Packrat Parsers Can Support Left Recursion [*]

Alessandro Warth

University of California, Los Angeles
and Viewpoints Research Institute
awarth@cs.ucla.edu

James R. Douglass

The Boeing Company
jamie.douglass@boeing.com

Todd Millstein

University of California, Los Angeles
todd@cs.ucla.edu

Abstract

Packrat parsing offers several advantages over other parsing techniques, such as the guarantee of linear parse times while supporting backtracking and unlimited look-ahead. Unfortunately, the limited support for left recursion in packrat parser implementations makes them difficult to use for a large class of grammars (Java's, for example). This paper presents a modification to the memoization mechanism used by packrat parser implementations that makes it possible for them to support (even indirectly or mutually) left-recursive rules. While it is possible for a packrat parser with our modification to yield super-linear parse times for some left-recursive grammars, our experiments show that this is not the case for typical uses of left recursion.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Parsing

General Terms Languages, Algorithms, Design, Performance

Keywords packrat parsing, left recursion

1. Introduction

Packrat parsers [2] are an attractive choice for programming language implementers because:

- They provide "the power and flexibility of backtracking and unlimited look-ahead, but nevertheless [guarantee] linear parse times." [2]

- They support syntactic and semantic predicates.

- They are easy to understand: because packrat parsers only support *ordered choice*—as opposed to *unordered choice*, as found in Context-Free Grammars (CFGs)—there are no ambiguities and no shift-reduce/reduce-reduce conflicts, which can be difficult to resolve.

- They impose no separation between lexical analysis and parsing. This feature, sometimes referred to as *scannerless parsing* [10], eliminates the need for *moded lexers* [9] when combining grammars (e.g., in Domain-Specific Embedded Language (DSEL) implementations).

Unfortunately, "like other recursive descent parsers, packrat parsers cannot support left-recursion" [6], which is typically used to express the syntax of left-associative operators. To better understand this limitation, consider the following rule for parsing expressions:

```
expr ::= <expr> "-" <num> / <num>
```

Note that the first alternative in expr begins with expr itself. Because the choice operator in packrat parsers (denoted here by "/") tries each alternative in order, this recursion will never terminate: an application of expr will result in another application of expr without consuming any input, which in turn will result in yet another application of expr, and so on. The second choice—the non-left-recursive case—will never be used.

We could change the order of the choices in expr,

```
expr ::= <num> / <expr> "-" <num>
```

but to no avail. Since all valid expressions begin with a number, the second choice—the left-recursive case—would never be used. For example, applying the expr rule to the input "1-2" would succeed after consuming only the "1", and leave the rest of the input, "-2", unprocessed.

Some packrat parser implementations, including *Pappy* [1] and *Rats!* [6], circumvent this limitation by automatically transforming *directly left-recursive* rules into equivalent non-left-recursive rules. This technique is called *left recursion elimination*. As an example, the left-recursive rule above can be transformed to

```
expr ::= <num> ("-" <num>)*
```

which is not left-recursive and therefore can be handled correctly by a packrat parser. Note that the transformation shown here is overly simplistic; a suitable transformation must preserve the left-associativity of the parse trees generated by the resulting non-left-recursive rule, as well as the meaning of the original rule's *semantic actions*.

Now consider the following minor modification to the original grammar, which has no effect on the language accepted by expr:

```
x    ::= <expr>
expr ::= <x> "-" <num> / <num>
```

When given this grammar, the *Pappy* packrat parser generator [1] reports the following error message:

```
Illegal left recursion: x -> expr -> x
```

This happens because expr is now *indirectly* left-recursive, and

---

Pappy does not support indirect left recursion (also referred to as mutual left recursion). In fact, to the best of our knowledge, none of the currently-available packrat parser implementations supports indirectly left-recursive rules.

Although this example is certainly contrived, indirect left recursion does in fact arise in real-world grammars. For instance, Roman Redziejowski [8] discusses the difficulty of implementing a packrat parser for Java [5], whose *Primary* rule (for expressions) is indirectly left-recursive with five other rules. While programmers can always refactor grammars manually in order to eliminate indirect left recursion, doing so is tedious and error-prone, and in the end it is generally difficult to be convinced that the resulting grammar is equivalent to the original.

This paper presents a modification to the memoization mechanism used by packrat parser implementations that enables them to support both direct and indirect left recursion directly (i.e., without first having to transform rules). While it is possible for a packrat parser with our modification to yield super-linear parse times for some left-recursive grammars, our experiments (Section 5) show that this is not the case for typical uses of left recursion.

The rest of this paper is structured as follows. Section 2 gives a brief overview of packrat parsing. Section 3 describes our modification to the memoization mechanism, first showing how direct left recursion can be supported, and then extending the approach to support indirect left recursion. Section 4 validates this work by showing that it enables packrat parsers to support a grammar that closely mirrors Java's heavily left-recursive *Primary* rule. Section 5 discusses the effects of our modification on parse times. Section 6 discusses related work, and Section 7 concludes.

2. An Overview of Packrat Parsing

Packrat parsers are able to guarantee linear parse times while supporting backtracking and unlimited look-ahead "by saving all intermediate parsing results as they are computed and ensuring that no result is evaluated more than once" [2]. For example, consider what happens when the rule

```
expr ::= <num> "+" <num>
       / <num> "-" <num>
```

(where num matches a sequence of digits) is applied to the input "1234-5".

Since choices are always evaluated in order, our parser begins by trying to match the input with the pattern

```
<num> "+" <num>
```

The first term in this pattern, `<num>`, successfully matches the first four characters of the input stream ("1234"). Next, the parser attempts to match the next character on the input stream, "-", with the next term in the pattern, "+". This match fails, and thus we backtrack to the position at which the previous choice started (0) and try the second alternative:

```
<num> "-" <num>
```

At this point, a conventional top-down backtracking parser would have to apply num to the input, just like we did while evaluating the first alternative. However, because packrat parsers *memoize* all intermediate results, no work is required this time around: our parser already knows that num succeeds at position 0, consuming the first four characters. It suffices to update the current position to 4 and carry on evaluating the remaining terms. The next pattern, "-", successfully matches the next character, and thus the current position is incremented to 5. Finally, `<num>` matches and consumes the "5", and the parse succeeds.

Intermediate parsing results are stored in the parser's memo table, which we shall model as a function

APPLY-RULE(R,P)
 let $m = \text{MEMO}(R,P)$
 if $m = \text{NIL}$
  then let $ans = \text{EVAL}(R.body)$
   $m \leftarrow$ **new** MEMOENTRY(ans, Pos)
   $\text{MEMO}(R,P) \leftarrow m$
   return ans
  else $Pos \leftarrow m.pos$
   return $m.ans$

Figure 1. The original APPLY-RULE procedure.

MEMO $:$ (RULE, POS) $\rightarrow$ MEMOENTRY

where

 MEMOENTRY $:$ ($ans :$ AST, $pos :$ POS)

In other words, MEMO maps a rule-position pair (R,P) to a tuple consisting of

- the AST (or the special value FAIL[1]) resulting from applying R at position P, and

- the position of the next character on the input stream.

or NIL, if there is no entry in the memo table for the given rule-position pair.

The APPLY-RULE procedure (see Figure 1), used in every rule application, ensures that no rule is ever evaluated more than once at a given position. When rule R is applied at position P, APPLY-RULE consults the memo table. If the memo table indicates that R was previously applied at P, the appropriate parse tree node is returned, and the parser's current position is updated accordingly. Otherwise, APPLY-RULE evaluates the rule, stores the result in the memo table, and returns the corresponding parse tree node.

By using the memo table as shown in this section, packrat parsers are able to support backtracking and unlimited look-ahead while guaranteeing linear parse times. In the next section, we present modifications to the memo table and the APPLY-RULE procedure that make it possible for packrat parsers to support left recursion.

3. Adding Support for Left Recursion

In Section 1, we showed informally that the original version of the expr rule,

```
expr ::= <expr> "-" <num> / <num>
```

causes packrat parsers to go into infinite recursion. We now revisit the same example, this time from Section 2's more detailed point of view.

Consider what happens when expr is applied to the input "1-2-3". Since the parser's current position is initially 0, this application is encoded as APPLY-RULE(expr, 0). APPLY-RULE, shown in Figure 1, begins by searching the parser's memo table for the result of expr at position 0. The memo table is initially empty, and thus MEMO(expr, 0) evaluates to NIL, indicating that expr has not yet been used at position 0. This leads APPLY-RULE to evaluate the body of the expr rule, which is made up of two choices. The first choice begins with `<expr>`, which, since the parser's current position is still 0, is encoded as the familiar APPLY-RULE(expr, 0). At this point, the memo table remains unchanged and thus we are back

---

[1] Failures are also memoized in order to avoid doing unnecessary work when backtracking occurs.

```
APPLY-RULE(R,P)
    let m = MEMO(R,P)
    if m = NIL
        then m ← new MEMOENTRY(FAIL,P)          *
              MEMO(R,P) ← m                      *
              let ans = EVAL(R.body)
              m.ans ← ans                        *
              m.pos ← Pos                        *
              return ans
        else Pos ← m.pos
              return m.ans
```

Figure 2. Avoiding non-termination by making left-recursive applications fail. (Lines marked with * are either new or have changed since the previous version.)

exactly where we started! The parser is doomed to repeat the same steps forever, or more precisely, until the computer eventually runs out of stack space.

The rest of this section presents a solution to this problem. First, we modify the algorithm to make left-recursive applications fail, in order to avoid infinite loops. We then build on this extension to properly support direct left recursion. Extending this idea to support indirect left recursion is conceptually straightforward; we present the intuition for this in Section 3.3. Finally, Section 3.4 focuses on the operational details of this extension.

3.1 Avoiding Infinite Recursion in Left-Recursive Rules

A simple way to avoid infinite recursion is for APPLY-RULE to store a result of FAIL in the memo table *before* it evaluates the body of a rule, as shown in Figure 2. This has the effect of making all left-recursive applications (both direct and indirect) fail.

Consider what happens when expr is applied to the input "1-2-3" using the new version of APPLY-RULE. Once again this application is encoded as APPLY-RULE(expr,0). APPLY-RULE first updates the memo table with a result of FAIL for expr at position 0, then goes on to evaluate the rule's body, starting with its first choice. The first choice begins with an application of expr, which, because the current position is still 0, is also encoded as APPLY-RULE(expr,0). This time, however, APPLY-RULE *will* find a result in the memo table, and thus will not evaluate the body of the rule. And because that result is FAIL, the current choice will be aborted. The parser will then move on to the second choice, <num>, which will succeed after consuming the "1", and leave the rest of the input, "-2-3", unprocessed.

3.2 Supporting Direct Left Recursion

While this is clearly not expr's intended behavior, the modified APPLY-RULE procedure shown in Figure 2 was a step in the right direction. Consider the side-effects of our application of *expr* at position 0:

1. The parser's current position was updated to 1, and

2. The parser's memo table was updated with a mapping from (expr, 0) to (expr→num→1, 1).

The parse shown above avoided all left-recursive terms; we call it the *seed parse*.

Now, suppose we backtrack to position 0 and evaluate expr's body one more time. Note that unlike evaluating APPLY-RULE(expr,0), which would simply retrieve the previous result stored in the memo table, evaluating the *body* of this rule (which we denote in pseudo-code as EVAL(*expr.body*)) will

```
GROW-LR(R,P,M,H)
    ...                                          ▷ line A
    while TRUE
        do
            Pos ← P
            ...                                  ▷ line B
            let ans = EVAL(R.body)
            if ans = FAIL or Pos ≤ M.pos
                then break
            M.ans ← ans
            M.pos ← Pos
        ...                                      ▷ line C
    Pos ← M.pos
    return M.ans
```

Figure 3. GROW-LR: support for direct left recursion.

sidestep one level of memoization and begin to evaluate each of its choices. The first choice,

```
<expr> "-" <num>
```

begins with a left-recursive application, just like before. This time, however, that application succeeds because the memo table now contains the seed parse. Next, the terms "-" and <num> successfully match and consume the "–" and "2" on the input, respectively. If we update the memo table with the new answer and repeat these steps one more time, we will have parsed "1-2-3", the entire input stream!

We refer to this iterative process as *growing the seed*; Figure 3 shows GROW-LR, which implements the seed-growing algorithm. GROW-LR tries to grow the parse of rule R at position P, given the seed parse in the MEMOENTRY M.[2] Note that each time the rule's body is evaluated, the parser must backtrack to P; this is accomplished with the statement "$Pos ← P$". At the start of each iteration, M contains the last successful result of the left recursion. The loop's terminating condition, "*ans* = FAIL or $Pos ≤ M.pos$", detects that no progress was made as a result of evaluating the rule's body. Once this condition is satisfied, the parser's current position is updated to the one associated with the last successful result.

GROW-LR can be used to compute the result of a left-recursive application. Before we can use it, however, we must be able to detect when a left recursion has occurred. We do this by introducing a new data type, LR, and modifying MEMOENTRY so that LRs may be stored in *ans*,

    LR: (detected: BOOLEAN)
    MEMOENTRY : (*ans* : AST or LR, *pos* : POSITION)

and modifying APPLY-RULE as shown in Figure 4.

To detect left-recursive applications, APPLY-RULE memoizes an LR with *detected* = FALSE before evaluating the body of the rule. A left-recursive application of the same rule will cause its associated LR's *detected* field to be set to TRUE, and yield a result of FAIL. When an application is found to be left-recursive and it has a successful seed parse, GROW-LR is invoked in order to grow the seed into the rule's final result.

The modifications shown in Figures 3 and 4 enable packrat parsers to support direct left recursion without the need for left recursion elimination transformations. This includes nested direct left recursion, such as

---

[2] Lines A, B, and C, and the argument H can be ignored at this point.

```
APPLY-RULE(R, P)
    let m = MEMO(R, P)
    if m = NIL
        then let lr = new LR(FALSE)                    *
            m ← new MEMOENTRY(lr, P)                    *
            MEMO(R, P) ← m
            let ans = EVAL(R.body)
            m.ans ← ans
            m.pos ← Pos
            if lr.detected and ans ≠ FAIL              *
                then return GROW-LR(R, P, m, NIL)       *
                else return ans                        *
        else Pos ← m.pos
            if m.ans is LR                             *
                then m.ans.detected ← TRUE             *
                    return FAIL                        *
                else return m.ans                      *
```

Figure 4. Detecting left recursion and growing the seed with GROW-LR. (Lines marked with * are either new or have changed since the previous version.)

```
term ::= <term> "+" <fact>
       / <term> "-" <fact>
       / <fact>
fact ::= <fact> "*" <num>
       / <fact> "/" <num>
       / <num>
```

In the remainder of this section, we will present additional modifications that will enable our parser to also support indirect left recursion.

3.3 Getting Ready For Indirect Left Recursion

Recall the following grammar, taken from the introduction,

```
x    ::= <expr>
expr ::= <x> "-" <num> / <num>
```

and consider what happens when x is applied to the input "4-3" using the new version of APPLY-RULE given in the previous section. First, the x rule is detected to be left-recursive with a seed parse of x→expr→num→4. GROW-LR then evaluates x's body once again to try to grow the seed. At this point, the memo table already has an answer for expr, namely expr→num→4; this causes the second evaluation of x to yield a parse identical to the seed parse. Because the last evaluation of x consumed no more input than the seed parse, the loop in GROW-LR terminates and the seed parse becomes the final result. This is clearly not the behavior we wanted.

The example above shows that the modifications for supporting left recursion described in the previous section are overly simplistic. GROW-LR repeatedly evaluates *a single rule* in order to grow the seed parse, which is not sufficient when more than one rule is involved in a left recursion.

We shall now introduce a few concepts that will play a key role in the next and final set of modifications to our parser. The first of these concepts is that of a **rule invocation stack**. Before a rule is evaluated, it is pushed onto the parser's rule invocation stack, only to be popped off the stack once it has finished computing a result. In Figure 5, (A) depicts the rule invocation stack just after the x rule invokes expr.

An invocation of rule R is left-recursive if R is already on the rule invocation stack, and the parser's position has not changed since that first invocation. In the example above, the invocation of x
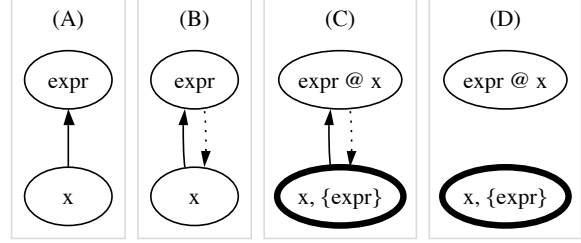
Figure 5. The rule invocation stack, shown at various stages during a left-recursive application.

by expr, shown in (B), is left-recursive. Left-recursive invocations form a loop in the rule invocation stack. We refer to the rule that started that loop as the **head rule** of the left recursion, and to the other rules in the loop as being **involved** in that left recursion. In (C), x's thicker border denotes that it is the head of a left recursion, and {expr}, inside the x node, represents the set of rules involved in that left recursion. The expr node, now labeled "expr @ x", indicates that expr is involved in a left recursion whose head rule is x.

We can use this information to handle the invocation of expr specially while growing x's seed parse. More specifically, we can force expr's body to be re-evaluated, ignoring the rule's previously memoized result. In general, when growing a left recursion result, we bypass the memo table and re-evaluate the body of any rule involved in the left recursion. This is the intuition for the final set of modifications to our parser, which are presented in the next section.

3.4 Adding Support for Indirect Left Recursion

The final version of the APPLY-RULE procedure is shown in Figure 6. It has been modified in order to maintain a rule invocation stack as described above. The stack is represented by the LR data type, which we have modified to look as follows:

LR : (*seed* : AST, *rule* : RULE, *head* : HEAD, *next* : LR)

The rule invocation stack is kept in the global variable *LRStack*, of type LR, and is represented as a linked list, using LR's *next* field.

LR's *seed* field holds the initial parse found for the associated rule, which is stored in the *rule* field. In place of LR's old *detected* field, we now have the *head* field which, for a left-recursive invocation, holds information pertinent to the left recursion (*head* is set to NIL for non-left-recursive invocations). The HEAD data type,

HEAD : (*rule* : RULE, *involvedSet*, *evalSet* : SET of RULE)

contains the head rule of the left recursion (*rule*), and the following two sets of rules:

- *involvedSet*, for the rules involved in the left recursion, and
- *evalSet*, which holds the subset of the involved rules that may still be evaluated during the current growth cycle.

These data structures are used to represent the information depicted in Figure 5.

Our parser must be able to determine whether left recursion growth is in progress, and if so, which head rule is being grown. Because only one left recursion can be grown at a time for any given position, a global variable, HEADS, is used to map a position to the HEAD of the left recursion which is currently being grown:

HEADS : POSITION → HEAD

HEADS is NIL at any position where left recursion growth is not underway.

The task of examining the rule invocation stack to find the head rule and its involved rule set is performed by the SETUP-LR procedure, shown in Figure 7.[3] In our example, SETUP-LR is invoked when the stack is at stage (B), in Figure 5, and leaves the stack as shown in stage (C).

GROW-LR, shown in Figure 3, must be modified to use the head rule and its involved rule set. Left recursion growth starts by changing Line A to

$$\text{HEADS}(P) \leftarrow H$$

which indicates that left recursion growth is in progress. For each cycle of growth, the involved rules are given a fresh opportunity for evaluation. This is implemented by changing Line B to

$$H.evalSet \leftarrow \text{COPY}(H.involvedSet)$$

In our example, this will cause `expr` to be included in the set of rules which will be re-evaluated if reached. Finally, when left recursion growth is completed, the head at the left recursion position must be removed. To accomplish this, Line C is changed to

$$\text{HEADS}(P) \leftarrow \text{NIL}$$

When a rule is applied, APPLY-RULE now invokes the RECALL procedure, shown in Figure 9, in order to retrieve previous parse results. In addition to fetching memoized results from the memo table, RECALL ensures that involved rules are evaluated during the growth phase. RECALL also prevents rules that were not previously evaluated as part of the left recursion seed construction from being parsed during the growth phase. This preserves the behavior that is expected of a Parsing Expression Grammar (PEG), namely that the first successful parse becomes the result of a rule.

Note that APPLY-RULE now invokes LR-ANSWER (see Figure 8), not GROW-LR, when it detects left recursion. If the current rule is the head of the left recursion, LR-ANSWER invokes GROW-LR just as we did before. Otherwise, the current rule is involved in the left recursion and must defer to the head rule to grow any left recursive parse, and pass its current parse to participate in the construction of a seed parse.

With these modifications, our parser supports both direct and indirect left recursion.

4. Case Study: Parsing Java's *Primary* Expressions

To validate our mechanism for supporting left recursion, we modified two existing packrat parser implementations,

- *Context-free Attributed Transformations* (*CAT*), a parser generator that supports PEGs as well as CFGs, and

- the back-end of the parsing and pattern-matching language *OMeta* [11]

to use the new APPLY-RULE procedure described in the previous section. We also constructed a grammar that closely mirrors Java's *Primary* rule, as found in chapter 15 of the Java Language Specification [5]. Because of its heavily mutually left-recursive nature, *Primary* cannot be supported directly by conventional packrat parsers [8].

Our process for composing this grammar, shown in Figure 10, started with a careful examination of the grammar of Java expressions. We then identified all other rules that are mutually left-recursive with *Primary*. All such rules, namely

---

[3] There are more efficient ways to implement SETUP-LR which avoid walking the stack; we chose to include this one because it is easier to understand.

```
APPLY-RULE(R, P)
    let m = RECALL(R, P)                              *
    if m = NIL
        then ▷ Create a new LR and push it onto the rule
             ▷ invocation stack.
             let lr = new LR(FAIL, R, NIL, LRStack)   *
             LRStack ← lr                             *
             ▷ Memoize lr, then evaluate R.
             m ← new MEMOENTRY(lr, P)
             MEMO(R, P) ← m
             let ans = EVAL(R.body)
             ▷ Pop lr off the rule invocation stack.
             LRStack ← LRStack.next                   *
             m.pos ← Pos
             if lr.head ≠ NIL                         *
                 then lr.seed ← ans                   *
                      return LR-ANSWER(R, P, m)       *
                 else m.ans ← ans                     *
                      return ans
        else Pos ← m.pos
             if m.ans is LR
                 then SETUP-LR(R, m.ans)              *
                      return m.ans.seed               *
                 else return m.ans
```

Figure 6. The final version of APPLY-RULE. (Lines marked with * are either new or have changed since the previous version.)

```
SETUP-LR(R, L)
    if L.head = NIL
        then L.head ← new HEAD(R, {}, {})
    let s = LRStack
    while s.head ≠ L.head
        do s.head ← L.head
           L.head.involvedSet ← L.head.involvedSet ∪ {s.rule}
           s ← s.next
```

Figure 7. The SETUP-LR procedure.

```
LR-ANSWER(R, P, M)
    let h = M.ans.head
    if h.rule ≠ R
        then return M.ans.seed
        else M.ans ← M.ans.seed
             if M.ans = FAIL
                 then return FAIL
                 else return GROW-LR(R, P, M, h)
```

Figure 8. The LR-ANSWER procedure.

```
RECALL(R, P)
    let m = MEMO(R, P)
    let h = HEADS(P)
    ▷ If not growing a seed parse, just return what is stored
    ▷ in the memo table.
    if h = NIL
        then return m
    ▷ Do not evaluate any rule that is not involved in this
    ▷ left recursion.
    if m = NIL and R ∉ {h.head} ∪ h.involvedSet
        then return new MEMOENTRY(FAIL, P)
    ▷ Allow involved rules to be evaluated, but only once,
    ▷ during a seed-growing iteration.
    if R ∈ h.evalSet
        then h.evalSet ← h.evalSet \ {R}
            let ans = EVAL(R.body)
            m.ans ← ans
            m.pos ← Pos
    return m
```

Figure 9. The RECALL procedure.

- *Primary*,
- *PrimaryNoNewArray*,
- *ClassInstanceCreationExpression*,
- *MethodInvocation*,
- *FieldAccess*, and
- *ArrayAccess*

are included in our grammar, as are "stubs" for the other rules they reference (*ClassName*, *InterfaceTypeName*, *Identifier*, *Method-Name*, *ExpressionName*, and *Expression*).

Next, we removed some of the uninteresting (i.e., non-left-recursive) choices from these rules, and ordered the remaining choices so that the rules would behave correctly when used in a packrat parser. For example, since the method invocation expression "this.m()" has a prefix of "this.m", which is also a valid field access expression, the *PrimaryNoNewArray* rule must try *MethodInvocation* before trying *FieldAccess*.

We then encoded the resulting grammar in the syntax accepted by the *Pappy* packrat parser generator [1]. Just as we expected, *Pappy* was unable to compile this grammar and displayed the error message "Illegal left recursion: Primary -> PrimaryNoNewArray -> ClassInstanceCreationExpression -> Primary".

Lastly, we encoded the same grammar in the syntax accepted by *CAT* and *OMeta*. The resulting parsers exhibit the correct behavior, as shown in Table 1.

5. Performance

A packrat parser's guarantee of linear parse times is based on its ability to compute the result of any single rule application in constant time. Our iterative process for growing the seed parse of a left-recursive application violates this assumption, thus making it possible for some left-recursive grammars to yield super-linear parse times. As an example, the grammar

```
start ::= <ones> "2" / "1" <start> / ε
ones  ::= <ones> "1" / "1"
```

accepts strings of zero or more "1"s in $O(n^2)$ time. The same inefficiency will arise for any grammar that causes the parser to
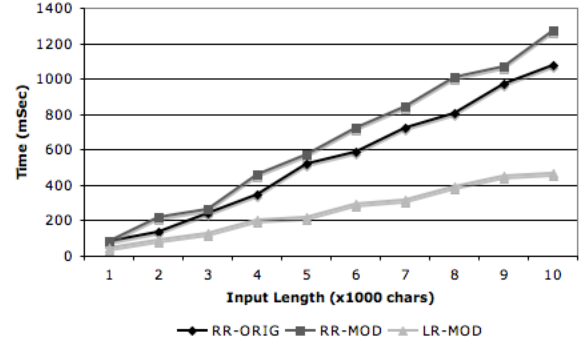

Figure 11. *RR-ORIG* shows the performance characteristics of rr in a traditional packrat parser implementation. *RR-MOD* and *LR-MOD* show the performance characteristics of rr and lr, respectively, in an implementation that was modified as described in Section 3.

backtrack to the middle of a previously computed left-recursive parse and then re-apply the same left-recursive rule. Fortunately, this problem—which is analogous to that of Ford's iterative combinators—does not manifest itself in practical grammars [1].

In order to gauge the expected performance of packrat parsers modified as described in this paper, we constructed the following two rules:

```
rr ::= "1" <rr> / "1"
lr ::= <lr> "1" / "1"
```

The first rule, rr, is right-recursive, and the second, lr, is left-recursive. Both recognize the same language, i.e., a string of one or more "1"s, and while the parse trees generated by these rules have different associativities, they have the same size.

We used these rules to recognize strings with lengths ranging from 1,000 to 10,000, in increments of 1,000. The results of this experiment are shown in Figure 11. The rr rule was first timed using a "vanilla" packrat parser implementation (*RR-ORIG*), and then using a version of the same implementation that was modified as described in Section 3 (*RR-MOD*). The lr rule was only timed using the modified implementation (*LR-MOD*), since left recursion is not supported in our "vanilla" implementation.

The following conclusions can be drawn from this experiment:

- **Our modifications to support left recursion do not introduce significant overhead for non-left-recursive rules.** Although the recognizing times for *RR-MOD* were consistently slower than those for *RR-ORIG*, the difference was rather small. Furthermore, *RR-MOD* and *RR-ORIG* appear to have the same slope.

- **The modified packrat parser implementation supports typical uses of left recursion in linear time**, as shown by *LR-MOD*.

- **Using left recursion can actually improve parse times.** The results of *LR-MOD* were consistently better than those of *RR-MOD* and *RR-ORIG*. More importantly, *LR-MOD*'s gentler slope tells us that it will scale much better than the others on larger input strings. This difference in performance is likely due to the fact that left recursion uses only a constant amount of stack space, while right recursion causes the stack to grow linearly with the size of the input.

In order to measure the effect of indirect left recursion on parse times, we constructed three more versions of the lr rule. The first,

Input String	Parse Tree (in s-expression form)
"this"	`this`
"this.x"	`(field-access this x)`
"this.x.y"	`(field-access (field-access this x) y)`
"this.x.m()"	`(method-invocation (field-access this x) m)`
"x[i][j].y"	`(field-access (array-access (array-access x i) j) y)`

Table 1. Some Java *Primary* expressions and their corresponding parse trees, as generated by a packrat parser modified as proposed in Section 3. The head of an s-expression denotes the type of the AST node.

Primary	::=	\<PrimaryNoNewArray\>
PrimaryNoNewArray	::=	\<ClassInstanceCreationExpression\>
	/	\<MethodInvocation\>
	/	\<FieldAccess\>
	/	\<ArrayAccess\>
	/	`this`
ClassInstanceCreationExpression	::=	`new` \<ClassOrInterfaceType\> `()`
	/	\<Primary\> `.` `new` \<Identifier\> `()`
MethodInvocation	::=	\<Primary\> `.` \<Identifier\> `()`
	/	\<MethodName\> `()`
FieldAccess	::=	\<Primary\> `.` \<Identifier\>
	/	`super .` \<Identifier\>
ArrayAccess	::=	\<Primary\> `[` \<Expression\> `]`
	/	\<ExpressionName\> `[` \<Expression\> `]`
ClassOrInterfaceType	::=	\<ClassName\> / \<InterfaceTypeName\>
ClassName	::=	`C / D`
InterfaceTypeName	::=	`I / J`
Identifier	::=	`x / y /` \<ClassOrInterfaceType\>
MethodName	::=	`m / n`
ExpressionName	::=	\<Identifier\>
Expression	::=	`i / j`

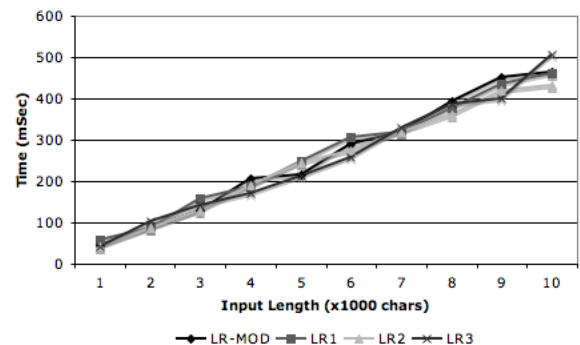**Figure 10.** Java's *Primary* expressions.

```
lr1 ::= <x> "1" / "1"
x   ::= <lr1>
```

is indirectly left-recursive "one rule deep" (`lr`, which is directly left-recursive, may be considered to be indirectly left-recursive zero rules deep). The two other rules, `lr2` and `lr3` (not shown), are indirectly left-recursive two and three rules deep, respectively.

Figure 12 shows the timing results for the new rules, in addition to the original `lr` rule. These results indicate that adding simple indirection to left-recursive rules does not have a significant effect on parse times. We expect that more complex instances of indirect left recursion found in real-world grammars, like Java's *Primary* rule, will exhibit steeper, but still linear, *input length vs. parse time* curves (this should be attributed to the complexity of the grammar, and not its use of left recursion).

Figure 12. The effect of indirect left recursion on parse times.

6. Related Work

As discussed in Section 1, some packrat parser implementations, including *Pappy* [1] and *Rats!* [6], support *directly left-recursive* rules by transforming them into equivalent non-left-recursive rules. Unfortunately, because neither of these implementations supports *mutually left-recursive* rules, implementing parsers for grammars containing such rules (such as Java's) using these systems is not trivial. The programmer must carefully analyze the grammar and rewrite certain rules, which can be tricky. And because the resulting parser is no longer obviously equivalent to the grammar for which

it was written, it is difficult to be certain that it does indeed parse the language for which it was intended.

While it may be possible to adapt the "transformational approach" of *Pappy* and *Rats!* to handle mutual left recursion by performing a global analysis on the grammar,

- the presence of syntactic and semantic predicate terms may complicate this task significantly, and

- such a global analysis is not a good fit for modular parsing frameworks such as *Rats!*, in which rules may be overridden in "sub-parsers".

The approach described here works seamlessly with syntactic and semantic predicates, as well as modular parsing. In fact, our implementation of *OMeta* [11], an object-oriented language for parsing and pattern-matching which (like *Rats!*) allows rules to be overridden, uses the mechanisms described in this paper.

In [4], Frost and Hafiz present a technique for supporting left recursion in top-down parsers that involves limiting the depth of the (otherwise) infinite recursion that arises from left-recursive rules to the length of the remaining input plus 1. While their technique is applicable to any kind of top-down parser (including packrat parsers), it cannot be used when the length of the input stream is unknown, as is the case with interactive input (e.g., read-eval-print loops and network sockets). While the approach presented here does not have this limitation and is significantly more efficient, its need to interact with the memo table makes it applicable only to packrat parsers.

In [7], Johnson describes a technique based on memoization and Continuation-Passing Style (CPS) for implementing top-down parsers that support left recursion and polynomial parse times. This technique was developed for CFGs and relies heavily on the non-determinism of the CFG choice operator; for this reason, we believe that it would be difficult (if at all possible) to adapt it for use in packrat parser implementations, where the ordering of the choices is significant.

Jamie Douglass, one of the authors of this paper, had previously developed a memoization-based technique for supporting left recursion in an earlier version of *CAT* which only supported CFG rules. That technique's memoization mechanism was restricted to only the head and involved rules, and used only while growing a seed parse.

7. Conclusions and Future Work

We have described a modification to the memoization mechanism used by packrat parser implementations that enables them to support both direct and indirect (or mutual) left recursion. This modification obviates the need for left recursion elimination transformations, and supports typical uses of left recursion without sacrificing linear parse times.

We have applied our modification to the packrat parser implementations of the *CAT* framework and *OMeta*, a parsing and pattern-matching language. This enabled both of these systems to support the heavily left-recursive portion of the Java grammar discussed in Section 4.

One of the anonymous reviewers noted that the compelling simplicity of packrat parsing is "to a large extent lost in the effort to support indirect left recursion." We, like this reviewer, believe that the final version of the algorithm presented in Section 3 can be simplified, and plan to do so in future work.

Packrat parsing was originally developed by Bryan Ford to support PEGs [3]. By extending packrat parsers with support for left recursion, we have also extended the class of grammars they are able to parse (which is now a superset of PEGs). Therefore, it may be interesting to develop a formalism for this new class of grammars that can serve as the theoretical foundation for this new style of packrat parsing.

References

[1] Bryan Ford. Packrat Parsing: a practical linear-time algorithm with backtracking. Master's thesis, Massachusetts Institute of Technology, September 2002.

[2] Bryan Ford. Packrat Parsing: Simple, Powerful, Lazy, Linear Time (functional pearl). In *ICFP '02: Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming*, pages 36–47, New York, NY, USA, 2002. ACM Press.

[3] Bryan Ford. Parsing Expression Grammars: a Recognition-Based Syntactic Foundation. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 111–122, New York, NY, USA, 2004. ACM Press.

[4] Richard A. Frost and Rahmatullah Hafiz. A new top-down parsing algorithm to accommodate ambiguity and left recursion in polynomial time. *SIGPLAN Notices*, 41(5):46–54, 2006.

[5] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, 2005.

[6] Robert Grimm. Better Extensibility Through Modular Syntax. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 38–51, New York, NY, USA, 2006. ACM Press.

[7] Mark Johnson. Memoization in top-down parsing. *Computational Linguistics*, 21(3):405–417, 1995.

[8] Roman Redziejowski. Parsing Expression Grammar as a primitive recursive-descent parser with backtracking. In G. Lindemann and H. Schlingloff, editors, *Proceedings of the CS&P'2006 Workshop*, volume 3 of *Informatik-Bericht Nr. 206*, pages 304–315. Humboldt-Universität zu Berlin, 2006. To appear in *Fundamenta Informaticae*.

[9] Eric Van Wyk and August Schwerdfeger. Context-Aware Scanning for Parsing Extensible Languages. In *Intl. Conf. on Generative Programming and Component Engineering, GPCE 2007*. ACM Press, October 2007.

[10] Eelco Visser. Scannerless Generalized-LR Parsing. Technical Report P9707, Programming Research Group, University of Amsterdam.

[11] Alessandro Warth and Ian Piumarta. OMeta: an Object-Oriented Language for Pattern-Matching. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications*, New York, NY, USA, 2007. ACM Press.