DCGs + memoing = packrat parsing: but is it worth it?

Ralph Becket and Zoltan Somogyi

National ICT Australia, and
Department of Computer Science and Software Engineering
The University of Melbourne

Packrat parsing

Recursive descent parsing requires backtracking for grammars that aren't LL(1). In the worst case, that backtracking may lead to exponential complexity.

Packrat parsers ensure linear complexity for such grammars by testing each production rule at most once against each position in the input stream.

They do this by incrementally constructing a table mapping each *(non-terminal, input position)* pair to *unknown*, *failed*, or *succeeded n*, where parsing succeeded consuming *n* input tokens (or characters in a scannerless parser).

The success entries may also contain other information, such as the abstract syntax tree of the matched input fragment.

Definite clause grammars

The standard way to write recursive descent parsers in logic programming languages is to use DCGs:

```
% DCG rule before expansion:
package_declaration --->
    keyword("package"),
    qualified_identifier,
    punct(";").

% DCG rule after expansion:
package_declaration(S0, S) :-
    keyword("package", S0, S1),
    qualified_identifier(S1, S2),
    punct(";", S2, S).
```

Parser state representations

chars: the state is a list of the characters remaining in the input.

single: the state is a triple: a string giving the entire contents of the input file, the length of that string, and the current offset.

global: the state is just an integer, the current offset. The input string and its length are stored in global variables, and accessed using impure foreign language code.

pass1: the state is just an integer, the current offset, but the input string and its length are passed around to every recognition predicate as a pair in one extra input argument.

pass2: the state is just an integer, the current offset, but the input string and its length are passed around to every recognition predicate as two separate extra input arguments.

Grammar to DCG

```
BlockStatement ::= LocalVariableDeclarationStatement
                |   ClassOrInterfaceDeclaration
                |   [Identifier :] Statement
```

```
% straightforward translation: allows multiple parses
block_statement -->
    local_variable_declaration_statement.
block_statement -->
    class_or_interface_declaration.
block_statement -->
    optional(label), statement.

% packrat-like translation: commits to first successful parse
block_statement -->
    ( if local_variable_declaration_statement then
        []
    else if class_or_interface_declaration then
        []
    else
        optional(label), statement
    ).
```

Memoing

```
:- pred fib(int::in, int::out) is det.
:- pragma memo(fib(in, out),
    [specified(value, output), allow_reset]).

fib(N, F) :-
    ( if N < 2 then
        F = 1
    else
        fib(N - 1, F1), fib(N - 2, F2), F = F1 + F2
    ).
```

The Mercury compiler will surround this with new code that

- looks up `N` in `fib`'s hash table, and
- if it is there, returns the recorded value of `F`, while
- if it is not there, executes the body and records the value of `F`.

Memoing applied to DCGs

Each input argument can be tabled by

- its value (which requires traversing the entire input term),
- by the address of that term (which is a single lookup), or
- not at all, if its value is promised to be implied by the other input arguments. (An argument whose value is constant between table resets qualifies.)

```
:- pred package_declaration(string::in, int::in,
   int::in, int::out) is semidet.

:- pragma memo(package_declaration/4, [allow_reset,
   specified([promise_implied, promise_implied, addr, output])]).

package_declaration(Str, Length) -->
   keyword("package", Str, Length),
   qualified_identifier(Str, Length),
   punct(";", Str, Length).
```

DCGs + memoing = packrat parsing

Packrat parsing uses a single 2D array:

	p1	p2	p3	p4	p5	p6	p7	p8	p9	p10
NT1		fail				succ 2		fail		
NT2			succ 3							
NT3									succ 1	

Memoing a recognizer predicate in a DCG parser uses a hash table that corresponds to that nonterminal's row in the packrat table.

Despite the structural difference (expandable hash table versus fixed size array), this has the same function: ensuring the parser won't try to match the nonterminal at a given input position more than once.

Packrat parser performance

To compare the performance of packrat and memoed DCG parsers, we used Sun's Java grammar. The test load was 735 large Java source files totalling over 900,000 lines and 9.6 megabytes. The test task was recognition only; no parse tree was built.

Parser version	Null load	Test load	Difference
unoptimized	0.56s	7.54s	6.98s
optimized	0.52s	6.92s	6.40s

The packrat parser we tested was the `xtc` Java parser generated with the Rats! packrat parser generator.

Rats! generates Java code, and uses more than a dozen optimizations, with one being including fewer nonterminals in the table. Our DCG parsers are written in Mercury, and have no other optimization.

DCG parser performance

Backend	Input	Best	None memoed	All memoed
high level C	chars	3.56s	4.60s (1.29, 77th)	14.08s (3.96, 98th)
high level C	single	3.38s	4.14s (1.22, 77th)	13.44s (3.98, 98th)
high level C	global	1.30s	1.34s (1.03, 16th)	10.63s (8.18, 98th)
high level C	pass1	1.35s	1.36s (1.01, 2nd)	10.66s (7.90, 98th)
high level C	pass2	1.24s	1.24s (1.00, 2nd)	10.65s (8.59, 98th)
low level C	chars	5.01s	5.03s (1.00, 2nd)	16.58s (3.31, 98th)
low level C	single	4.76s	5.01s (1.05, 4th)	15.94s (3.35, 98th)
low level C	global	1.82s	1.90s (1.04, 65th)	12.89s (7.08, 98th)
low level C	pass1	1.87s	1.92s (1.02, 13th)	13.18s (7.05, 98th)
low level C	pass2	2.13s	2.29s (1.08, 85th)	13.71s (6.44, 98th)

We tried 98 combinations of what to memo, including 92 that each memoed just one recognizer. Memoing everything was always the slowest, while memoing nothing was never anywhere that bad.

How can memoing a recognizer lead to a slowdown?

The point of memoing is to save time by avoiding reexecutions. However, memoing has costs in both time and space.

- If a recognizer is never called more than once in the same situation, memoing it incurs costs but yields no benefits.
- If the time taken by a call to a recognizer predicate is less than the cost of consulting that predicate's table, memoing that recognizer will lead to a slowdown *even if almost every call is a duplicate*.
- Suppose you have two recognizers r1 and r2 where r1 executes for N times as long as r2. The memory cost of r2 is likely to be much more than 1/Nth of the memory cost of r1, whether measured in bytes in memory, bytes in cache, traffic on the memory bus, or TLB entries tied down.

When should a recognizer be memoed?

Despite the apples-to-oranges nature of our experiments, we are pretty sure that one should only memo recognizer predicates

- which are reexecuted in the same input position reasonably frequently,
- whose time taken per-call is significantly more than their per-call tabling overhead (building a parse tree helps with this), and
- which consume nontrivial execution time overall.

For the Java grammar, very few recognizers pass all three tests.

Rats! starts by memoing everything and then decides what *should not be* memoed.

We believe one should start by memoing nothing and then decide what *should be* memoed.

Conclusion and further work

Memoing a recognizer only when doing so is likely to yield a speedup will typically fail to provide the linear time guarantee provided by memoing everything, or by packrat parsing.

However, memoing everything risks a slowdown from DRAM speed to disk speed. Technically, time complexity may still be linear, but users will still complain.

Our guidelines for when to memo recognizers would have more weight behind them if they were confirmed by repeats of our experiments for other grammars and other kinds of grammars (e.g. natural languages, which should have more ambiguity).

A more apples-to-apples comparison would unfortunately require either a packrat parser generator for a logic programming language or a DCG parser for a language like Java; neither is likely.

DCGs with computation

```
% DCG rules before expansion:
nat(N)       --> digit(D), digits(D, N).
digit(D)     --> [X], { char.digit_to_int(X, D) }.
digits(M, N) -->
    ( if digit(D) then digits(10 * M + D, N) else { N = M } ).

% DCG rules after expansion:
nat(N, S0, S)    :- digit(D, S0, S1), digits(D, N, S1, S).
digit(D, S0, S)  :- S0 = [X | S], char.digit_to_int(X, D).
digits(M, N, S0, S) :-
    ( if digit(D, S0, S1) then
        digits(10 * M + D, N, S1, S)
    else
        N = M, S = S0
    ).
```

Grammar patterns as higher order predicates

```
:- pred optional(
    pred(ps, ps)::in(pred(in, out) is semidet),
    ps::in, ps::out) is semidet.

optional(P) -->
    ( if P then
        []
    else
        { semidet_succeed }
    ).
```

Grammar patterns as higher order predicates

```
:- pred optional(
    pred(string, int, int, int)::
        in(pred(in, in, in, out) is semidet),
    string::in, int::in, int::in, int::out) is semidet.

optional(P, Str, Length) -->
    ( if P(Str, Length) then
        []
    else
        { semidet_succeed }
    ).
```