DCGs + Memoing = Packrat Parsing But is it worth it?

Ralph Becket and Zoltan Somogyi

NICTA and
Department of Computer Science and Software Engineering
The University of Melbourne, 111 Barry Street, Parkville
Victoria 3010, Australia
{rafe,zs}@csse.unimelb.edu.au

Abstract. Packrat parsing is a newly popular technique for efficiently implementing recursive descent parsers. Packrat parsing avoids the potential exponential costs of recursive descent parsing with backtracking by ensuring that each production rule in the grammar is tested at most once against each position in the input stream. This paper argues that (a) packrat parsers can be trivially implemented using a combination of definite clause grammar rules and memoing, and that (b) packrat parsing may actually be significantly *less* efficient than plain recursive descent with backtracking, but (c) memoing the recognizers of just one or two nonterminals, selected in accordance with Amdahl's law, can sometimes yield speedups. We present experimental evidence to support these claims.

Keywords: Mercury, parsing, packrat, recursive descent, DCG, memoing, tabling.

1 Introduction

Recursive descent parsing has many attractions: (a) it is simple to understand and implement; (b) all the features of the implementation language are available to the parser developer; (c) complex rules can be implemented easily (for instance, longest match, or A not followed by B); (d) parsing rules may if necessary depend on 'state' such as the current contents of the symbol table; and (e) higher order rules may be used to abstract away common grammatical features (e.g. comma-separated lists).

However, recursive descent parsing requires backtracking for grammars that aren't LL(1) and, in the worst case, that backtracking may lead to exponential complexity. Consider the grammar in figure 1. Assuming a top-down, left-to-right, recursive descent strategy, matching the nonterminal "`a`" against the input string "`xxxzzz`" involves testing all possible length six expansions of `a` before succeeding.

```
a  ::=  b | c
b  ::=  'x' d 'y'
c  ::=  'x' d 'z'
d  ::=  a | epsilon
```

Fig. 1. *A grammar with pathological recursive descent behaviour:* `a`, `b`, `c`, `d` *are non-terminals;* `'x'`, `'y'`, `'z'` *are terminals;* `epsilon` *matches the empty string.*

Packrat parsing ensures linear complexity for such grammars by testing each production rule at most once against each position in the input stream. This is typically done by incrementally constructing a table mapping each *(non-terminal, input position)* pair to *unknown*, *failed*, or a number n where parsing succeeded consuming n input tokens. (In practice the 'succeeded' entries may also contain other information, such as abstract syntax tree representations of the matched input fragment.) Figure 2 gives an example of a packrat table being filled out.

	Input position					
	1	2	3	4	5	6
a	6_{-15-}	4_{-11-}	2_{-7-}	$failed_{-3-}$		
b	$failed_{-13-}$	$failed_{-9-}$	$failed_{-5-}$	$failed_{-1-}$		
c	6_{-14-}	4_{-10-}	2_{-6-}	$failed_{-2-}$		
d		4_{-12-}	2_{-8-}	0_{-4-}		

Fig. 2. *Filling out a packrat table matching non-terminal* `a` *from figure 1 against "*`xxxzzz`*". Blank entries denote* unknown*, 'failed' denotes a non-match, numbers denote successful parsing consuming that number of input tokens, and –subscripts– show the order in which entries were added.*

Packrat parsing has recently been made popular by packages such as Pappy (Bryan Ford's Haskell package [2]) and Rats! (Robert Grimm's Java package [3]).

There are two key problems with the packrat approach. First, the table can consume prodigious amounts of memory (the Java parser generated by Pappy requires up to 400 bytes of memory for every byte of input). Second, it cannot easily be extended to handle contextual information, such as the current line number. Another, perhaps less significant,

problem is that the packrat table cannot be used for nondeterministic grammars.

In this paper, we describe an alternative approach to packrat parsing that can avoid these problems. First, section 2 shows that recursive descent parsers can be constructed quite easily in Mercury [8] using Definite Clause Grammars [6]. Section 3 goes on to show how Mercury DCG parsers can be trivially converted into packrat parsers by memoing the recognition predicates of all the nonterminals, and that being selective about what to memo has its advantages. Section 4 gives some interesting performance results, comparing the Mercury approach with Robert Grimm's Rats! packrat parser generator. Section 5 concludes with a discussion of the relative merits of packrat parsers and plain DCG parsers.

2 Definite Clause Grammars

Logic programming languages such as Prolog and Mercury have built-in support for coding recursive descent parsers in the form of definite clause grammars (DCGs). The Mercury syntax of a DCG rule is

```
H --> B.
```

where `H` is the head of the rule (the non-terminal) and `B` is its body. The syntax for the body is

```
B  ::=  [x1, ..., xn]
    |   B1, B2
    |   ( if B1 then B2 else B3 )
    |   B1 ; B2
    |   { G }
```

Body terms match as follows: `[x1, ..., xn]` succeeds iff the next `n` items of input unify with `x1` to `xn` respectively (these items are consumed by the match); `B1, B2` matches `B1` followed by `B2`; `( if B1 then B2 else B3 )` matches either `B1, B2`, or just `B3` in the case that `B1` does not match at the current position; `B1 ; B2` matches either `B1` or `B2`; `{ G }` succeeds iff the ordinary Mercury goal `G` succeeds. A body item `not B` is syntactic sugar for `( if B then { false } else { true } )`, where the goal `true` always succeeds and the goal `false` always fails.

The compiler uses a simple source-to-source transformation to convert DCG rules into ordinary Mercury. Every DCG rule becomes a predicate with two extra arguments threaded through, corresponding respectively to the representation of the input remaining before and after matching. Figure 3 shows the transformation algorithm.

Transform `H --> B` into `H(S0, S) :- <<B, S0, S>>` where

```
<<[x1, ..., xn],   S0, S>> =  S0 = [x1, ..., xn | S]
<<(C1, C2),        S0, S>> =  some [S1] (<<C1, S0, S1>>,
                                         <<C2, S1, S >>)
<<( if C then T            =  some [S1] ( if   <<C, S0, S1>>
          else E ), S0, S>>                  then <<T, S1, S >>
                                             else <<E, S0, S >> )
<<(D1 ; D2),       S0, S>> =  (<<D1, S0, S>> ; <<D2, S0, S>>)
<<{ G }            S0, S>> =  G, S = S0
```

Fig. 3. *Transforming DCG rules into plain Mercury.* `S0`*,* `S1`*,* `S` *are input stream states.*

By way of an example, the following DCG rule `nat` matches natural numbers (as in Prolog, Mercury variable names start with an upper case letter, while predicate names and function symbols start with a lower case letter):

```
nat    --> digit, digits.
digit  --> [X], { char.is_digit(X) }.
digits --> ( if digit then digits else { true } ).
```

`char.is_digit` is a predicate in the Mercury standard library; it succeeds iff its argument is a character between '0' and '9'. Note that the last rule implements longest match.

DCGs are very flexible: rules can take arguments and compute results. The following version of `nat` returns the number matched as an integer:

```
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 } ).
```

Here, `digits` takes `M` (the numeric value of the digits read so far) as a parameter and computes `N` (the numeric value of the entire digit sequence matched by the nonterminal) as the result. `char.digit_to_int(X, D)` succeeds iff `X` is a digit, unifying `D` with its integer value.

Negation can allow for elegant disambiguation between rules:

```
integer(S, I) --> sign(S), nat(I), not frac_part(_F).
real(S, I, F) --> sign(S), nat(I), frac_part(F).
frac_part(F)  --> ['.'], digit(D), frac(100.0, float(D) / 10.0, F).
```

```
frac(M, F0, F) --> ( if   digit(D)
                     then frac(M * 10.0, F0 + float(D) / M, F)
                     else { F = F0 } ).
sign(S)        --> ( if ['-'] then { S = -1 } else { S = 1 } ).
```

The pattern `not frac_part(_F)` succeeds iff `frac_part(_F)` fails, hence the `integer` rule only matches natural numbers that are not followed by a fractional part. Without this negated pattern, `integer` would match the initial part of the lexeme of every `real` number.

Higher order DCG rules can be used to abstract out common grammatical patterns, such as these:

```
optional(P, X) -->
    ( if P(Y) then
        { X = yes(Y) }
    else
        { X = no }
    ).

zero_or_more(P, List) -->
    ( if P(Head) then
        zero_or_more(P, Tail),
        { List = [Head | Tail] }.
    else
        { List = [] }
    ).

one_or_more(P, [Head | Tail]) -->
    P(Head),
    zero_or_more(P, Tail).

comma_separated_list(P, [Head | Tail]) -->
    P(Head),
    zero_or_more(comma_followed_by(P), Tail).

comma_followed_by(P, X) -->
    [','],
    P(X).
```

(In each of these cases `P` must be a DCG rule computing a single result.)

Using these higher order rules is quite simple. For example, one can match a comma-separated list of natural numbers just by calling by calling `comma_separated_list(nat, Nats)`.

3 Using Memoing to Create Packrat Parsers

The Mercury compiler provides extensive support for several forms of tabled evaluation. Memoing is one of these forms. When a memoized

predicate is called, it looks up its call table to see whether it has been called with these arguments before. If not, it enters the input arguments in the call table, executes as usual, and then records the output arguments of each answer. On the other hand, if the predicate *has* been called with these input arguments before, it returns the answers from the call's answer table directly, *without* executing the predicate's code.

To memoize a Mercury predicate or function one need only add the appropriate pragma. For example:

```
:- pragma memo(nat/3, <attributes>).
:- pragma memo(integer/4, <attributes>).
:- pragma memo(real/5, <attributes>).
```

The first argument of the `memo` pragma is the *name/arity* pair identifying the predicate to be memoized. For DCG rules the arity component is that of the Mercury predicate resulting from the DCG transformation of figure 3, which adds two extra arguments to the DCG rule.

The second argument to the `memo` pragma is a list of attributes controlling the memoization transformation. Valid attributes are:

- `allow_reset` tells the compiler to generate a predicate that the user can call to clear the predicate's memo table.
- `statistics` tells the compiler to keep statistics about all the accesses to the predicate's memo table, and to generate a predicate that user code can call to retrieve these statistics.
- `specified([...])` tells the compiler how to construct the call table. The list should contain one element for each predicate argument. If the argument is an input argument, this element can be `value`, `addr` or `promise_implied`.
  - `value` tells the compiler to table the full value of the argument. If the term bound to this argument at runtime contains n function symbols, this will take $O(n)$ time and can create $O(n)$ new nodes in the call tree, so it can be slow.
  - `addr` tells the compiler to table only the address of the argument, which uses only constant time and space. The downside is that while equal addresses imply equal values, nonequal addresses do *not* imply unequal values. Therefore if any arguments are tabled by address, two calls in which the values of all input arguments are equal may nevertheless not be recognized as being the same call, leading to the unnecessary recomputation of some previously stored answers.
  - `promise_implied` asserts that the corresponding argument need not be stored or looked up in the call table, because the user

promises its value to be a function of the values of the other input arguments.

If the argument is an output argument, the corresponding element should be `output`.

– `fast_loose` tells the compiler to table all input arguments by address; it is equivalent to a longer `specified([...])` annotation.

For parsing applications, asking for all input arguments to be tabled by address is almost always optimal.

Memoizing all the rules in a DCG parser essentially converts a recursive descent parser into a packrat parser. The memoized predicates generated by the Mercury compiler employ hash tables with separate chaining. Tables start small, but are automatically expanded when their load factors exceed a threshold. They therefore have O(1) expected lookup times, the same order as packrat tables. Since hash tables are more complex than mere 2D arrays, their constant factor is higher, but in practice, the hash table approach may well be superior. This is because packrat tables will nearly always be sparsely populated, so the hash tables probably occupy less memory and are therefore likely to be more memory-system friendly — one would expect significantly less paging when parsing any sizeable amount of input. Of course, the packrat table can be compressed, as in [4], but that erodes, eliminates, or even reverses the constant factor advantage. Furthermore, it is a *specialized* optimization; with memoed DCGs, any optimization of the memo tables is of *general* use. The more general memoing mechanism has two further advantages: it works for parsing nondeterministic grammars, and, more importantly, it supports parameterized parsing rules, such as `comma_separated_list`.

4 Performance Evaluation

To see how memoized DCG parsers perform compared to packrat parsers, we need both kinds of parsers for the same language, preferably a language with lots of programs available as test data. We implemented a parser for the Java language to allow comparison with results published for other packrat parsers (e.g. [2], [3]).

4.1 Parser Structure

Our implementation is an almost direct transliteration of the grammar provided in Sun's Java Language Specification (Second Edition) which can be found on-line at `http:`

`//java.sun.com/docs/books/jls/second_edition/html/syntax.doc.html`. (Section 3 of that document specifies the lexical structure of identifiers, primitive literals and so on). We did the translation into Mercury done by hand (although doing so automatically would not have been hard were it not for the occasional error in Sun's published grammar, such as the handling of `instanceof` expressions). We did not take advantage of any opportunities for optimization.

We implemented a rule from Sun's grammar such as

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

as the Mercury DCG predicate

```
block_statement -->
    ( if local_variable_declaration_statement then
        []
    else if class_or_interface_declaration then
        []
    else
        optional(label), statement
    ).

label -->
    identifier,
    punct(":").
```

`[]` is the DCG equivalent of a no-op goal and the grammatical convenience `[Identifier :]` is replaced with a higher order call to one of the predicates we showed in section 2. The higher order argument of that call representing the identifier-colon pattern could have been an anonymous lambda-expression, but making it a separate named predicate (`label`) is more readable.

The reason why the predicate is only a recognizer (i.e. it doesn't return anything useful like a parse tree) is to remove any differences in the parse tree construction process as a source of unwanted variability in our comparison with the Rats! packrat parser generator. Since Rats! views nonterminals as being defined by an *ordered* list of productions, which stops looking for parses using later productions after finding a parse using an earlier production, this code also replaces the apparent nondeterminism of the original grammar with a deterministic if-then-else chain. In the absence of code to build parse trees, the identity of the production that matches a nonterminal like `block_statement` doesn't matter anyway. Adapting the implementation to also construct an abstract syntax

tree would simply require an extra parameter in each DCG predicate's argument list, together with some simple code in each production to compute its value. In fact, the definition of `optional` in section 2 assumes that we are building parse trees; if we are not, we need to use this simplified definition instead:

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

4.2 Experimental Setup

Our test load consisted of the 735 largest Java source files taken from a randomly chosen large Java program, the Batik SVG toolkit. (More source files would have exceeded the 20 Kbyte limit on the length of command line argument vectors.) The input files range in size from a few hundred lines to more than 10,000 lines; they total more than 900,000 lines and 9.6 Mbytes.

To evaluate packrat parsing, we used the `xtc` Java parser generated with the Rats! optimizing packrat parser generator (version 1.12.0, released on 18 July 2007). We took the grammar specification from the Rats! web site, so it should match the version used in Grimm's paper [4]. We ran the generated parser both with and without the Java optimization option (-Xms20m) recommended by Grimm, which starts the system with a 20 Mb memory allocation pool. The startup times were nontrivial either way, so figure 4 reports not just the time taken by each version to parse the test load, but also the time taken by each version on the null load (a single empty file), and the difference between them. The figures represent user time in seconds; they were obtained by averaging the times from 22 runs.

We also tried to test a packrat parser generated by Pappy, but we could not get it to work.

To evaluate DCG parsing and memoing, we wrote a script that could take a template Java parser written in Mercury and create several hundred different versions of it. These versions varied along the following dimensions.

Input representation. There are several possible ways to represent the input and the current position of the parser in it. We tested five of these.

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.

The `chars` approach increases the size of the input to be parsed eightfold (it uses one eight-byte cons cell per character), while the `single` approach requires allocating a three-word cell on the heap for every character match, so these should be slower than the other three.

All these alternatives assume that the entire input is available when parsing starts. For non-interactive applications, this is ok. Interactive applications can use other representations; they will probably be slower, but interactive applications typically don't care about that, since in such systems the user is usually by far the slowest component.

Mercury backend. The Mercury compiler can generate either low level C code that is effectively assembler [8] or high level C code that actually uses C constructs such as functions, local variables, while loops and so on [5].

Memoed predicates. The parser has 92 recognition predicates, and we could memo an arbitrary subset of these predicates. We ran tests with all recognition predicates memoed, with no predicates memoed, and 92 versions each with a single predicate memoed. Based on preliminary results, we also selected four versions with two or three interesting predicates memoed.

None of the Mercury versions had measurable startup times, so we don't report them. We also don't have room to report timing results for all versions of the Mercury parser, so figure 5 reports only a selection. (The full set of raw data are available from the Mercury web site, right next to this paper.) The figures represent user time in seconds; they were obtained by averaging the times from 22 runs. For each of the ten possible combinations of backend and input representation, we present times for three out of the 98 variations along the memoized predicates that we explored:

best: the best time from all the versions we tested;

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

Fig. 4. Times for Rats! packrat parser

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)

Fig. 5. Times for Mercury DCG parser

none: the time with no predicates memoed (equivalent to pure recursive descent parsing);

all: the time with all predicates memoed (equivalent to packrat parsing).

The `none` and `all` columns also contain the ratio between the time in that column and the best time, and its position in the list of all the 98 times along the "memoed predicates" dimension from best to worst.

Rats! emits parser code in Java whereas the Mercury compiler emits C code (compiled with `gcc`), which makes this aspect something of an apples to oranges performance comparison.

The Java compiler we used was build 2.3 of IBM's J9 suite (released April 2007). The Mercury compiler we used was version `rotd-2007-08-18`; the generated C code was compiled with gcc 3.4.4 (20050314). The test machine was a PC with a 2.4 GHz Pentium IV CPU with 512 Kb of cache, 512 Mb of main memory, running Linux kernel version 2.6.8-2.

4.3 Performance Analysis

There are two main kinds of observations we can make about the performance data in figures 4 and 5: comparisons between packrat parsing using Rats! and memoed DCG parsing using Mercury, and comparisons

among the various versions of memoed DCG parsing using Mercury. We start with the latter.

It is clear from figure 5 that memoing all recognition predicates is never a good idea. For every combination of the other dimensions (backend and input representation), memoing everything was *always* the worst possible choice in the "memoed predicates" dimension. The raw data also shows that it was always worst by a very wide margin. This effect is so strong that we would be very surprised if memoing everything turned out *not* to be the worst choice (from a similar range of possibilities) for any other grammar.

Figure 5 also shows that memoing nothing, i.e. using a plain recursive descent parser, is usually quite close to being the optimal choice. In several cases it is separated from the best choice by less time than the typical measurement error.

With the low level C backend and using "global" as the input representation, the speeds of the best 65 versions are all within 4% of each other. This shows that for most predicates, memoing just that predicate and nothing else is likely to have only an insignificant effect. again often within measurement error. Our raw data confirms that the trend also holds for the other nine rows of figure 5, though in those the clustering is slightly looser.

However, there are some predicates whose memoing leads to significant effects. For example, memoing the recognizer for the `punct` nonterminal (whose definition is `punct(Punct) --> match_string(Punct), whitespace`) always leads to significant slowdowns. In each row, it leads to one of the three worst times in that row, the only worse choices (amongst the ones we tested) being memoing everything and memoing `punct` *and* two other predicates. On the other hand, memoing the recognizer for the `modifiers_opt` nonterminal almost always lead to speedups; the version with only this predicate memoed was the fastest version for four of the ten rows, and was second, third and fifth fastest respectively in three other rows.

As it happens, the recognizer predicate for `punct` is called very frequently, but in more than 95% of cases it fails immediately, so a call to the recognizer typically does very little. On the other hand, `modifiers_opt` looks for zero or more occurrences of `modifier`, which requires looking for any one of eleven keywords, so even in the typical case where none of these is present, it requires a nontrivial amount of work to recognize this fact.

`punct` and `modifiers_opt` are also at opposite ends of the scale when it comes to the cost of memoing. `modifiers_opt` has no input apart from the current input position, and so (for every one of the input representations we tested) checking whether we have already looked for this nonterminal at this position requires only a single hash table lookup. [1] On the other hand, `punct` also has another input, the string to be matched. Computing the string's hash value requires scanning it, and comparing the probe string with a hash slot's occupant requires scanning it again (at least in the case of a hit), so the table lookup will take significantly more than twice as long for `punct` as for `modifiers_opt`. (Using the address of a string as its hash value could speed this up, but Mercury doesn't yet support tabling strings by their addresses; we are in the process of fixing this.)

These observations are a simple consequence of Amdahl's law [1]. The effect on performance of memoing a recognizer predicate depends on

1. the fraction of the runtime of the parser that the predicate accounts for,
2. the ratio of the time taken to execute the recognizer predicate's body compared to the time taken to perform the table lookup that could avoid that execution, and
3. the probability that the table lookup fails, so you have to execute the predicate body anyway.

The first point is the reason why memoing most predicates doesn't have a measurable impact. If a predicate accounts for only a small part of the execution time, memoing it can't have more than a small effect either, in which case not memoing it is better since it does not waste any space (in memory, or more importantly, *in the cache*) on the memo table. However, the key point is the second one: if the table lookup takes at least as long as executing the predicate body, then *memoing will yield a slowdown, not a speedup*, even if almost all lookups are hits. Pure recognizers are particularly vulnerable to this effect. Adding code to build ASTs and/or to evaluate semantic predicates to the bodies of nonterminals will reduce the relative if not the absolute costs of tabling.

Comparing the performance of the Rats!-generated packrat parser in figure 4 with the performance of the the all-memoed Mercury DCG parser

[1] With the pass1 and pass2 input representations, the extra arguments are always the same, so we specify `promise_implied` to ensure that these arguments are not memoed. This is OK, since we always reset all tables when switching from one file to another.

in figure 5 is very difficult because the difference between packrat parsing (including all the optimizations applied by Rats!) and memoed DCG parsing is confounded by other differences, chiefly in how the parsers' executables are generated (Rats! generates Java, whereas Mercury generates C). If Rats! is ever modified to generate C, or if the Mercury compiler's Java backend is ever completed, this confounding factor would be removed.

Grimm reports [4] that the set of 17 optimizations performed by Rats! (some of which rely on the presence of hand-written annotations) yield a cumulative speedup of a factor of 8.9. That was on a different machine and on a different test load, but by nevertheless applying that factor to the data in figure 4, we can estimate (*very* roughly) that a totally unoptimized packrat parser in Java would take 50 to 60 seconds on our load on our test machine ($6.4\text{s} * 8.9 = 56.96\text{s}$). At 16.58 seconds, even the slowest Mercury DCG parser is significantly faster than that. While some of this is almost certainly due to the Java vs C difference, part of it is probably due to differences in how the two systems manage their tables. Likewise, all ten versions of a plain Mercury DCG parser with no memoing are much faster than the fully optimized Rats! generated packrat parser.

The best parser generated by Rats! (the one with all optimizations applied, which avoids memoing many nonterminals) is a factor of 8.9 faster than the worst Rats! parser (the one with no optimizations applied, which memoes all nonterminals). In most rows of figure 5, the difference between the best and worst versions is smaller than that. That tells us that Rats! is tapping some sources of speedup that we don't. This is not surprising, given that section 8 of [4] gives a long list of optimizations that we haven't even tried to apply, even though they would be worthwhile. However, the fact that we get speedups in the factor of 3 to factor of 8 range by simply not memoing anything, and some slight speedups beyond that by carefully selecting one or two predicates to memo, shows that with this single optimization (memoing almost nothing) we are also tapping an important source of speedup that Rats! doesn't.

Rats! actually has an optimization that reduces the set of memoed nonterminals, and figure 4 of [4] shows that of the 17 optimizations applied by Rats, this one gets one of the two biggest speedups (almost a factor of 2). However, our data shows that memoing even fewer nonterminals can yield even bigger speedups.

We think the Rats! technique of starting off by memoing all nonterminals, and then applying heuristics to choose some nonterminals to *not*

be memoed, is approaching the problem from the wrong end. We think the right approach is to start by memoing nothing, and then applying heuristics to choose some nonterminals to *be* memoed. Rats! uses heuristics based on properties of the grammar. We think that while these have their place, it is much more important to pay attention to Amdahl's law and memo a nonterminal only if the expected speedup due to this step is (a) positive and (b) nontrivial, i.e. likely to be measurable.

The best way to estimate the expected speedup is via feedback from a profiler that can record the program's overall execution time, the average time to execute each nonterminal's predicate, the average time to table the arguments of that predicate, and the average hit rate of each table. In the absence of such feedback, the system can either try to estimate that information from the structure of the grammar (this approach has had some success in other contexts, e.g. granularity analysis), or ask the programmer to annotate the predicates that should be tabled. Running a few experiments with different sets of annotations doesn't take very long (figure 5 is based on several thousand such experiments, all driven by a single script), and in fact may be *less* work than annotating all the "transient" nonterminals in a Rats! parser specification.

Of course, adopting no memoing as the default approach means abandoning the linear time guarantee of packrat parsing; with few or no nonterminals memoed, large chunks of the input may in theory be scanned an exponential number of times. However, we don't think this is a problem. First, as many others have noted [7], exponential behavior just doesn't seem to happen in practice anyway. Second, the linear time guarantee always had problems. Tabling everything consumes main memory at a high rate, and so risks starting thrashing, thus dropping the program from DRAM speed to disk speed. While a theoretician may say the performance is still linear, that won't prevent complaints from users. The fact that many languages nowadays (including Java and Mercury) include a garbage collector (which must scan the tables at least once in a while, but won't be able to recover memory from them) just makes this even worse. Given these facts, we think that in the absence of a genuine need for a guaranteed linear upper bound, focusing just on the expected case makes much more sense.

5 Discussion and Conclusion

The PEGs (parsing expression grammars) that underlie packrat parsers and DCGs (definite clause grammars) have very similar expressive power.

While PEGs usually require support from a specialized tool, DCGs can be implemented directly in a general purpose programming language. This brings some advantages, for example the ability to use higher order code to abstract away common patterns and the ability to use standard tools such as debuggers and profilers. In this paper, we have shown another of these advantages, which is that if the host language has support for memoization, then *any* DCG parser can be turned into a packrat parser simply by memoizing the predicates implementing the production rules. This is by far the simplest way to construct a packrat parser: it uses a general purpose language feature, it handles the arguments representing the current offset in the input the same way as any other arguments (such as those representing a symbol table), and doesn't even require any new implementation effort.

However, while it is trivial to turn any DCG parser into a packrat parser, our data shows that this is almost always a performance loss, not a win. Our data shows that not memoing any predicates is consistently *much* faster than memoing *all* predicates, and that memoing nothing is in fact usually pretty close to optimal. While generalizing from a sample of one (the Java grammar) is always dangerous, we believe this result is very likely to hold for the grammar of any programming language, since these tend to have only relatively few ambiguous components. (Grammars for natural languages can be expected to have much more pervasive ambiguity.) Most predicates don't contribute significantly to the parser's runtime, so tabling them just adds overhead in both space and time. For memoing to yield a benefit, the memoed predicate must contribute significantly to the runtime of the parser, and the average running time of one of its invocations multiplied by the hit rate of the table (the expected savings), must exceed the time taken by the tabling operations themselves (the cost). We propose that this be the chief consideration in deciding what predicates to memo in a recursive descent parser. This consideration is so important that respecting it, and tabling only a minimal set of predicates (usually only one, sometimes none) leads to a parser that is significantly faster than the one generated by Rats!, even though the Rats! applies a whole host of other optimizations we don't.

The best of both worlds would be a system that respected Amdahl's law in choosing what to memo but also applied all the other optimizations applied by Rats!. Some of them (e.g. factoring out common prefixes that occur in the conditions of a chain of if-then-elses) are generic enough that it makes sense to add them to the implementations of general purpose programming languages such as Mercury. Some (e.g. turning the code

that recognizes `public | private | protected` from a chain of if-then-elses into a decision tree) are specific to parsers, and thus are appropriate only for a parser generator. The same is true for Rats!'s support for left-recursive productions.

References

1. Gene Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the AFIPS Spring Join Computer Conference*, pages 483–485, Atlantic City, New Jersey, 1967.
2. B. Ford. Packrat parsing: Simple, powerful, lazy, linear time. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, pages 36–47, Pittsburgh, Pennsylvania, 2002.
3. R. Grimm. Practical packrat parsing. *New York University Technical Report, Dept. of Computer Science*, TR2004-854, 2004.
4. R. Grimm. Better extensibility through modular syntax. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pages 38–51, Ottawa, Canada, 2006.
5. Fergus Henderson and Zoltan Somogyi. Compiling Mercury to high-level C code. In Nigel Horspool, editor, *Proceedings of the 2002 International Conference on Compiler Construction*, Grenoble, France, April 2002. Springer-Verlag.
6. F. Perreira and D. Warren. Definite clause grammars for language analysis — a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13:231–278, 1980.
7. R. Redziejowski. Parsing expression grammar as a primitive recursive-descent parser with backtracking. *Fundamenta Informaticae*, 79 (1-4):513–524, 2007.
8. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29 (1-3):17–64, 1996.