B, C, D, D),
    list(A), list(B), list(C), D, D).
:- mode list.foldl_corresponding3(pred(in, in, in, in, out) is det,
    in, in, in, in, out) is det.
:- mode list.foldl_corresponding3(pred(in, in, in, mdi, muo) is det,
    in, in, in, mdi, muo) is det.
:- mode list.foldl_corresponding3(pred(in, in, in, di, uo) is det,
    in, in, in, di, uo) is det.
:- mode list.foldl_corresponding3(pred(in, in, in, in, out) is semidet,
    in, in, in, in, out) is semidet.
:- mode list.foldl_corresponding3(pred(in, in, in, mdi, muo) is semidet,
    in, in, in, mdi, muo) is semidet.
:- mode list.foldl_corresponding3(pred(in, in, in, di, uo) is semidet,
```

```
    in, in, in, di, uo) is semidet.

    % list.foldl2_corresponding3(P, As, Bs, Cs, !Acc1, !Acc2):
    % like list.foldl_corresponding3 but with two accumulators.
    %
:- pred list.foldl2_corresponding3(pred(A, B, C, D, D, E, E),
    list(A), list(B), list(C), D, D, E, E).
:- mode list.foldl2_corresponding3(pred(in, in, in, in, out, in, out) is det,
    in, in, in, in, out, in, out) is det.
:- mode list.foldl2_corresponding3(pred(in, in, in, in, out, mdi, muo) is det,
    in, in, in, in, out, mdi, muo) is det.
:- mode list.foldl2_corresponding3(pred(in, in, in, in, out, di, uo) is det,
    in, in, in, in, out, di, uo) is det.
:- mode list.foldl2_corresponding3(
    pred(in, in, in, in, out, in, out) is semidet,
    in, in, in, in, out, in, out) is semidet.
:- mode list.foldl2_corresponding3(
    pred(in, in, in, in, out, mdi, muo) is semidet,
    in, in, in, in, out, mdi, muo) is semidet.
:- mode list.foldl2_corresponding3(
    pred(in, in, in, in, out, di, uo) is semidet,
    in, in, in, in, out, di, uo) is semidet.

    % list.foldl3_corresponding3(P, As, Bs, Cs, !Acc1, !Acc2, !Acc3):
    % like list.foldl_corresponding3 but with three accumulators.
    %
:- pred list.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 list.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 list.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 list.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 list.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 list.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 list.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.
```

```
    % list.foldl4_corresponding3(P, As, Bs, Cs, !Acc1, !Acc2, !Acc3, !Acc4):
    % like list.foldl_corresponding3 but with four accumulators.
    %
:- pred list.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 list.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 list.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 list.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 list.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 list.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 list.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.

    % list.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 list.map_foldl(pred(L, M, A, A), list(L), list(M), A, A).
:- mode list.map_foldl(pred(in, out, in, out) is det, in, out, in, out)
    is det.
:- mode list.map_foldl(pred(in, out, mdi, muo) is det, in, out, mdi, muo)
    is det.
:- mode list.map_foldl(pred(in, out, di, uo) is det, in, out, di, uo)
    is det.
:- mode list.map_foldl(pred(in, out, in, out) is semidet, in, out, in, out)
    is semidet.
:- mode list.map_foldl(pred(in, out, mdi, muo) is semidet, in, out, mdi, muo)
    is semidet.
:- mode list.map_foldl(pred(in, out, di, uo) is semidet, in, out, di, uo)
    is semidet.
:- mode list.map_foldl(pred(in, in, di, uo) is semidet, in, in, di, uo)
    is semidet.
:- mode list.map_foldl(pred(in, out, in, out) is nondet, in, out, in, out)
    is nondet.
```

```
:- mode list.map_foldl(pred(in, out, mdi, muo) is nondet, in, out, mdi, muo)
    is nondet.
:- mode list.map_foldl(pred(in, out, in, out) is cc_multi, in, out, in, out)
    is cc_multi.
:- mode list.map_foldl(pred(in, out, mdi, muo) is cc_multi, in, out, mdi, muo)
    is cc_multi.
:- mode list.map_foldl(pred(in, out, di, uo) is cc_multi, in, out, di, uo)
    is cc_multi.

    % Same as list.map_foldl, but with two accumulators.
    %
:- pred list.map_foldl2(pred(L, M, A, A, B, B), list(L), list(M), A, A, B, B).
:- mode list.map_foldl2(pred(in, out, in, out, in, out) is det,
    in, out, in, out, in, out) is det.
:- mode list.map_foldl2(pred(in, out, in, out, mdi, muo) is det,
    in, out, in, out, mdi, muo) is det.
:- mode list.map_foldl2(pred(in, out, in, out, di, uo) is det,
    in, out, in, out, di, uo) is det.
:- mode list.map_foldl2(pred(in, out, in, out, in, out) is semidet,
    in, out, in, out, in, out) is semidet.
:- mode list.map_foldl2(pred(in, out, in, out, mdi, muo) is semidet,
    in, out, in, out, mdi, muo) is semidet.
:- mode list.map_foldl2(pred(in, out, in, out, di, uo) is semidet,
    in, out, in, out, di, uo) is semidet.
:- mode list.map_foldl2(pred(in, in, in, out, di, uo) is semidet,
    in, in, in, out, di, uo) is semidet.
:- mode list.map_foldl2(pred(in, out, in, out, in, out) is cc_multi,
    in, out, in, out, in, out) is cc_multi.
:- mode list.map_foldl2(pred(in, out, in, out, mdi, muo) is cc_multi,
    in, out, in, out, mdi, muo) is cc_multi.
:- mode list.map_foldl2(pred(in, out, in, out, di, uo) is cc_multi,
    in, out, in, out, di, uo) is cc_multi.
:- mode list.map_foldl2(pred(in, out, in, out, in, out) is nondet,
    in, out, in, out, in, out) is nondet.

    % Same as list.map_foldl, but with three accumulators.
    %
:- pred list.map_foldl3(pred(L, M, A, A, B, B, C, C), list(L), list(M),
    A, A, B, B, C, C).
:- mode list.map_foldl3(pred(in, out, in, out, in, out, di, uo) is det,
    in, out, in, out, in, out, di, uo) is det.
:- mode list.map_foldl3(pred(in, out, in, out, in, out, in, out) is det,
    in, out, in, out, in, out, in, out) is det.
:- mode list.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 list.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 list.map_foldl3(pred(in, out, in, out, in, out, in, out) is semidet,
    in, out, in, out, in, out, in, out) is semidet.
:- mode list.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 list.map_foldl, but with four accumulators.
    %
:- pred list.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 list.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 list.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 list.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 list.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 list.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 list.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 list.map_foldl, but with five accumulators.
    %
:- pred list.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 list.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 list.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 list.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 list.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 list.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 list.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 list.map_foldl, but with six accumulators.
    %
:- pred list.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 list.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 list.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 list.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 list.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 list.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 list.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 list.map_foldl, but with two mapped outputs.
    %
:- pred list.map2_foldl(pred(L, M, N, A, A), list(L), list(M), list(N),
    A, A).
:- mode list.map2_foldl(pred(in, out, out, in, out) is det, in, out, out,
    in, out) is det.
:- mode list.map2_foldl(pred(in, out, out, mdi, muo) is det, in, out, out,
    mdi, muo) is det.
:- mode list.map2_foldl(pred(in, out, out, di, uo) is det, in, out, out,
    di, uo) is det.
:- mode list.map2_foldl(pred(in, out, out, in, out) is semidet, in, out, out,
    in, out) is semidet.
:- mode list.map2_foldl(pred(in, out, out, mdi, muo) is semidet, in, out, out,
    mdi, muo) is semidet.
:- mode list.map2_foldl(pred(in, out, out, di, uo) is semidet, in, out, out,
    di, uo) is semidet.
```

```
:- mode list.map2_foldl(pred(in, out, out, in, out) is nondet, in, out, out,
    in, out) is nondet.
:- mode list.map2_foldl(pred(in, out, out, mdi, muo) is nondet, in, out, out,
    mdi, muo) is nondet.
:- mode list.map2_foldl(pred(in, out, out, in, out) is cc_multi, in, out, out,
    in, out) is cc_multi.
:- mode list.map2_foldl(pred(in, out, out, di, uo) is cc_multi, in, out, out,
    di, uo) is cc_multi.

    % Same as list.map_foldl, but with two mapped outputs and two
    % accumulators.
    %
:- pred list.map2_foldl2(pred(L, M, N, A, A, B, B), list(L), list(M), list(N),
    A, A, B, B).
:- mode list.map2_foldl2(pred(in, out, out, in, out, di, uo) is det,
    in, out, out, in, out, di, uo) is det.
:- mode list.map2_foldl2(pred(in, out, out, in, out, in, out) is det,
    in, out, out, in, out, in, out) is det.
:- mode list.map2_foldl2(pred(in, out, out, in, out, di, uo) is cc_multi,
    in, out, out, in, out, di, uo) is cc_multi.
:- mode list.map2_foldl2(pred(in, out, out, in, out, in, out) is cc_multi,
    in, out, out, in, out, in, out) is cc_multi.
:- mode list.map2_foldl2(pred(in, out, out, in, out, in, out) is semidet,
    in, out, out, in, out, in, out) is semidet.
:- mode list.map2_foldl2(pred(in, out, out, in, out, in, out) is nondet,
    in, out, out, in, out, in, out) is nondet.

    % Same as list.map_foldl, but with two mapped outputs and three
    % accumulators.
    %
:- pred list.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 list.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 list.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 list.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 list.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 list.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 list.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 list.map_foldl, but with two mapped outputs and four
    % accumulators.
    %
:- pred list.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 list.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 list.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 list.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 list.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 list.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 list.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 list.map_foldl, but with three mapped outputs.
    %
:- pred list.map3_foldl(pred(L, M, N, O, A, A), list(L), list(M), list(N),
    list(O), A, A).
:- mode list.map3_foldl(pred(in, out, out, out, in, out) is det, in, out, out,
    out, in, out) is det.
:- mode list.map3_foldl(pred(in, out, out, out, mdi, muo) is det, in, out, out,
    out, mdi, muo) is det.
:- mode list.map3_foldl(pred(in, out, out, out, di, uo) is det, in, out, out,
    out, di, uo) is det.
:- mode list.map3_foldl(pred(in, out, out, out, in, out) is semidet, in, out,
    out, out, in, out) is semidet.
:- mode list.map3_foldl(pred(in, out, out, out, mdi, muo) is semidet, in, out,
    out, out, mdi, muo) is semidet.
:- mode list.map3_foldl(pred(in, out, out, out, di, uo) is semidet, in, out,
    out, out, di, uo) is semidet.
:- mode list.map3_foldl(pred(in, out, out, out, in, out) is nondet, in, out,
    out, out, in, out) is nondet.
:- mode list.map3_foldl(pred(in, out, out, out, mdi, muo) is nondet, in, out,
```

```
    out, out, mdi, muo) is nondet.
:- mode list.map3_foldl(pred(in, out, out, out, in, out) is cc_multi, in, out,
    out, out, in, out) is cc_multi.
:- mode list.map3_foldl(pred(in, out, out, out, di, uo) is cc_multi, in, out,
    out, out, di, uo) is cc_multi.

    % Same as list.map_foldl, but with three mapped outputs and two
    % accumulators.
    %
:- pred list.map3_foldl2(pred(L, M, N, O, A, A, B, B), list(L),
    list(M), list(N), list(O), A, A, B, B).
:- mode list.map3_foldl2(pred(in, out, out, out, in, out, di, uo) is det,
    in, out, out, out, in, out, di, uo) is det.
:- mode list.map3_foldl2(pred(in, out, out, out, in, out, in, out) is det,
    in, out, out, out, in, out, in, out) is det.
:- mode list.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 list.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 list.map3_foldl2(pred(in, out, out, out, in, out, in, out) is semidet,
    in, out, out, out, in, out, in, out) is semidet.
:- mode list.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 list.map_foldl, but with four mapped outputs.
    %
:- pred list.map4_foldl(pred(L, M, N, O, P, A, A), list(L), list(M), list(N),
    list(O), list(P), A, A).
:- mode list.map4_foldl(pred(in, out, out, out, out, in, out) is det,
    in, out, out, out, out, in, out) is det.
:- mode list.map4_foldl(pred(in, out, out, out, out, mdi, muo) is det,
    in, out, out, out, out, mdi, muo) is det.
:- mode list.map4_foldl(pred(in, out, out, out, out, di, uo) is det,
    in, out, out, out, out, di, uo) is det.
:- mode list.map4_foldl(pred(in, out, out, out, out, in, out) is semidet,
    in, out, out, out, out, in, out) is semidet.
:- mode list.map4_foldl(pred(in, out, out, out, out, mdi, muo) is semidet,
    in, out, out, out, out, mdi, muo) is semidet.
:- mode list.map4_foldl(pred(in, out, out, out, out, di, uo) is semidet,
    in, out, out, out, out, di, uo) is semidet.
:- mode list.map4_foldl(pred(in, out, out, out, out, in, out) is nondet,
    in, out, out, out, out, in, out) is nondet.
:- mode list.map4_foldl(pred(in, out, out, out, out, mdi, muo) is nondet,
    in, out, out, out, out, mdi, muo) is nondet.
:- mode list.map4_foldl(pred(in, out, out, out, out, in, out) is cc_multi,
    in, out, out, out, out, in, out) is cc_multi.
:- mode list.map4_foldl(pred(in, out, out, out, out, di, uo) is cc_multi,
```

```
    in, out, out, out, out, di, uo) is cc_multi.

    % list.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 list.map_foldr(pred(L, M, A, A), list(L), list(M), A, A).
:- mode list.map_foldr(pred(in, out, in, out) is det, in, out, in, out)
    is det.
:- mode list.map_foldr(pred(in, out, mdi, muo) is det, in, out, mdi, muo)
    is det.
:- mode list.map_foldr(pred(in, out, di, uo) is det, in, out, di, uo)
    is det.
:- mode list.map_foldr(pred(in, out, in, out) is semidet, in, out, in, out)
    is semidet.
:- mode list.map_foldr(pred(in, out, mdi, muo) is semidet, in, out, mdi, muo)
    is semidet.
:- mode list.map_foldr(pred(in, out, di, uo) is semidet, in, out, di, uo)
    is semidet.
:- mode list.map_foldr(pred(in, in, di, uo) is semidet, in, in, di, uo)
    is semidet.

    % list.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 list.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.

    % list.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 list.all_true(pred(X)::in(pred(in) is semidet), list(X)::in)
    is semidet.

    % list.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 list.all_true_corresponding(pred(X, Y)::in(pred(in, in) is semidet),
```

```
    list(X)::in, list(Y)::in) is semidet.

    % list.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 list.all_false(pred(X)::in(pred(in) is semidet), list(X)::in)
    is semidet.

    % list.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 list.all_false_corresponding(pred(X, Y)::in(pred(in, in) is semidet),
    list(X)::in, list(Y)::in) is semidet.

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

    % list.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.
    %
:- pred list.filter(pred(X)::in(pred(in) is semidet), list(X)::in,
    list(X)::out) is det.
:- func list.filter(pred(X)::in(pred(in) is semidet), list(X)::in)
    = (list(X)::out) is det.

    % list.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 true.
    %
:- pred list.negated_filter(pred(X)::in(pred(in) is semidet), list(X)::in,
    list(X)::out) is det.
:- func list.negated_filter(pred(X)::in(pred(in) is semidet), list(X)::in)
    = (list(X)::out) is det.

    % list.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 true.
    %
:- pred list.filter(pred(X)::in(pred(in) is semidet), list(X)::in,
    list(X)::out, list(X)::out) is det.

    % list.filter_map(Transformer, List, TrueList) 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.
    %
:- pred list.filter_map(pred(X, Y)::in(pred(in, out) is semidet),
    list(X)::in, list(Y)::out) is det.

:- func list.filter_map(func(X) = Y, list(X)) = list(Y).
:- mode list.filter_map(func(in) = out is semidet, in) = out is det.

    % list.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 list.filter_map(pred(X, Y)::in(pred(in, out) is semidet),
    list(X)::in, list(Y)::out, list(X)::out) is det.

    % Same as list.filter_map/3 except that it only returns the first
    % match:
    %   find_first_map(X, Y, Z) <=> list.filter_map(X, Y, [Z | _])
    %
:- pred list.find_first_map(pred(X, Y)::in(pred(in, out) is semidet),
    list(X)::in, Y::out) is semidet.

    % Same as list.find_first_map, except with two outputs.
    %
:- pred list.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 list.find_first_map, except with three outputs.
    %
:- pred list.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 an item in the list for which Match is true where the
    % first element in the list has the index Index0.
```

```
    %
:- pred list.find_index_of_match(pred(T), list(T), int, int).
:- mode list.find_index_of_match(pred(in) is semidet, in, in, out) is semidet.

    % list.takewhile(Predicate, List, UptoList, AfterList) takes a
    % closure with one input argument, and calls it on successive members
    % of List as long as the calls succeed. The elements for which
    % the call succeeds are placed in UptoList and the first element for
    % which the call fails, and all the remaining elements of List are
    % placed in AfterList.
    %
:- pred list.takewhile(pred(T)::in(pred(in) is semidet), list(T)::in,
    list(T)::out, list(T)::out) is det.

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

    % list.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 the predicate
    % term 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).
    %
:- pred list.sort(comparison_pred(X)::in(comparison_pred), list(X)::in,
    list(X)::out) is det.
:- func list.sort(comparison_func(X), list(X)) = list(X).

    % list.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 list.sort_and_remove_dups(comparison_pred(X)::in(comparison_pred),
    list(X)::in, list(X)::out) is det.

    % list.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 list.remove_adjacent_dups(comparison_pred(X)::in(comparison_pred),
    list(X)::in, list(X)::out) is det.

    % list.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.
    %
:- pred list.merge(comparison_pred(X)::in(comparison_pred),
    list(X)::in, list(X)::in, list(X)::out) is det.

:- func list.merge(comparison_func(X), list(X), list(X)) = list(X).

    % list.merge_and_remove_dups(P, 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.
    %
:- pred list.merge_and_remove_dups(comparison_pred(X)::in(comparison_pred),
    list(X)::in, list(X)::in, list(X)::out) is det.

:- func list.merge_and_remove_dups(comparison_func(X), list(X), list(X))
    = list(X).

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

    % list.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 list.series(T, pred(T), func(T) = T) = list(T).
:- mode list.series(in, pred(in) is semidet, func(in) = out is det) = out
    is det.

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

    % Lo `..` Hi = [Lo, Lo + 1, ..., Hi] if Lo =< Hi
    %            =                    [] otherwise
    %
:- func int `..` int = list(int).

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

:- func list.head(list(T)) = T is semidet.
```

```
:- func list.tail(list(T)) = list(T) is semidet.

    % list.det_head(List) returns the first element of List,
    % calling error/1 if List is empty.
    %
:- func list.det_head(list(T)) = T.

    % list.det_tail(List) returns the tail of List,
    % calling error/1 if List is empty.
    %
:- func list.det_tail(list(T)) = list(T).

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

    % Convert a list to a pretty_printer.doc for formatting.
    %
:- func list_to_doc(list(T)) = pretty_printer.doc.

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

40 map

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 1993-2012 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% File: map.m.
% Main author: fjh, conway.
% Stability: high.
%
% This file provides the '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.
%
% The implementation is using balanced binary 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.  Therefore, two maps with the same set of key-value pairs may have
% different internal representations.  This means that two maps with the
% same set of key-value pairs that may fail to unify and may compare as
% unequal, for example if items were inserted into one of the maps in a
% different order.  See map.equal/2 below which can be used to test if two
% maps have the same set of key-value pairs.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module map.
:- interface.

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

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

:- type map(_K, _V).

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

    % Initialize an empty map.
    %
:- pred map.init(map(_, _)::uo) is det.
:- func map.init = (map(K, V)::uo) is det.

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

    % Check whether a map is empty.
    %
:- pred map.is_empty(map(_, _)::in) is semidet.

    % 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 map.equal(map(K, V)::in, map(K, V)::in) is semidet.
```

```
    % Check whether map contains key
    %
:- pred map.contains(map(K, _V)::in, K::in) is semidet.

:- pred map.member(map(K, V)::in, K::out, V::out) is nondet.

    % Search map for key.
    %
:- func map.search(map(K, V), K) = V is semidet.
:- pred map.search(map(K, V)::in, K::in, V::out) is semidet.

    % Search map for key, but abort if search fails.
    %
:- func map.lookup(map(K, V), K) = V.
:- pred map.lookup(map(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 map.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.
    % Aborts if there is no key with the given or lower value.
    %
:- pred map.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 map.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.
    % Aborts if there is no key with the given or higher value.
    %
:- pred map.upper_bound_lookup(map(K, V)::in, K::in, K::out, V::out) is det.

    % Return the largest key in the map, if there is one.
    %
:- func map.max_key(map(K, V)) = K is semidet.

    % As above, but abort if there is no largest key.
```

```
    %
:- func map.det_max_key(map(K, V)) = K.

    % Return the smallest key in the map, if there is one.
    %
:- func map.min_key(map(K,V)) = K is semidet.

    % As above, but abort if there is no smallest key.
    %
:- func map.det_min_key(map(K, V)) = K.

    % Search map for data.
    %
:- pred map.inverse_search(map(K, V)::in, V::in, K::out) is nondet.

    % Insert a new key and corresponding value into a map.
    % Fail if the key already exists.
    %
:- func map.insert(map(K, V), K, V) = map(K, V) is semidet.
:- pred map.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.
    % Abort if the key already exists.
    %
:- func map.det_insert(map(K, V), K, V) = map(K, V).
:- pred map.det_insert(K::in, V::in, map(K, V)::in, map(K, V)::out) is det.

    % Apply map.det_insert to key - value pairs from corresponding lists.
    %
:- func map.det_insert_from_corresponding_lists(map(K, V), list(K), list(V))
    = map(K, V).
:- pred map.det_insert_from_corresponding_lists(list(K)::in,
    list(V)::in, map(K, V)::in, map(K, V)::out) is det.

    % Apply map.det_insert to key - value pairs from the assoc_lists.
    %
:- func map.det_insert_from_assoc_list(map(K, V), assoc_list(K, V))
    = map(K, V).
:- pred map.det_insert_from_assoc_list(assoc_list(K, V)::in,
    map(K, V)::in, map(K, V)::out) is det.

    % Apply map.set to key - value pairs from corresponding lists.
    %
:- func map.set_from_corresponding_lists(map(K, V), list(K), list(V))
    = map(K, V).
:- pred map.set_from_corresponding_lists(list(K)::in, list(V)::in,
    map(K, V)::in, map(K, V)::out) is det.
```

```
:- func map.set_from_assoc_list(map(K, V), assoc_list(K, V)) = map(K, V).
:- pred map.set_from_assoc_list(assoc_list(K, V)::in,
    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 map.update(map(K, V), K, V) = map(K, V) is semidet.
:- pred map.update(K::in, V::in, map(K, V)::in, map(K, V)::out) is semidet.

    % Update the value corresponding to a given key
    % Abort if the key doesn't already exist.
    %
:- func map.det_update(map(K, V), K, V) = map(K, V).
:- pred map.det_update(K::in, V::in, map(K, V)::in, map(K, V)::out) is det.

    % map.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.
    %
:- pred map.search_insert(K::in, V::in, maybe(V)::out,
    map(K, V)::in, map(K, V)::out) is det.

    % 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 map.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 aborts instead of failing if the
    % key is not found.
    %
:- func map.det_transform_value(func(V) = V, K, map(K, V)) = map(K, V).
:- pred map.det_transform_value(pred(V, V)::in(pred(in, out) is det), K::in,
    map(K, V)::in, map(K, V)::out) is det.

    % Update value if the key is already present, otherwise
    % insert new key and value.
    %
:- func map.set(map(K, V), K, V) = map(K, V).
:- pred map.set(K::in, V::in, map(K, V)::in, map(K, V)::out) is det.

    % Given a map, return a list of all the keys in the map.
```

```
    %
:- func map.keys(map(K, _V)) = list(K).
:- pred map.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 map.sorted_keys(map(K, _V)) = list(K).
:- pred map.sorted_keys(map(K, _V)::in, list(K)::out) is det.

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

:- pred map.keys_and_values(map(K, V)::in, list(K)::out, list(V)::out) is det.

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

    % Convert an association list to a map.
    %
:- func map.from_assoc_list(assoc_list(K, V)) = map(K, V).
:- pred map.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 map.from_sorted_assoc_list(assoc_list(K, V)) = map(K, V).
:- pred map.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 map.from_rev_sorted_assoc_list(assoc_list(K, V)) = map(K, V).
:- pred map.from_rev_sorted_assoc_list(assoc_list(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 map.delete(map(K, V), K) = map(K, V).
:- pred map.delete(K::in, map(K, V)::in, map(K, V)::out) is det.

    % Apply map.delete/3 to a list of keys.
    %
:- func map.delete_list(map(K, V), list(K)) = map(K, V).
:- pred map.delete_list(list(K)::in, map(K, V)::in, map(K, V)::out) is det.

    % Apply map.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 result will be either an abort or incorrect output.)
    %
:- func map.delete_sorted_list(map(K, V), list(K)) = map(K, V).
:- pred map.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 map.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.
    % Abort if the key is not present.
    %
:- pred map.det_remove(K::in, V::out, map(K, V)::in, map(K, V)::out) is det.

    % Count the number of elements in the map.
    %
:- func map.count(map(K, V)) = int.
:- pred map.count(map(K, V)::in, int::out) is det.

    % Convert a pair of lists (which must be of the same length) to a map.
    %
:- func map.from_corresponding_lists(list(K), list(V)) = map(K, V).
:- pred map.from_corresponding_lists(list(K)::in, list(V)::in, map(K, V)::out)
    is det.

    % 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 map.merge(map(K, V), map(K, V)) = map(K, V).
:- pred map.merge(map(K, V)::in, map(K, V)::in, map(K, V)::out) is det.
```

```
    % For map.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 map.overlay(map(K, V), map(K, V)) = map(K, V).
:- pred map.overlay(map(K, V)::in, map(K, V)::in, map(K, V)::out) is det.

    % map.overlay_large_map(MapA, MapB, Map) performs the same task as
    % map.overlay(MapA, MapB, Map). However, while map.overlay takes time
    % proportional to the size of MapB, map.overlay_large_map takes time
    % proportional to the size of MapA. In other words, it preferable when
    % MapB is a large map.
    %
:- func map.overlay_large_map(map(K, V), map(K, V)) = map(K, V).
:- pred map.overlay_large_map(map(K, V)::in, map(K, V)::in, map(K, V)::out)
    is det.

    % map.select takes a map and a set of keys, and returns a map
    % containing the keys in the set and their corresponding values.
    %
:- func map.select(map(K, V), set(K)) = map(K, V).
:- pred map.select(map(K, V)::in, set(K)::in, map(K, V)::out) is det.

    % map.select_sorted_list takes a map and a sorted list of keys, and returns
    % a map containing the keys in the list and their corresponding values.
    %
:- func map.select_sorted_list(map(K, V), list(K)) = map(K, V).
:- pred map.select_sorted_list(map(K, V)::in, list(K)::in, map(K, V)::out)
    is det.

    % Given a list of keys, produce a list of their corresponding
    % values in a specified map.
    %
:- func map.apply_to_list(list(K), map(K, V)) = list(V).
:- pred map.apply_to_list(list(K)::in, map(K, V)::in, list(V)::out) is det.

    % Declaratively, a NOP.
    % 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.
    %
:- func map.optimize(map(K, V)) = map(K, V).
:- pred map.optimize(map(K, V)::in, map(K, V)::out) is det.

    % Remove the smallest item from the map, fail if the map is empty.
    %
```

```
:- pred map.remove_smallest(K::out, V::out, map(K, V)::in, map(K, V)::out)
    is semidet.

    % Perform an inorder traversal of the map, applying
    % an accumulator predicate for each key-value pair.
    %
:- func map.foldl(func(K, V, A) = A, map(K, V), A) = A.
:- pred map.foldl(pred(K, V, A, A), map(K, V), A, A).
:- mode map.foldl(pred(in, in, in, out) is det, in, in, out) is det.
:- mode map.foldl(pred(in, in, mdi, muo) is det, in, mdi, muo) is det.
:- mode map.foldl(pred(in, in, di, uo) is det, in, di, uo) is det.
:- mode map.foldl(pred(in, in, in, out) is semidet, in, in, out) is semidet.
:- mode map.foldl(pred(in, in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode map.foldl(pred(in, in, di, uo) is semidet, in, di, uo) is semidet.
:- mode map.foldl(pred(in, in, in, out) is cc_multi, in, in, out) is cc_multi.
:- mode map.foldl(pred(in, in, di, uo) is cc_multi, in, di, uo) is cc_multi.
:- mode map.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 map.foldl, this is often
    % a more convenient format, and a little more efficient).
    %
:- pred map.foldl2(pred(K, V, A, A, B, B), map(K, V), A, A, B, B).
:- mode map.foldl2(pred(in, in, in, out, in, out) is det,
    in, in, out, in, out) is det.
:- mode map.foldl2(pred(in, in, in, out, mdi, muo) is det,
    in, in, out, mdi, muo) is det.
:- mode map.foldl2(pred(in, in, in, out, di, uo) is det,
    in, in, out, di, uo) is det.
:- mode map.foldl2(pred(in, in, di, uo, di, uo) is det,
    in, di, uo, di, uo) is det.
:- mode map.foldl2(pred(in, in, in, out, in, out) is semidet,
    in, in, out, in, out) is semidet.
:- mode map.foldl2(pred(in, in, in, out, mdi, muo) is semidet,
    in, in, out, mdi, muo) is semidet.
:- mode map.foldl2(pred(in, in, in, out, di, uo) is semidet,
    in, in, out, di, uo) is semidet.
:- mode map.foldl2(pred(in, in, in, out, in, out) is cc_multi,
    in, in, out, in, out) is cc_multi.
:- mode map.foldl2(pred(in, in, in, out, mdi, muo) is cc_multi,
    in, in, out, mdi, muo) is cc_multi.
:- mode map.foldl2(pred(in, in, in, out, di, uo) is cc_multi,
    in, in, out, di, uo) is cc_multi.
:- mode map.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 map.foldl, this is often
    % a more convenient format, and a little more efficient).
    %
:- pred map.foldl3(pred(K, V, A, A, B, B, C, C), map(K, V), A, A, B, B, C, C).
:- mode map.foldl3(pred(in, in, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out) is det.
:- mode map.foldl3(pred(in, in, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, mdi, muo) is det.
:- mode map.foldl3(pred(in, in, in, out, in, out, di, uo) is det,
    in, in, out, in, out, di, uo) is det.
:- mode map.foldl3(pred(in, in, in, out, di, uo, di, uo) is det,
    in, in, out, di, uo, di, uo) is det.
:- mode map.foldl3(pred(in, in, di, uo, di, uo, di, uo) is det,
    in, di, uo, di, uo, di, uo) is det.
:- mode map.foldl3(pred(in, in, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out) is semidet.
:- mode map.foldl3(pred(in, in, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, mdi, muo) is semidet.
:- mode map.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 map.foldl, this is often
    % a more convenient format, and a little more efficient).
    %
:- pred map.foldl4(pred(K, V, A, A, B, B, C, C, D, D), map(K, V),
    A, A, B, B, C, C, D, D).
:- mode map.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 map.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 map.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 map.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 map.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 map.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.

    % Perform an inorder traversal by key of the map, applying an accumulator
    % predicate for value.
    %
```

```
:- pred map.foldl_values(pred(V, A, A), map(K, V), A, A).
:- mode map.foldl_values(pred(in, in, out) is det, in, in, out) is det.
:- mode map.foldl_values(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode map.foldl_values(pred(in, di, uo) is det, in, di, uo) is det.
:- mode map.foldl_values(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode map.foldl_values(pred(in, mdi, muo) is semidet, in, mdi, muo)
    is semidet.
:- mode map.foldl_values(pred(in, di, uo) is semidet, in, di, uo) is semidet.
:- mode map.foldl_values(pred(in, in, out) is cc_multi, in, in, out)
    is cc_multi.
:- mode map.foldl_values(pred(in, di, uo) is cc_multi, in, di, uo) is cc_multi.
:- mode map.foldl_values(pred(in, mdi, muo) is cc_multi, in, mdi, muo)
    is cc_multi.

    % As above, but with two accumulators.
    %
:- pred map.foldl2_values(pred(V, A, A, B, B), map(K, V), A, A, B, B).
:- mode map.foldl2_values(pred(in, in, out, in, out) is det, in,
    in, out, in, out) is det.
:- mode map.foldl2_values(pred(in, in, out, mdi, muo) is det, in,
    in, out, mdi, muo) is det.
:- mode map.foldl2_values(pred(in, in, out, di, uo) is det, in,
    in, out, di, uo) is det.
:- mode map.foldl2_values(pred(in, in, out, in, out) is semidet, in,
    in, out, in, out) is semidet.
:- mode map.foldl2_values(pred(in, in, out, mdi, muo) is semidet, in,
    in, out, mdi, muo) is semidet.
:- mode map.foldl2_values(pred(in, in, out, di, uo) is semidet, in,
    in, out, di, uo) is semidet.
:- mode map.foldl2_values(pred(in, in, out, in, out) is cc_multi, in,
    in, out, in, out) is cc_multi.
:- mode map.foldl2_values(pred(in, in, out, mdi, muo) is cc_multi, in,
    in, out, mdi, muo) is cc_multi.
:- mode map.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 map.foldl3_values(pred(V, A, A, B, B, C, C), map(K, V),
    A, A, B, B, C, C).
:- mode map.foldl3_values(pred(in, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out) is det.
:- mode map.foldl3_values(pred(in, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, mdi, muo) is det.
:- mode map.foldl3_values(pred(in, in, out, in, out, di, uo) is det, in,
    in, out, in, out, di, uo) is det.
:- mode map.foldl3_values(pred(in, in, out, in, out, in, out) is semidet, in,
```

```
    in, out, in, out, in, out) is semidet.
:- mode map.foldl3_values(pred(in, in, out, in, out, mdi, muo) is semidet, in,
    in, out, in, out, mdi, muo) is semidet.
:- mode map.foldl3_values(pred(in, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, di, uo) is semidet.
:- mode map.foldl3_values(pred(in, in, out, in, out, in, out) is cc_multi, in,
    in, out, in, out, in, out) is cc_multi.
:- mode map.foldl3_values(pred(in, in, out, in, out, mdi, muo) is cc_multi, in,
    in, out, in, out, mdi, muo) is cc_multi.
:- mode map.foldl3_values(pred(in, in, out, in, out, di, uo) is cc_multi, in,
    in, out, in, out, di, uo) is cc_multi.

:- func map.foldr(func(K, V, A) = A, map(K, V), A) = A.
:- pred map.foldr(pred(K, V, A, A), map(K, V), A, A).
:- mode map.foldr(pred(in, in, in, out) is det, in, in, out) is det.
:- mode map.foldr(pred(in, in, mdi, muo) is det, in, mdi, muo) is det.
:- mode map.foldr(pred(in, in, di, uo) is det, in, di, uo) is det.
:- mode map.foldr(pred(in, in, in, out) is semidet, in, in, out) is semidet.
:- mode map.foldr(pred(in, in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode map.foldr(pred(in, in, di, uo) is semidet, in, di, uo) is semidet.
:- mode map.foldr(pred(in, in, in, out) is cc_multi, in, in, out) is cc_multi.
:- mode map.foldr(pred(in, in, mdi, muo) is cc_multi, in, mdi, muo)
    is cc_multi.
:- mode map.foldr(pred(in, in, di, uo) is cc_multi, in, di, uo) is cc_multi.

:- pred map.foldr2(pred(K, V, A, A, B, B), map(K, V), A, A, B, B).
:- mode map.foldr2(pred(in, in, in, out, in, out) is det,
    in, in, out, in, out) is det.
:- mode map.foldr2(pred(in, in, in, out, mdi, muo) is det,
    in, in, out, mdi, muo) is det.
:- mode map.foldr2(pred(in, in, in, out, di, uo) is det,
    in, in, out, di, uo) is det.
:- mode map.foldr2(pred(in, in, di, uo, di, uo) is det,
    in, di, uo, di, uo) is det.
:- mode map.foldr2(pred(in, in, in, out, in, out) is semidet,
    in, in, out, in, out) is semidet.
:- mode map.foldr2(pred(in, in, in, out, mdi, muo) is semidet,
    in, in, out, mdi, muo) is semidet.
:- mode map.foldr2(pred(in, in, in, out, di, uo) is semidet,
    in, in, out, di, uo) is semidet.

:- pred map.foldr3(pred(K, V, A, A, B, B, C, C), map(K, V), A, A, B, B, C, C).
:- mode map.foldr3(pred(in, in, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out) is det.
:- mode map.foldr3(pred(in, in, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, mdi, muo) is det.
:- mode map.foldr3(pred(in, in, in, out, in, out, di, uo) is det,
```

```
    in, in, out, in, out, di, uo) is det.
:- mode map.foldr3(pred(in, in, in, out, di, uo, di, uo) is det,
    in, in, out, di, uo, di, uo) is det.
:- mode map.foldr3(pred(in, in, di, uo, di, uo, di, uo) is det,
    in, di, uo, di, uo, di, uo) is det.
:- mode map.foldr3(pred(in, in, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out) is semidet.
:- mode map.foldr3(pred(in, in, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, mdi, muo) is semidet.
:- mode map.foldr3(pred(in, in, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, di, uo) is semidet.

:- pred map.foldr4(pred(K, V, A, A, B, B, C, C, D, D), map(K, V),
    A, A, B, B, C, C, D, D).
:- mode map.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 map.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 map.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 map.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 map.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 map.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 map.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 map.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 map.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.

    % Apply a transformation predicate to all the values in a map.
    %
:- func map.map_values(func(K, V) = W, map(K, V)) = map(K, W).
:- pred map.map_values(pred(K, V, W), map(K, V), map(K, W)).
:- mode map.map_values(pred(in, in, out) is det, in, out) is det.
:- mode map.map_values(pred(in, in, out) is semidet, in, out) is semidet.

    % Same as map.map_values, but do not pass the key to the given predicate.
    %
:- func map.map_values_only(func(V) = W, map(K, V)) = map(K, W).
:- pred map.map_values_only(pred(V, W), map(K, V), map(K, W)).
:- mode map.map_values_only(pred(in, out) is det, in, out) is det.
:- mode map.map_values_only(pred(in, out) is semidet, in, out) is semidet.
```

```
    % Perform an inorder traversal by key of the map, applying a transformation
    % predicate to each value while updating an accumulator.
    %
:- pred map.map_foldl(pred(K, V, W, A, A), map(K, V), map(K, W), A, A).
:- mode map.map_foldl(pred(in, in, out, in, out) is det, in, out, in, out)
    is det.
:- mode map.map_foldl(pred(in, in, out, mdi, muo) is det, in, out, mdi, muo)
    is det.
:- mode map.map_foldl(pred(in, in, out, di, uo) is det, in, out, di, uo)
    is det.
:- mode map.map_foldl(pred(in, in, out, in, out) is semidet, in, out,
    in, out) is semidet.
:- mode map.map_foldl(pred(in, in, out, mdi, muo) is semidet, in, out,
    mdi, muo) is semidet.
:- mode map.map_foldl(pred(in, in, out, di, uo) is semidet, in, out,
    di, uo) is semidet.

    % As map.map_foldl, but with two accumulators.
    %
:- pred map.map_foldl2(pred(K, V, W, A, A, B, B), map(K, V), map(K, W),
    A, A, B, B).
:- mode map.map_foldl2(pred(in, in, out, in, out, in, out) is det,
    in, out, in, out, in, out) is det.
:- mode map.map_foldl2(pred(in, in, out, in, out, mdi, muo) is det,
    in, out, in, out, mdi, muo) is det.
:- mode map.map_foldl2(pred(in, in, out, in, out, di, uo) is det,
    in, out, in, out, di, uo) is det.
:- mode map.map_foldl2(pred(in, in, out, di, uo, di, uo) is det,
    in, out, di, uo, di, uo) is det.
:- mode map.map_foldl2(pred(in, in, out, in, out, in, out) is semidet,
    in, out, in, out, in, out) is semidet.
:- mode map.map_foldl2(pred(in, in, out, in, out, mdi, muo) is semidet,
    in, out, in, out, mdi, muo) is semidet.
:- mode map.map_foldl2(pred(in, in, out, in, out, di, uo) is semidet,
    in, out, in, out, di, uo) is semidet.

    % As map.map_foldl, but with three accumulators.
    %
:- pred map.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.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.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.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.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.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.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.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.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.map_foldl, but without passing the key to the predicate.
    %
:- pred map.map_values_foldl(pred(V, W, A, A), map(K, V), map(K, W), A, A).
:- mode map.map_values_foldl(pred(in, out, di, uo) is det,
    in, out, di, uo) is det.
:- mode map.map_values_foldl(pred(in, out, in, out) is det,
    in, out, in, out) is det.
:- mode map.map_values_foldl(pred(in, out, in, out) is semidet,
    in, out, in, out) is semidet.

    % As map.map_values_foldl, but with two accumulators.
    %
:- pred map.map_values_foldl2(pred(V, W, A, A, B, B), map(K, V), map(K, W),
    A, A, B, B).
:- mode map.map_values_foldl2(pred(in, out, di, uo, di, uo) is det,
    in, out, di, uo, di, uo) is det.
:- mode map.map_values_foldl2(pred(in, out, in, out, di, uo) is det,
    in, out, in, out, di, uo) is det.
:- mode map.map_values_foldl2(pred(in, out, in, out, in, out) is det,
    in, out, in, out, in, out) is det.
:- mode map.map_values_foldl2(pred(in, out, in, out, in, out) is semidet,
    in, out, in, out, in, out) is semidet.

    % As map.map_values_foldl, but with three accumulators.
    %
:- pred map.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.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.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.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.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.map_values_foldl3(
    pred(in, out, in, out, in, out, in, out) is semidet,
```

```
    in, out, in, out, in, out, in, out) is semidet.

    % 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.
    %
:- pred map.intersect(pred(V, V, V), map(K, V), map(K, V), map(K, V)).
:- mode map.intersect(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode map.intersect(pred(in, in, out) is det, in, in, out) is det.

:- func map.intersect(func(V, V) = V, map(K, V), map(K, V)) = map(K, V).

    % Calls map.intersect. Aborts if map.intersect fails.
    %
:- pred map.det_intersect(pred(V, V, V), map(K, V), map(K, V), map(K, V)).
:- mode map.det_intersect(pred(in, in, out) is semidet, in, in, out) is det.

:- func map.det_intersect(func(V, V) = V, map(K, V), map(K, V)) = map(K, V).
:- mode map.det_intersect(func(in, in) = out is semidet, in, in) = 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.
    %
    % map.common_subset is very similar to map.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 map.common_subset(map(K, V), map(K, V)) = map(K, V).

    % 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 map.union(func(V, V) = V, map(K, V), map(K, V)) = map(K, V).
```

```
:- pred map.union(pred(V, V, V), map(K, V), map(K, V), map(K, V)).
:- mode map.union(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode map.union(pred(in, in, out) is det, in, in, out) is det.

    % Calls map.union. Aborts if map.union fails.
    %
:- pred map.det_union(pred(V, V, V), map(K, V), map(K, V), map(K, V)).
:- mode map.det_union(pred(in, in, out) is semidet, in, in, out) is det.

:- func map.det_union(func(V, V) = V, map(K, V), map(K, V)) = map(K, V).
:- mode map.det_union(func(in, in) = out is semidet, in, in) = out is det.

    % 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 map.reverse_map(map(K, V)) = map(V, set(K)).

    % Field selection for maps.

    % Map ^ elem(Key) = map.search(Map, Key).
    %
:- func map.elem(K, map(K, V)) = V is semidet.

    % Map ^ det_elem(Key) = map.lookup(Map, Key).
    %
:- func map.det_elem(K, map(K, V)) = V.

    % Field update for maps.

    % (Map ^ elem(Key) := Value) = map.set(Map, Key, Value).
    %
:- func 'elem :='(K, map(K, V), V) = map(K, V).

    % (Map ^ det_elem(Key) := Value) = map.det_update(Map, Key, Value).
    %
:- func 'det_elem :='(K, map(K, V), V) = map(K, V).

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

41 math

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
```

```
%-----------------------------------------------------%
% Copyright (C) 1995-2007, 2011-2012 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% 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.

    % A domain error exception, indicates that the inputs to a function
    % were outside the domain of the function.  The string indicates
    % where the error occurred.
    %
:- type domain_error ---> domain_error(string).

%-----------------------------------------------------%
%
% Mathematical constants
%

    % Pythagoras' number.
    %
:- func math.pi = float.

    % Base of natural logarithms.
    %
```

```
:- func math.e = float.

%-----------------------------------------------%
%
% "Next integer" operations
%

    % math.ceiling(X) = Ceil is true if Ceil is the smallest integer
    % not less than X.
    %
:- func math.ceiling(float) = float.

    % math.floor(X) = Floor is true if Floor is the largest integer
    % not greater than X.
    %
:- func math.floor(float) = float.

    % math.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.
    %
:- func math.round(float) = float.

    % math.truncate(X) = Trunc is true if Trunc is the integer closest to X
    % such that |Trunc| =< |X|.
    %
:- func math.truncate(float) = float.

%-----------------------------------------------%
%
% Polynomial roots
%

    % math.sqrt(X) = Sqrt is true if Sqrt is the positive square root of X.
    %
    % Domain restriction: X >= 0
    %
:- func math.sqrt(float) = float.
:- func math.unchecked_sqrt(float) = float.

:- type math.quadratic_roots
    --->    no_roots
    ;       one_root(float)
    ;       two_roots(float, float).

    % math.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 math.solve_quadratic(float, float, float) = quadratic_roots.

%-----------------------------------------------------%
%
% Power/logarithm operations
%

    % math.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 math.pow(float, float) = float.
:- func math.unchecked_pow(float, float) = float.

    % math.exp(X) = Exp is true if Exp is e raised to the power of X.
    %
:- func math.exp(float) = float.

    % math.ln(X) = Log is true if Log is the natural logarithm of X.
    %
    % Domain restriction: X > 0
    %
:- func math.ln(float) = float.
:- func math.unchecked_ln(float) = float.

    % math.log10(X) = Log is true if Log is the logarithm to base 10 of X.
    %
    % Domain restriction: X > 0
    %
:- func math.log10(float) = float.
:- func math.unchecked_log10(float) = float.

    % math.log2(X) = Log is true if Log is the logarithm to base 2 of X.
    %
    % Domain restriction: X > 0
    %
:- func math.log2(float) = float.
:- func math.unchecked_log2(float) = float.

    % math.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 math.log(float, float) = float.
:- func math.unchecked_log(float, float) = float.
```

```
%---------------------------------------------------%
%
% Trigonometric operations
%

    % math.sin(X) = Sin is true if Sin is the sine of X.
    %
:- func math.sin(float) = float.

    % math.cos(X) = Cos is true if Cos is the cosine of X.
    %
:- func math.cos(float) = float.

    % math.tan(X) = Tan is true if Tan is the tangent of X.
    %
:- func math.tan(float) = float.

    % math.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 math.asin(float) = float.
:- func math.unchecked_asin(float) = float.

    % math.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 math.acos(float) = float.
:- func math.unchecked_acos(float) = float.

    % math.atan(X) = ATan is true if ATan is the inverse tangent of X,
    % where ATan is in the range [-pi/2,pi/2].
    %
:- func math.atan(float) = float.

    % math.atan2(Y, X) = ATan is true if ATan is the inverse tangent of Y/X,
    % where ATan is in the range [-pi,pi].
    %
:- func math.atan2(float, float) = float.

%---------------------------------------------------%
%
% Hyperbolic functions
```

```
%
    % math.sinh(X) = Sinh is true if Sinh is the hyperbolic sine of X.
    %
:- func math.sinh(float) = float.

    % math.cosh(X) = Cosh is true if Cosh is the hyperbolic cosine of X.
    %
:- func math.cosh(float) = float.

    % math.tanh(X) = Tanh is true if Tanh is the hyperbolic tangent of X.
    %
:- func math.tanh(float) = float.

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

42 maybe

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%---------------------------------------------------%
% Copyright (C) 1994-2006, 2010-2011 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%---------------------------------------------------%
%
% File: maybe.m.
% Main author: fjh.
% Stability: high.
%
% This module defines the "maybe" type.
%
%---------------------------------------------------%
%---------------------------------------------------%

:- module maybe.
:- interface.

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

:- type maybe(T)
    --->    no
    ;       yes(T).
```

```
:- inst maybe(I)
    --->    no
    ;       yes(I).

:- inst maybe_yes(I)
    --->    yes(I).

:- type maybe_error
    --->    ok
    ;       error(string).

:- type maybe_error(T) == maybe_error(T, string).

:- type maybe_error(T, E)
    --->    ok(T)
    ;       error(E).

:- inst maybe_error(I)
    --->    ok(I)
    ;       error(ground).

:- inst maybe_error_ok(I)
    --->    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).
    % 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.

    % fold_maybe(_, no, Acc) = Acc.
    % fold_maybe(F, yes(Value), Acc0) = F(Acc0).
    %
:- func fold_maybe(func(T, U) = U, maybe(T), U) = U.

    % 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.

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

43 multi_map

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%---------------------------------------------------%
% Copyright (C) 1995, 1997, 2000, 2002-2006, 2011 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%---------------------------------------------------%
%
% File: multi_map.m.
% Main author: dylan.  Based on map.m, by fjh, conway.
% Stability: low.
%
% This file provides the 'multi_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 multi_map is similar, though allows a one to many relationship
% between keys and data.
%
% 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(Key, Data) == map(Key, list(Data)).

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

    % Initialize an empty multi_map.
    %
:- func multi_map.init = multi_map(_, _).
:- pred multi_map.init(multi_map(_, _)::uo) is det.

    % Check whether a multi_map is empty.
    %
:- pred multi_map.is_empty(multi_map(_, _)::in) is semidet.

    % Check whether multi_map contains key.
    %
:- pred multi_map.contains(multi_map(K, _V)::in, K::in) is semidet.

:- pred multi_map.member(multi_map(K, V)::in, K::out, V::out) is nondet.

    % Search multi_map for given key.
    %
:- pred multi_map.search(multi_map(K, V)::in, K::in, list(V)::out) is semidet.

    % Search multi_map for given key.
    %
:- pred multi_map.nondet_search(multi_map(K, V)::in, K::in, V::out) is nondet.

    % Search multi_map for key, but abort if search fails.
    %
:- func multi_map.lookup(multi_map(K, V), K) = list(V).
:- pred multi_map.lookup(multi_map(K, V)::in, K::in, list(V)::out) is det.

    % Search multi_map for key.
    %
:- pred multi_map.nondet_lookup(multi_map(K, V)::in, K::in, V::out) is nondet.
```

```
    % Search multi_map for data.
    %
:- pred multi_map.inverse_search(multi_map(K, V)::in, V::in, K::out) is nondet.

    % Insert a new key and corresponding value into a multi_map.
    % Fail if the key already exists.
    %
:- pred multi_map.insert(K::in, V::in,
    multi_map(K, V)::in, multi_map(K, V)::out) is semidet.

    % Insert a new key and corresponding value into a multi_map.
    % Aborts if the key already exists.
    %
:- func multi_map.det_insert(multi_map(K, V), K, V) = multi_map(K, V).
:- pred multi_map.det_insert(K::in, V::in,
    multi_map(K, V)::in, multi_map(K, V)::out) is det.

    % Update (add) the value corresponding to a given key.
    % Fails if the key does not already exist.
    %
:- pred multi_map.update(K::in, V::in,
    multi_map(K, V)::in, multi_map(K, V)::out) is semidet.

    % Update (add) the value corresponding to a given key.
    % Aborts if the key does not already exist.
    %
:- func multi_map.det_update(multi_map(K, V), K, V) = multi_map(K, V).
:- pred multi_map.det_update(K::in, V::in,
    multi_map(K, V)::in, multi_map(K, V)::out) is det.

    % Update (replace) the value corresponding to a given key.
    % Fails if the key does not already exist.
    %
:- pred multi_map.replace(K::in, list(V)::in,
    multi_map(K, V)::in, multi_map(K, V)::out) is semidet.

    % Update (replace) the value corresponding to a given key.
    % Aborts if the key does not already exist.
    %
:- func multi_map.det_replace(multi_map(K, V), K, list(V)) = multi_map(K, V).
:- pred multi_map.det_replace(K::in, list(V)::in,
    multi_map(K, V)::in, multi_map(K, V)::out) is det.

    % Update (add) value if the key is already present, otherwise
    % insert the new key and value.
    %
:- func multi_map.set(multi_map(K, V), K, V) = multi_map(K, V).
```

```
:- pred multi_map.set(K::in, V::in,
    multi_map(K, V)::in, multi_map(K, V)::out) is det.

:- func multi_map.add(multi_map(K, V), K, V) = multi_map(K, V).
:- pred multi_map.add(K::in, V::in,
    multi_map(K, V)::in, multi_map(K, V)::out) is det.

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

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

    % Convert a multi_map to an association list.
    %
:- func multi_map.to_flat_assoc_list(multi_map(K, V)) = assoc_list(K, V).
:- pred multi_map.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 multi_map.from_flat_assoc_list(assoc_list(K, V)) = multi_map(K, V).
:- pred multi_map.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 multi_map.to_assoc_list(multi_map(K, V)) = assoc_list(K, list(V)).
:- pred multi_map.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 multi_map.from_assoc_list(assoc_list(K, list(V))) = multi_map(K, V).
:- pred multi_map.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 multi_map.from_sorted_assoc_list(assoc_list(K, list(V)))
```

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

    % Delete a key and data from a multi_map
    % if the key is not present, leave the multi_map unchanged.
    %
:- func multi_map.delete(multi_map(K, V), K) = multi_map(K, V).
:- pred multi_map.delete(K::in,
    multi_map(K, V)::in, multi_map(K, V)::out) is det.

    % Delete a data value from a key in a multi_map
    % if the key is not present, leave the multi_map unchanged.
    %
:- func multi_map.delete(multi_map(K, V), K, V) = multi_map(K, V).
:- pred multi_map.delete(K::in, V::in,
    multi_map(K, V)::in, multi_map(K, V)::out) is det.

    % Delete a key-value pair from a multi_map and return the value.
    % Fails if the key is not present.
    %
:- pred multi_map.remove(K::in, list(V)::out,
    multi_map(K, V)::in, multi_map(K, V)::out) is semidet.

    % Delete a key-value pair from a multi_map and return the value.
    % Aborts if the key is not present.
    %
:- pred multi_map.det_remove(K::in, list(V)::out,
    multi_map(K, V)::in, multi_map(K, V)::out) is det.

    % Count the number of elements (keys) in the multi_map.
    %
:- func multi_map.count(multi_map(K, V)) = int.
:- pred multi_map.count(multi_map(K, V)::in, int::out) is det.

    % Count the number of data elements in the multi_map.
    %
:- func multi_map.all_count(multi_map(K, V)) = int.
:- pred multi_map.all_count(multi_map(K, V)::in, int::out) is det.

    % Convert a pair of lists (which must be of the same length)
    % to a multi_map.
    %
:- func multi_map.from_corresponding_lists(list(K), list(V))
    = multi_map(K, V).
:- pred multi_map.from_corresponding_lists(list(K)::in, list(V)::in,
    multi_map(K, V)::out) is det.
```

```
    % Convert a pair of lists (which must be of the same length)
    % to a multi_map.
    %
:- func multi_map.from_corresponding_list_lists(list(K), list(list(V)))
    = multi_map(K, V).
:- pred multi_map.from_corresponding_list_lists(list(K)::in, list(list(V))::in,
    multi_map(K, V)::out) is det.

    % multi_map.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 corresponding to that key from 'MultiMapA' and
    % 'MultiMapB'.
    %
:- func multi_map.merge(multi_map(K, V), multi_map(K, V))
    = multi_map(K, V).
:- pred multi_map.merge(multi_map(K, V)::in, multi_map(K, V)::in,
    multi_map(K, V)::out) is det.

    % multi_map.select takes a multi_map and a set of keys and returns
    % a multi_map containing the keys in the set and their corresponding
    % values.
    %
:- func multi_map.select(multi_map(K, V), set(K)) = multi_map(K, V).
:- pred multi_map.select(multi_map(K, V)::in, set(K)::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 multi_map.apply_to_list(list(K), multi_map(K, V)) = list(V).
:- pred multi_map.apply_to_list(list(K)::in, multi_map(K, V)::in,
    list(V)::out) is det.

    % Declaratively, a NOP.
    % Operationally, a suggestion that the implementation
    % optimize the representation of the multi_map in the expectation
    % of a number of lookups but few or no modifications.
    %
:- func multi_map.optimize(multi_map(K, V)) = multi_map(K, V).
:- pred multi_map.optimize(multi_map(K, V)::in, multi_map(K, V)::out) is det.

    % Remove the smallest item from the multi_map.
    % Fails if the multi_map is empty.
    %
```

```
:- pred multi_map.remove_smallest(K::out, list(V)::out,
    multi_map(K, V)::in, multi_map(K, V)::out) is semidet.

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

44 ops

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%---------------------------------------------------%
% Copyright (C) 1995-2008, 2010, 2012 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%---------------------------------------------------%
%
% 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 ops.class describes what structure terms constructed with an operator
    % of that class are allowed to take.
:- type ops.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 ops.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 ops.priority == int.

:- type ops.op_info
    --->    op_info(
                ops.class,
                ops.priority
            ).

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

:- typeclass ops.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 ops.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 ops.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 ops.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 ops.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 ops.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 ops.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 ops.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 ops.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 ops.mercury_op_table.
:- instance ops.op_table(ops.mercury_op_table).
```

```
:- func ops.init_mercury_op_table = (ops.mercury_op_table::uo) is det.

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

45 pair

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%---------------------------------------------------%
% Copyright (C) 1994-2006 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%---------------------------------------------------%
%
% 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)
    --->    (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).

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

46 parser

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%---------------------------------------------------%
% Copyright (C) 1995-2001, 2003-2008, 2011-2012 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%---------------------------------------------------%
%
% 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_token_list predicate is similar,
% but it takes a list of tokens rather than a string.
%
% The parser and lexer are intended to exactly follow ISO Prolog
% syntax, but there are some departures from that for three reasons:
%
%   (1) I wrote some of the code at home when the ISO Prolog draft
%       was at uni - so in some places I just guessed.
%   (2) In some places the lexer reports an error when it shouldn't.
%   (3) There are a couple of hacks to make it compatible with NU-Prolog
%       syntax.
%
% 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 in 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):
    %
    % Reads a Mercury term from the current input stream.
    %
:- pred read_term(read_term(T)::out, io::di, io::uo) is det.

    % read_term_with_op_table(Result):
    %
    % Reads a term from the current input 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).

    % read_term_filename(FileName, Result, !IO):
    %
    % Reads a term from the current input stream. The string is the filename
    % to use for the current input 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.

    % read_term_filename_with_op_table(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).

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

    % 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 six-argument version 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_string(FileName, String, MaxOffset, StartPos,
    %   EndPos, Term).
    %
:- pred read_term_from_substring(string::in, string::in, int::in,
    posn::in, posn::out, read_term(T)::out) is det.

    % read_term_from_string_with_op_table(Ops, FileName, String,
    %   MaxOffset, StartPos, EndPos, 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).

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

    % 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).

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

47 parsing_utils

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
```

```
%-----------------------------------------------------%
% Copyright (C) 2009-2012 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% 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 a parser state (ps)
% input/output pair 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.

    % 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.

    % 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.

    % 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).
    %
:- type line_numbers.

:- 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.

    % 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.

    % 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 upper and lowercase unaccented Latin letters are treated specially.
    %
:- 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.

    % 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.

    % 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.

    % 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.

    % 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.

    % 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.

    % 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.

    % 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.

% The following parser combinators are equivalent to the above, except that
% a separate state argument is threaded through the computation (e.g., for
% parsers that incrementally construct a symbol table).

    % 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, !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, !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, !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, !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, !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.

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

48 pprint

```
%-----------------------------------------------------%
% vim:ts=4 sw=4 expandtab tw=0 wm=0 ft=mercury
%-----------------------------------------------------%
% Copyright (C) 2000-2007, 2010-2011 The University of Melbourne
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% 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(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 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(string.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.
    %
:- 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) ).

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

49 pqueue

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%-----------------------------------------------------%
% Copyright (C) 1994-1995, 1997, 1999, 2003-2007, 2009 The University of
% Melbourne.
%
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% 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 pqueue.init = pqueue(K, V).
:- pred pqueue.init(pqueue(K, V)::out) is det.
```

```
    % True iff the priority queue is empty.
    %
:- pred pqueue.is_empty(pqueue(K, V)::in) is semidet.

    % Insert a value V with key K into a priority queue
    % and return the new priority queue.
    %
:- func pqueue.insert(pqueue(K, V), K, V) = pqueue(K, V).
:- pred pqueue.insert(K::in, V::in, pqueue(K, V)::in, pqueue(K, V)::out)
    is det.

    % Extract the smallest key-value pair from the priority queue without
    % removing it.  Fails if the priority queue is empty.
    %
:- pred pqueue.peek(pqueue(K, V)::in, K::out, V::out) is semidet.

    % Extract the smallest key from the priority queue without removing it.
    % Fails if the priority queue is empty.
    %
:- pred pqueue.peek_key(pqueue(K, V)::in, K::out) is semidet.

    % Extract the smallest value from the priority queue without removing
    % it.  Fails if the priority queue is empty.
    %
:- pred pqueue.peek_value(pqueue(K, V)::in, V::out) is semidet.

    % As above, but calls error/1 if the priority queue is empty.
    %
:- pred pqueue.det_peek(pqueue(K, V)::in, K::out, V::out) is det.
:- func pqueue.det_peek_key(pqueue(K, V)) = K.
:- func pqueue.det_peek_value(pqueue(K, V)) = V.

    % Remove the smallest item from the priority queue.
    % Fails if the priority queue is empty.
    %
:- pred pqueue.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 pqueue.det_remove(K::out, V::out, pqueue(K, V)::in, pqueue(K, V)::out)
    is det.

    % Merges all the entries of one priority queue with another, returning
    % the merged list.
    %
:- func pqueue.merge(pqueue(K, V), pqueue(K, V)) = pqueue(K, V).
```

```
:- pred pqueue.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 pqueue.to_assoc_list(pqueue(K, V)) = assoc_list(K, V).
:- pred pqueue.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 pqueue.assoc_list_to_pqueue(assoc_list(K, V)) = pqueue(K, V).
:- pred pqueue.assoc_list_to_pqueue(assoc_list(K, V)::in, pqueue(K, V)::out)
    is det.

    % A synonym for pqueue.assoc_list_to_pqueue/1.
    %
:- func pqueue.from_assoc_list(assoc_list(K, V)) = pqueue(K, V).

    % length(PQueue) = Length.
    %
    % Length is the number of items in PQueue
    %
:- func pqueue.length(pqueue(K, V)) = int.

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

50 pretty_printer

```
%---------------------------------------------------%
% vim: ts=4 sw=4 expandtab tw=0 wm=0 ft=mercury
%---------------------------------------------------%
% Copyright (C) 2007, 2009-2011 The University of Melbourne
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%---------------------------------------------------%
%
% 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 operator precedence and
% bracketing rules.
%
% 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 num-
ber of
% lines to be output, and a limit on the depth for formatting arbitrary terms.
% Output is replaced with ellipsis ("...") when limits are 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. It cannot be exploited by
    % user code.
    %
```

```
:- type pp_internal.

    % 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(docs) = 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.

    % The pretty-printer limit type, used to control conversion by
    % format_univ, format_list, and format_term.
    %
    % 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 a
    % limit of 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 formatting_limit
    --->    linear(int)                 % Print this many functors.
    ;       triangular(int).            % Print first arg with limit N-
1,
                                        % second arg with limit N-2, ...

    % 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.
    %
:- func set_formatter(string, string, int, formatter, formatter_map) =
    formatter_map.



    % format(Stream, FMap, LineWidth, MaxLines, Limit, Doc, !State):
    %
    % Format Doc to fit on lines of LineWidth chars, truncating after
    % MaxLines lines, fomatting format_univ(_) docs using specialised
    % formatters Formatters starting with pretty-printer limits Limit.
    %
:- pred write_doc_to_stream(Stream, noncanon_handling, formatter_map, int, int,
    formatting_limit, doc, State, State)
    <= stream.writer(Stream, string, State).
:- mode write_doc_to_stream(in, in(canonicalize), in, in, in, in, in,
    di, uo) is det.
:- mode write_doc_to_stream(in, in(include_details_cc), in, in, in, in, in,
    di, uo) is cc_multi.
```

```
    % Convenience predicates.  A user-configurable set of type-specific
    % formatters and formatting parameters are attached to the I/O state.
    % The I/O state-specific format predicate below uses this settings.
    %
:- type pp_params
    --->    pp_params(
                pp_line_width   :: int,             % Line width in characters.
                pp_max_lines    :: int,             % Max lines to output.
                pp_limit        :: formatting_limit % Term formatting limit.
            ).

    % An initial default formatter_map is provided for the most commonly
    % used types in the Mercury standard library (array, char, float,
    % int, map, string, etc.)
    %
    % The default formatter_map may also be updated by users' modules
    % (e.g., in initialisation goals).
    %
    % These defaults are thread local (i.e., 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.
:- pred set_default_formatter(string::in, string::in, int::in, formatter::in,
    io::di, io::uo) is det.

    % The initial default pp_params are pp_params(78, 100, triangular(100)).
    % These defaults are thread local (i.e., 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.

    % write_doc(Doc, !IO):
    % write_doc(FileStream, Doc, !IO):
    %
    % Format Doc to io.stdout_stream or FileStream respectively, using
    % write_doc_to stream, 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.

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

51 prolog

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%-----------------------------------------------------%
% Copyright (C) 1997-2003, 2005-2006, 2012 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% 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.

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

52 queue

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

```
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%-----------------------------------------------------%
% Copyright (C) 1994-1995, 1997-1999, 2003-2006, 2011 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% 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 (queue.put) and remove them from
% the front of the queue (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).

    % ‘queue.init(Queue)’ is true iff ‘Queue’ is an empty queue.
    %
:- func queue.init = queue(T).
:- pred queue.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 queue.equal(queue(T)::in, queue(T)::in) is semidet.

    % ‘queue.is_empty(Queue)’ is true iff ‘Queue’ is an empty queue.
    %
:- pred queue.is_empty(queue(T)::in) is semidet.

    % ‘queue.is_full(Queue)’ is intended to be true iff ‘Queue’ is a queue
```

```
    % whose capacity is exhausted. This implementation allows arbitrary-
sized
    % queues, so queue.is_full always fails.
    %
:- pred queue.is_full(queue(T)::in) is semidet.

    % ‘queue.put(Elem, Queue0, Queue)’ is true iff ‘Queue’ is the queue
    % which results from appending ‘Elem’ onto the end of ‘Queue0’.
    %
:- func queue.put(queue(T), T) = queue(T).
:- pred queue.put(T::in, queue(T)::in, queue(T)::out) is det.

    % ‘queue.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 queue.put_list(queue(T), list(T)) = queue(T).
:- pred queue.put_list(list(T)::in, queue(T)::in, queue(T)::out) is det.

    % ‘queue.first(Queue, Elem)’ is true iff ‘Queue’ is a non-empty queue
    % whose first element is ‘Elem’.
    %
:- pred queue.first(queue(T)::in, T::out) is semidet.

    % ‘queue.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 queue.get(T::out, queue(T)::in, queue(T)::out) is semidet.

    % ‘queue.length(Queue, Length)’ is true iff ‘Queue’ is a queue
    % containing ‘Length’ elements.
    %
:- func queue.length(queue(T)) = int.
:- pred queue.length(queue(T)::in, int::out) is det.

    % ‘queue.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 queue.list_to_queue(list(T)) = queue(T).
:- pred queue.list_to_queue(list(T)::in, queue(T)::out) is det.

    % A synonym for queue.list_to_queue/1.
    %
:- func queue.from_list(list(T)) = queue(T).

    % ‘queue.to_list(Queue) = List’ is the inverse of queue.from_list/1.
```

```
    %
:- func queue.to_list(queue(T)) = list(T).

    % ‘queue.delete_all(Elem, Queue0, Queue)’ is true iff ‘Queue’ is the same
    % queue as ‘Queue0’ with all occurrences of ‘Elem’ removed from it.
    %
:- func queue.delete_all(queue(T), T) = queue(T).
:- pred queue.delete_all(T::in, queue(T)::in, queue(T)::out) is det.

    % ‘queue.put_on_front(Queue0, Elem) = Queue’ pushes ‘Elem’ on to
    % the front of ‘Queue0’, giving ‘Queue’.
    %
:- func queue.put_on_front(queue(T), T) = queue(T).
:- pred queue.put_on_front(T::in, queue(T)::in, queue(T)::out) is det.

    % ‘queue.put_list_on_front(Queue0, Elems) = Queue’ pushes ‘Elems’
    % on to the front of ‘Queue0’, giving ‘Queue’ (the Nth member
    % of ‘Elems’ becomes the Nth member from the front of ‘Queue’).
    %
:- func queue.put_list_on_front(queue(T), list(T)) = queue(T).
:- pred queue.put_list_on_front(list(T)::in, queue(T)::in, queue(T)::out)
    is det.

    % ‘queue.get_from_back(Elem, Queue0, Queue)’ removes ‘Elem’ from
    % the back of ‘Queue0’, giving ‘Queue’.
    %
:- pred queue.get_from_back(T::out, queue(T)::in, queue(T)::out) is semidet.

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

53 random

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%---------------------------------------------------%
% Copyright (C) 1994-1998,2001-2006, 2011 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%---------------------------------------------------%
%
% File: random.m
% Main author: conway
% Stability: low
%
```

```
% 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.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module random.
:- interface.

:- import_module list.

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

% The type `random.supply' represents a supply of random numbers.
    %
```

```
:- type random.supply.

% random.init(Seed, RS): creates a supply of random numbers RS
% using the specified Seed.
%
:- pred random.init(int::in, random.supply::uo) is det.

% random.random(Num, RS0, RS): extracts a number Num in the
% range 0 .. RandMax from the random number supply RS0, and
% binds RS to the new state of the random number supply.
    %
:- pred random.random(int, random.supply, random.supply).
:- mode random.random(out, mdi, muo) is det.
:- mode random.random(out, in, out) is det.

% random.random(Low, Range, Num, RS0, RS): extracts a number Num
% in the range Low .. (Low + Range - 1) from the random number
% supply RS0, and binds RS to the new state of the random number
% supply.  For best results, the value of Range should be no greater
% than about 100.
%
:- pred random.random(int, int, int, random.supply, random.supply).
:- mode random.random(in, in, out, mdi, muo) is det.
:- mode random.random(in, in, out, in, out) is det.

% random.randmax(RandMax, RS0, RS): binds RandMax to the maximum
% random number that can be returned from the random number
% supply RS0, and returns RS = RS0.
%
:- pred random.randmax(int, random.supply, random.supply).
:- mode random.randmax(out, mdi, muo) is det.
:- mode random.randmax(out, in, out) is det.

% random.randcount(RandCount, RS0, RS): binds RandCount to the
% number of distinct random numbers that can be returned from the
% random number supply RS0, and returns RS = RS0.  This will be one
% more than the number returned by randmax/3.
%
:- pred random.randcount(int, random.supply, random.supply).
:- mode random.randcount(out, mdi, muo) is det.
:- mode random.randcount(out, in, out) is det.

% random.permutation(List0, List, RS0, RS):
% binds List to a random permutation of List0,
% and binds RS to the new state of the random number supply.
%
:- pred random.permutation(list(T), list(T), random.supply, random.supply).
```

```
:- mode random.permutation(in, out, mdi, muo) is det.
:- mode random.permutation(in, out, in, out) is det.

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

54 rational

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%---------------------------------------------------%
% Copyright (C) 1997-1998, 2003-2006 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%---------------------------------------------------%
%
% 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.

:- 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.rational(int) = rational.
```

```
:- func rational.rational(int, int) = rational.

:- func rational.from_integer(integer) = rational.

:- func rational.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 rational.numer(rational) = integer.

:- func rational.denom(rational) = integer.

:- func rational.abs(rational) = rational.

:- func rational.reciprocal(rational) = rational.

:- func rational.one = rational.

:- func rational.zero = rational.

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

55 rbtree

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%---------------------------------------------------%
% Copyright (C) 1995-2000, 2003-2007, 2011 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%---------------------------------------------------%
%
```

```
% 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:
%   Aborts program 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 rbtree.init = rbtree(K, V).
```

```
:- pred rbtree.init(rbtree(K, V)::uo) is det.

    % Initialise an rbtree containing the given key-value pair.
    %
:- func rbtree.singleton(K, V) = rbtree(K, V).

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

    % Inserts a new key-value pair into the tree.
    % Fails if key already in the tree.
    %
:- pred rbtree.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 rbtree.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 rbtree.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 rbtree.set(rbtree(K, V), K, V) = rbtree(K, V).
:- pred rbtree.set(K::in, V::in, rbtree(K, V)::in, rbtree(K, V)::out) is det.

    % Insert a duplicate key into the tree.
    %
:- func rbtree.insert_duplicate(rbtree(K, V), K, V) = rbtree(K, V).
:- pred rbtree.insert_duplicate(K::in, V::in,
    rbtree(K, V)::in, rbtree(K, V)::out) is det.

:- pred rbtree.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 rbtree.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 rbtree.lookup(rbtree(K, V), K) = V.
:- pred rbtree.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 rbtree.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 rbtree.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 rbtree.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 rbtree.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 rbtree.delete(rbtree(K, V), K) = rbtree(K, V).
:- pred rbtree.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 rbtree.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 rbtree.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 rbtree.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 rbtree.keys(rbtree(K, V)) = list(K).
:- pred rbtree.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 rbtree.values(rbtree(K, V)) = list(V).
:- pred rbtree.values(rbtree(K, V)::in, list(V)::out) is det.

    % Count the number of elements in the tree.
    %
:- func rbtree.count(rbtree(K, V)) = int.
:- pred rbtree.count(rbtree(K, V)::in, int::out) is det.

:- func rbtree.assoc_list_to_rbtree(assoc_list(K, V)) = rbtree(K, V).
:- pred rbtree.assoc_list_to_rbtree(assoc_list(K, V)::in, rbtree(K, V)::out)
    is det.

:- func rbtree.from_assoc_list(assoc_list(K, V)) = rbtree(K, V).

:- func rbtree.rbtree_to_assoc_list(rbtree(K, V)) = assoc_list(K, V).
:- pred rbtree.rbtree_to_assoc_list(rbtree(K, V)::in, assoc_list(K, V)::out)
    is det.

:- func rbtree.to_assoc_list(rbtree(K, V)) = assoc_list(K, V).

:- func rbtree.foldl(func(K, V, T) = T, rbtree(K, V), T) = T.
:- pred rbtree.foldl(pred(K, V, T, T), rbtree(K, V), T, T).
:- mode rbtree.foldl(pred(in, in, in, out) is det, in, in, out) is det.
:- mode rbtree.foldl(pred(in, in, mdi, muo) is det, in, mdi, muo) is det.
:- mode rbtree.foldl(pred(in, in, di, uo) is det, in, di, uo) is det.
:- mode rbtree.foldl(pred(in, in, in, out) is semidet, in, in, out)
    is semidet.
:- mode rbtree.foldl(pred(in, in, mdi, muo) is semidet, in, mdi, muo)
```

```
    is semidet.
:- mode rbtree.foldl(pred(in, in, di, uo) is semidet, in, di, uo)
    is semidet.

:- pred rbtree.foldl2(pred(K, V, T, T, U, U), rbtree(K, V), T, T, U, U).
:- mode rbtree.foldl2(pred(in, in, in, out, in, out) is det,
    in, in, out, in, out) is det.
:- mode rbtree.foldl2(pred(in, in, in, out, mdi, muo) is det,
    in, in, out, mdi, muo) is det.
:- mode rbtree.foldl2(pred(in, in, in, out, di, uo) is det,
    in, in, out, di, uo) is det.
:- mode rbtree.foldl2(pred(in, in, di, uo, di, uo) is det,
    in, di, uo, di, uo) is det.
:- mode rbtree.foldl2(pred(in, in, in, out, in, out) is semidet,
    in, in, out, in, out) is semidet.
:- mode rbtree.foldl2(pred(in, in, in, out, mdi, muo) is semidet,
    in, in, out, mdi, muo) is semidet.
:- mode rbtree.foldl2(pred(in, in, in, out, di, uo) is semidet,
    in, in, out, di, uo) is semidet.

:- pred rbtree.foldl3(pred(K, V, T, T, U, U, W, W), rbtree(K, V),
    T, T, U, U, W, W).
:- mode rbtree.foldl3(pred(in, in, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out) is det.
:- mode rbtree.foldl3(pred(in, in, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out) is semidet.
:- mode rbtree.foldl3(pred(in, in, in, out, in, out, di, uo) is det,
    in, in, out, in, out, di, uo) is det.
:- mode rbtree.foldl3(pred(in, in, in, out, di, uo, di, uo) is det,
    in, in, out, di, uo, di, uo) is det.
:- mode rbtree.foldl3(pred(in, in, di, uo, di, uo, di, uo) is det,
    in, di, uo, di, uo, di, uo) is det.

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

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

56 require

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
```

```
%-----------------------------------------------------%
% Copyright (C) 1993-1999, 2003, 2005-2006, 2010-2011 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% 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.

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

    % 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.

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

57 rtree

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

```
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%-----------------------------------------------------%
% Copyright (C) 2006-2007 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% 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 rtree.init = (rtree(K, V)::uo) is det <= region(K).

    % Succeeds iff the given rtree is empty.
    %
:- pred rtree.is_empty(rtree(K, V)::in) is semidet.

    % Insert a new key and corresponding value into an rtree.
    %
:- func rtree.insert(K, V, rtree(K, V)) = rtree(K, V) <= region(K).
:- pred rtree.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 rtree.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 rtree.search_intersects(rtree(K, V), K) = list(V) <= region(K).

    % Search for all values with keys that contain the query key.
    %
:- func rtree.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 rtree.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 rtree.search_first(pred(K, L), pred(V, L), rtree(K, V), L, V, L).
:- mode rtree.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 rtree.search_general_fold(pred(K), pred(K, V, A, A), rtree(K, V),
    A, A).
:- mode rtree.search_general_fold(pred(in) is semidet,
    pred(in, in, in, out) is det, in, in, out) is det.
:- mode rtree.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 rtree.fold(pred(K, V, A, A), rtree(K, V), A, A).
:- mode rtree.fold(pred(in, in, in, out) is det, in, in, out) is det.
:- mode rtree.fold(pred(in, in, di, uo) is det, in, di, uo) is det.
:- mode rtree.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 rtree.map_values(pred(K, V, W), rtree(K, V), rtree(K, W)).
:- mode rtree.map_values(pred(in, in, out) is det, in, out) is det.
:- mode rtree.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).

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

58 set

```
%-----------------------------------------------------%
% vim: ts=4 sw=4 et ft=mercury
%-----------------------------------------------------%
% Copyright (C) 1994-1997, 1999-2012 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% 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).

    % ‘set.init(Set)’ is true iff ‘Set’ is an empty set.
    %
```

```
:- func set.init = set(T).
:- pred set.init(set(T)::uo) is det.

    % 'set.list_to_set(List, Set)' is true iff 'Set' is the set
    % containing only the members of 'List'.
    %
:- pred set.list_to_set(list(T)::in, set(T)::out) is det.
:- func set.list_to_set(list(T)) = set(T).

    % Synonyms for set.list_to_set/1.
    %
:- func set.from_list(list(T)) = set(T).
:- func set.set(list(T)) = set(T).

    % 'set.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.
    %
:- pred set.sorted_list_to_set(list(T)::in, set(T)::out) is det.
:- func set.sorted_list_to_set(list(T)) = set(T).

    % A synonym for set.sorted_list_to_set/1.
    %
:- func set.from_sorted_list(list(T)) = set(T).

    % 'set.to_sorted_list(Set, List)' is true iff 'List' is the list
    % of all the members of 'Set', in sorted order without any
    % duplicates.
    %
:- pred set.to_sorted_list(set(T)::in, list(T)::out) is det.
:- func set.to_sorted_list(set(T)) = list(T).

    % 'set.singleton_set(Elem, Set)' is true iff 'Set' is the set
    % containing just the single element 'Elem'.
    %
:- pred set.singleton_set(T, set(T)).
:- mode set.singleton_set(in, out) is det.
:- mode set.singleton_set(out, in) is semidet.

:- func set.make_singleton_set(T) = set(T).

:- pred set.is_singleton(set(T)::in, T::out) is semidet.

    % 'set.equal(SetA, SetB)' is true iff
    % 'SetA' and 'SetB' contain the same elements.
    %
:- pred set.equal(set(T)::in, set(T)::in) is semidet.
```

```
:- pred set.empty(set(T)::in) is semidet.

:- pred set.non_empty(set(T)::in) is semidet.

    % A synonym for set.empty/1.
    %
:- pred set.is_empty(set(T)::in) is semidet.

    % 'set.subset(SetA, SetB)' is true iff 'SetA' is a subset of 'SetB'.
    %
:- pred set.subset(set(T)::in, set(T)::in) is semidet.

    % 'set.superset(SetA, SetB)' is true iff 'SetA' is a
    % superset of 'SetB'.
    %
:- pred set.superset(set(T)::in, set(T)::in) is semidet.

    % 'set.member(X, Set)' is true iff 'X' is a member of 'Set'.
    %
:- pred set.member(T, set(T)).
:- mode set.member(in, in) is semidet.
:- mode set.member(out, in) is nondet.

    % 'set_is_member(X, Set, Result)' returns
    % 'Result = yes' iff 'X' is a member of 'Set'.
    %
:- pred set.is_member(T::in, set(T)::in, bool::out) is det.

    % 'set.contains(Set, X)' is true iff 'X' is a member of 'Set'.
    %
:- pred set.contains(set(T)::in, T::in) is semidet.

    % 'set.insert(X, Set0, Set)' is true iff 'Set' is the union of
    % 'Set0' and the set containing only 'X'.
    %
:- func set.insert(set(T), T) = set(T).
:- pred set.insert(T::in, set(T)::in, set(T)::out) is det.

    % 'set.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 set.insert_new(T::in, set(T)::in, set(T)::out) is semidet.

    % 'set.insert_list(Xs, Set0, Set)' is true iff 'Set' is the union of
    % 'Set0' and the set containing only the members of 'Xs'.
    %
```

```
:- func set.insert_list(set(T), list(T)) = set(T).
:- pred set.insert_list(list(T)::in, set(T)::in, set(T)::out) is det.

    % 'set.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 set.delete(set(T), T) = set(T).
:- pred set.delete(T::in, set(T)::in, set(T)::out) is det.

    % 'set.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 set.delete_list(set(T), list(T)) = set(T).
:- pred set.delete_list(list(T)::in, set(T)::in, set(T)::out) is det.

    % 'set.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 set.remove(T::in, set(T)::in, set(T)::out) is semidet.

    % 'set.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'.
    %
:- pred set.remove_list(list(T)::in, set(T)::in, set(T)::out) is semidet.

    % 'set.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 set.remove_least(T::out, set(T)::in, set(T)::out) is semidet.

    % 'set.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 set.union(set(T), set(T)) = set(T).
:- pred set.union(set(T)::in, set(T)::in, set(T)::out) is det.

    % ‘set.union_list(A, B)’ is true iff ‘B’ is the union of
    % all the sets in ‘A’.
    %
:- func set.union_list(list(set(T))) = set(T).

    % ‘set.power_union(A, B)’ is true iff ‘B’ is the union of
    % all the sets in ‘A’.
    %
:- func set.power_union(set(set(T))) = set(T).
:- pred set.power_union(set(set(T))::in, set(T)::out) is det.

    % ‘set.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 set.intersect(set(T), set(T)) = set(T).
:- pred set.intersect(set(T)::in, set(T)::in, set(T)::out) is det.

    % ‘set.power_intersect(A, B)’ is true iff ‘B’ is the intersection of
    % all the sets in ‘A’.
    %
:- func set.power_intersect(set(set(T))) = set(T).
:- pred set.power_intersect(set(set(T))::in, set(T)::out) is det.

    % ‘set.intersect_list(A, B)’ is true iff ‘B’ is the intersection of
    % all the sets in ‘A’.
    %
:- func set.intersect_list(list(set(T))) = set(T).

    % ‘set.difference(SetA, SetB, Set)’ is true iff ‘Set’ is the
    % set containing all the elements of ‘SetA’ except those that
    % occur in ‘SetB’.
    %
:- func set.difference(set(T), set(T)) = set(T).
:- pred set.difference(set(T)::in, set(T)::in, set(T)::out) is det.
```

```
    % 'set.count(Set, Count)' is true iff 'Set' has 'Count' elements.
    % i.e. 'Count' is the cardinality (size) of the set.
    %
:- func set.count(set(T)) = int.
:- pred set.count(set(T)::in, int::out) is det.

    % Support for higher order set processing.

    % map(F, S) =
    %   list_to_set(list.map(F, to_sorted_list(S))).
    %
:- func set.map(func(T1) = T2, set(T1)) = set(T2).
:- pred set.map(pred(T1, T2), set(T1), set(T2)).
:- mode set.map(pred(in, out) is det, in, out) is det.
:- mode set.map(pred(in, out) is cc_multi, in, out) is cc_multi.
:- mode set.map(pred(in, out) is semidet, in, out) is semidet.
:- mode set.map(pred(in, out) is multi, in, out) is multi.
:- mode set.map(pred(in, out) is nondet, in, out) is nondet.

    % set.map_fold(P, S0, S, A0, A) :-
    %   L0 = set.to_sorted_list(S0),
    %   list.map_foldl(P, L0, L, A0, A),
    %   S = set.list_to_set(L).
    %
:- pred set.map_fold(pred(T1, T2, T3, T3), set(T1), set(T2), T3, T3).
:- mode set.map_fold(pred(in, out, in, out) is det, in, out, in, out) is det.
:- mode set.map_fold(pred(in, out, mdi, muo) is det, in, out, mdi, muo) is det.
:- mode set.map_fold(pred(in, out, di, uo) is det, in, out, di, uo) is det.
:- mode set.map_fold(pred(in, out, in, out) is semidet, in, out,
    in, out) is semidet.
:- mode set.map_fold(pred(in, out, mdi, muo) is semidet, in, out,
    mdi, muo) is semidet.
:- mode set.map_fold(pred(in, out, di, uo) is semidet, in, out,
    di, uo) is semidet.

    % Return the set of items for which the given predicate succeeds.
    % set.filter(P, S) =
    %   sorted_list_to_set(list.filter(P, to_sorted_list(S))).
    %
:- func set.filter(pred(T1), set(T1)) = set(T1).
:- mode set.filter(pred(in) is semidet, in) = out is det.
:- pred set.filter(pred(T1), set(T1), set(T1)).
:- mode set.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 set.filter(pred(T1), set(T1), set(T1), set(T1)).
:- mode set.filter(pred(in) is semidet, in, out, out) is det.

    % set.filter_map(PF, S) =
    %   list_to_set(list.filter_map(PF, to_sorted_list(S))).
    %
:- pred set.filter_map(pred(T1, T2), set(T1), set(T2)).
:- mode set.filter_map(in(pred(in, out) is semidet), in, out) is det.
:- func set.filter_map(func(T1) = T2, set(T1)) = set(T2).
:- mode set.filter_map(func(in) = out is semidet, in) = out is det.

    % set.fold(F, S, A) =
    %   list.foldl(F, to_sorted_list(S), A).
    %
:- func set.fold(func(T, A) = A, set(T), A) = A.
:- func set.foldl(func(T, A) = A, set(T), A) = A.

:- pred set.fold(pred(T, A, A), set(T), A, A).
:- mode set.fold(pred(in, in, out) is det, in, in, out) is det.
:- mode set.fold(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode set.fold(pred(in, di, uo) is det, in, di, uo) is det.
:- mode set.fold(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode set.fold(pred(in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode set.fold(pred(in, di, uo) is semidet, in, di, uo) is semidet.

:- pred set.foldl(pred(T, A, A), set(T), A, A).
:- mode set.foldl(pred(in, in, out) is det, in, in, out) is det.
:- mode set.foldl(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode set.foldl(pred(in, di, uo) is det, in, di, uo) is det.
:- mode set.foldl(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode set.foldl(pred(in, mdi, muo) is semidet, in, mdi, muo) is semidet.
:- mode set.foldl(pred(in, di, uo) is semidet, in, di, uo) is semidet.

:- pred set.fold2(pred(T, A, A, B, B), set(T), A, A, B, B).
:- mode set.fold2(pred(in, in, out, in, out) is det, in,
    in, out, in, out) is det.
:- mode set.fold2(pred(in, in, out, mdi, muo) is det, in,
    in, out, mdi, muo) is det.
:- mode set.fold2(pred(in, in, out, di, uo) is det, in,
    in, out, di, uo) is det.
:- mode set.fold2(pred(in, in, out, in, out) is semidet,
    in, in, out, in, out) is semidet.
:- mode set.fold2(pred(in, in, out, mdi, muo) is semidet,
    in, in, out, mdi, muo) is semidet.
:- mode set.fold2(pred(in, in, out, di, uo) is semidet,
    in, in, out, di, uo) is semidet.
```

```
:- pred set.foldl2(pred(T, A, A, B, B), set(T), A, A, B, B).
:- mode set.foldl2(pred(in, in, out, in, out) is det, in,
    in, out, in, out) is det.
:- mode set.foldl2(pred(in, in, out, mdi, muo) is det, in,
    in, out, mdi, muo) is det.
:- mode set.foldl2(pred(in, in, out, di, uo) is det, in,
    in, out, di, uo) is det.
:- mode set.foldl2(pred(in, in, out, in, out) is semidet,
    in, in, out, in, out) is semidet.
:- mode set.foldl2(pred(in, in, out, mdi, muo) is semidet,
    in, in, out, mdi, muo) is semidet.
:- mode set.foldl2(pred(in, in, out, di, uo) is semidet,
    in, in, out, di, uo) is semidet.

:- pred set.fold3(pred(T, A, A, B, B, C, C), set(T), A, A, B, B, C, C).
:- mode set.fold3(pred(in, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out) is det.
:- mode set.fold3(pred(in, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, mdi, muo) is det.
:- mode set.fold3(pred(in, in, out, in, out, di, uo) is det, in,
    in, out, in, out, di, uo) is det.
:- mode set.fold3(pred(in, in, out, in, out, in, out) is semidet, in,
    in, out, in, out, in, out) is semidet.
:- mode set.fold3(pred(in, in, out, in, out, mdi, muo) is semidet, in,
    in, out, in, out, mdi, muo) is semidet.
:- mode set.fold3(pred(in, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, di, uo) is semidet.

:- pred set.foldl3(pred(T, A, A, B, B, C, C), set(T), A, A, B, B, C, C).
:- mode set.foldl3(pred(in, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out) is det.
:- mode set.foldl3(pred(in, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, mdi, muo) is det.
:- mode set.foldl3(pred(in, in, out, in, out, di, uo) is det, in,
    in, out, in, out, di, uo) is det.
:- mode set.foldl3(pred(in, in, out, in, out, in, out) is semidet, in,
    in, out, in, out, in, out) is semidet.
:- mode set.foldl3(pred(in, in, out, in, out, mdi, muo) is semidet, in,
    in, out, in, out, mdi, muo) is semidet.
:- mode set.foldl3(pred(in, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, di, uo) is semidet.

:- pred set.fold4(pred(T, A, A, B, B, C, C, D, D), set(T), A, A, B, B,
    C, C, D, D).
:- mode set.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 set.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 set.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 set.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 set.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 set.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 set.foldl4(pred(T, A, A, B, B, C, C, D, D), set(T), A, A, B, B,
    C, C, D, D).
:- mode set.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 set.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 set.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 set.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 set.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 set.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 set.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 set.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 set.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 set.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 set.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 set.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 set.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 set.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 set.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 set.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 set.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 set.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 set.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 set.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 set.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 set.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 set.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 set.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 set.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 set.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 set.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 set.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 set.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 set.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 set.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 set.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 set.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 set.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.

    % 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.

    % set.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 set.divide(pred(T)::in(pred(in) is semidet), set(T)::in,
    set(T)::out, set(T)::out) is det.

    % set_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 set.divide_by_set(set(T)::in, set(T)::in, set(T)::out, set(T)::out)
    is det.

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

59 set_bbbtree

```
%---------------------------------------------------%
% vim: ts=4 sw=4 et ft=mercury
%---------------------------------------------------%
% Copyright (C) 1995-1997, 1999-2006, 2010-2012 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
```

```
%----------------------------------------------------%
%
% 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).

    % `set_bbbtree.init(Set)' returns an initialized empty set.
    %
:- func set_bbbtree.init = set_bbbtree(T).
:- pred set_bbbtree.init(set_bbbtree(T)::uo) is det.

    % `set_bbbtree.empty(Set) is true iff `Set' is contains no elements.
    %
:- pred set_bbbtree.empty(set_bbbtree(T)::in) is semidet.

    % A synonym for the above.
    %
:- pred set_bbbtree.is_empty(set_bbbtree(T)::in) is semidet.

:- pred set_bbbtree.non_empty(set_bbbtree(T)::in) is semidet.

    % `set_bbbtree.count(Set, Count)' is true iff `Set' has `Count' elements.
    % i.e. `Count' is the cardinality (size) of the set.
    %
:- func set_bbbtree.count(set_bbbtree(T)) = int.
:- pred set_bbbtree.count(set_bbbtree(T)::in, int::out) is det.

    % `set_bbbtree.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 set_bbbtree.member(T, set_bbbtree(T)).
:- mode set_bbbtree.member(in, in) is semidet.
```

```
:- mode set_bbbtree.member(out, in) is nondet.

    % 'set_bbbtree.is_member(X, Set, Result)' is true iff 'X' is a member
    % of 'Set'.
    %
:- pred set_bbbtree.is_member(T::in, set_bbbtree(T)::in, bool::out) is det.

    % 'set_bbbtree.contains(Set, X)' is true iff 'X' is a member of 'Set'.
    % O(lg n).
    %
:- pred set_bbbtree.contains(set_bbbtree(T)::in, T::in) is semidet.

    % 'set_bbbtree.least(Set, X)' is true iff 'X' is smaller than all
    % the other members of 'Set'.
    %
:- pred set_bbbtree.least(set_bbbtree(T), T).
:- mode set_bbbtree.least(in, out) is semidet.
:- mode set_bbbtree.least(in, in) is semidet.

    % 'set_bbbtree.largest(Set, X)' is true iff 'X' is larger than all
    % the other members of 'Set'.
    %
:- pred set_bbbtree.largest(set_bbbtree(T), T).
:- mode set_bbbtree.largest(in, out) is semidet.
:- mode set_bbbtree.largest(in, in) is semidet.

    % 'set_bbbtree.singleton_set(X, Set)' is true iff 'Set' is the set
    % containing just the single element 'X'.
    %
:- pred set_bbbtree.singleton_set(T, set_bbbtree(T)).
:- mode set_bbbtree.singleton_set(in, out) is det.
:- mode set_bbbtree.singleton_set(in, in) is semidet.
:- mode set_bbbtree.singleton_set(out, in) is semidet.

:- func set_bbbtree.make_singleton_set(T) = set_bbbtree(T).

:- pred set_bbbtree.is_singleton(set_bbbtree(T)::in, T::out) is semidet.

    % 'set_bbbtree.equal(SetA, SetB)' is true iff 'SetA' and 'SetB'
    % contain the same elements.
    %
:- pred set_bbbtree.equal(set_bbbtree(T)::in, set_bbbtree(T)::in) is semidet.

    % 'set_bbbtree.insert(X, Set0, Set)' is true iff 'Set' is the union of
    % 'Set0' and the set containing only 'X'.
    %
:- pred set_bbbtree.insert(T, set_bbbtree(T), set_bbbtree(T)).
```

```
:- mode set_bbbtree.insert(di, di, uo) is det.
:- mode set_bbbtree.insert(in, in, out) is det.

:- func set_bbbtree.insert(set_bbbtree(T), T) = set_bbbtree(T).

    % 'set_bbbtree.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 set_bbbtree.insert_new(T::in,
    set_bbbtree(T)::in, set_bbbtree(T)::out) is semidet.

    % 'set_bbbtree.insert_list(Xs, Set0, Set)' is true iff 'Set' is
    % the union of 'Set0' and the set containing only the members of 'Xs'.
    %
:- pred set_bbbtree.insert_list(list(T)::in,
    set_bbbtree(T)::in, set_bbbtree(T)::out) is det.

:- func set_bbbtree.insert_list(set_bbbtree(T), list(T)) = set_bbbtree(T).

    % 'set_bbbtree.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'.
    %
:- pred set_bbbtree.delete(T, set_bbbtree(T), set_bbbtree(T)).
:- mode set_bbbtree.delete(in, di, uo) is det.
:- mode set_bbbtree.delete(in, in, out) is det.

:- func set_bbbtree.delete(set_bbbtree(T), T) = set_bbbtree(T).

    % 'set_bbbtree.delete_list(Xs, Set0, Set)' is true iff 'Set' is the
    % relative complement of 'Set0' and the set containing only the members
    % of 'Xs'.
    %
:- pred set_bbbtree.delete_list(list(T)::in,
    set_bbbtree(T)::in, set_bbbtree(T)::out) is det.

:- func set_bbbtree.delete_list(set_bbbtree(T), list(T)) = set_bbbtree(T).

    % 'set_bbbtree.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 set_bbbtree.remove(T::in, set_bbbtree(T)::in, set_bbbtree(T)::out)
    is semidet.
```

```
    % 'set_bbbtree.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'.
    %
:- pred set_bbbtree.remove_list(list(T)::in,
    set_bbbtree(T)::in, set_bbbtree(T)::out) is semidet.

    % 'set_bbbtree.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 set_bbbtree.remove_least(T::out,
    set_bbbtree(T)::in, set_bbbtree(T)::out) is semidet.

    % 'set_bbbtree.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 set_bbbtree.remove_largest(T::out,
    set_bbbtree(T)::in, set_bbbtree(T)::out) is semidet.

    % 'set_bbbtree.list_to_set(List, Set)' is true iff 'Set' is the set
    % containing only the members of 'List'. O(n lg n)
    %
:- pred set_bbbtree.list_to_set(list(T)::in, set_bbbtree(T)::out) is det.

:- func set_bbbtree.list_to_set(list(T)) = set_bbbtree(T).

    % A synonym for set_bbtree.list_to_set/1.
    %
:- func set_bbbtree.from_list(list(T)) = set_bbbtree(T).

    % 'set_bbbtree.sorted_list_to_set(List, Set)' is true iff 'Set' is the
    % set containing only the members of 'List'.
    % 'List' must be sorted. O(n).
    %
:- pred set_bbbtree.sorted_list_to_set(list(T)::in, set_bbbtree(T)::out)
    is det.

:- func set_bbbtree.sorted_list_to_set(list(T)) = set_bbbtree(T).

    % A synonym for set_bbbtree.sorted_list_to_set/1.
    %
:- func set_bbbtree.from_sorted_list(list(T)) = set_bbbtree(T).
```

```
    % 'set_bbbtree.sorted_list_to_set_len(List, Set, N)' is true iff
    % 'Set' is the set containing only the members of 'List' and 'N'
    % is the length of the list. If the length of the list is already known
    % then a noticeable speed improvement can be expected over
    % 'set_bbbtree.sorted_list_to_set' as a significant cost involved
    % with 'set_bbbtree.sorted_list_to_set' is the call to list.length.
    % 'List' must be sorted. O(n).
    %
:- pred set_bbbtree.sorted_list_to_set_len(list(T)::in, set_bbbtree(T)::out,
    int::in) is det.

    % 'set_bbbtree.to_sorted_list(Set, List)' is true iff 'List' is the
    % list of all the members of 'Set', in sorted order. O(n).
    %
:- pred set_bbbtree.to_sorted_list(set_bbbtree(T), list(T)).
:- mode set_bbbtree.to_sorted_list(di, uo) is det.
:- mode set_bbbtree.to_sorted_list(in, out) is det.

:- func set_bbbtree.to_sorted_list(set_bbbtree(T)) = list(T).

    % 'set_bbbtree.union(SetA, SetB, Set)' is true iff 'Set' is the union
    % of 'SetA' and 'SetB'.
    %
:- pred set_bbbtree.union(set_bbbtree(T)::in, set_bbbtree(T)::in,
    set_bbbtree(T)::out) is det.

:- func set_bbbtree.union(set_bbbtree(T), set_bbbtree(T)) = set_bbbtree(T).

    % 'set_bbbtree.union_list(Sets) = Set' is true iff 'Set' is the union
    % of all the sets in 'Sets'
    %
:- func set_bbbtree.union_list(list(set_bbbtree(T))) = set_bbbtree(T).

    % 'set_bbbtree.power_union(Sets, Set)' is true iff 'Set' is the union
    % of all the sets in 'Sets'
    %
:- pred set_bbbtree.power_union(set_bbbtree(set_bbbtree(T))::in,
    set_bbbtree(T)::out) is det.

:- func set_bbbtree.power_union(set_bbbtree(set_bbbtree(T))) = set_bbbtree(T).

    % 'set_bbbtree.intersect(SetA, SetB, Set)' is true iff 'Set' is the
    % intersection of 'SetA' and 'SetB'.
    %
:- pred set_bbbtree.intersect(set_bbbtree(T)::in, set_bbbtree(T)::in,
    set_bbbtree(T)::out) is det.
```

```
:- func set_bbbtree.intersect(set_bbbtree(T), set_bbbtree(T)) = set_bbbtree(T).

    % 'set_bbbtree.power_intersect(Sets, Set) is true iff 'Set' is the
    % intersection of the sets in 'Sets'.
    %
:- pred set_bbbtree.power_intersect(set_bbbtree(set_bbbtree(T))::in,
    set_bbbtree(T)::out) is det.

:- func set_bbbtree.power_intersect(set_bbbtree(set_bbbtree(T)))
    = set_bbbtree(T).

    % 'set_bbbtree.intersect_list(Sets) = Set is true iff 'Set' is the
    % intersection of the sets in 'Sets'.
    %
:- func set_bbbtree.intersect_list(list(set_bbbtree(T))) = set_bbbtree(T).

    % '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'.
    %
:- pred set_bbbtree.difference(set_bbbtree(T)::in, set_bbbtree(T)::in,
    set_bbbtree(T)::out) is det.

:- func set_bbbtree.difference(set_bbbtree(T), set_bbbtree(T))
    = set_bbbtree(T).

    % 'set_bbbtree.subset(SetA, SetB)' is true iff all the elements of
    % 'SetA' are also elements of 'SetB'.
    %
:- pred set_bbbtree.subset(set_bbbtree(T)::in, set_bbbtree(T)::in) is semidet.

    % 'set_bbbtree.superset(SetA, SetB)' is true iff all the elements of
    % 'SetB' are also elements of 'SetA'.
    %
:- pred set_bbbtree.superset(set_bbbtree(T)::in, set_bbbtree(T)::in)
    is semidet.

:- func set_bbbtree.fold(func(T1, T2) = T2, set_bbbtree(T1), T2) = T2.
:- pred set_bbbtree.fold(pred(T1, T2, T2), set_bbbtree(T1), T2, T2).
:- mode set_bbbtree.fold(pred(in, in, out) is det, in, in, out) is det.
:- mode set_bbbtree.fold(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode set_bbbtree.fold(pred(in, di, uo) is det, in, di, uo) is det.
:- mode set_bbbtree.fold(pred(in, in, out) is semidet, in, in, out)
    is semidet.
:- mode set_bbbtree.fold(pred(in, mdi, muo) is semidet, in, mdi, muo)
    is semidet.
:- mode set_bbbtree.fold(pred(in, di, uo) is semidet, in, di, uo)
```

```
    is semidet.

:- pred set_bbbtree.fold2(pred(T1, T2, T2, T3, T3), set_bbbtree(T1),
    T2, T2, T3, T3).
:- mode set_bbbtree.fold2(pred(in, in, out, in, out) is det, in,
    in, out, in, out) is det.
:- mode set_bbbtree.fold2(pred(in, in, out, mdi, muo) is det, in,
    in, out, mdi, muo) is det.
:- mode set_bbbtree.fold2(pred(in, in, out, di, uo) is det, in,
    in, out, di, uo) is det.
:- mode set_bbbtree.fold2(pred(in, in, out, in, out) is semidet, in,
    in, out, in, out) is semidet.
:- mode set_bbbtree.fold2(pred(in, in, out, mdi, muo) is semidet, in,
    in, out, mdi, muo) is semidet.
:- mode set_bbbtree.fold2(pred(in, in, out, di, uo) is semidet, in,
    in, out, di, uo) is semidet.

:- pred set_bbbtree.fold3(pred(T1, T2, T2, T3, T3, T4, T4),
    set_bbbtree(T1), T2, T2, T3, T3, T4, T4).
:- mode set_bbbtree.fold3(pred(in, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out) is det.
:- mode set_bbbtree.fold3(pred(in, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, mdi, muo) is det.
:- mode set_bbbtree.fold3(pred(in, in, out, in, out, di, uo) is det, in,
    in, out, in, out, di, uo) is det.
:- mode set_bbbtree.fold3(pred(in, in, out, in, out, in, out) is semidet, in,
    in, out, in, out, in, out) is semidet.
:- mode set_bbbtree.fold3(pred(in, in, out, in, out, mdi, muo) is semidet, in,
    in, out, in, out, mdi, muo) is semidet.
:- mode set_bbbtree.fold3(pred(in, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, di, uo) is semidet.

:- pred set_bbbtree.fold4(pred(T1, T2, T2, T3, T3, T4, T4, T5, T5),
    set_bbbtree(T1), T2, T2, T3, T3, T4, T4, T5, T5).
:- mode set_bbbtree.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 set_bbbtree.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 set_bbbtree.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 set_bbbtree.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 set_bbbtree.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 set_bbbtree.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 set_bbbtree.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 set_bbbtree.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 set_bbbtree.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 set_bbbtree.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 set_bbbtree.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 set_bbbtree.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 set_bbbtree.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 set_bbbtree.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 set_bbbtree.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 set_bbbtree.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 set_bbbtree.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 set_bbbtree.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 set_bbbtree.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 set_bbbtree.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.

    % 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.

:- func set_bbbtree.map(func(T1) = T2, set_bbbtree(T1)) = set_bbbtree(T2).

:- func set_bbbtree.filter_map(func(T1) = T2, set_bbbtree(T1))
    = set_bbbtree(T2).
:- mode set_bbbtree.filter_map(func(in) = out is semidet, in) = out is det.

    % set_bbbtree.filter(Pred, Items, Trues):
    % Return the set of items for which Pred succeeds.
    %
:- pred set_bbbtree.filter(pred(T)::in(pred(in) is semidet),
    set_bbbtree(T)::in, set_bbbtree(T)::out) is det.

    % set_bbbtree.filter(Pred, Items, Trues, Falses):
    % Return the set of items for which Pred succeeds,
    % and the set for which it fails.
    %
:- pred set_bbbtree.filter(pred(T)::in(pred(in) is semidet),
    set_bbbtree(T)::in, set_bbbtree(T)::out, set_bbbtree(T)::out) is det.

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

60 set_ctree234

```
%-----------------------------------------------------%
% vim: ts=4 sw=4 et ft=mercury
%-----------------------------------------------------%
% Copyright (C) 2005-2006, 2010-2012 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% 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).

    % 'set_ctree234.init = Set' is true iff 'Set' is an empty set.
    %
:- func set_ctree234.init = set_ctree234(T).

    % 'set_ctree234.singleton_set(Elem, Set)' is true iff 'Set' is the set
    % containing just the single element 'Elem'.
    %
:- pred set_ctree234.singleton_set(T, set_ctree234(T)).
:- mode set_ctree234.singleton_set(in, out) is det.
:- mode set_ctree234.singleton_set(out, in) is semidet.

:- func set_ctree234.make_singleton_set(T) = set_ctree234(T).

:- pred set_ctree234.is_singleton(set_ctree234(T)::in, T::out) is semidet.

    % 'set_ctree234.empty(Set)' is true iff 'Set' is an empty set.
    %
:- pred set_ctree234.empty(set_ctree234(_T)::in) is semidet.

:- pred set_ctree234.is_empty(set_ctree234(_T)::in) is semidet.

:- pred set_ctree234.non_empty(set_ctree234(T)::in) is semidet.
```

```
    % 'set_ctree234.member(X, Set)' is true iff 'X' is a member of 'Set'.
    %
:- pred set_ctree234.member(T, set_ctree234(T)).
:- mode set_ctree234.member(in, in) is semidet.
:- mode set_ctree234.member(out, in) is nondet.

    % 'set_ctree234.one_member(Set, X)' is true iff 'X' is a member of 'Set'.
    %
:- pred set_ctree234.one_member(set_ctree234(T)::in, T::out) is nondet.

    % 'set_ctree234.is_member(Set, X, Result)' returns
    % 'Result = yes' iff 'X' is a member of 'Set'.
    %
:- func set_ctree234.is_member(set_ctree234(T), T) = bool.
:- pred set_ctree234.is_member(set_ctree234(T)::in, T::in, bool::out) is det.

    % 'set_ctree234.contains(Set, X)' is true iff 'X' is a member of 'Set'.
    %
:- pred set_ctree234.contains(set_ctree234(T)::in, T::in) is semidet.

    % 'set_ctree234.list_to_set(List) = Set' is true iff 'Set' is the set
    % containing only the members of 'List'.
    %
:- func set_ctree234.list_to_set(list(T)) = set_ctree234(T).

:- func set_ctree234.from_list(list(T)) = set_ctree234(T).

    % 'set_ctree234.sorted_list_to_set(List) = Set' is true iff 'Set' is
    % the set containing only the members of 'List'. 'List' must be sorted.
    %
:- func set_ctree234.sorted_list_to_set(list(T)) = set_ctree234(T).

    % 'set_ctree234.to_sorted_list(Set) = List' is true iff 'List' is the
    % list of all the members of 'Set', in sorted order.
    %
:- func set_ctree234.to_sorted_list(set_ctree234(T)) = list(T).

    % 'set_ctree234.equal(SetA, SetB)' is true iff
    % 'SetA' and 'SetB' contain the same elements.
    %
:- pred set_ctree234.equal(set_ctree234(T)::in, set_ctree234(T)::in)
    is semidet.

    % 'set_ctree234.subset(SetA, SetB)' is true iff 'SetA' is a subset of
    % 'SetB'.
    %
:- pred set_ctree234.subset(set_ctree234(T)::in, set_ctree234(T)::in)
```

```
    is semidet.

    % 'set_ctree234.superset(SetA, SetB)' is true iff 'SetA' is a
    % superset of 'SetB'.
    %
:- pred set_ctree234.superset(set_ctree234(T)::in, set_ctree234(T)::in)
    is semidet.

    % 'set_ctree234.insert(X, Set0, Set)' is true iff 'Set' is the union
    % of 'Set0' and the set containing only 'X'.
    %
:- func set_ctree234.insert(T, set_ctree234(T)) = set_ctree234(T).
:- pred set_ctree234.insert(T::in, set_ctree234(T)::in, set_ctree234(T)::out)
    is det.

    % 'set_ctree234.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 set_ctree234.insert_new(T::in,
    set_ctree234(T)::in, set_ctree234(T)::out) is semidet.

    % 'set_ctree234.insert_list(Xs, Set0, Set)' is true iff 'Set' is the
    % union of 'Set0' and the set containing only the members of 'Xs'.
    %
:- func set_ctree234.insert_list(list(T), set_ctree234(T)) = set_ctree234(T).
:- pred set_ctree234.insert_list(list(T)::in,
    set_ctree234(T)::in, set_ctree234(T)::out) is det.

    % 'set_ctree234.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 set_ctree234.delete(T, set_ctree234(T)) = set_ctree234(T).
:- pred set_ctree234.delete(T::in, set_ctree234(T)::in, set_ctree234(T)::out)
    is det.

    % 'set_ctree234.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 set_ctree234.delete_list(list(T), set_ctree234(T)) = set_ctree234(T).
:- pred set_ctree234.delete_list(list(T)::in,
    set_ctree234(T)::in, set_ctree234(T)::out) is det.

    % 'set_ctree234.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 set_ctree234.remove(T::in, set_ctree234(T)::in, set_ctree234(T)::out)
    is semidet.

    % 'set_ctree234.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'.
    %
:- pred set_ctree234.remove_list(list(T)::in,
    set_ctree234(T)::in, set_ctree234(T)::out) is semidet.

    % 'set_ctree234.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 set_ctree234.remove_least(T::out,
    set_ctree234(T)::in, set_ctree234(T)::out) is semidet.

    % 'set_ctree234.union(SetA, SetB) = Set' is true iff 'Set' is the union
    % of 'SetA' and 'SetB'.
    %
:- func set_ctree234.union(set_ctree234(T), set_ctree234(T)) = set_ctree234(T).
:- pred set_ctree234.union(set_ctree234(T)::in, set_ctree234(T)::in,
    set_ctree234(T)::out) is det.

    % 'set_ctree234.union_list(A, B)' is true iff 'B' is the union of
    % all the sets in 'A'
    %
:- func set_ctree234.union_list(list(set_ctree234(T))) = set_ctree234(T).
:- pred set_ctree234.union_list(list(set_ctree234(T))::in,
    set_ctree234(T)::out) is det.

    % 'set_ctree234.power_union(A) = B' is true iff 'B' is the union of
    % all the sets in 'A'
    %
:- func set_ctree234.power_union(set_ctree234(set_ctree234(T)))
    = set_ctree234(T).
:- pred set_ctree234.power_union(set_ctree234(set_ctree234(T))::in,
    set_ctree234(T)::out) is det.

    % 'set_ctree234.intersect(SetA, SetB) = Set' is true iff 'Set' is the
    % intersection of 'SetA' and 'SetB'.
    %
```

```
:- func set_ctree234.intersect(set_ctree234(T), set_ctree234(T))
    = set_ctree234(T).
:- pred set_ctree234.intersect(set_ctree234(T)::in, set_ctree234(T)::in,
    set_ctree234(T)::out) is det.

    % 'set_ctree234.power_intersect(A, B)' is true iff 'B' is the
    % intersection of all the sets in 'A'.
    %
:- func set_ctree234.power_intersect(set_ctree234(set_ctree234(T)))
    = set_ctree234(T).

    % 'set_ctree234.intersect_list(A) = B' is true iff 'B' is the
    % intersection of all the sets in 'A'.
    %
:- func set_ctree234.intersect_list(list(set_ctree234(T))) = set_ctree234(T).

    % 'set_ctree234.difference(SetA, SetB, Set)' is true iff 'Set' is the
    % set containing all the elements of 'SetA' except those that
    % occur in 'SetB'.
    %
:- func set_ctree234.difference(set_ctree234(T), set_ctree234(T))
    = set_ctree234(T).
:- pred set_ctree234.difference(set_ctree234(T)::in, set_ctree234(T)::in,
    set_ctree234(T)::out) is det.

    % 'set_ctree234.count(Set, Count)' is true iff 'Set' has
    % 'Count' elements.
    %
:- func set_ctree234.count(set_ctree234(T)) = int.

:- func set_ctree234.map(func(T1) = T2, set_ctree234(T1)) = set_ctree234(T2).
:- pred set_ctree234.map(pred(T1, T2)::in(pred(in, out) is det),
    set_ctree234(T1)::in, set_ctree234(T2)::out) is det.

:- pred set_ctree234.filter_map(pred(T1, T2)::in(pred(in, out) is semidet),
    set_ctree234(T1)::in, set_ctree234(T2)::out) is det.

:- func set_ctree234.filter_map(func(T1) = T2, set_ctree234(T1))
    = set_ctree234(T2).
:- mode set_ctree234.filter_map(func(in) = out is semidet, in) = out is det.

:- func set_ctree234.fold(func(T1, T2) = T2, set_ctree234(T1), T2) = T2.
:- pred set_ctree234.fold(pred(T1, T2, T2), set_ctree234(T1), T2, T2).
:- mode set_ctree234.fold(pred(in, in, out) is det, in, in, out) is det.
:- mode set_ctree234.fold(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode set_ctree234.fold(pred(in, di, uo) is det, in, di, uo) is det.
:- mode set_ctree234.fold(pred(in, in, out) is semidet, in, in, out)
```

```
    is semidet.
:- mode set_ctree234.fold(pred(in, mdi, muo) is semidet, in, mdi, muo)
    is semidet.
:- mode set_ctree234.fold(pred(in, di, uo) is semidet, in, di, uo)
    is semidet.

:- pred set_ctree234.fold2(pred(T1, T2, T2, T3, T3), set_ctree234(T1),
    T2, T2, T3, T3) is det.
:- mode set_ctree234.fold2(pred(in, in, out, in, out) is det,
    in, in, out, in, out) is det.
:- mode set_ctree234.fold2(pred(in, in, out, mdi, muo) is det,
    in, in, out, mdi, muo) is det.
:- mode set_ctree234.fold2(pred(in, in, out, di, uo) is det,
    in, in, out, di, uo) is det.
:- mode set_ctree234.fold2(pred(in, in, out, in, out) is semidet,
    in, in, out, in, out) is semidet.
:- mode set_ctree234.fold2(pred(in, in, out, mdi, muo) is semidet,
    in, in, out, mdi, muo) is semidet.
:- mode set_ctree234.fold2(pred(in, in, out, di, uo) is semidet,
    in, in, out, di, uo) is semidet.

:- pred set_ctree234.fold3(
    pred(T1, T2, T2, T3, T3, T4, T4), set_ctree234(T1),
    T2, T2, T3, T3, T4, T4).
:- mode set_ctree234.fold3(pred(in, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out) is det.
:- mode set_ctree234.fold3(pred(in, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, mdi, muo) is det.
:- mode set_ctree234.fold3(pred(in, in, out, in, out, di, uo) is det, in,
    in, out, in, out, di, uo) is det.
:- mode set_ctree234.fold3(pred(in, in, out, in, out, in, out) is semidet, in,
    in, out, in, out, in, out) is semidet.
:- mode set_ctree234.fold3(pred(in, in, out, in, out, mdi, muo) is semidet, in,
    in, out, in, out, mdi, muo) is semidet.
:- mode set_ctree234.fold3(pred(in, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, di, uo) is semidet.

:- pred set_ctree234.fold4(
    pred(T1, T2, T2, T3, T3, T4, T4, T5, T5), set_ctree234(T1),
    T2, T2, T3, T3, T4, T4, T5, T5).
:- mode set_ctree234.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 set_ctree234.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 set_ctree234.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 set_ctree234.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 set_ctree234.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 set_ctree234.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 set_ctree234.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 set_ctree234.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 set_ctree234.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 set_ctree234.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 set_ctree234.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 set_ctree234.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 set_ctree234.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 set_ctree234.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 set_ctree234.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 set_ctree234.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 set_ctree234.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 set_ctree234.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 set_ctree234.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 set_ctree234.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.

    % all_true(Pred, Set) succeeds iff Pred(Element) succeeds
    % for all the elements of Set.
    %
:- pred set_ctree234.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 set_ctree234.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 set_ctree234.filter(pred(T)::in(pred(in) is semidet),
    set_ctree234(T)::in, set_ctree234(T)::out, set_ctree234(T)::out) is det.

    % set_ctree234.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 set_ctree234.divide(pred(T)::in(pred(in) is semidet),
    set_ctree234(T)::in, set_ctree234(T)::out, set_ctree234(T)::out) is det.

    % set_ctree234.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 set_ctree234.divide_by_set(set_ctree234(T)::in, set_ctree234(T)::in,
    set_ctree234(T)::out, set_ctree234(T)::out) is det.

:- pred verify_depths(set_ctree234(T)::in, list(int)::out) is det.

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

61 set_ordlist

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 1996-1997,1999-2002, 2004-2006, 2008-2012 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%---------------------------------------------------%
%
% 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).

    % ‘set_ordlist.init(Set)’ is true iff ‘Set’ is an empty set.
    %
:- pred set_ordlist.init(set_ordlist(_T)::uo) is det.
:- func set_ordlist.init = set_ordlist(T).

    % ‘set_ordlist.list_to_set(List, Set)’ is true iff ‘Set’ is the set
    % containing only the members of ‘List’.
    %
:- pred set_ordlist.list_to_set(list(T)::in, set_ordlist(T)::out) is det.
:- func set_ordlist.list_to_set(list(T)) = set_ordlist(T).

    % A synonym for set_ordlist.list_to_set/1.
    %
:- func set_ordlist.from_list(list(T)) = set_ordlist(T).

    % ‘set_ordlist.sorted_list_to_set(List, Set)’ is true iff ‘Set’ is
    % the set containing only the members of ‘List’.  ‘List’ must be sorted.
```

```
    %
:- pred set_ordlist.sorted_list_to_set(list(T)::in, set_ordlist(T)::out)
    is det.
:- func set_ordlist.sorted_list_to_set(list(T)) = set_ordlist(T).

    % A synonym for set_ordlist.sorted_list_to_set/1.
    %
:- func set_ordlist.from_sorted_list(list(T)) = set_ordlist(T).

    % 'set_ordlist.to_sorted_list(Set, List)' is true iff 'List' is the
    % list of all the members of 'Set', in sorted order.
    %
:- pred set_ordlist.to_sorted_list(set_ordlist(T)::in, list(T)::out) is det.
:- func set_ordlist.to_sorted_list(set_ordlist(T)) = list(T).

    % 'set_ordlist.singleton_set(Elem, Set)' is true iff 'Set' is the set
    % containing just the single element 'Elem'.
    %
:- pred set_ordlist.singleton_set(T, set_ordlist(T)).
:- mode set_ordlist.singleton_set(in, out) is det.
:- mode set_ordlist.singleton_set(out, in) is semidet.

:- func set_ordlist.make_singleton_set(T) = set_ordlist(T).
:- pred set_ordlist.is_singleton(set_ordlist(T)::in, T::out) is semidet.

    % 'set_ordlist.equal(SetA, SetB)' is true iff
    % 'SetA' and 'SetB' contain the same elements.
    %
:- pred set_ordlist.equal(set_ordlist(T)::in, set_ordlist(T)::in) is semidet.

    % 'set_ordlist.empty(Set)' is true iff 'Set' is an empty set.
    %
:- pred set_ordlist.empty(set_ordlist(_T)::in) is semidet.
:- pred set_ordlist.is_empty(set_ordlist(T)::in) is semidet.

:- pred set_ordlist.non_empty(set_ordlist(T)::in) is semidet.
:- pred set_ordlist.is_non_empty(set_ordlist(T)::in) is semidet.

    % 'set_ordlist.subset(SetA, SetB)' is true iff 'SetA' is a subset of
    % 'SetB'.
    %
:- pred set_ordlist.subset(set_ordlist(T)::in, set_ordlist(T)::in) is semidet.

    % 'set_ordlist.superset(SetA, SetB)' is true iff 'SetA' is a
    % superset of 'SetB'.
    %
:- pred set_ordlist.superset(set_ordlist(T)::in, set_ordlist(T)::in)
```

```
    is semidet.

    % 'set_ordlist.member(X, Set)' is true iff 'X' is a member of 'Set'.
    %
:- pred set_ordlist.member(T, set_ordlist(T)).
:- mode set_ordlist.member(in, in) is semidet.
:- mode set_ordlist.member(out, in) is nondet.

    % 'set_ordlist.is_member(X, Set, Result)' returns
    % 'Result = yes' iff 'X' is a member of 'Set'.
    %
:- pred set_ordlist.is_member(T::in, set_ordlist(T)::in, bool::out) is det.

    % 'set_ordlist.contains(Set, X)' is true iff 'X' is a member of 'Set'.
    %
:- pred set_ordlist.contains(set_ordlist(T)::in, T::in) is semidet.

    % 'set_ordlist.insert(X, Set0, Set)' is true iff 'Set' is the union
    % of 'Set0' and the set containing only 'X'.
    %
:- pred set_ordlist.insert(T::in, set_ordlist(T)::in, set_ordlist(T)::out)
    is det.
:- func set_ordlist.insert(set_ordlist(T), T) = set_ordlist(T).

    % 'set_ordlist.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 set_ordlist.insert_new(T::in,
    set_ordlist(T)::in, set_ordlist(T)::out) is semidet.

    % 'set_ordlist.insert_list(Xs, Set0, Set)' is true iff 'Set' is the
    % union of 'Set0' and the set containing only the members of 'Xs'.
    %
:- pred set_ordlist.insert_list(list(T)::in,
    set_ordlist(T)::in, set_ordlist(T)::out) is det.
:- func set_ordlist.insert_list(set_ordlist(T), list(T)) = set_ordlist(T).

    % 'set_ordlist.delete(Set0, X, 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'.
    %
:- pred set_ordlist.delete(T::in, set_ordlist(T)::in, set_ordlist(T)::out)
    is det.
:- func set_ordlist.delete(set_ordlist(T), T) = set_ordlist(T).
```

```
    % 'set_ordlist.delete_list(Xs, Set0, Set)' is true iff 'Set' is the
    % relative complement of 'Set0' and the set containing only the members
    % of 'Xs'.
    %
:- pred set_ordlist.delete_list(list(T)::in,
    set_ordlist(T)::in, set_ordlist(T)::out) is det.
:- func set_ordlist.delete_list(set_ordlist(T), list(T)) = set_ordlist(T).

    % 'set_ordlist.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 set_ordlist.remove(T::in, set_ordlist(T)::in, set_ordlist(T)::out)
    is semidet.

    % 'set_ordlist.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'.
    %
:- pred set_ordlist.remove_list(list(T)::in,
    set_ordlist(T)::in, set_ordlist(T)::out) is semidet.

    % 'set_ordlist.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 set_ordlist.remove_least(T::out,
    set_ordlist(T)::in, set_ordlist(T)::out) is semidet.

    % 'set_ordlist.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.
    %
:- pred set_ordlist.union(set_ordlist(T)::in, set_ordlist(T)::in,
    set_ordlist(T)::out) is det.
:- func set_ordlist.union(set_ordlist(T), set_ordlist(T)) = set_ordlist(T).

    % 'set_ordlist.union_list(A, B)' is true iff 'B' is the union of
    % all the sets in 'A'
    %
:- pred set_ordlist.union_list(list(set_ordlist(T))::in, set_ordlist(T)::out)
    is det.
:- func set_ordlist.union_list(list(set_ordlist(T))) = set_ordlist(T).
```

```
    % 'set_ordlist.power_union(A, B)' is true iff 'B' is the union of
    % all the sets in 'A'
    %
:- pred set_ordlist.power_union(set_ordlist(set_ordlist(T))::in,
    set_ordlist(T)::out) is det.
:- func set_ordlist.power_union(set_ordlist(set_ordlist(T))) = set_ordlist(T).

    % 'set_ordlist.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.
    %
:- pred set_ordlist.intersect(set_ordlist(T), set_ordlist(T), set_ordlist(T)).
:- mode set_ordlist.intersect(in, in, out) is det.
:- mode set_ordlist.intersect(in, in, in) is semidet.
:- func set_ordlist.intersect(set_ordlist(T), set_ordlist(T))
    = set_ordlist(T).

    % 'set_ordlist.power_intersect(A, B)' is true iff 'B' is the
    % intersection of all the sets in 'A'.
    %
:- pred set_ordlist.power_intersect(set_ordlist(set_ordlist(T))::in,
    set_ordlist(T)::out) is det.
:- func set_ordlist.power_intersect(set_ordlist(set_ordlist(T)))
    = set_ordlist(T).

    % 'set_ordlist.intersect_list(A) = B' is true iff 'B' is the
    % intersection of all the sets in 'A'.
    %
:- func set_ordlist.intersect_list(list(set_ordlist(T))) = set_ordlist(T).
:- pred set_ordlist.intersect_list(list(set_ordlist(T))::in,
    set_ordlist(T)::out) is det.

    % 'set_ordlist.difference(SetA, SetB, Set)' is true iff 'Set' is the
    % set containing all the elements of 'SetA' except those that
    % occur in 'SetB'.
    %
:- pred set_ordlist.difference(set_ordlist(T)::in, set_ordlist(T)::in,
    set_ordlist(T)::out) is det.
:- func set_ordlist.difference(set_ordlist(T), set_ordlist(T))
    = set_ordlist(T).

    % 'set_ordlist.count(Set, Count)' is true iff 'Set' has
    % 'Count' elements.
    %
:- pred set_ordlist.count(set_ordlist(T)::in, int::out) is det.
:- func set_ordlist.count(set_ordlist(T)) = int.
```

```
    % Return the set of items for which the given predicate succeeds.
    %
:- func set_ordlist.filter(pred(T1), set_ordlist(T1)) = set_ordlist(T1).
:- mode set_ordlist.filter(pred(in) is semidet, in) = out is det.
:- pred set_ordlist.filter(pred(T1), set_ordlist(T1), set_ordlist(T1)).
:- mode set_ordlist.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 set_ordlist.filter(pred(T1), set_ordlist(T1),
    set_ordlist(T1), set_ordlist(T1)).
:- mode set_ordlist.filter(pred(in) is semidet, in, out, out) is det.

:- func set_ordlist.map(func(T1) = T2, set_ordlist(T1)) = set_ordlist(T2).

:- func set_ordlist.filter_map(func(T1) = T2, set_ordlist(T1))
    = set_ordlist(T2).
:- mode set_ordlist.filter_map(func(in) = out is semidet, in) = out is det.

:- pred set_ordlist.filter_map(pred(T1, T2), set_ordlist(T1),
    set_ordlist(T2)).
:- mode set_ordlist.filter_map(pred(in, out) is semidet, in, out) is det.

:- func set_ordlist.fold(func(T1, T2) = T2, set_ordlist(T1), T2) = T2.
:- pred set_ordlist.fold(pred(T1, T2, T2), set_ordlist(T1), T2, T2).
:- mode set_ordlist.fold(pred(in, in, out) is det, in, in, out) is det.
:- mode set_ordlist.fold(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode set_ordlist.fold(pred(in, di, uo) is det, in, di, uo) is det.
:- mode set_ordlist.fold(pred(in, in, out) is semidet, in, in, out)
    is semidet.
:- mode set_ordlist.fold(pred(in, mdi, muo) is semidet, in, mdi, muo)
    is semidet.
:- mode set_ordlist.fold(pred(in, di, uo) is semidet, in, di, uo)
    is semidet.

:- func set_ordlist.foldl(func(T1, T2) = T2, set_ordlist(T1), T2) = T2.
:- pred set_ordlist.foldl(pred(T1, T2, T2), set_ordlist(T1), T2, T2).
:- mode set_ordlist.foldl(pred(in, in, out) is det, in, in, out) is det.
:- mode set_ordlist.foldl(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode set_ordlist.foldl(pred(in, di, uo) is det, in, di, uo) is det.
:- mode set_ordlist.foldl(pred(in, in, out) is semidet, in, in, out)
    is semidet.
:- mode set_ordlist.foldl(pred(in, mdi, muo) is semidet, in, mdi, muo)
    is semidet.
:- mode set_ordlist.foldl(pred(in, di, uo) is semidet, in, di, uo)
    is semidet.
```

```
:- pred set_ordlist.fold2(pred(T1, T2, T2, T3, T3), set_ordlist(T1),
    T2, T2, T3, T3).
:- mode set_ordlist.fold2(pred(in, in, out, in, out) is det, in,
    in, out, in, out) is det.
:- mode set_ordlist.fold2(pred(in, in, out, mdi, muo) is det, in,
    in, out, mdi, muo) is det.
:- mode set_ordlist.fold2(pred(in, in, out, di, uo) is det, in,
    in, out, di, uo) is det.
:- mode set_ordlist.fold2(pred(in, in, out, in, out) is semidet, in,
    in, out, in, out) is semidet.
:- mode set_ordlist.fold2(pred(in, in, out, mdi, muo) is semidet, in,
    in, out, mdi, muo) is semidet.
:- mode set_ordlist.fold2(pred(in, in, out, di, uo) is semidet, in,
    in, out, di, uo) is semidet.

:- pred set_ordlist.foldl2(pred(T1, T2, T2, T3, T3), set_ordlist(T1),
    T2, T2, T3, T3).
:- mode set_ordlist.foldl2(pred(in, in, out, in, out) is det, in,
    in, out, in, out) is det.
:- mode set_ordlist.foldl2(pred(in, in, out, mdi, muo) is det, in,
    in, out, mdi, muo) is det.
:- mode set_ordlist.foldl2(pred(in, in, out, di, uo) is det, in,
    in, out, di, uo) is det.
:- mode set_ordlist.foldl2(pred(in, in, out, in, out) is semidet, in,
    in, out, in, out) is semidet.
:- mode set_ordlist.foldl2(pred(in, in, out, mdi, muo) is semidet, in,
    in, out, mdi, muo) is semidet.
:- mode set_ordlist.foldl2(pred(in, in, out, di, uo) is semidet, in,
    in, out, di, uo) is semidet.

:- pred set_ordlist.fold3(pred(T1, T2, T2, T3, T3, T4, T4),
    set_ordlist(T1), T2, T2, T3, T3, T4, T4).
:- mode set_ordlist.fold3(pred(in, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out) is det.
:- mode set_ordlist.fold3(pred(in, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, mdi, muo) is det.
:- mode set_ordlist.fold3(pred(in, in, out, in, out, di, uo) is det, in,
    in, out, in, out, di, uo) is det.
:- mode set_ordlist.fold3(pred(in, in, out, in, out, in, out) is semidet, in,
    in, out, in, out, in, out) is semidet.
:- mode set_ordlist.fold3(pred(in, in, out, in, out, mdi, muo) is semidet, in,
    in, out, in, out, mdi, muo) is semidet.
:- mode set_ordlist.fold3(pred(in, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, di, uo) is semidet.

:- pred set_ordlist.foldl3(pred(T1, T2, T2, T3, T3, T4, T4),
```

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

:- pred set_ordlist.fold4(pred(T1, T2, T2, T3, T3, T4, T4, T5, T5),
    set_ordlist(T1), T2, T2, T3, T3, T4, T4, T5, T5).
:- mode set_ordlist.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 set_ordlist.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 set_ordlist.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 set_ordlist.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 set_ordlist.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 set_ordlist.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 set_ordlist.foldl4(pred(T1, T2, T2, T3, T3, T4, T4, T5, T5),
    set_ordlist(T1), T2, T2, T3, T3, T4, T4, T5, T5).
:- mode set_ordlist.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 set_ordlist.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 set_ordlist.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 set_ordlist.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 set_ordlist.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 set_ordlist.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 set_ordlist.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 set_ordlist.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 set_ordlist.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 set_ordlist.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 set_ordlist.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 set_ordlist.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 set_ordlist.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 set_ordlist.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 set_ordlist.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 set_ordlist.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 set_ordlist.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 set_ordlist.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 set_ordlist.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 set_ordlist.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 set_ordlist.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 set_ordlist.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 set_ordlist.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 set_ordlist.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 set_ordlist.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 set_ordlist.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 set_ordlist.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 set_ordlist.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 set_ordlist.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 set_ordlist.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 set_ordlist.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 set_ordlist.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 set_ordlist.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 set_ordlist.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.
```

```
    % 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.

    % set_ordlist.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 set_ordlist.divide(pred(T)::in(pred(in) is semidet),
    set_ordlist(T)::in, set_ordlist(T)::out, set_ordlist(T)::out) is det.

    % set_ordlist.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 set_ordlist.divide_by_set(set_ordlist(T)::in, set_ordlist(T)::in,
    set_ordlist(T)::out, set_ordlist(T)::out) is det.

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

62 set_tree234

```
%---------------------------------------------------%
% vim: ts=4 sw=4 et ft=mercury
%---------------------------------------------------%
% Copyright (C) 2005-2006, 2009-2012 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%---------------------------------------------------%
%
% 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).

    % 'set_tree234.init = Set' is true iff 'Set' is an empty set.
    %
:- func set_tree234.init = set_tree234(T).

    % 'set_tree234.singleton_set(Elem, Set)' is true iff 'Set' is the set
    % containing just the single element 'Elem'.
    %
:- pred set_tree234.singleton_set(T, set_tree234(T)).
:- mode set_tree234.singleton_set(in, out) is det.
:- mode set_tree234.singleton_set(out, in) is semidet.

:- func set_tree234.make_singleton_set(T) = set_tree234(T).

:- pred set_tree234.is_singleton(set_tree234(T)::in, T::out) is semidet.

    % 'set_tree234.empty(Set)' is true iff 'Set' is an empty set.
    %
:- pred set_tree234.empty(set_tree234(_T)::in) is semidet.

    % A synonym for the above.
    %
:- pred set_tree234.is_empty(set_tree234(_T)::in) is semidet.

:- pred set_tree234.non_empty(set_tree234(T)::in) is semidet.
:- pred set_tree234.is_non_empty(set_tree234(T)::in) is semidet.

    % 'set_tree234.member(X, Set)' is true iff 'X' is a member of 'Set'.
    %
:- pred set_tree234.member(T, set_tree234(T)).
:- mode set_tree234.member(in, in) is semidet.
:- mode set_tree234.member(out, in) is nondet.

    % 'set_tree234.is_member(Set, X, Result)' returns
    % 'Result = yes' iff 'X' is a member of 'Set'.
    %
:- func set_tree234.is_member(set_tree234(T), T) = bool.
:- pred set_tree234.is_member(set_tree234(T)::in, T::in, bool::out) is det.
```

```
    % 'set_tree234.contains(Set, X)' is true iff 'X' is a member of 'Set'.
    %
:- pred set_tree234.contains(set_tree234(T)::in, T::in) is semidet.

    % 'set_tree234.list_to_set(List) = Set' is true iff 'Set' is the set
    % containing only the members of 'List'.
    %
:- func set_tree234.list_to_set(list(T)) = set_tree234(T).
:- pred set_tree234.list_to_set(list(T)::in, set_tree234(T)::out) is det.

:- func set_tree234.from_list(list(T)) = set_tree234(T).
:- pred set_tree234.from_list(list(T)::in, set_tree234(T)::out) is det.

    % 'set_tree234.sorted_list_to_set(List) = Set' is true iff 'Set' is
    % the set containing only the members of 'List'. 'List' must be sorted.
    %
:- func set_tree234.sorted_list_to_set(list(T)) = set_tree234(T).
:- pred set_tree234.sorted_list_to_set(list(T)::in, set_tree234(T)::out) is det.

    % '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).

    % 'set_tree234.to_sorted_list(Set) = List' is true iff 'List' is the
    % list of all the members of 'Set', in sorted order.
    %
:- func set_tree234.to_sorted_list(set_tree234(T)) = list(T).
:- pred set_tree234.to_sorted_list(set_tree234(T)::in, list(T)::out) is det.

    % '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).

    % 'set_tree234.equal(SetA, SetB)' is true iff
    % 'SetA' and 'SetB' contain the same elements.
    %
:- pred set_tree234.equal(set_tree234(T)::in, set_tree234(T)::in) is semidet.

    % 'set_tree234.subset(SetA, SetB)' is true iff 'SetA' is a subset of
    % 'SetB'.
    %
:- pred set_tree234.subset(set_tree234(T)::in, set_tree234(T)::in) is semidet.

    % 'set_tree234.superset(SetA, SetB)' is true iff 'SetA' is a
    % superset of 'SetB'.
```

```
    %
:- pred set_tree234.superset(set_tree234(T)::in, set_tree234(T)::in)
    is semidet.

    % 'set_tree234.insert(X, Set0, Set)' is true iff 'Set' is the union
    % of 'Set0' and the set containing only 'X'.
    %
:- func set_tree234.insert(T, set_tree234(T)) = set_tree234(T).
:- pred set_tree234.insert(T::in, set_tree234(T)::in, set_tree234(T)::out)
    is det.

    % 'set_tree234.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 set_tree234.insert_new(T::in,
    set_tree234(T)::in, set_tree234(T)::out) is semidet.

    % 'set_tree234.insert_list(Xs, Set0, Set)' is true iff 'Set' is the
    % union of 'Set0' and the set containing only the members of 'Xs'.
    %
:- func set_tree234.insert_list(list(T), set_tree234(T)) = set_tree234(T).
:- pred set_tree234.insert_list(list(T)::in,
    set_tree234(T)::in, set_tree234(T)::out) is det.

    % 'set_tree234.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 set_tree234.delete(T, set_tree234(T)) = set_tree234(T).
:- pred set_tree234.delete(T::in, set_tree234(T)::in, set_tree234(T)::out)
    is det.

    % 'set_tree234.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 set_tree234.delete_list(list(T), set_tree234(T)) = set_tree234(T).
:- pred set_tree234.delete_list(list(T)::in,
    set_tree234(T)::in, set_tree234(T)::out) is det.

    % 'set_tree234.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 set_tree234.remove(T::in, set_tree234(T)::in, set_tree234(T)::out)
    is semidet.

    % 'set_tree234.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'.
    %
:- pred set_tree234.remove_list(list(T)::in,
    set_tree234(T)::in, set_tree234(T)::out) is semidet.

    % 'set_tree234.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 set_tree234.remove_least(T::out,
    set_tree234(T)::in, set_tree234(T)::out) is semidet.

    % 'set_tree234.union(SetA, SetB) = Set' is true iff 'Set' is the union
    % of 'SetA' and 'SetB'.
    %
:- func set_tree234.union(set_tree234(T), set_tree234(T)) = set_tree234(T).
:- pred set_tree234.union(set_tree234(T)::in, set_tree234(T)::in,
    set_tree234(T)::out) is det.

    % 'set_tree234.union_list(A, B)' is true iff 'B' is the union of
    % all the sets in 'A'
    %
:- func set_tree234.union_list(list(set_tree234(T))) = set_tree234(T).
:- pred set_tree234.union_list(list(set_tree234(T))::in, set_tree234(T)::out)
    is det.

    % 'set_tree234.power_union(A) = B' is true iff 'B' is the union of
    % all the sets in 'A'
    %
:- func set_tree234.power_union(set_tree234(set_tree234(T))) = set_tree234(T).
:- pred set_tree234.power_union(set_tree234(set_tree234(T))::in,
    set_tree234(T)::out) is det.

    % 'set_tree234.intersect(SetA, SetB) = Set' is true iff 'Set' is the
    % intersection of 'SetA' and 'SetB'.
    %
:- func set_tree234.intersect(set_tree234(T), set_tree234(T)) = set_tree234(T).
:- pred set_tree234.intersect(set_tree234(T)::in, set_tree234(T)::in,
    set_tree234(T)::out) is det.

    % 'set_tree234.power_intersect(A, B)' is true iff 'B' is the
```

```
    % intersection of all the sets in ‘A’.
    %
:- func set_tree234.power_intersect(set_tree234(set_tree234(T)))
    = set_tree234(T).
:- pred set_tree234.power_intersect(set_tree234(set_tree234(T))::in,
    set_tree234(T)::out) is det.

    % ‘set_tree234.intersect_list(A, B)’ is true iff ‘B’ is the
    % intersection of all the sets in ‘A’.
    %
:- func set_tree234.intersect_list(list(set_tree234(T))) = set_tree234(T).
:- pred set_tree234.intersect_list(list(set_tree234(T))::in,
    set_tree234(T)::out) is det.

    % ‘set_tree234.difference(SetA, SetB, Set)’ is true iff ‘Set’ is the
    % set containing all the elements of ‘SetA’ except those that
    % occur in ‘SetB’.
    %
:- func set_tree234.difference(set_tree234(T), set_tree234(T))
    = set_tree234(T).
:- pred set_tree234.difference(set_tree234(T)::in, set_tree234(T)::in,
    set_tree234(T)::out) is det.

    % ‘set_tree234.count(Set, Count)’ is true iff ‘Set’ has
    % ‘Count’ elements.
    %
:- func set_tree234.count(set_tree234(T)) = int.

:- func set_tree234.map(func(T1) = T2, set_tree234(T1)) = set_tree234(T2).
:- pred set_tree234.map(pred(T1, T2)::in(pred(in, out) is det),
    set_tree234(T1)::in, set_tree234(T2)::out) is det.

:- pred set_tree234.filter_map(pred(T1, T2)::in(pred(in, out) is semidet),
    set_tree234(T1)::in, set_tree234(T2)::out) is det.

:- func set_tree234.filter_map(func(T1) = T2, set_tree234(T1))
    = set_tree234(T2).
:- mode set_tree234.filter_map(func(in) = out is semidet, in) = out is det.

:- func set_tree234.fold(func(T1, T2) = T2, set_tree234(T1), T2) = T2.
:- pred set_tree234.fold(pred(T1, T2, T2), set_tree234(T1), T2, T2).
:- mode set_tree234.fold(pred(in, in, out) is det, in, in, out) is det.
:- mode set_tree234.fold(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode set_tree234.fold(pred(in, di, uo) is det, in, di, uo) is det.
:- mode set_tree234.fold(pred(in, in, out) is semidet, in, in, out)
    is semidet.
:- mode set_tree234.fold(pred(in, mdi, muo) is semidet, in, mdi, muo)
```

```
    is semidet.
:- mode set_tree234.fold(pred(in, di, uo) is semidet, in, di, uo)
    is semidet.

:- func set_tree234.foldl(func(T1, T2) = T2, set_tree234(T1), T2) = T2.
:- pred set_tree234.foldl(pred(T1, T2, T2), set_tree234(T1), T2, T2).
:- mode set_tree234.foldl(pred(in, in, out) is det, in, in, out) is det.
:- mode set_tree234.foldl(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode set_tree234.foldl(pred(in, di, uo) is det, in, di, uo) is det.
:- mode set_tree234.foldl(pred(in, in, out) is semidet, in, in, out)
    is semidet.
:- mode set_tree234.foldl(pred(in, mdi, muo) is semidet, in, mdi, muo)
    is semidet.
:- mode set_tree234.foldl(pred(in, di, uo) is semidet, in, di, uo)
    is semidet.

:- pred set_tree234.fold2(pred(T1, T2, T2, T3, T3), set_tree234(T1),
    T2, T2, T3, T3).
:- mode set_tree234.fold2(pred(in, in, out, in, out) is det, in,
    in, out, in, out) is det.
:- mode set_tree234.fold2(pred(in, in, out, mdi, muo) is det, in,
    in, out, mdi, muo) is det.
:- mode set_tree234.fold2(pred(in, in, out, di, uo) is det, in,
    in, out, di, uo) is det.
:- mode set_tree234.fold2(pred(in, in, out, in, out) is semidet, in,
    in, out, in, out) is semidet.
:- mode set_tree234.fold2(pred(in, in, out, mdi, muo) is semidet, in,
    in, out, mdi, muo) is semidet.
:- mode set_tree234.fold2(pred(in, in, out, di, uo) is semidet, in,
    in, out, di, uo) is semidet.

:- pred set_tree234.foldl2(pred(T1, T2, T2, T3, T3), set_tree234(T1),
    T2, T2, T3, T3).
:- mode set_tree234.foldl2(pred(in, in, out, in, out) is det, in,
    in, out, in, out) is det.
:- mode set_tree234.foldl2(pred(in, in, out, mdi, muo) is det, in,
    in, out, mdi, muo) is det.
:- mode set_tree234.foldl2(pred(in, in, out, di, uo) is det, in,
    in, out, di, uo) is det.
:- mode set_tree234.foldl2(pred(in, in, out, in, out) is semidet, in,
    in, out, in, out) is semidet.
:- mode set_tree234.foldl2(pred(in, in, out, mdi, muo) is semidet, in,
    in, out, mdi, muo) is semidet.
:- mode set_tree234.foldl2(pred(in, in, out, di, uo) is semidet, in,
    in, out, di, uo) is semidet.

:- pred set_tree234.fold3(
```

```
    pred(T1, T2, T2, T3, T3, T4, T4), set_tree234(T1),
    T2, T2, T3, T3, T4, T4).
:- mode set_tree234.fold3(pred(in, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out) is det.
:- mode set_tree234.fold3(pred(in, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, mdi, muo) is det.
:- mode set_tree234.fold3(pred(in, in, out, in, out, di, uo) is det, in,
    in, out, in, out, di, uo) is det.
:- mode set_tree234.fold3(pred(in, in, out, in, out, in, out) is semidet, in,
    in, out, in, out, in, out) is semidet.
:- mode set_tree234.fold3(pred(in, in, out, in, out, mdi, muo) is semidet, in,
    in, out, in, out, mdi, muo) is semidet.
:- mode set_tree234.fold3(pred(in, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, di, uo) is semidet.

:- pred set_tree234.foldl3(
    pred(T1, T2, T2, T3, T3, T4, T4), set_tree234(T1),
    T2, T2, T3, T3, T4, T4).
:- mode set_tree234.foldl3(pred(in, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out) is det.
:- mode set_tree234.foldl3(pred(in, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, mdi, muo) is det.
:- mode set_tree234.foldl3(pred(in, in, out, in, out, di, uo) is det, in,
    in, out, in, out, di, uo) is det.
:- mode set_tree234.foldl3(pred(in, in, out, in, out, in, out) is semidet, in,
    in, out, in, out, in, out) is semidet.
:- mode set_tree234.foldl3(pred(in, in, out, in, out, mdi, muo) is semidet, in,
    in, out, in, out, mdi, muo) is semidet.
:- mode set_tree234.foldl3(pred(in, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, di, uo) is semidet.

:- pred set_tree234.fold4(
    pred(T1, T2, T2, T3, T3, T4, T4, T5, T5), set_tree234(T1),
    T2, T2, T3, T3, T4, T4, T5, T5).
:- mode set_tree234.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 set_tree234.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 set_tree234.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 set_tree234.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 set_tree234.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 set_tree234.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 set_tree234.foldl4(
    pred(T1, T2, T2, T3, T3, T4, T4, T5, T5), set_tree234(T1),
    T2, T2, T3, T3, T4, T4, T5, T5).
:- mode set_tree234.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 set_tree234.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 set_tree234.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 set_tree234.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 set_tree234.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 set_tree234.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 set_tree234.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 set_tree234.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 set_tree234.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 set_tree234.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 set_tree234.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 set_tree234.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 set_tree234.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 set_tree234.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 set_tree234.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 set_tree234.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 set_tree234.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 set_tree234.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 set_tree234.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 set_tree234.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 set_tree234.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 set_tree234.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 set_tree234.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 set_tree234.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 set_tree234.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 set_tree234.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 set_tree234.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 set_tree234.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 set_tree234.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 set_tree234.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 set_tree234.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 set_tree234.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 set_tree234.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 set_tree234.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.

    % all_true(Pred, Set) succeeds iff Pred(Element) succeeds
    % for all the elements of Set.
    %
:- pred set_tree234.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 set_tree234.filter(pred(T)::in(pred(in) is semidet),
    set_tree234(T)::in) = (set_tree234(T)::out) is det.
:- pred set_tree234.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 set_tree234.filter(pred(T)::in(pred(in) is semidet),
    set_tree234(T)::in, set_tree234(T)::out, set_tree234(T)::out) is det.

    % set_tree234.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 set_tree234.divide(pred(T)::in(pred(in) is semidet),
    set_tree234(T)::in, set_tree234(T)::out, set_tree234(T)::out) is det.

    % set_tree234.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 set_tree234.divide_by_set(set_tree234(T)::in, set_tree234(T)::in,
    set_tree234(T)::out, set_tree234(T)::out) is det.

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

63 set_unordlist

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 1995-1997,1999-2002, 2004-2006, 2010-2012 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%---------------------------------------------------%
%
% 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).

    % ‘set_unordlist.init(Set)’ is true iff ‘Set’ is an empty set.
    %
:- func set_unordlist.init = set_unordlist(T).
:- pred set_unordlist.init(set_unordlist(_T)::uo) is det.

    % ‘set_unordlist.list_to_set(List, Set)’ is true iff ‘Set’ is the set
    % containing only the members of ‘List’.
    %
:- func set_unordlist.list_to_set(list(T)) = set_unordlist(T).
```

```
:- pred set_unordlist.list_to_set(list(T)::in, set_unordlist(T)::out) is det.

    % A synonym for set_unordlist.list_to_set/1.
    %
:- func set_unordlist.from_list(list(T)) = set_unordlist(T).

    % 'set_unordlist.sorted_list_to_set(List, Set)' is true iff 'Set' is
    % the set containing only the members of 'List'.  'List' must be sorted.
    %
:- pred set_unordlist.sorted_list_to_set(list(T)::in, set_unordlist(T)::out)
    is det.
:- func set_unordlist.sorted_list_to_set(list(T)) = set_unordlist(T).

    % A synonym for set_unordlist.sorted_list_to_set/1.
    %
:- func set_unordlist.from_sorted_list(list(T)) = set_unordlist(T).

    % 'set_unordlist.to_sorted_list(Set, List)' is true iff 'List' is the
    % list of all the members of 'Set', in sorted order.
    %
:- pred set_unordlist.to_sorted_list(set_unordlist(T)::in, list(T)::out)
    is det.
:- func set_unordlist.to_sorted_list(set_unordlist(T)) = list(T).

    % 'set_unordlist.singleton_set(Elem, Set)' is true iff 'Set' is the set
    % containing just the single element 'Elem'.
    %
:- pred set_unordlist.singleton_set(T, set_unordlist(T)).
:- mode set_unordlist.singleton_set(in, out) is det.
:- mode set_unordlist.singleton_set(in, in) is semidet.     % Implied.
:- mode set_unordlist.singleton_set(out, in) is semidet.

:- func set_unordlist.make_singleton_set(T) = set_unordlist(T).

:- pred set_unordlist.is_singleton(set_unordlist(T)::in, T::out) is semidet.

    % 'set_unordlist.equal(SetA, SetB)' is true iff
    % 'SetA' and 'SetB' contain the same elements.
    %
:- pred set_unordlist.equal(set_unordlist(T)::in, set_unordlist(T)::in)
    is semidet.

    % 'set_unordlist.empty(Set)' is true iff 'Set' is an empty set.
    %
:- pred set_unordlist.empty(set_unordlist(_T)::in) is semidet.

:- pred set_unordlist.non_empty(set_unordlist(_T)::in) is semidet.
```

```
:- pred set_unordlist.is_empty(set_unordlist(_T)::in) is semidet.

    % 'set_unordlist.subset(SetA, SetB)' is true iff 'SetA' is a subset of
    % 'SetB'.
    %
:- pred set_unordlist.subset(set_unordlist(T)::in, set_unordlist(T)::in)
    is semidet.

    % 'set_unordlist.superset(SetA, SetB)' is true iff 'SetA' is a
    % superset of 'SetB'.
    %
:- pred set_unordlist.superset(set_unordlist(T)::in, set_unordlist(T)::in)
    is semidet.

    % 'set_unordlist.member(X, Set)' is true iff 'X' is a member of 'Set'.
    %
:- pred set_unordlist.member(T, set_unordlist(T)).
:- mode set_unordlist.member(in, in) is semidet.
:- mode set_unordlist.member(out, in) is nondet.

    % 'set_unordlist.is_member(X, Set, Result)' returns
    % 'Result = yes' iff 'X' is a member of 'Set'.
    %
:- pred set_unordlist.is_member(T::in, set_unordlist(T)::in, bool::out)
    is det.

    % 'set_unordlist.contains(Set, X)' is true iff
    % 'X' is a member of 'Set'.
    %
:- pred set_unordlist.contains(set_unordlist(T)::in, T::in) is semidet.

    % 'set_unordlist.insert(X, Set0, Set)' is true iff 'Set' is the union
    % of 'Set0' and the set containing only 'X'.
    %
:- pred set_unordlist.insert(T, set_unordlist(T), set_unordlist(T)).
:- mode set_unordlist.insert(di, di, uo) is det.
:- mode set_unordlist.insert(in, in, out) is det.

:- func set_unordlist.insert(set_unordlist(T), T) = set_unordlist(T).

    % 'set_unordlist.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 set_unordlist.insert_new(T::in,
    set_unordlist(T)::in, set_unordlist(T)::out) is semidet.
```

```
    % 'set_unordlist.insert_list(Xs, Set0, Set)' is true iff 'Set' is the
    % union of 'Set0' and the set containing only the members of 'Xs'.
    %
:- pred set_unordlist.insert_list(list(T)::in,
    set_unordlist(T)::in, set_unordlist(T)::out) is det.

:- func set_unordlist.insert_list(set_unordlist(T), list(T))
    = set_unordlist(T).

    % 'set_unordlist.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'.
    %
:- pred set_unordlist.delete(T, set_unordlist(T), set_unordlist(T)).
:- mode set_unordlist.delete(in, di, uo) is det.
:- mode set_unordlist.delete(in, in, out) is det.

:- func set_unordlist.delete(set_unordlist(T), T) = set_unordlist(T).

    % 'set_unordlist.delete_list(Xs, Set0, Set)' is true iff 'Set' is the
    % relative complement of 'Set0' and the set containing only the members
    % of 'Xs'.
    %
:- pred set_unordlist.delete_list(list(T)::in,
    set_unordlist(T)::in, set_unordlist(T)::out) is det.

:- func set_unordlist.delete_list(set_unordlist(T), list(T))
    = set_unordlist(T).

    % 'set_unordlist.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 set_unordlist.remove(T::in,
    set_unordlist(T)::in, set_unordlist(T)::out) is semidet.

    % 'set_unordlist.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'.
    %
:- pred set_unordlist.remove_list(list(T)::in,
    set_unordlist(T)::in, set_unordlist(T)::out) is semidet.
```

```
    % 'set_unordlist.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 set_unordlist.remove_least(T::out,
    set_unordlist(T)::in, set_unordlist(T)::out) is semidet.

    % 'set_unordlist.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.
    %
:- pred set_unordlist.union(set_unordlist(T)::in, set_unordlist(T)::in,
    set_unordlist(T)::out) is det.

:- func set_unordlist.union(set_unordlist(T), set_unordlist(T))
    = set_unordlist(T).

    % 'set_unordlist.union_list(A) = B' is true iff 'B' is the union of
    % all the sets in 'A'
    %
:- func set_unordlist.union_list(list(set_unordlist(T))) = set_unordlist(T).

    % 'set_unordlist.power_union(A, B)' is true iff 'B' is the union of
    % all the sets in 'A'
    %
:- pred set_unordlist.power_union(set_unordlist(set_unordlist(T))::in,
    set_unordlist(T)::out) is det.

:- func set_unordlist.power_union(set_unordlist(set_unordlist(T)))
    = set_unordlist(T).

    % 'set_unordlist.intersect(SetA, SetB, Set)' is true iff 'Set' is the
    % intersection of 'SetA' and 'SetB'.
    %
:- pred set_unordlist.intersect(set_unordlist(T)::in, set_unordlist(T)::in,
    set_unordlist(T)::out) is det.

:- func set_unordlist.intersect(set_unordlist(T), set_unordlist(T))
    = set_unordlist(T).

    % 'set_unordlist.power_intersect(A, B)' is true iff 'B' is the
    % intersection of all the sets in 'A'
    %
:- pred set_unordlist.power_intersect(set_unordlist(set_unordlist(T))::in,
    set_unordlist(T)::out) is det.

:- func set_unordlist.power_intersect(set_unordlist(set_unordlist(T)))
```

```
    = set_unordlist(T).

    % 'set_unordlist.intersect_list(A, B)' is true iff 'B' is the
    % intersection of all the sets in 'A'
    %
:- func set_unordlist.intersect_list(list(set_unordlist(T)))
    = set_unordlist(T).

    % 'set_unordlist.difference(SetA, SetB, Set)' is true iff 'Set' is the
    % set containing all the elements of 'SetA' except those that
    % occur in 'SetB'
    %
:- pred set_unordlist.difference(set_unordlist(T)::in, set_unordlist(T)::in,
    set_unordlist(T)::out) is det.

:- func set_unordlist.difference(set_unordlist(T), set_unordlist(T))
    = set_unordlist(T).

:- func set_unordlist.count(set_unordlist(T)) = int.
:- pred set_unordlist.count(set_unordlist(T)::in, int::out) is det.

:- func set_unordlist.map(func(T1) = T2, set_unordlist(T1))
    = set_unordlist(T2).

:- func set_unordlist.filter_map(func(T1) = T2, set_unordlist(T1))
    = set_unordlist(T2).
:- mode set_unordlist.filter_map(func(in) = out is semidet, in) = out is det.

:- func set_unordlist.fold(func(T1, T2) = T2, set_unordlist(T1), T2) = T2.
:- pred set_unordlist.fold(pred(T1, T2, T2), set_unordlist(T1), T2, T2).
:- mode set_unordlist.fold(pred(in, in, out) is det, in, in, out) is det.
:- mode set_unordlist.fold(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode set_unordlist.fold(pred(in, di, uo) is det, in, di, uo) is det.
:- mode set_unordlist.fold(pred(in, in, out) is semidet, in, in, out)
    is semidet.
:- mode set_unordlist.fold(pred(in, mdi, muo) is semidet, in, mdi, muo)
    is semidet.
:- mode set_unordlist.fold(pred(in, di, uo) is semidet, in, di, uo)
    is semidet.

:- pred set_unordlist.fold2(pred(T1, T2, T2, T3, T3), set_unordlist(T1),
    T2, T2, T3, T3).
:- mode set_unordlist.fold2(pred(in, in, out, in, out) is det, in,
    in, out, in, out) is det.
:- mode set_unordlist.fold2(pred(in, in, out, mdi, muo) is det, in,
    in, out, mdi, muo) is det.
:- mode set_unordlist.fold2(pred(in, in, out, di, uo) is det, in,
```

```
    in, out, di, uo) is det.
:- mode set_unordlist.fold2(pred(in, in, out, in, out) is semidet, in,
    in, out, in, out) is semidet.
:- mode set_unordlist.fold2(pred(in, in, out, mdi, muo) is semidet, in,
    in, out, mdi, muo) is semidet.
:- mode set_unordlist.fold2(pred(in, in, out, di, uo) is semidet, in,
    in, out, di, uo) is semidet.

:- pred set_unordlist.fold3(pred(T1, T2, T2, T3, T3, T4, T4),
    set_unordlist(T1), T2, T2, T3, T3, T4, T4).
:- mode set_unordlist.fold3(pred(in, in, out, in, out, in, out) is det, in,
    in, out, in, out, in, out) is det.
:- mode set_unordlist.fold3(pred(in, in, out, in, out, mdi, muo) is det, in,
    in, out, in, out, mdi, muo) is det.
:- mode set_unordlist.fold3(pred(in, in, out, in, out, di, uo) is det, in,
    in, out, in, out, di, uo) is det.
:- mode set_unordlist.fold3(pred(in, in, out, in, out, in, out) is semidet, in,
    in, out, in, out, in, out) is semidet.
:- mode set_unordlist.fold3(pred(in, in, out, in, out, mdi, muo) is semidet, in,
    in, out, in, out, mdi, muo) is semidet.
:- mode set_unordlist.fold3(pred(in, in, out, in, out, di, uo) is semidet, in,
    in, out, in, out, di, uo) is semidet.

:- pred set_unordlist.fold4(pred(T1, T2, T2, T3, T3, T4, T4, T5, T5),
    set_unordlist(T1), T2, T2, T3, T3, T4, T4, T5, T5).
:- mode set_unordlist.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 set_unordlist.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 set_unordlist.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 set_unordlist.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 set_unordlist.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 set_unordlist.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 set_unordlist.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 set_unordlist.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 set_unordlist.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 set_unordlist.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 set_unordlist.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 set_unordlist.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 set_unordlist.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 set_unordlist.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 set_unordlist.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 set_unordlist.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 set_unordlist.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 set_unordlist.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 set_unordlist.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 set_unordlist.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.

    % all_true(Pred, Set) succeeds iff Pred(Element) succeeds
    % for all the elements of Set.
    %
:- pred set_unordlist.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 set_unordlist.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 set_unordlist.filter(pred(T)::in(pred(in) is semidet),
    set_unordlist(T)::in, set_unordlist(T)::out, set_unordlist(T)::out) is det.

    % set_unordlist.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 set_unordlist.divide(pred(T)::in(pred(in) is semidet),
    set_unordlist(T)::in, set_unordlist(T)::out, set_unordlist(T)::out) is det.

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

64 solutions

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%-----------------------------------------------------%
% Copyright (C) 1994-2007 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% 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, Accumulator, Acc0, Acc) <=>
    %   solutions(Generator, Solutions),
    %   list.foldl(Accumulator, 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, Accumulator, AccA0, AccA, AccB0, AccB) <=>
    %   solutions(Generator, Solutions),
    %   list.foldl2(Accumulator, 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, Accumulator, Acc0, Acc) <=>
    %   unsorted_solutions(Generator, Solutions),
    %   list.foldl(Accumulator, Solutions, Acc0, Acc).
    %
    % Operationally, however, unsorted_aggregate/4 will call the
    % Accumulator 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, Accumulator, !Acc1, !Acc2) <=>
    %   unsorted_solutions(Generator, Solutions),
    %   list.foldl2(Accumulator, Solutions, !Acc1, !Acc2).
    %
    % Operationally, however, unsorted_aggregate2/6 will call the
    % Accumulator 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)
    %       ;
    %           true
    %       ).
    %
    % 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.

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

65 sparse_bitset

```
%---------------------------------------------------%
% vim: ts=4 sw=4 et ft=mercury
%---------------------------------------------------%
% Copyright (C) 2000-2007, 2011-2012 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%---------------------------------------------------%
%
% 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 for 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).

    % Return an empty set.
```

```
    %
:- func init = sparse_bitset(T).
:- pred init(sparse_bitset(T)::out) is det.

:- pred empty(sparse_bitset(T)).
:- mode empty(in) is semidet.
:- mode empty(out) is det.

:- pred is_empty(sparse_bitset(T)::in) is semidet.

:- pred is_non_empty(sparse_bitset(T)::in) is semidet.

    % '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.

    % '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).

    % '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 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).

    % '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).
```

```
    % 'make_singleton_set(Elem)' returns a set containing just the single
    % element 'Elem'.
    %
:- func make_singleton_set(T) = sparse_bitset(T) <= enum(T).

    % 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).

    % 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).

    % '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.

    % '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).

    % '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.

    % '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).

    % ‘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.

    % 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).

    % 'count(Set)' returns the number of elements in 'Set'.
    % Takes O(card(Set)) time.
    %
:- func count(sparse_bitset(T)) = int <= enum(T).

    % '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, 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, 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), sparse_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, sparse_bitset(T), U) = U <= enum(T).

:- pred foldr(pred(T, U, U), sparse_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), sparse_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.

    % 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.

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

66 stack

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%---------------------------------------------------%
% Copyright (C) 1994-1995, 1997-1999, 2005-2006, 2011-2012 The University of Melbourne
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%---------------------------------------------------%
%
% 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).

% 'stack.init(Stack)' is true iff 'Stack' is an empty stack.
%
:- pred stack.init(stack(T)::out) is det.
:- func stack.init = stack(T).
```

```
% ‘stack.is_empty(Stack)’ is true iff ‘Stack’ is an empty stack.
%
:- pred stack.is_empty(stack(T)::in) is semidet.

% ‘stack.is_full(Stack)’ is intended to be true iff ‘Stack’
% is a stack whose capacity is exhausted.  This implementation
% allows arbitrary-sized stacks, so stack.is_full always fails.
%
:- pred stack.is_full(stack(T)::in) is semidet.

% ‘stack.push(Elem, Stack0, Stack)’ is true iff ‘Stack’ is
% the stack which results from pushing ‘Elem’ onto the top
% of ‘Stack0’.
%
:- pred stack.push(T::in, stack(T)::in, stack(T)::out) is det.
:- func stack.push(stack(T), T) = stack(T).

% ‘stack.push_list(Elems, Stack0, Stack)’ is true iff ‘Stack’
% is the stack which results from pushing the elements of the
% list ‘Elems’ onto the top of ‘Stack0’.
%
:- pred stack.push_list(list(T)::in, stack(T)::in, stack(T)::out) is det.
:- func stack.push_list(stack(T), list(T)) = stack(T).

% ‘stack.top(Stack, Elem)’ is true iff ‘Stack’ is a non-empty
% stack whose top element is ‘Elem’.
%
:- pred stack.top(stack(T)::in, T::out) is semidet.

% ‘stack.det_top’ is like ‘stack.top’ except that it will
% call error/1 rather than failing if given an empty stack.
%
:- pred stack.det_top(stack(T)::in, T::out) is det.
:- func stack.det_top(stack(T)) = T.

% ‘stack.pop(Elem, Stack0, Stack)’ is 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 stack.pop(T::out, stack(T)::in, stack(T)::out) is semidet.

% ‘stack.det_pop’ is like ‘stack.pop’ except that it will
% call error/1 rather than failing if given an empty stack.
%
:- pred stack.det_pop(T::out, stack(T)::in, stack(T)::out) is det.
```

```
% ‘stack.depth(Stack, Depth)’ is true iff ‘Stack’ is a stack
% containing ‘Depth’ elements.
%
:- pred stack.depth(stack(T)::in, int::out) is det.
:- func stack.depth(stack(T)) = int.

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

67 std_util

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%-----------------------------------------------------%
% Copyright (C) 1994-2006, 2008 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% 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.
    % XXX It would be nice to have infix ‘o’ or somesuch for this.
    %
:- 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.

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

    % 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.
    %
:- pred maybe_pred(pred(T1, T2), T1, maybe(T2)).
:- mode maybe_pred(pred(in, out) is semidet, in, out) is det.

:- func maybe_func(func(T1) = T2, T1) = maybe(T2).
:- mode maybe_func(func(in) = out is semidet, in) = out is det.

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

    % 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.

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

68 store

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%-----------------------------------------------------%
% Copyright (C) 1994-1997, 2000-2008, 2010-2011 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% 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 store.new (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.state).
:- instance store(store(S)).

    % Initialize a new store.
    %
:- some [S] pred store.init(store(S)::uo) is det.

%-----------------------------------------------%
%
% Mutvars
%

    % generic_mutvar(T, S):
    % A mutable variable holding a value of type T in store S.
    %
:- type generic_mutvar(T, S).
:- type io_mutvar(T) == generic_mutvar(T, io.state).
:- type store_mutvar(T, S) == generic_mutvar(T, store(S)).

    % Create a new mutable variable, initialized with the specified value.
    %
:- pred store.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 store.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 store.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 store.set_mutvar(generic_mutvar(T, S)::in, T::in,
    S::di, S::uo) is det <= store(S).

    % new_cyclic_mutvar(Func, Mutvar):
    %
```

```
    % 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 can't 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, mutvar(clist(T, S))).
    %
    %   :- pred init_cl(T::in, clist(T, S)::out,
    %       store(S)::di, store(S)::uo) is det.
    %
    %   init_cl(X, CList, !Store) :-
    %       store.new_cyclic_mutvar(func(CL) = node(X, CL), CList,
    %       !Store).
    %
:- pred store.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.
    %
:- type generic_ref(T, S).
:- type io_ref(T, S) == generic_ref(T, io.state).
:- 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 store.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 store.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 store.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 store.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 store.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 store.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 store.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 store.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 store.unsafe_arg_ref(generic_ref(T, S)::in, int::in,
    generic_ref(ArgT, S)::out, S::di, S::uo) is det <= store(S).
```

```
:- pred store.unsafe_new_arg_ref(T::di, int::in, generic_ref(ArgT, S)::out,
    S::di, S::uo) is det <= store(S).

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

69 stream

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 2006-2007, 2010 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%---------------------------------------------------%
%
% File: stream.m.
% Authors: juliensf, maclarty.
% Stability: low
%
% This module provides a family of typeclasses for defining streams
% in Mercury.  It also provides some generic predicates that operate
% on instances of these typeclasses.
%
%---------------------------------------------------%
%---------------------------------------------------%

:- module stream.
:- interface.

:- import_module bool.
:- import_module char.
:- import_module list.

:- include_module string_writer.

%---------------------------------------------------%
%
% Types used by streams.
%

:- type stream.name == string.

:- type stream.result(Error)
```

```
    --->    ok
    ;       eof
    ;       error(Error).

:- type stream.result(T, Error)
    --->    ok(T)
    ;       eof
    ;       error(Error).

:- type stream.res(Error)
    --->    ok
    ;       error(Error).

:- type stream.res(T, Error)
    --->    ok(T)
    ;       error(Error).

    % stream.maybe_partial_res is used when it is possible to return
    % a partial result when an error occurs.
    %
:- type stream.maybe_partial_res(T, Error)
    --->    ok(T)
    ;       error(T, Error).

%---------------------------------------------------%
%
% Stream errors.
%

:- typeclass stream.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(Stream, State) <= (Stream -> State) where
[
        % Returns a descriptive name for the stream.
        % Intended for use in error messages.
        %
        pred name(Stream::in, stream.name::out, State::di, State::uo) is det
].

%---------------------------------------------------%
%
% Input streams.
%

    % An input stream is a source of data.
    %
:- typeclass stream.input(Stream, State) <= stream(Stream, State) where [].

    % A reader stream is a subclass of 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 stream.reader(Stream, Unit, State, Error)
    <= (stream.input(Stream, State), stream.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 or
    % bulk_get/9 for that stream return `eof'.  If a call to get/4
    % returns `error(...)', all further calls to get/4 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 get(Stream::in, stream.result(Unit, Error)::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 stream.bulk_reader(Stream, Index, Store, State, Error)
    <= (stream.input(Stream, State), stream.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 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
    % 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, stream.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 stream.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 stream.writer(Stream, Unit, State)
    <= stream.output(Stream, State) where
[
    % Write the next unit to the given stream.
    % Blocks if the whole unit can't 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 stream.duplex(Stream, State)
    <= (stream.input(Stream, State), stream.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 stream.putback(Stream, Unit, State, Error)
    <= stream.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 stream.unbounded_putback(Stream, Unit, State, Error)
    <= stream.putback(Stream, Unit, State, Error) where [].

%---------------------------------------------------%
%
% Seekable streams.
%

    % stream.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 stream.whence
    --->    set
    ;       cur
    ;       end.

:- typeclass stream.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, stream.whence::in, int::in, State::di, State::uo)
        is det
].

%---------------------------------------------------%
%
% Line oriented streams.
%
```

```
    % A line oriented stream is a stream that keeps track of line numbers.
    %
:- typeclass stream.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 stream.input_stream_fold(Stream, pred(Unit, T, T), T,
    stream.maybe_partial_res(T, Error), State, State)
    <= stream.reader(Stream, Unit, State, Error).
:- mode stream.input_stream_fold(in, in(pred(in, in, out) is det),
    in, out, di, uo) is det.
:- mode stream.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 stream.input_stream_fold_state(Stream, pred(Unit, State, State),
    stream.res(Error), State, State)
    <= stream.reader(Stream, Unit, State, Error).
:- mode stream.input_stream_fold_state(in, in(pred(in, di, uo) is det),
    out, di, uo) is det.
:- mode stream.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 stream.input_stream_fold2_state(Stream,
    pred(Unit, T, T, State, State), T, stream.maybe_partial_res(T, Error),
```

```
    State, State) <= stream.reader(Stream, Unit, State, Error).
:- mode stream.input_stream_fold2_state(in,
    in(pred(in, in, out, di, uo) is det),
    in, out, di, uo) is det.
:- mode stream.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 stream.input_stream_fold2_state_maybe_stop(Stream,
    pred(Unit, bool, T, T, State, State),
    T, stream.maybe_partial_res(T, Error), State, State)
    <= stream.reader(Stream, Unit, State, Error).
:- mode stream.input_stream_fold2_state_maybe_stop(in,
    in(pred(in, out, in, out, di, uo) is det), in, out, di, uo) is det.
:- mode stream.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 from the specified stream.
    %
:- pred stream.ignore_whitespace(Stream::in, stream.result(Error)::out,
    State::di, State::uo)
    is det <= stream.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)
    <= stream.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.

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

70 stream.string_writer

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 2006-2007, 2011 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%---------------------------------------------------%
%
% 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_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(string.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 univ, then it
    % will print out the value stored in the univ, but not the type.
    %
    % 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(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.
```

```
:- pred print_cc(Stream::in, T::in, State::di, State::uo) is cc_multi
    <= (stream.writer(Stream, string, State),
    stream.writer(Stream, char, State)).

    % 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.  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(Stream, deconstruct.noncanon_handling, T, State, State) is det
    <= (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.

:- pred write_cc(Stream::in, T::in, State::di, State::uo) is cc_multi
    <= (stream.writer(Stream, string, State),
    stream.writer(Stream, char, State)).

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

71 string.builder

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 2006-2007 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% 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.

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

72 string

```
%-----------------------------------------------------%
% vim: ts=4 sw=4 et ft=mercury
%-----------------------------------------------------%
% Copyright (C) 1993-2012 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% File: string.m.
% Main authors: fjh, petdr.
% Stability: medium to high.
%
% This modules provides basic string handling facilities.
%
% 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.
%
% When Mercury is compiled to C, strings are UTF-8 encoded, using 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, strings are represented as Java 'String's.
% When Mercury is compiled to C# code, strings are represented as
% 'System.String's.  In both cases, strings are UTF-16 encoded.  A sin-
gle 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 builtin comparison operation on strings is also implementation dependent.
% In the current implementation, when Mercury is compiled to C, string
% comparison is implemented using C's strcmp() function.  When Mercury
% is compiled to Java, string comparison is implemented using Java's
% String.compareTo() method.  When Mercury is compiled to C#, string comparison
% is implemented using C#'s System.String.CompareOrdinal() method.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- 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.

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

    % This type is used for defining stream typeclass instances where the raw
    % string type would be ambiguous. 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).

    % This type is used for defining stream typeclass instances where the raw
    % string type would be ambiguous. A text file is a possibly empty sequence
    % of characters terminated by the end of file.
    %
:- type text_file
    --->    text_file(string).

    % 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 string.length(string::in) = (int::uo) is det.
:- pred string.length(string, int).
:- mode string.length(in, uo) is det.
:- mode string.length(ui, uo) is det.

    % Synonyms for string.length.
    %
:- func string.count_code_units(string) = int.
:- pred string.count_code_units(string::in, int::out) is det.

    % Determine the number of code points in a string.
    %
:- func string.count_codepoints(string) = int.
:- pred string.count_codepoints(string::in, int::out) is det.

    % Determine the number of code units required to represent a string
    % in UTF-8 encoding.
    %
:- func string.count_utf8_code_units(string) = int.

    % string.codepoint_offset(String, CodePointCount, CodePointOffset):
    % Equivalent to `string.codepoint_offset(String, 0, CodePointCount,
    % CodePointOffset)'.
    %
:- pred string.codepoint_offset(string::in, int::in, int::out) is semidet.

    % string.codepoint_offset(String, StartOffset, CodePointCount,
    %   CodePointOffset):
    %
    % Return the offset into `String' where, starting from `StartOffset',
    % `CodePointCount' code points are skipped.  Fails if either `StartOffset'
    % or `CodePointOffset' are out of range.
    %
:- pred string.codepoint_offset(string::in, int::in, int::in, int::out)
    is semidet.

    % Append two strings together.
    %
:- func string.append(string::in, string::in) = (string::uo) is det.

:- pred string.append(string, string, string).
:- mode string.append(in, in, in) is semidet.  % implied
:- mode string.append(in, uo, in) is semidet.
```

```
:- mode string.append(in, in, uo) is det.
:- mode string.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 string.remove_suffix instead.
% :- mode string.append(out, in, in) is semidet.

    % S1 ++ S2 = S :- string.append(S1, S2, S).
    %
    % Nicer syntax.
:- func string ++ string = string.
:- mode in ++ in = uo is det.

    % string.remove_suffix(String, Suffix, Prefix):
    % The same as string.append(Prefix, Suffix, String) except that
    % this is semidet whereas string.append(out, in, in) is nondet.
    %
:- pred string.remove_suffix(string::in, string::in, string::out) is semidet.

    % string.det_remove_suffix(String, Suffix) returns the same value
    % as string.remove_suffix, except it aborts if String does not end
    % with Suffix.
    %
:- func string.det_remove_suffix(string, string) = string.

    % string.remove_suffix_if_present(Suffix, String) returns `String' minus
    % `Suffix' if `String' ends with `Suffix', `String' otherwise.
    %
:- func string.remove_suffix_if_present(string, string) = string.

    % string.remove_prefix(Prefix, String, Suffix):
    % This is a synonym for string.append(Prefix, Suffix, String) but with
    % the arguments in a more convenient order for use with higher-order code.
    %
:- pred string.remove_prefix(string::in, string::in, string::out) is semidet.

    % string.remove_prefix_if_present(Prefix, String) = Suffix returns `String'
    % minus `Prefix' if `String' begins with `Prefix', `String' otherwise.
    %
:- func string.remove_prefix_if_present(string, string) = string.

    % string.prefix(String, Prefix) is true iff Prefix is a prefix of String.
    % Same as string.append(Prefix, _, String).
    %
:- pred string.prefix(string, string).
:- mode string.prefix(in, in) is semidet.
```

```
:- mode string.prefix(in, out) is multi.

    % string.suffix(String, Suffix) is true iff Suffix is a suffix of String.
    % Same as string.append(_, Suffix, String).
    %
:- pred string.suffix(string, string).
:- mode string.suffix(in, in) is semidet.
:- mode string.suffix(in, out) is multi.

    % string.string(X): Returns a canonicalized string representation
    % of the value X using the standard Mercury operators.
    %
:- func string.string(T) = string.

    % As above, but using the supplied table of operators.
    %
:- func string.string_ops(ops.table, T) = string.

    % 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.string_ops_noncanon(noncanon_handling, ops.table, T, string).
:- mode string.string_ops_noncanon(in(do_not_allow), in, in, out) is det.
:- mode string.string_ops_noncanon(in(canonicalize), in, in, out) is det.
:- mode string.string_ops_noncanon(in(include_details_cc), in, in, out)
    is cc_multi.
:- mode string.string_ops_noncanon(in, in, in, out) is cc_multi.

    % string.char_to_string(Char, String).
    % Converts a character (code point) to a string or vice versa.
    %
:- func string.char_to_string(char::in) = (string::uo) is det.
:- pred string.char_to_string(char, string).
:- mode string.char_to_string(in, uo) is det.
:- mode string.char_to_string(out, in) is semidet.

    % A synonym for string.char_to_string/1.
```

```
    %
:- func string.from_char(char::in) = (string::uo) is det.

    % Convert an integer to a string.
    %
:- func string.int_to_string(int::in) = (string::uo) is det.
:- pred string.int_to_string(int::in, string::uo) is det.

    % A synonym for string.int_to_string/1.
    %
:- func string.from_int(int::in) = (string::uo) is det.

    % Convert an integer to a string with commas as thousand separators.
    %
:- func string.int_to_string_thousands(int::in) = (string::uo) is det.

    % string.int_to_base_string(Int, Base, String):
    % Convert an integer to a string in a given Base.
    % An exception is thrown if Base is not between 2 and 36.
    %
:- func string.int_to_base_string(int::in, int::in) = (string::uo) is det.
:- pred string.int_to_base_string(int::in, int::in, string::uo) is det.

    % string.int_to_base_string_group(Int, Base, GroupLength, Separator,
    %   String):
    % Convert an integer to a string in a given Base (between 2 and 36)
    % and insert Separator between every GroupLength digits.
    % If GroupLength is less than one then no separators will appear in the
    % output.  An exception is thrown if Base is not between 2 and 36.
    % Useful for formatting numbers like "1,300,000".
    %
:- func string.int_to_base_string_group(int, int, int, string) = string.
:- mode string.int_to_base_string_group(in, in, in, in) = 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 to allow a successful
    % decimal -> binary conversion of the float.
    %
:- func string.float_to_string(float::in) = (string::uo) is det.
:- pred string.float_to_string(float::in, string::uo) is det.

    % A synonym for string.float_to_string/1.
    %
```

```
:- func string.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 string.c_pointer_to_string(c_pointer::in) = (string::uo) is det.
:- pred string.c_pointer_to_string(c_pointer::in, string::uo) is det.

    % A synonym for string.c_pointer_to_string/1.
    %
:- func string.from_c_pointer(c_pointer::in) = (string::uo) is det.

    % string.first_char(String, Char, Rest) is true iff Char is the first
    % character (code point) of String, and Rest is the remainder.
    %
    % WARNING: string.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. Repeated use of string.first_char
    % to iterate over a string will result in very poor performance.
    % Use string.foldl or string.to_char_list instead.
    %
:- pred string.first_char(string, char, string).
:- mode string.first_char(in, in, in) is semidet.  % implied
:- mode string.first_char(in, uo, in) is semidet.  % implied
:- mode string.first_char(in, in, uo) is semidet.  % implied
:- mode string.first_char(in, uo, uo) is semidet.
:- mode string.first_char(uo, in, in) is det.

    % string.replace(String0, Search, Replace, String):
    % string.replace replaces the first occurrence of Search in String0
    % with Replace to give String. It fails if Search does not occur
    % in String0.
    %
:- pred string.replace(string::in, string::in, string::in, string::uo)
    is semidet.

    % string.replace_all(String0, Search, Replace, String):
    % string.replace_all replaces any occurrences of Search in String0
    % with Replace to give String.
    %
:- func string.replace_all(string::in, string::in, string::in) = (string::uo)
    is det.
:- pred string.replace_all(string::in, string::in, string::in, string::uo)
    is det.

    % Converts a string to lowercase.
    % Note that this only converts unaccented Latin letters.
```

```
    %
:- func string.to_lower(string::in) = (string::uo) is det.
:- pred string.to_lower(string, string).
:- mode string.to_lower(in, uo) is det.
:- mode string.to_lower(in, in) is semidet.        % implied

    % Converts a string to uppercase.
    % Note that this only converts unaccented Latin letters.
    %
:- func string.to_upper(string::in) = (string::uo) is det.
:- pred string.to_upper(string, string).
:- mode string.to_upper(in, uo) is det.
:- mode string.to_upper(in, in) is semidet.        % implied

    % Convert the first character (if any) of a string to uppercase.
    % Note that this only converts unaccented Latin letters.
    %
:- func string.capitalize_first(string) = string.
:- pred string.capitalize_first(string::in, string::out) is det.

    % Convert the first character (if any) of a string to lowercase.
    % Note that this only converts unaccented Latin letters.
    %
:- func string.uncapitalize_first(string) = string.
:- pred string.uncapitalize_first(string::in, string::out) is det.

    % Convert the string to a list of characters (code points).
    % Throws an exception if the list of characters contains a null character.
    %
    % NOTE: in future the same treatment may be afforded surrogate code points.
    %
:- func string.to_char_list(string) = list(char).
:- pred string.to_char_list(string, list(char)).
:- mode string.to_char_list(in, out) is det.
:- mode string.to_char_list(uo, in) is det.

    % Convert a list of characters (code points) to a string.
    % Throws an exception if the list of characters contains a null character.
    %
    % NOTE: in future the same treatment may be afforded surrogate code points.
    %
:- func string.from_char_list(list(char)::in) = (string::uo) is det.
:- pred string.from_char_list(list(char), string).
:- mode string.from_char_list(in, uo) is det.
:- mode string.from_char_list(out, in) is det.

    % As above, but fail instead of throwing an exception if the
```

```
    % list contains a null character.
    %
    % NOTE: in future the same treatment may be afforded surrogate code points.
    %
:- pred string.semidet_from_char_list(list(char)::in, string::uo) is semidet.

    % Same as string.from_char_list, except that it reverses the order
    % of the characters.
    % Throws an exception if the list of characters contains a null character.
    %
    % NOTE: in future the same treatment may be afforded surrogate code points.
    %
:- func string.from_rev_char_list(list(char)::in) = (string::uo) is det.
:- pred string.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.
    %
    % NOTE: in future the same treatment may be afforded surrogate code points.
    %
:- pred string.semidet_from_rev_char_list(list(char)::in, string::uo)
    is semidet.

    % Convert a string into a list of code units.
    %
:- pred string.to_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.
    %
:- pred string.from_code_unit_list(list(int)::in, string::uo) is semidet.

    % Converts a signed base 10 string to an int; throws an exception
    % if the string argument does not match the regexp [+-]?[0-9]+
    % or the number is not in the range [int.min_int+1, int.max_int].
    %
:- func string.det_to_int(string) = int.

    % 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
    % [int.min_int+1, int.max_int], string.to_int fails.
    %
:- pred string.to_int(string::in, int::out) is semidet.
```

```
    % 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 base is 10 and the number is not in the range
    % [int.min_int, int.max_int], the predicate fails.
    %
:- pred string.base_string_to_int(int::in, string::in, int::out) is semidet.

    % Converts 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, if the base is 10, the number
    % is in the range [int.min_int, int.max_int].
    %
:- func string.det_base_string_to_int(int, string) = int.

    % Convert a string to a float. Throws an exception if the string is not
    % a syntactically correct float literal.
    %
:- func string.det_to_float(string) = float.

    % Convert a string to a float. If the string is not a syntactically correct
    % float literal, string.to_float fails.
    %
:- pred string.to_float(string::in, float::out) is semidet.

    % True if string contains only alphabetic characters [A-Za-z].
    %
:- pred string.is_all_alpha(string::in) is semidet.

    % True if string contains only alphabetic characters [A-Za-z] and
    % underscores.
    %
:- pred string.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 string.is_all_alnum_or_underscore(string::in) is semidet.

    % True if the string contains only decimal digits (0-9).
    %
:- pred string.is_all_digits(string::in) is semidet.

    % string.all_match(TestPred, String):
    %
    % True if TestPred is true when applied to each character (code point) in
```

```
    % String or if String is the empty string.
    %
:- pred string.all_match(pred(char)::in(pred(in) is semidet), string::in)
    is semidet.

    % string.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 string.pad_left(string, char, int) = string.
:- pred string.pad_left(string::in, char::in, int::in, string::out) is det.

    % string.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 string.pad_right(string, char, int) = string.
:- pred string.pad_right(string::in, char::in, int::in, string::out) is det.

    % string.duplicate_char(Char, Count, String):
    % Construct a string consisting of 'Count' occurrences of 'Char'
    % code points in sequence.
    %
:- func string.duplicate_char(char::in, int::in) = (string::uo) is det.
:- pred string.duplicate_char(char::in, int::in, string::uo) is det.

    % string.contains_char(String, Char):
    % Succeed if the code point 'Char' occurs in 'String'.
    %
:- pred string.contains_char(string::in, char::in) is semidet.

    % string.index(String, Index, Char):
    % 'Char' is the character (code point) in 'String', beginning at the
    % code unit 'Index'.  Fails if 'Index' is out of range (negative, or
    % greater than or equal to the length of 'String').
    %
    % Calls error/1 if an illegal sequence is detected.
    %
:- pred string.index(string::in, int::in, char::uo) is semidet.

    % string.det_index(String, Index, Char):
    % 'Char' is the character (code point) in 'String', beginning at the
    % code unit 'Index'.
    % Calls error/1 if 'Index' is out of range (negative, or greater than
    % or equal to the length of 'String'), or if an illegal sequence is
```

```
    % detected.
    %
:- func string.det_index(string, int) = char.
:- pred string.det_index(string::in, int::in, char::uo) is det.

    % A synonym for det_index/2:
    % String ^ elem(Index) = string.det_index(String, Index).
    %
:- func string ^ elem(int) = char.

    % string.unsafe_index(String, Index, Char):
    % 'Char' is the character (code point) in 'String', beginning at the
    % code unit 'Index'.
    % 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 string.det_index
    % may be linear in the length of the string. Use with care!
    %
:- func string.unsafe_index(string, int) = char.
:- pred string.unsafe_index(string::in, int::in, char::uo) is det.

    % A synonym for unsafe_index/2:
    % String ^ unsafe_elem(Index) = string.unsafe_index(String, Index).
    %
:- func string ^ unsafe_elem(int) = char.

    % string.index_next(String, Index, NextIndex, Char):
    % Like 'string.index'/3 but also returns the position of the code unit
    % that follows the code point beginning at 'Index',
    % i.e. NextIndex = Index + num_code_units_to_encode(Char).
    %
:- pred string.index_next(string::in, int::in, int::out, char::uo) is semidet.

    % string.unsafe_index_next(String, Index, NextIndex, Char):
    % 'Char' is the character (code point) in 'String', beginning at the
    % code unit 'Index'. 'NextIndex' is the offset following the encoding
    % of 'Char'. 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 string.unsafe_index_next(string::in, int::in, int::out, char::uo)
    is semidet.

    % string.prev_index(String, Index, CharIndex, Char):
    % 'Char' is the character (code point) in 'String' immediately _before_
    % the code unit 'Index'.  Fails if 'Index' is out of range (non-positive,
    % or greater than the length of 'String').
```

```
    %
:- pred string.prev_index(string::in, int::in, int::out, char::uo) is semidet.

    % string.unsafe_prev_index(String, Index, CharIndex, Char):
    % ‘Char’ is the character (code point) in ‘String’ immediately _before_
    % the code unit ‘Index’. ‘CharIndex’ is the offset of the beginning of
    % ‘Char’. Fails if ‘Index’ is zero.
    % WARNING: behavior is UNDEFINED if ‘Index’ is out of range
    % (negative, or greater than or equal to the length of ‘String’).
    %
:- pred string.unsafe_prev_index(string::in, int::in, int::out, char::uo)
    is semidet.

    % string.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 string.unsafe_index_code_unit(string::in, int::in, int::out) is det.

    % string.chomp(String):
    % ‘String’ minus any single trailing newline character.
    %
:- func string.chomp(string) = string.

    % string.lstrip(String):
    % ‘String’ minus any initial whitespace characters in the ASCII range.
    %
:- func string.lstrip(string) = string.

    % string.rstrip(String):
    % ‘String’ minus any trailing whitespace characters in the ASCII range.
    %
:- func string.rstrip(string) = string.

    % string.strip(String):
    % ‘String’ minus any initial and trailing whitespace characters in the
    % ASCII range.
    %
:- func string.strip(string) = string.

    % string.lstrip_pred(Pred, String):
    % ‘String’ minus the maximal prefix consisting entirely of characters
    % (code points) satisfying ‘Pred’.
    %
:- func string.lstrip_pred(pred(char)::in(pred(in) is semidet), string::in)
    = (string::out) is det.
```

```
    % string.rstrip_pred(Pred, String):
    % ‘String’ minus the maximal suffix consisting entirely of characters
    % (code points) satisfying ‘Pred’.
    %
:- func string.rstrip_pred(pred(char)::in(pred(in) is semidet), string::in)
    = (string::out) is det.

    % string.prefix_length(Pred, String):
    % The length (in code units) of the maximal prefix of ‘String’ consisting
    % entirely of characters (code points) satisfying Pred.
    %
:- func string.prefix_length(pred(char)::in(pred(in) is semidet), string::in)
    = (int::out) is det.

    % string.suffix_length(Pred, String):
    % The length (in code units) of the maximal suffix of ‘String’ consisting
    % entirely of characters (code points) satisfying Pred.
    %
:- func string.suffix_length(pred(char)::in(pred(in) is semidet), string::in)
    = (int::out) is det.

    % string.set_char(Char, Index, String0, String):
    % ‘String’ is ‘String0’, with the code point beginning at code unit
    % ‘Index’ removed and replaced by ‘Char’.
    % Fails if ‘Index’ is out of range (negative, or greater than or equal to
    % the length of ‘String0’).
    %
:- pred string.set_char(char, int, string, string).
:- mode string.set_char(in, in, in, out) is semidet.
% XXX This mode is disabled because the compiler puts constant
% strings into static data even when they might be updated.
%:- mode string.set_char(in, in, di, uo) is semidet.

    % string.det_set_char(Char, Index, String0, String):
    % ‘String’ is ‘String0’, with the code point beginning at code unit
    % ‘Index’ removed and replaced by ‘Char’.
    % Calls error/1 if ‘Index’ is out of range (negative, or greater than
    % or equal to the length of ‘String0’).
    %
:- func string.det_set_char(char, int, string) = string.
:- pred string.det_set_char(char, int, string, string).
:- mode string.det_set_char(in, in, in, out) is det.
% XXX This mode is disabled because the compiler puts constant
% strings into static data even when they might be updated.
%:- mode string.det_set_char(in, in, di, uo) is det.
```

```
    % string.unsafe_set_char(Char, Index, String0, String):
    % ‘String’ is ‘String0’, with the code point beginning at code unit
    % ‘Index’ removed and replaced by ‘Char’.
    % WARNING: behavior is UNDEFINED if ‘Index’ is out of range
    % (negative, or greater than or equal to the length of ‘String0’).
    % This version is constant time, whereas string.det_set_char
    % may be linear in the length of the string. Use with care!
    %
:- func string.unsafe_set_char(char, int, string) = string.
:- mode string.unsafe_set_char(in, in, in) = out is det.
% XXX This mode is disabled because the compiler puts constant
% strings into static data even when they might be updated.
%:- mode string.unsafe_set_char(in, in, di) = uo is det.
:- pred string.unsafe_set_char(char, int, string, string).
:- mode string.unsafe_set_char(in, in, in, out) is det.
% XXX This mode is disabled because the compiler puts constant
% strings into static data even when they might be updated.
%:- mode string.unsafe_set_char(in, in, di, uo) is det.

    % string.foldl(Closure, String, !Acc):
    % ‘Closure’ is an accumulator predicate which is to be called for each
    % character (code point) of the string ‘String’ in turn. The initial
    % value of the accumulator is ‘!.Acc’ and the final value is ‘!:Acc’.
    % (string.foldl is equivalent to
    %   string.to_char_list(String, Chars),
    %   list.foldl(Closure, Chars, !Acc)
    % but is implemented more efficiently.)
    %
:- func string.foldl(func(char, A) = A, string, A) = A.
:- pred string.foldl(pred(char, A, A), string, A, A).
:- mode string.foldl(pred(in, di, uo) is det, in, di, uo) is det.
:- mode string.foldl(pred(in, in, out) is det, in, in, out) is det.
:- mode string.foldl(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode string.foldl(pred(in, in, out) is nondet, in, in, out) is nondet.
:- mode string.foldl(pred(in, in, out) is multi, in, in, out) is multi.

    % string.foldl2(Closure, String, !Acc1, !Acc2):
    % A variant of string.foldl with two accumulators.
    %
:- pred string.foldl2(pred(char, A, A, B, B), string, A, A, B, B).
:- mode string.foldl2(pred(in, di, uo, di, uo) is det,
    in, di, uo, di, uo) is det.
:- mode string.foldl2(pred(in, in, out, di, uo) is det,
    in, in, out, di, uo) is det.
:- mode string.foldl2(pred(in, in, out, in, out) is det,
    in, in, out, in, out) is det.
:- mode string.foldl2(pred(in, in, out, in, out) is semidet,
```

```
    in, in, out, in, out) is semidet.
:- mode string.foldl2(pred(in, in, out, in, out) is nondet,
    in, in, out, in, out) is nondet.
:- mode string.foldl2(pred(in, in, out, in, out) is multi,
    in, in, out, in, out) is multi.

    % string.foldl_between(Closure, String, Start, End, !Acc)
    % is equivalent to string.foldl(Closure, SubString, !Acc)
    % where SubString = string.between(String, Start, End).
    %
    % ‘Start’ and ‘End’ are in terms of code units.
    %
:- func string.foldl_between(func(char, A) = A, string, int, int, A) = A.
:- pred string.foldl_between(pred(char, A, A), string, int, int, A, A).
:- mode string.foldl_between(pred(in, in, out) is det, in, in, in,
    in, out) is det.
:- mode string.foldl_between(pred(in, di, uo) is det, in, in, in,
    di, uo) is det.
:- mode string.foldl_between(pred(in, in, out) is semidet, in, in, in,
    in, out) is semidet.
:- mode string.foldl_between(pred(in, in, out) is nondet, in, in, in,
    in, out) is nondet.
:- mode string.foldl_between(pred(in, in, out) is multi, in, in, in,
    in, out) is multi.

    % string.foldl2_between(Closure, String, Start, End, !Acc1, !Acc2)
    % A variant of string.foldl_between with two accumulators.
    %
    % ‘Start’ and ‘End’ are in terms of code units.
    %
:- pred string.foldl2_between(pred(char, A, A, B, B),
    string, int, int, A, A, B, B).
:- mode string.foldl2_between(pred(in, di, uo, di, uo) is det,
    in, in, in, di, uo, di, uo) is det.
:- mode string.foldl2_between(pred(in, in, out, di, uo) is det,
    in, in, in, in, out, di, uo) is det.
:- mode string.foldl2_between(pred(in, in, out, in, out) is det,
    in, in, in, in, out, in, out) is det.
:- mode string.foldl2_between(pred(in, in, out, in, out) is semidet,
    in, in, in, in, out, in, out) is semidet.
:- mode string.foldl2_between(pred(in, in, out, in, out) is nondet,
    in, in, in, in, out, in, out) is nondet.
:- mode string.foldl2_between(pred(in, in, out, in, out) is multi,
    in, in, in, in, out, in, out) is multi.

    % string.foldr(Closure, String, !Acc):
    % As string.foldl/4, except that processing proceeds right-to-left.
```

```
    %
:- func string.foldr(func(char, T) = T, string, T) = T.
:- pred string.foldr(pred(char, T, T), string, T, T).
:- mode string.foldr(pred(in, in, out) is det, in, in, out) is det.
:- mode string.foldr(pred(in, di, uo) is det, in, di, uo) is det.
:- mode string.foldr(pred(in, in, out) is semidet, in, in, out) is semidet.
:- mode string.foldr(pred(in, in, out) is nondet, in, in, out) is nondet.
:- mode string.foldr(pred(in, in, out) is multi, in, in, out) is multi.

    % string.foldr_between(Closure, String, Start, End, !Acc)
    % is equivalent to string.foldr(Closure, SubString, !Acc)
    % where SubString = string.between(String, Start, End).
    %
    % 'Start' and 'End' are in terms of code units.
    %
:- func string.foldr_between(func(char, T) = T, string, int, int, T) = T.
:- pred string.foldr_between(pred(char, T, T), string, int, int, T, T).
:- mode string.foldr_between(pred(in, in, out) is det, in, in, in,
    in, out) is det.
:- mode string.foldr_between(pred(in, di, uo) is det, in, in, in,
    di, uo) is det.
:- mode string.foldr_between(pred(in, in, out) is semidet, in, in, in,
    in, out) is semidet.
:- mode string.foldr_between(pred(in, in, out) is nondet, in, in, in,
    in, out) is nondet.
:- mode string.foldr_between(pred(in, in, out) is multi, in, in, in,
    in, out) is multi.

    % string.foldl_substring(Closure, String, Start, Count, !Acc)
    % Please use string.foldl_between instead.
    %
:- pragma obsolete(string.foldl_substring/5).
:- pragma obsolete(string.foldl_substring/6).
:- func string.foldl_substring(func(char, A) = A, string, int, int, A) = A.
:- pred string.foldl_substring(pred(char, A, A), string, int, int, A, A).
:- mode string.foldl_substring(pred(in, in, out) is det, in, in, in,
    in, out) is det.
:- mode string.foldl_substring(pred(in, di, uo) is det, in, in, in,
    di, uo) is det.
:- mode string.foldl_substring(pred(in, in, out) is semidet, in, in, in,
    in, out) is semidet.
:- mode string.foldl_substring(pred(in, in, out) is nondet, in, in, in,
    in, out) is nondet.
:- mode string.foldl_substring(pred(in, in, out) is multi, in, in, in,
    in, out) is multi.

    % string.foldl2_substring(Closure, String, Start, Count, !Acc1, !Acc2)
```

```
    % Please use string.foldl2_between instead.
    %
:- pragma obsolete(string.foldl2_substring/8).
:- pred string.foldl2_substring(pred(char, A, A, B, B),
    string, int, int, A, A, B, B).
:- mode string.foldl2_substring(pred(in, di, uo, di, uo) is det,
    in, in, in, di, uo, di, uo) is det.
:- mode string.foldl2_substring(pred(in, in, out, di, uo) is det,
    in, in, in, in, out, di, uo) is det.
:- mode string.foldl2_substring(pred(in, in, out, in, out) is det,
    in, in, in, in, out, in, out) is det.
:- mode string.foldl2_substring(pred(in, in, out, in, out) is semidet,
    in, in, in, in, out, in, out) is semidet.
:- mode string.foldl2_substring(pred(in, in, out, in, out) is nondet,
    in, in, in, in, out, in, out) is nondet.
:- mode string.foldl2_substring(pred(in, in, out, in, out) is multi,
    in, in, in, in, out, in, out) is multi.

    % string.foldr_substring(Closure, String, Start, Count, !Acc)
    % Please use string.foldr_between instead.
    %
:- pragma obsolete(string.foldr_substring/5).
:- pragma obsolete(string.foldr_substring/6).
:- func string.foldr_substring(func(char, T) = T, string, int, int, T) = T.
:- pred string.foldr_substring(pred(char, T, T), string, int, int, T, T).
:- mode string.foldr_substring(pred(in, in, out) is det, in, in, in,
    in, out) is det.
:- mode string.foldr_substring(pred(in, di, uo) is det, in, in, in,
    di, uo) is det.
:- mode string.foldr_substring(pred(in, in, out) is semidet, in, in, in,
    in, out) is semidet.
:- mode string.foldr_substring(pred(in, in, out) is nondet, in, in, in,
    in, out) is nondet.
:- mode string.foldr_substring(pred(in, in, out) is multi, in, in, in,
    in, out) is multi.

    % string.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 characters (code points) matched by SepP.
    % For example,
    %
    % string.words_separator(char.is_whitespace, " the cat  sat on the  mat") =
    %   ["the", "cat", "sat", "on", "the", "mat"]
    %
    % Note the difference to string.split_at_separator.
    %
:- func string.words_separator(pred(char), string) = list(string).
```

```
:- mode string.words_separator(pred(in) is semidet, in) = out is det.

    % string.words(String) =
    %   string.words_separator(char.is_whitespace, String).
    %
:- func string.words(string) = list(string).

    % string.split_at_separator(SepP, String) returns the list of
    % substrings of String (in first to last order) that are delimited
    % by characters (code points) matched by SepP. For example,
    %
    % string.split_at_separator(char.is_whitespace, " a cat  sat on the  mat")
    %   = ["", "a", "cat", "", "sat", "on", "the", "", "mat"]
    %
    % Note the difference to string.words_separator.
    %
:- func string.split_at_separator(pred(char), string) = list(string).
:- mode string.split_at_separator(pred(in) is semidet, in) = out is det.

    % string.split_at_char(Char, String) =
    %     string.split_at_separator(unify(Char), String)
    %
:- func string.split_at_char(char, string) = list(string).

    % string.split_at_string(Separator, String) returns the list of substrings
    % of String that are delimited by Separator. For example,
    %
    % string.split_at_string("|||", "|||fld2|||fld3") = ["", "fld2", [fld3"]
    %
    % Always the first match of Separator is used to break the String, for
    % example: string.split_at_string("aa", "xaaayaaaz") = ["x", "ay", "az"]
    %
:- func string.split_at_string(string, string) = list(string).

    % string.split(String, Index, LeftSubstring, RightSubstring):
    % Split a string into two substrings, at the code unit 'Index'.
    % (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 string.split(string::in, int::in, string::out, string::out) is det.

    % string.split_by_codepoint(String, Count, LeftSubstring, RightSubstring):
    % 'LeftSubstring' is the left-most 'Count' characters (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 string.split_by_codepoint(string::in, int::in, string::out, string::out)
    is det.

    % string.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 string.left(string::in, int::in) = (string::out) is det.
:- pred string.left(string::in, int::in, string::out) is det.

    % string.left_by_codepoint(String, Count, LeftSubstring):
    % 'LeftSubstring' is the left-most 'Count' characters (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 string.left_by_codepoint(string::in, int::in) = (string::out) is det.
:- pred string.left_by_codepoint(string::in, int::in, string::out) is det.

    % string.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 string.right(string::in, int::in) = (string::out) is det.
:- pred string.right(string::in, int::in, string::out) is det.

    % string.right_by_codepoint(String, Count, RightSubstring):
    % 'RightSubstring' is the right-most 'Count' characters (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 string.right_by_codepoint(string::in, int::in) = (string::out) is det.
:- pred string.right_by_codepoint(string::in, int::in, string::out) is det.

    % string.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 string.between(string::in, int::in, int::in) = (string::uo) is det.
:- pred string.between(string::in, int::in, int::in, string::uo) is det.
```

```
    % string.substring(String, Start, Count, Substring):
    % Please use string.between instead.
    %
:- pragma obsolete(string.substring/3).
:- pragma obsolete(string.substring/4).
:- func string.substring(string::in, int::in, int::in) = (string::uo) is det.
:- pred string.substring(string::in, int::in, int::in, string::uo) is det.

    % string.between_codepoints(String, Start, End, Substring):
    % ‘Substring’ is the part of ‘String’ between the code point positions
    % ‘Start’ and ‘End’.
    % (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 string.between_codepoints(string::in, int::in, int::in)
    = (string::uo) is det.
:- pred string.between_codepoints(string::in, int::in, int::in, string::uo)
    is det.

    % string.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 string.substring may take time proportional to the length
    % of the whole string.
    %
:- func string.unsafe_between(string::in, int::in, int::in) = (string::uo)
    is det.
:- pred string.unsafe_between(string::in, int::in, int::in, string::uo)
    is det.

    % string.unsafe_substring(String, Start, Count, Substring):
    % Please use string.unsafe_between instead.
    %
:- pragma obsolete(string.unsafe_substring/3).
:- pragma obsolete(string.unsafe_substring/4).
:- func string.unsafe_substring(string::in, int::in, int::in) = (string::uo)
    is det.
:- pred string.unsafe_substring(string::in, int::in, int::in, string::uo)
    is det.
```

```
    % Append a list of strings together.
    %
:- func string.append_list(list(string)::in) = (string::uo) is det.
:- pred string.append_list(list(string)::in, string::uo) is det.

    % string.join_list(Separator, Strings) = JoinedString:
    % Appends together the strings in Strings, putting Separator between
    % adjacent strings. If Strings is the empty list, returns the empty string.
    %
:- func string.join_list(string::in, list(string)::in) = (string::uo) is det.

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

    % Two other hash functions for strings.
    %
:- func string.hash2(string) = int.
:- func string.hash3(string) = int.

    % string.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 string.sub_string_search(string::in, string::in, int::out) is semidet.

    % string.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.
    %
:- pred string.sub_string_search_start(string::in, string::in, int::in,
    int::out) is semidet.

    % A function similar to sprintf() in C.
    %
    % For example,
    %   string.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. string.format will not return the length of the string.
    %
    % conv  var     output form.        effect of '#'.
    % char. type.
    %
    % d     int     signed integer
    % i     int     signed integer
    % o     int     signed octal        with '0' prefix
    % x,X   int     signed hex          with '0x', '0X' prefix
    % u     int     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     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'.
    % (With 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 string.format(string, list(string.poly_type)) = string.
:- pred string.format(string::in, list(string.poly_type)::in, string::out)
    is det.

:- type string.poly_type
    --->    f(float)
    ;       i(int)
    ;       s(string)
```

```
    ;       c(char).

    % format_table(Columns, Separator) = Table
    % format_table/2 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. A newline character is inserted
    % between each row. If the columns are not all the same length then
    % an exception is thrown. 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 string.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 an maximum width with each column.
    %
:- func string.format_table_max(assoc_list(justified_column, maybe(int)),
    string) = string.

:- type justified_column
    --->    left(list(string))
    ;       right(list(string)).

    % 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 a line and each
    % line contains as many whole words as possible. If any one word ex-
ceeds N
    % code point in length then it will be broken over two (or more) lines.
    % Sequences of whitespace characters are replaced by a single space.
    %
:- func string.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 string.word_wrap_separator(string, int, string) = string.

    % Convert a string to a pretty_printer.doc for formatting.
    %
:- func string.string_to_doc(string) = pretty_printer.doc.

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

73 table_statistics

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%-----------------------------------------------------%
% Copyright (C) 2007 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% 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.

:- 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.

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

74 term

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%-----------------------------------------------------%
% Copyright (C) 1993-2000, 2003-2009, 2011-2012 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% File: term.m.
% Main author: fjh.
% Stability: medium.
%
% This file provides a type ‘term’ used to represent Prolog terms,
% and various predicates to manipulate terms and substitutions.
% Terms are polymorphic so that terms representing different kinds of
% thing can be made to be of different types so they don’t get mixed up.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module term.
:- interface.

:- import_module enum.
:- import_module list.
:- import_module map.
:- import_module type_desc.
:- import_module univ.

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

```
:- type term(T)
    --->    functor(
                const,
                list(term(T)),
                term.context
            )
    ;       variable(
                var(T),
                term.context
            ).

:- type const
    --->    atom(string)
    ;       integer(int)
    ;       string(string)
    ;       float(float)
    ;       implementation_defined(string).

:- type term.context
    --->    context(string, int).
            % file name, line number.

:- type var(T).
:- type var_supply(T).

:- type generic
    --->    generic.

:- type term    ==  term(generic).
:- type var     ==  var(generic).

:- func get_term_context(term(T)) = term.context.

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

    % The following predicates can convert values of (almost) any type
    % to the type 'term' and back again.

:- 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).

    % term.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.

:- 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
            ).

    % 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.

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

    % 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.

:- type substitution(T) == map(var(T), term(T)).
:- type substitution    == substitution(generic).

    % unify_term(Term1, Term2, Bindings0, Bindings):
    %
    % Unify (with occur check) two terms with respect to a set of bindings
    % and possibly update the set of bindings.
    %
:- pred unify_term(term(T)::in, term(T)::in, substitution(T)::in,
    substitution(T)::out) is semidet.

    % As above, but unify the corresponding elements of two lists of terms.
    % Fails 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(Term1, Term2, BoundVars, !Bindings):
    %
    % Unify (with occur check) two terms with respect to a set of bindings
    % and possibly update the set of bindings. Fails if any of the variables
    % in BoundVars 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.

    % As above, but unify the corresponding elements of two lists of terms.
    % Fails if the lists are not of equal length.
    %
:- 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.

    % list_subsumes(Terms1, Terms2, Subst) succeeds iff the list
    % Terms1 subsumes (is more general than) Terms2, producing a substitution
    % which when applied to Terms1 will give Terms2.
    %
:- pred list_subsumes(list(term(T))::in, list(term(T))::in,
    substitution(T)::out) is semidet.

    % substitute(Term0, Var, Replacement, Term):
    %
    % Replace all occurrences of Var in Term0 with Replacement,
    % and return the result in 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.

    % As above, except for a list of terms rather than a single
    %
:- 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.

    % substitute_corresponding(Vars, Repls, Term0, Term):
    %
    % Replace all occurrences of variables in Vars with the corresponding
    % term in Repls, and return the result in Term. If Vars contains
    % duplicates, or if Vars is not the same length as Repls, 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.

    % As above, except applies to a list of terms rather than a single term.
    %
:- 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.

    % apply_rec_substitution(Term0, Substitution, Term):
    %
    % Recursively apply substitution to Term0 until no more substitutions
    % can be applied, and then return the result in 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.

    % As above, except applies to a list of terms rather than a single term.
    %
:- 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.

    % apply_substitution(Term0, Substitution, Term):
    %
```

```
    % Apply substitution to Term0 and return the result in Term.
    %
:- func apply_substitution(term(T), substitution(T)) = term(T).
:- pred apply_substitution(term(T)::in, substitution(T)::in,
    term(T)::out) is det.

    % As above, except applies to a list of terms rather than a single 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.

    % occurs(Term0, Var, Substitution):
    % True iff Var occurs in the term resulting after applying Substitution
    % to Term0. Var variable 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.

    % relabel_variable(Term0, OldVar, NewVar, Term):
    %
    % Replace all occurrences of OldVar in Term0 with NewVar and put the result
    % in 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.

    % As above, except applies to a list of terms rather than a single term.
    % XXX the name of the predicate is misleading.
    %
:- 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.

    % 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), map(var(T), var(T))) = term(T).
:- pred apply_variable_renaming(term(T)::in, map(var(T), var(T))::in,
    term(T)::out) is det.
```

```
    % Applies apply_variable_renaming to a list of terms.
    %
:- func apply_variable_renaming_to_list(list(term(T)),
    map(var(T), var(T))) = list(term(T)).
:- pred apply_variable_renaming_to_list(list(term(T))::in,
    map(var(T), var(T))::in, list(term(T))::out) is det.

    % Applies apply_variable_renaming to a var.
    %
:- func apply_variable_renaming_to_var(map(var(T), var(T)),
    var(T)) = var(T).
:- pred apply_variable_renaming_to_var(map(var(T), var(T))::in,
    var(T)::in, var(T)::out) is det.

    % Applies apply_variable_renaming to a list of vars.
    %
:- func apply_variable_renaming_to_vars(map(var(T), var(T)),
    list(var(T))) = list(var(T)).
:- pred apply_variable_renaming_to_vars(map(var(T), var(T))::in,
    list(var(T))::in, list(var(T))::out) is det.

    % is_ground_in_bindings(Term, Bindings) is true iff no variables contained
    % in Term are non-ground in Bindings.
    %
:- pred is_ground_in_bindings(term(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.

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

    % To manage a supply of variables, use the following 2 predicates.
    % (We might want to give these a unique mode later.)

    % 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(VarSupply0, Variable, 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.

    % var_id(Variable):
    % Returns a unique number associated with this variable w.r.t.
    % its originating var_supply.
    %
:- func var_id(var(T)) = int.

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

    % from_int/1 should only be applied to integers returned by to_int/1.
    % This instance declaration is needed to allow sets of variables to be
    % represented using sparse_bitset.m.
:- instance enum(var(_)).

    % Convert a variable to an int. Different variables map to different ints.
    % Other than that, the mapping is unspecified.
    %
:- func var_to_int(var(T)) = int.
:- pred var_to_int(var(T)::in, int::out) is det.

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

    % 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.

    % Used to initialize the term context when reading in
    % (or otherwise constructing) a term.
    %
:- func context_init = context.
:- pred context_init(context::out) is det.
:- func context_init(string, int) = context.
:- pred context_init(string::in, int::in, context::out) is det.

    % Convert a list of terms which are all vars into a list of vars.
    % Abort (call error/1) 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.

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

75 term_io

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 1994-2006, 2009, 2011-2012 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
```

```
%
% 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).

    % term_io.read_term(Result, !IO):
    %
    % Read a term from standard input. 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 term_io.read_term(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 term_io.read_term_with_op_table(Ops::in, read_term(T)::out,
    io::di, io::uo) is det <= op_table(Ops).

    % Writes a term to standard output. Uses the variable names specified
```

```
    % by the varset. Writes _N for all unnamed variables, with N start-
ing at 0.
    %
:- pred term_io.write_term(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 term_io.write_term_with_op_table(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 term_io.write_term_nl(varset(T)::in, term(T)::in, io::di, io::uo)
    is det.

    % As above, except it appends a period and new-line.
    %
:- pred term_io.write_term_nl_with_op_table(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 stdout.
    %
:- pred term_io.write_constant(const::in, io::di, io::uo) is det.

    % Like term_io.write_constant, but return the result in a string.
    %
:- func term_io.format_constant(const) = string.

    % Writes a variable to stdout.
    %
:- pred term_io.write_variable(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 term_io.write_variable_with_op_table(Ops::in, var(T)::in,
    varset(T)::in, io::di, io::uo) is det <= op_table(Ops).

    % Given a string S, write S in double-quotes, with characters
    % escaped if necessary, to stdout.
    %
:- pred term_io.quote_string(string::in, io::di, io::uo) is det.

:- pred term_io.quote_string(Stream::in, string::in,
    State::di, State::uo) is det
```

```
    <= (stream.writer(Stream, string, State),
    stream.writer(Stream, char, State)).

    % Like term_io.quote_string, but return the result in a string.
    %
:- func term_io.quoted_string(string) = string.

    % Given an atom-name A, write A, enclosed in single-quotes if necessary,
    % with characters escaped if necessary, to stdout.
    %
:- pred term_io.quote_atom(string::in, io::di, io::uo) is det.

:- pred term_io.quote_atom(Stream::in, string::in,
    State::di, State::uo) is det
    <= (stream.writer(Stream, string, State),
    stream.writer(Stream, char, State)).

    % Like term_io.quote_atom, but return the result in a string.
    %
:- func term_io.quoted_atom(string) = string.

    % Given a character C, write C in single-quotes,
    % escaped if necessary, to stdout.
    %
:- pred term_io.quote_char(char::in, io::di, io::uo) is det.

:- pred term_io.quote_char(Stream::in, char::in,
    State::di, State::uo) is det
    <= (stream.writer(Stream, string, State),
    stream.writer(Stream, char, State)).

    % Like term_io.quote_char, but return the result in a string.
    %
:- func term_io.quoted_char(char) = string.

    % Given a character C, write C, escaped if necessary, to stdout.
    % The character is not enclosed in quotes.
    %
:- pred term_io.write_escaped_char(char::in, io::di, io::uo) is det.

:- pred term_io.write_escaped_char(Stream::in, char::in,
    State::di, State::uo) is det
    <= (stream.writer(Stream, string, State),
    stream.writer(Stream, char, State)).

    % Like term_io.write_escaped_char, but return the result in a string.
    %
```

```
:- func term_io.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, with characters escaped if necessary,
    % to stdout. The string is not enclosed in quotes.
    %
:- pred term_io.write_escaped_string(string::in, io::di, io::uo) is det.

:- pred term_io.write_escaped_string(Stream::in, string::in,
    State::di, State::uo) is det
    <= (stream.writer(Stream, string, State),
    stream.writer(Stream, char, State)).

    % Like term_io.write_escaped_char, but return the result in a string.
    %
:- func term_io.escaped_string(string) = string.

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

76 term_to_xml

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 1993-2007, 2010-2011 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%---------------------------------------------------%
%
% 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 1 allows values of a specific type to be mapped to arbitrary XML
% elements with arbitrary children and arbitrary attributes.
% In 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.
% In method 2 a DTD can be automatically generated. In method 1 DTDs cannot
% be automatically generated.
%
% 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 arbitrary 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 an XML document or a portion of
    % an XML document.
    %
:- 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 the substring "]]>".
            % If it does then invalid XML will be generated.

    ;       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
```

```
    --->    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
    --->    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_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)).

    % 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).

%---------------------------------------------------%
%
% 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', `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 `Int', `String', `Float' and `Char'
    %   will have their `functor', `arity', `type' and `field' (if
    %   there is a field name) attributes set. `Int', `String',
    %   `Float' and `Char' elements 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
    --->    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 then 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 also not supported.
    %
    % 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 duplicate element
                    % and a list of types whose functors map to that
                    % element is given.
                    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 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).

    % 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).

    % 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.

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

77 thread.channel

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 2000-2001, 2006-2007 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% 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 channel.init(channel(T)::out, io::di, io::uo) is det.

    % Put an item at the end of the channel.
    %
:- pred channel.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 channel.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 channel.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 channel.duplicate call.
    %
:- pred channel.duplicate(channel(T)::in, channel(T)::out, io::di, io::uo)
    is det.

    % Place an item back at the start of the channel.
    %
:- pred channel.untake(channel(T)::in, T::in, io::di, io::uo) is det.
```

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

78 thread

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 2000-2001,2003-2004, 2006-2008, 2010-2011 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% File: thread.m.
% Main author: conway.
% Stability: medium.
%
% This module defines the Mercury concurrency interface.
%
% The term ‘concurrency’ here refers to threads, not necessarily to parallel
% execution.  (The latter is also possible if you are using one of the .par
% grades and the lowlevel C backend, e.g. grade asm_fast.par.gc).
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- module thread.
:- interface.

:- import_module io.

:- include_module channel.
:- include_module mvar.
:- include_module semaphore.

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

    % can_spawn succeeds if spawn/3 is supported in the current grade.
    %
:- pred can_spawn 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.
    %
:- pred spawn(pred(io, io)::in(pred(di, uo) is cc_multi),
    io::di, io::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.

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

79 thread.mvar

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 2000-2003, 2006-2007, 2011 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% 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.
    %
:- impure func mvar.init = (mvar(T)::uo) is det.

    % Create an empty mvar.
    %
:- pred mvar.init(mvar(T)::out, io::di, io::uo) is det.

    % Take the contents of the mvar out leaving the mvar empty.
    % If the mvar is empty, block until some thread fills the mvar.
    %
:- pred mvar.take(mvar(T)::in, T::out, io::di, io::uo) is det.

    % Take the contents of the mvar out leaving the mvar empty.
    % Returns immediately with no if the mvar was empty, or yes(X) if
    % the mvar contained X.
    %
:- pred mvar.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 block until it becomes empty.
    %
:- pred mvar.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, return no immediately without blocking.
    %
:- pred mvar.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, block until it is full.
    % This is equivalent to mvar.take followed by mvar.put.
    %
:- pred mvar.read(mvar(T)::in, T::out, io::di, io::uo) is det.

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

80 thread.semaphore

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 2000-2001,2003-2004, 2006-2007, 2009-2011 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% File: thread.semaphore.m.
% Main author: conway.
% Stability: medium.
%
% This module implements a simple semaphore data type for allowing
% coroutines 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(Sem, !IO) creates a new semaphore ‘Sem’ with its counter
    % initialized to 0.
    %
:- pred semaphore.init(semaphore::out, io::di, io::uo) is det.

    % Returns a new semaphore ‘Sem’ with its counter initialized to Count.
    %
:- impure func semaphore.init(int::in) = (semaphore::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 semaphore.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 semaphore.try_wait(semaphore::in, bool::out, io::di, io::uo) is det.

    % 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 coroutines that are waiting on this semaphore (if
    % any).
    %
:- pred semaphore.signal(semaphore::in, io::di, io::uo) is det.

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

81 time

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%---------------------------------------------------%
% 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.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%---------------------------------------------------%
%
% 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 'time.clock' or 'time.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

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

    % time.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 `time.clocks_per_sec'.
    %
    % On Java the elapsed time for the calling thread is returned.
    %
:- pred time.clock(clock_t::out, io::di, io::uo) is det.

    % time.clocks_per_sec:
    %
    % Returns the number of "clocks" per second as defined by CLOCKS_PER_SEC.
    % A `clock_t' value returned by `time.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 time.clock.
    %
:- func time.clocks_per_sec = int.

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

    % time.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(time_t::out, io::di, io::uo) is det.

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

    % time.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 ‘time.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 time.times(tms::out, clock_t::out, io::di, io::uo) is det.

    % time.clk_tck:
    %
    % Returns the number of "clock ticks" per second as defined by
    % sysconf(_SC_CLK_TCK). A ‘clock_t’ value returned by ‘time.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 time.clk_tck = int.

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

    % time.difftime(Time1, Time0) = Diff:
    %
    % Computes the number of seconds elapsed between ‘Time1’ and ‘Time0’.
    %
:- func time.difftime(time_t, time_t) = float.

    % time.localtime(Time) = TM:
    %
    % Converts the calendar time ‘Time’ to a broken-down representation,
    % expressed relative to the user’s specified time zone.
    %
:- func time.localtime(time_t) = tm.

    % time.gmtime(Time) = TM:
    %
    % Converts the calendar time ‘Time’ to a broken-down representation,
    % expressed as UTC (Universal Coordinated Time).
    %
```

```
:- func time.gmtime(time_t) = tm.

    % time.mktime(TM) = Time:
    %
    % Converts the broken-down local time value to calendar time.
    % It also normalises the value by filling in day of week and day of year
    % based on the other components.
    %
:- func time.mktime(tm) = time_t.

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

    % time.asctime(TM) = String:
    %
    % Converts the broken-down time value `TM' to a string in a standard
    % format.
    %
:- func time.asctime(tm) = string.

    % time.ctime(Time) = String:
    %
    % Converts the calendar time value `Time' to a string in a standard format
    % (i.e. same as "asctime (localtime (<time>))").
    %
:- func time.ctime(time_t) = string.

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

82 tree234

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%-----------------------------------------------------%
% Copyright (C) 1994-1997,1999-2000,2002-2012 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% File: tree234.m.
% Main author: conway.
% Stability: medium.
%
% This module implements a map (dictionary) using 2-3-4 trees - see
% map.m for futher documentation.
```

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

:- module tree234.
:- interface.

:- import_module assoc_list.
:- import_module list.
:- import_module pretty_printer.
:- import_module term.

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

:- type tree234(K, V).

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

:- func tree234.init = tree234(K, V).
:- pred tree234.init(tree234(K, V)::uo) is det.

:- func tree234.singleton(K, V) = tree234(K, V).

:- pred tree234.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 tree234.equal(tree234(K, V)::in, tree234(K, V)::in) is semidet.

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

:- pred tree234.member(tree234(K, V)::in, K::out, V::out) is nondet.

:- pred tree234.search(tree234(K, V)::in, K::in, V::out) is semidet.

:- func tree234.lookup(tree234(K, V), K) = V.
:- pred tree234.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 tree234.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.
    % Aborts if there is no key with the given or lower value.
    %
:- pred tree234.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 tree234.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.
    % Aborts if there is no key with the given or higher value.
    %
:- pred tree234.upper_bound_lookup(tree234(K, V)::in, K::in, K::out, V::out)
    is det.

:- func tree234.max_key(tree234(K, V)) = K is semidet.

:- func tree234.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 tree234.insert(K::in, V::in, tree234(K, V)::in, tree234(K, V)::out)
    is semidet.

    % tree234.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.
    %
:- pred tree234.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 tree234.update(K::in, V::in, tree234(K, V)::in, tree234(K, V)::out)
    is semidet.

    % tree234.set(K, V, !Tree):
    %
    % Set the value corresponding to K to V, regardless of whether K is
    % already in the tree or not.
    %
:- func tree234.set(tree234(K, V), K, V) = tree234(K, V).
:- pred tree234.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 tree234.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 tree234.delete(tree234(K, V), K) = tree234(K, V).
:- pred tree234.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 tree234.remove(K, V, tree234(K, V), tree234(K, V)).
:- mode tree234.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 tree234.remove_smallest(K, V, tree234(K, V), tree234(K, V)).
:- mode tree234.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 tree234.keys(tree234(K, V)) = list(K).
```

```
:- pred tree234.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 tree234.values(tree234(K, V)) = list(V).
:- pred tree234.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 tree234.keys_and_values(tree234(K, V)::in, list(K)::out, list(V)::out)
    is det.

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

    % Count the number of elements in a tree.
    %
:- func tree234.count(tree234(K, V)) = int.
:- pred tree234.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.tree234_to_assoc_list(tree234(K, V)) = assoc_list(K, V).
:- pred tree234.tree234_to_assoc_list(tree234(K, V)::in,
    assoc_list(K, V)::out) is det.

:- func tree234.assoc_list_to_tree234(assoc_list(K, V)) = tree234(K, V).
:- pred tree234.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 tree234.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 tree234.from_rev_sorted_assoc_list(assoc_list(K, V)::in,
```

```
    tree234(K, V)::out) is det.

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

:- func tree234.foldl(func(K, V, A) = A, tree234(K, V), A) = A.

:- pred tree234.foldl(pred(K, V, A, A), tree234(K, V), A, A).
:- mode tree234.foldl(pred(in, in, in, out) is det, in, in, out) is det.
:- mode tree234.foldl(pred(in, in, mdi, muo) is det, in, mdi, muo) is det.
:- mode tree234.foldl(pred(in, in, di, uo) is det, in, di, uo) is det.
:- mode tree234.foldl(pred(in, in, in, out) is semidet, in, in, out)
    is semidet.
:- mode tree234.foldl(pred(in, in, mdi, muo) is semidet, in, mdi, muo)
    is semidet.
:- mode tree234.foldl(pred(in, in, di, uo) is semidet, in, di, uo)
    is semidet.
:- mode tree234.foldl(pred(in, in, in, out) is cc_multi, in, in, out)
    is cc_multi.
:- mode tree234.foldl(pred(in, in, di, uo) is cc_multi, in, di, uo)
    is cc_multi.
:- mode tree234.foldl(pred(in, in, mdi, muo) is cc_multi, in, mdi, muo)
    is cc_multi.

:- pred tree234.foldl2(pred(K, V, A, A, B, B), tree234(K, V), A, A, B, B).
:- mode tree234.foldl2(pred(in, in, in, out, in, out) is det,
    in, in, out, in, out) is det.
:- mode tree234.foldl2(pred(in, in, in, out, mdi, muo) is det,
    in, in, out, mdi, muo) is det.
:- mode tree234.foldl2(pred(in, in, in, out, di, uo) is det,
    in, in, out, di, uo) is det.
:- mode tree234.foldl2(pred(in, in, di, uo, di, uo) is det,
    in, di, uo, di, uo) is det.
:- mode tree234.foldl2(pred(in, in, in, out, in, out) is semidet,
    in, in, out, in, out) is semidet.
:- mode tree234.foldl2(pred(in, in, in, out, mdi, muo) is semidet,
    in, in, out, mdi, muo) is semidet.
:- mode tree234.foldl2(pred(in, in, in, out, di, uo) is semidet,
    in, in, out, di, uo) is semidet.
:- mode tree234.foldl2(pred(in, in, in, out, in, out) is cc_multi,
    in, in, out, in, out) is cc_multi.
:- mode tree234.foldl2(pred(in, in, in, out, mdi, muo) is cc_multi,
    in, in, out, mdi, muo) is cc_multi.
:- mode tree234.foldl2(pred(in, in, in, out, di, uo) is cc_multi,
    in, in, out, di, uo) is cc_multi.
:- mode tree234.foldl2(pred(in, in, di, uo, di, uo) is cc_multi,
    in, di, uo, di, uo) is cc_multi.
```

```
:- pred tree234.foldl3(pred(K, V, A, A, B, B, C, C), tree234(K, V),
A, A, B, B, C, C).
:- mode tree234.foldl3(pred(in, in, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out) is det.
:- mode tree234.foldl3(pred(in, in, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, mdi, muo) is det.
:- mode tree234.foldl3(pred(in, in, in, out, in, out, di, uo) is det,
    in, in, out, in, out, di, uo) is det.
:- mode tree234.foldl3(pred(in, in, in, out, di, uo, di, uo) is det,
    in, in, out, di, uo, di, uo) is det.
:- mode tree234.foldl3(pred(in, in, di, uo, di, uo, di, uo) is det,
    in, di, uo, di, uo, di, uo) is det.
:- mode tree234.foldl3(pred(in, in, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out) is semidet.
:- mode tree234.foldl3(pred(in, in, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, mdi, muo) is semidet.
:- mode tree234.foldl3(pred(in, in, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, di, uo) is semidet.

:- pred tree234.foldl4(pred(K, V, A, A, B, B, C, C, D, D), tree234(K, V),
A, A, B, B, C, C, D, D).
:- mode tree234.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 tree234.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 tree234.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 tree234.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 tree234.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 tree234.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.

:- pred tree234.foldl_values(pred(V, A, A), tree234(K, V), A, A).
:- mode tree234.foldl_values(pred(in, in, out) is det, in, in, out) is det.
:- mode tree234.foldl_values(pred(in, mdi, muo) is det, in, mdi, muo) is det.
:- mode tree234.foldl_values(pred(in, di, uo) is det, in, di, uo) is det.
:- mode tree234.foldl_values(pred(in, in, out) is semidet, in, in, out)
    is semidet.
:- mode tree234.foldl_values(pred(in, mdi, muo) is semidet, in, mdi, muo)
    is semidet.
:- mode tree234.foldl_values(pred(in, di, uo) is semidet, in, di, uo)
    is semidet.
:- mode tree234.foldl_values(pred(in, in, out) is cc_multi, in, in, out)
```

```
    is cc_multi.
:- mode tree234.foldl_values(pred(in, di, uo) is cc_multi, in, di, uo)
    is cc_multi.
:- mode tree234.foldl_values(pred(in, mdi, muo) is cc_multi, in, mdi, muo)
    is cc_multi.

:- pred tree234.foldl2_values(pred(V, A, A, B, B), tree234(K, V), A, A, B, B).
:- mode tree234.foldl2_values(pred(in, in, out, in, out) is det, in,
    in, out, in, out) is det.
:- mode tree234.foldl2_values(pred(in, in, out, mdi, muo) is det, in,
    in, out, mdi, muo) is det.
:- mode tree234.foldl2_values(pred(in, in, out, di, uo) is det, in,
    in, out, di, uo) is det.
:- mode tree234.foldl2_values(pred(in, in, out, in, out) is semidet, in,
    in, out, in, out) is semidet.
:- mode tree234.foldl2_values(pred(in, in, out, mdi, muo) is semidet, in,
    in, out, mdi, muo) is semidet.
:- mode tree234.foldl2_values(pred(in, in, out, di, uo) is semidet, in,
    in, out, di, uo) is semidet.
:- mode tree234.foldl2_values(pred(in, in, out, in, out) is cc_multi, in,
    in, out, in, out) is cc_multi.
:- mode tree234.foldl2_values(pred(in, in, out, mdi, muo) is cc_multi, in,
    in, out, mdi, muo) is cc_multi.
:- mode tree234.foldl2_values(pred(in, in, out, di, uo) is cc_multi, in,
    in, out, di, uo) is cc_multi.

:- pred tree234.foldl3_values(pred(V, A, A, B, B, C, C), tree234(K, V),
    A, A, B, B, C, C).
:- mode tree234.foldl3_values(pred(in, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out) is det.
:- mode tree234.foldl3_values(pred(in, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, mdi, muo) is det.
:- mode tree234.foldl3_values(pred(in, in, out, in, out, di, uo) is det,
    in, in, out, in, out, di, uo) is det.
:- mode tree234.foldl3_values(pred(in, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out) is semidet.
:- mode tree234.foldl3_values(pred(in, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, mdi, muo) is semidet.
:- mode tree234.foldl3_values(pred(in, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, di, uo) is semidet.
:- mode tree234.foldl3_values(pred(in, in, out, in, out, in, out) is cc_multi,
    in, in, out, in, out, in, out) is cc_multi.
:- mode tree234.foldl3_values(pred(in, in, out, in, out, mdi, muo) is cc_multi,
    in, in, out, in, out, mdi, muo) is cc_multi.
:- mode tree234.foldl3_values(pred(in, in, out, in, out, di, uo) is cc_multi,
    in, in, out, in, out, di, uo) is cc_multi.
```

```
:- func tree234.foldr(func(K, V, A) = A, tree234(K, V), A) = A.

:- pred tree234.foldr(pred(K, V, A, A), tree234(K, V), A, A).
:- mode tree234.foldr(pred(in, in, in, out) is det, in, in, out) is det.
:- mode tree234.foldr(pred(in, in, mdi, muo) is det, in, mdi, muo) is det.
:- mode tree234.foldr(pred(in, in, di, uo) is det, in, di, uo) is det.
:- mode tree234.foldr(pred(in, in, in, out) is semidet, in, in, out)
    is semidet.
:- mode tree234.foldr(pred(in, in, mdi, muo) is semidet, in, mdi, muo)
    is semidet.
:- mode tree234.foldr(pred(in, in, di, uo) is semidet, in, di, uo)
    is semidet.
:- mode tree234.foldr(pred(in, in, in, out) is cc_multi, in, in, out)
    is cc_multi.
:- mode tree234.foldr(pred(in, in, di, uo) is cc_multi, in, di, uo)
    is cc_multi.
:- mode tree234.foldr(pred(in, in, mdi, muo) is cc_multi, in, mdi, muo)
    is cc_multi.

:- pred tree234.foldr2(pred(K, V, A, A, B, B), tree234(K, V), A, A, B, B).
:- mode tree234.foldr2(pred(in, in, in, out, in, out) is det,
    in, in, out, in, out) is det.
:- mode tree234.foldr2(pred(in, in, in, out, mdi, muo) is det,
    in, in, out, mdi, muo) is det.
:- mode tree234.foldr2(pred(in, in, in, out, di, uo) is det,
    in, in, out, di, uo) is det.
:- mode tree234.foldr2(pred(in, in, di, uo, di, uo) is det,
    in, di, uo, di, uo) is det.
:- mode tree234.foldr2(pred(in, in, in, out, in, out) is semidet,
    in, in, out, in, out) is semidet.
:- mode tree234.foldr2(pred(in, in, in, out, mdi, muo) is semidet,
    in, in, out, mdi, muo) is semidet.
:- mode tree234.foldr2(pred(in, in, in, out, di, uo) is semidet,
    in, in, out, di, uo) is semidet.

:- pred tree234.foldr3(pred(K, V, A, A, B, B, C, C), tree234(K, V),
A, A, B, B, C, C).
:- mode tree234.foldr3(pred(in, in, in, out, in, out, in, out) is det,
    in, in, out, in, out, in, out) is det.
:- mode tree234.foldr3(pred(in, in, in, out, in, out, mdi, muo) is det,
    in, in, out, in, out, mdi, muo) is det.
:- mode tree234.foldr3(pred(in, in, in, out, in, out, di, uo) is det,
    in, in, out, in, out, di, uo) is det.
:- mode tree234.foldr3(pred(in, in, in, out, di, uo, di, uo) is det,
    in, in, out, di, uo, di, uo) is det.
:- mode tree234.foldr3(pred(in, in, di, uo, di, uo, di, uo) is det,
    in, di, uo, di, uo, di, uo) is det.
```

```
:- mode tree234.foldr3(pred(in, in, in, out, in, out, in, out) is semidet,
    in, in, out, in, out, in, out) is semidet.
:- mode tree234.foldr3(pred(in, in, in, out, in, out, mdi, muo) is semidet,
    in, in, out, in, out, mdi, muo) is semidet.
:- mode tree234.foldr3(pred(in, in, in, out, in, out, di, uo) is semidet,
    in, in, out, in, out, di, uo) is semidet.

:- pred tree234.foldr4(pred(K, V, A, A, B, B, C, C, D, D), tree234(K, V),
A, A, B, B, C, C, D, D).
:- mode tree234.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 tree234.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 tree234.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 tree234.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 tree234.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 tree234.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 tree234.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 tree234.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 tree234.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.

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

:- func tree234.map_values(func(K, V) = W, tree234(K, V)) = tree234(K, W).

:- pred tree234.map_values(pred(K, V, W), tree234(K, V), tree234(K, W)).
:- mode tree234.map_values(pred(in, in, out) is det, in, out) is det.
:- mode tree234.map_values(pred(in, in, out) is semidet, in, out) is semidet.

:- func tree234.map_values_only(func(V) = W, tree234(K, V)) = tree234(K, W).

:- pred tree234.map_values_only(pred(V, W), tree234(K, V), tree234(K, W)).
:- mode tree234.map_values_only(pred(in, out) is det, in, out) is det.
:- mode tree234.map_values_only(pred(in, out) is semidet, in, out) is semidet.
```

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

:- pred tree234.map_foldl(pred(K, V, W, A, A), tree234(K, V), tree234(K, W),
    A, A).
:- mode tree234.map_foldl(pred(in, in, out, in, out) is det,
    in, out, in, out) is det.
:- mode tree234.map_foldl(pred(in, in, out, mdi, muo) is det,
    in, out, mdi, muo) is det.
:- mode tree234.map_foldl(pred(in, in, out, di, uo) is det,
    in, out, di, uo) is det.
:- mode tree234.map_foldl(pred(in, in, out, in, out) is semidet,
    in, out, in, out) is semidet.
:- mode tree234.map_foldl(pred(in, in, out, mdi, muo) is semidet,
    in, out, mdi, muo) is semidet.
:- mode tree234.map_foldl(pred(in, in, out, di, uo) is semidet,
    in, out, di, uo) is semidet.

:- pred tree234.map_foldl2(pred(K, V, W, A, A, B, B),
    tree234(K, V), tree234(K, W), A, A, B, B).
:- mode tree234.map_foldl2(pred(in, in, out, in, out, in, out) is det,
    in, out, in, out, in, out) is det.
:- mode tree234.map_foldl2(pred(in, in, out, in, out, mdi, muo) is det,
    in, out, in, out, mdi, muo) is det.
:- mode tree234.map_foldl2(pred(in, in, out, di, uo, di, uo) is det,
    in, out, di, uo, di, uo) is det.
:- mode tree234.map_foldl2(pred(in, in, out, in, out, di, uo) is det,
    in, out, in, out, di, uo) is det.
:- mode tree234.map_foldl2(pred(in, in, out, in, out, in, out) is semidet,
    in, out, in, out, in, out) is semidet.
:- mode tree234.map_foldl2(pred(in, in, out, in, out, mdi, muo) is semidet,
    in, out, in, out, mdi, muo) is semidet.
:- mode tree234.map_foldl2(pred(in, in, out, in, out, di, uo) is semidet,
    in, out, in, out, di, uo) is semidet.

:- pred tree234.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 tree234.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 tree234.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 tree234.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 tree234.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 tree234.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 tree234.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 tree234.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 tree234.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 tree234.map_values_foldl(pred(V, W, A, A),
    tree234(K, V), tree234(K, W), A, A).
:- mode tree234.map_values_foldl(pred(in, out, di, uo) is det,
    in, out, di, uo) is det.
:- mode tree234.map_values_foldl(pred(in, out, in, out) is det,
    in, out, in, out) is det.
:- mode tree234.map_values_foldl(pred(in, out, in, out) is semidet,
    in, out, in, out) is semidet.

:- pred tree234.map_values_foldl2(pred(V, W, A, A, B, B),
    tree234(K, V), tree234(K, W), A, A, B, B).
:- mode tree234.map_values_foldl2(pred(in, out, di, uo, di, uo) is det,
    in, out, di, uo, di, uo) is det.
:- mode tree234.map_values_foldl2(pred(in, out, in, out, di, uo) is det,
    in, out, in, out, di, uo) is det.
:- mode tree234.map_values_foldl2(pred(in, out, in, out, in, out) is det,
    in, out, in, out, in, out) is det.
:- mode tree234.map_values_foldl2(pred(in, out, in, out, in, out) is semidet,
    in, out, in, out, in, out) is semidet.

:- pred tree234.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 tree234.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 tree234.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 tree234.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 tree234.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 tree234.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.

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

83 tree_bitset

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et
%---------------------------------------------------%
% Copyright (C) 2006, 2009-2012 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%---------------------------------------------------%
%
% 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,
% 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).

    % Return an empty set.
    %
:- func init = tree_bitset(T).

:- pred empty(tree_bitset(T)).
:- mode empty(in) is semidet.
:- mode empty(out) is det.

:- pred is_empty(tree_bitset(T)::in) is semidet.

:- pred is_non_empty(tree_bitset(T)::in) is semidet.

    % '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).

    % '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).

    % '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 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).

    % `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).

    % `make_singleton_set(Elem)' returns a set containing just the single
    % element `Elem'.
    %
:- func make_singleton_set(T) = tree_bitset(T) <= enum(T).

    % 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).

    % `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.

    % `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).

    % `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.

    % `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).

    % '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.

    % 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).

    % 'count(Set)' returns the number of elements in 'Set'.
    % Takes O(card(Set)) time.
    %
:- func count(tree_bitset(T)) = int <= enum(T).

    % '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.

    % 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.

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

84 type_desc

```
%---------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%---------------------------------------------------%
% Copyright (C) 2002-2007, 2009-2012 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%---------------------------------------------------%
%
% 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.

    % Convert a pseudo_type_desc describing a ground type to a type_desc.
    % If the pseudo_type_desc describes a non-ground type, abort.
    %
:- 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. Aborts 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.
```

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

85 unit

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%-----------------------------------------------------%
% Copyright (C) 1994-2006 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% 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.
%-----------------------------------------------------%
```

86 univ

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%-----------------------------------------------------%
% Copyright (C) 1994-2010 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
```

```
%
% 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 aborts
    % 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.

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

87 varset

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%-----------------------------------------------------%
% Copyright (C) 1993-2000,2002-2007, 2009-2011 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% File: varset.m.
% Main author: fjh.
% Stability: low.
%
% This file provides facilities for manipulating collections of
% variables and terms.
% It provides the 'varset' ADT. A varset is a set of variables.
% (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 variables.)
% Associated with each variable there can be both a name and a value
% (binding).
%
% There may be some design flaws in the relationship between varset.m, and
% term.m.  Once we have implemented unique modes and destructive assign-
ment, we
% will need to rethink the design;  we may end up modifying these modules
% considerably, or we may end up making new single-threaded versions of these
% 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 varset.init = varset(T).
:- pred varset.init(varset(T)::out) is det.

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

    % Create a new variable.
    %
:- pred varset.new_var(var(T)::out, varset(T)::in, varset(T)::out) is det.

    % Create a new named variable.
    %
:- pred varset.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 varset.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 varset.new_maybe_named_var(maybe(string)::in, var(T)::out,
    varset(T)::in, varset(T)::out) is det.

    % Create multiple new variables.
    %
:- pred varset.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 varset.delete_var(varset(T), var(T)) = varset(T).
:- pred varset.delete_var(var(T)::in, varset(T)::in, varset(T)::out) is det.

    % Delete the names and values for a list of variables.
    %
:- func varset.delete_vars(varset(T), list(var(T))) = varset(T).
:- pred varset.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 result will be either an abort
    % or incorrect output.)
    %
:- func varset.delete_sorted_vars(varset(T), list(var(T))) = varset(T).
:- pred varset.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 varset.vars(varset(T)) = list(var(T)).
:- pred varset.vars(varset(T)::in, list(var(T))::out) is det.

    % Set the name of a variable.
    %
:- func varset.name_var(varset(T), var(T), string) = varset(T).
:- pred varset.name_var(var(T)::in, string::in,
    varset(T)::in, varset(T)::out) is det.
```

```
    % Lookup the name of a variable;
    % create one if it doesn't have one using V_ as a prefix.
    %
:- func varset.lookup_name(varset(T), var(T)) = string.
:- pred varset.lookup_name(varset(T)::in, var(T)::in, string::out) is det.

    % Lookup the name of a variable;
    % create one if it doesn't have one using the specified prefix
    %
:- func varset.lookup_name(varset(T), var(T), string) = string.
:- pred varset.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 varset.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 varset.bind_var(varset(T), var(T), term(T)) = varset(T).
:- pred varset.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 varset.bind_vars(varset(T), substitution(T)) = varset(T).
:- pred varset.bind_vars(substitution(T)::in,
    varset(T)::in, varset(T)::out) is det.

    % Lookup the value of a variable.
    %
:- pred varset.search_var(varset(T)::in, var(T)::in, term(T)::out) is semidet.

    % Get the bindings for all the bound variables.
    %
:- func varset.lookup_vars(varset(T)) = substitution(T).
:- pred varset.lookup_vars(varset(T)::in, substitution(T)::out) is det.

    % Combine two different varsets, renaming apart:
    % varset.merge_renaming(VarSet0, NewVarSet, VarSet, Subst) 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 map the variables in NewVarSet into the corresponding
    % fresh variable in VarSet.
```

```
    %
:- pred varset.merge_renaming(varset(T)::in, varset(T)::in, varset(T)::out,
    map(var(T), var(T))::out) is det.

    % Does the same job as varset.merge_renaming, but returns the renaming
    % as a general substitution in which all the terms in the range happen
    % to be variables.
    %
    % Consider using varset.merge_renaming instead.
    %
:- pred varset.merge_subst(varset(T)::in, varset(T)::in, varset(T)::out,
    substitution(T)::out) is det.

    % varset.merge(VarSet0, NewVarSet, Terms0, VarSet, Terms):
    %
    % As varset.merge_renaming, except instead of returning the renaming,
    % this predicate applies it to the given list of terms.
    %
:- pred varset.merge(varset(T)::in, varset(T)::in, list(term(T))::in,
    varset(T)::out, list(term(T))::out) is det.

    % Same as varset.merge_renaming, except that the names of variables
    % in NewVarSet are not included in the final varset.
    % This is useful if varset.create_name_var_map needs to be used
    % on the resulting varset.
    %
:- pred varset.merge_renaming_without_names(varset(T)::in,
    varset(T)::in, varset(T)::out, map(var(T), var(T))::out) is det.

    % Same as varset.merge_subst, except that the names of variables
    % in NewVarSet are not included in the final varset.
    % This is useful if varset.create_name_var_map needs to be used
    % on the resulting varset.
    %
    % Consider using varset.merge_renaming_without_names instead.
    %
:- pred varset.merge_subst_without_names(varset(T)::in,
    varset(T)::in, varset(T)::out, substitution(T)::out) is det.

    % Same as varset.merge, except that the names of variables
    % in NewVarSet are not included in the final varset.
    % This is useful if varset.create_name_var_map needs to be used
    % on the resulting varset.
    %
:- pred varset.merge_without_names(varset(T)::in, varset(T)::in,
    list(term(T))::in, varset(T)::out, list(term(T))::out) is det.
```

```
    % Get the bindings for all the bound variables.
    %
:- func varset.get_bindings(varset(T)) = substitution(T).
:- pred varset.get_bindings(varset(T)::in, substitution(T)::out) is det.

    % Set the bindings for all the bound variables.
    %
:- func varset.set_bindings(varset(T), substitution(T)) = varset(T).
:- pred varset.set_bindings(varset(T)::in, substitution(T)::in,
    varset(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 varset.create_name_var_map(varset(T)) = map(string, var(T)).
:- pred varset.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 varset.var_name_list(varset(T)) = assoc_list(var(T), string).
:- pred varset.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 varset.ensure_unique_names(list(var(T)), string, varset(T))
    = varset(T).
:- pred varset.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 varset.select(varset(T), set(var(T))) = varset(T).
:- pred varset.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 varset.squash(varset(T)::in, list(var(T))::in,
    varset(T)::out, map(var(T), var(T))::out) is det.

    % Coerce the types of the variables in a varset.
    %
:- func varset.coerce(varset(T)) = varset(U).
:- pred varset.coerce(varset(T)::in, varset(U)::out) is det.

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

88 version_array

```
%-----------------------------------------------------%
% vim: ts=4 sw=4 et tw=0 wm=0 ft=mercury
%-----------------------------------------------------%
% Copyright (C) 2004-2012 The University of Melbourne.
% Copyright (C) 2014 The Mercury Team.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%-----------------------------------------------------%
%
% 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.
%
% XXX This implementation is not yet guaranteed to work with the agc (accurate
% garbage collection) grades. Specifically, MR_deep_copy and MR_agc_deep_copy
% currently do not recognise version arrays.
%
%-----------------------------------------------------%
%-----------------------------------------------------%

:- 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).

    % An obsolete synonym for the above.
    %
:- pragma obsolete(unsafe_new/2).
:- func unsafe_new(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).

    % A ^ elem(I) = X iff the Ith member of A is X (the first item has
    % index 0).
    %
:- func version_array(T) ^ elem(int) = T.
```

```
    % lookup(A, I) = A ^ elem(I).
    %
:- func lookup(version_array(T), int) = T.

    % (A ^ elem(I) := X) is a copy of array A with item I updated to be X.
    % An exception is thrown if I is out of bounds. set/4 is an equivalent
    % predicate.
    %
:- func (version_array(T) ^ elem(int) := T) = version_array(T).

:- pred set(int::in, T::in, version_array(T)::in, version_array(T)::out)
    is det.

    % 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.
    %
:- 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.

    % 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.

    % copy(A) is a copy of array A. Access to the copy is O(1).
    %
:- func copy(version_array(T)) = version_array(T).

    % 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.

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

89 version_array2d

```
%----------------------------------------------------%
% Copyright (C) 2004-2006, 2011 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%----------------------------------------------------%
%
% 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 forth.)
    %
:- 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).

    % 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.

    % 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.

    % 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).

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

90 version_bitmap

```
%-----------------------------------------------------%
% Copyright (C) 2004-2007, 2010-2011 The University of Melbourne
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%-----------------------------------------------------%
%
% 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.

    % Returns the number of bits in a version_bitmap.
    %
:- func num_bits(version_bitmap) = int.

    % 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.

    % Get the given bit.
    %
:- func version_bitmap ^ bit(int) = bool.

    % Set the given bit.
    %
:- func (version_bitmap ^ bit(int) := bool) = version_bitmap.

    % 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.

    % 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.

    % 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.

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

91 version_hash_table

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%-----------------------------------------------------%
% Copyright (C) 2004-2006, 2010-2012 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
```

```
% 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 growing
% performance penalty 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 descendents 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).

    % Default hash_preds for ints and strings and everything (buwahahaha!)
    %
:- pred int_hash(int::in, int::out) is det.
:- pred string_hash(string::in, int::out) is det.
:- pred char_hash(char::in, int::out) is det.
:- pred float_hash(float::in, int::out) is det.
:- pred generic_hash(T::in, int::out) is det.

    % Returns the number of buckets in a hash table.
    %
:- func num_buckets(version_hash_table(K, V)) = int.

    % Returns the number of occupants in a hash table.
    %
:- func num_occupants(version_hash_table(K, V)) = int.

    % 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).

    % Insert key-value binding into a hash table; if one is
    % already there then the previous value is overwritten.
    % A predicate version is also provided.
    %
:- 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.  A predicate version is also provided.
    %
:- 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.  An
    % exception is thrown if a binding for the key does not
    % already exist.  A predicate version is also provided.
    %
:- 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.  A predicate version is also
    % provided.
    %
:- 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.

    % Lookup the value associated with the given key.  An exception
```

```
    % is raised 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.

    % Like lookup, but just fails 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.

    % 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).

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

92 version_store

```
%-----------------------------------------------------%
% vim: ft=mercury ts=4 sw=4 et wm=0 tw=0
%-----------------------------------------------------%
% Copyright (C) 2004-2006, 2011 The University of Melbourne.
% This file may only be copied under the terms of the GNU Library General
% Public License - see the file COPYING.LIB in the Mercury distribution.
%-----------------------------------------------------%
%
% 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.  whose mut-
var 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.

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