Improved Packrat Parser Left Recursion Support

James R Douglass

The Boeing Company
Seal Beach, CA

jamie.douglass@boeing.com

Abstract

Packrat parsers offer the guarantee of linear parse times while supporting backtracking and unlimited look-ahead. However, in packrat parser implementations, left recursion support is limited. This unfortunately makes them difficult to use for a large class of grammars. In our previous paper, we described a modification to the packrat parser memoization mechanism that made it possible to support—even indirectly or mutually—left recursive rules. This paper presents a simpler solution for supporting left recursion that is easier to understand and implement. While it is possible for some left recursive grammars to yield super-linear parse times (a problem which also occurs with iterative combinators) these grammars are a non-typical use of left recursion. For practical grammars with or without left recursion, the parsing solution presented in this paper continues to exhibit linear parse times and preserves the other advantages offered by packrat parsers.

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

General Terms Algorithms, Performance, Design, Languages

Keywords packrat parsing, left recursion

1. Introduction[1]

Packrat parsers [3] are an attractive choice for implementing programming languages because:

- They provide "the power and flexibility of backtracking and unlimited look-ahead, but nevertheless [guarantee] linear parse times." [3]
- 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 a Context-Free Grammar (CFG). As a result, there have no ambiguities and therefore 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 [11], eliminates the need for *moded lexers* [10] when combining grammars (e.g. in Domain-Specific Embedded Language (DSEL) implementations).

Unfortunately, "like other recursive descent parsers, packrat parsers cannot support left-recursion" [7], 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>        (1)
```

Each alternative definition in the ordered choice (denoted here by "/") is evaluated in the left to right order that the alternatives occur. Packrat parsers use the first successful matching alternative definition within the ordered choice.

In rule (1), the first alternative definition begins with rule (1) referencing itself. Therefore, each application attempts to match the first alternative which immediately applies rule (1) again. Unless, the application of rule expr at the current input position is marked to catch left recursion as an error, the repeated application of rule (1) without consuming any input creates an infinite loop condition. The second alternative definition—the case without left recursion—is never used.

We could change the order of the choices in rule (1),

```
expr ::= <num> / <num> "-" <expr>        (2)
```

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 rule (2) 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* [2] and *Rats!* [7], 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>)*              (3)
```

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*.

[1] Major of introduction is taken from previous paper. [13]

SIGPLAN'05 June 12–15, 2005, Location, State, Country.

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

Figure 1. The original APPLY-RULE procedure.

Now consider the following minor modification to the original grammar, which has no effect on the language accepted by expr in rule (1):

```
x    ::= <expr>
expr ::= <x> "-" <num> / <num>
```
(4)

When given this grammar, the *Pappy* packrat parser generator [2] reports the following error message:

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

This happens because expr in rule (4) is now *indirectly* left-recursive, and *Pappy* does not support indirect left recursion (or mutual left recursion).

Although this example is certainly contrived, indirect left recursion does in fact arise in real-world grammars. For instance, Roman Redziejowski [9] discusses the difficulty of implementing a packrat parser for Java [6], 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.

These limitations, however, were largely overcome in our previous work [13] by modifications 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). A reviewer of our previous paper commented, "One of the compelling features of packrat parsing is the simplicity and elegance of the algorithm. That simplicity is, however, to a large extend lost in the effort to support indirect left recursion."

This paper presents a simpler solution for supporting left recursion in an effort to regain the simplicity and elegance of the original packrat parsing algorithm. The solution presented in this paper is easier to understand and implement. While it is possible for some left recursive grammars to yield super-linear parse times (a problem which also occurs with iterative combinators) these grammars are an unusual use of left recursion. For practical grammars with or without left recursion, the parsing solution presented in this paper continues to exhibit linear parse times and preserves the other advantages offered by packrat parsers.

The rest of this paper is structured as follows. Section 2. gives a brief overview of packrat parsing. Section 3. describes the anatomy of left recursive grammars and defines terminology that is used to examine the left recursion solution presented in later sections. Section 4. summarizes our previously presented modification to the memoization mechanism to support left recursion along with a description of the complexity it introduces to packrat parsing. Section 5. describes the simplified solution for supporting left recursion in a packrat parser. Section 6. validates this work by examining several example grammars including a grammar that closely mirrors Java's heavily left-recursive *Primary* rule. Section 7. discusses the effects our improved left recursion support has on parse times. Section 8. discusses related work, and Section 9. concludes with a summary of results and future work plans.

2. An Overview of Packrat Parsing[2]

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" [3].

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.

For example, consider what happens when the rule

```
expr ::=  <num> "+" <num>
        / <num> "-" <num>
```
(5)

(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 (position 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 updates the current position to 4 and carries 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

$$\text{MEMO} : (\text{RULE}, \text{POS}) \rightarrow \text{MEMOENTRY}$$

where

$$\text{MEMOENTRY} : (\textit{ans}: \text{AST}, \textit{pos} : \text{POS})$$

In other words, MEMO maps a rule-position pair (*R*,*P*) to a tuple consisting of the AST (or the special value FAIL[3]), and the

[2] Section is taken from previous paper. [13]

position of the next character on the input stream. The AST is the result of applying rule *R* at position *P*. If there is no entry in the memo table for the given rule-position pair then the function returns a value of NIL.

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 examine left recursive grammars in more detail and define terminology that will be used to describe the left recursion solutions in later sections.

3. Left Recurive Grammars

This section will informally define a normalized grammar used to explore the behavior of left recursive rules. Terms are defined for describing the various parts formed while parsing a left recursive rule or set of rules. The section will also describe the different forms of left recursive rules and rule sets. Each kind of left recursion must be considered to insure a particular parsing solution for left recursion support works correctly.

3.1 A Parsing Expression Grammar (PEG)

The grammar we will use to examine left recursion is a Parsing Expression Grammar (PEG) written in normal form. Each Nonterminal is defined by a Definition in a single rule as follows.

Nonterminal ::= Definition

The Nonterminal is represented by an identifier, while the Definition itself is either a *simple definition* or an *ordered choice*. A *simple definition* is one of the following:

1. <*Nonterminal*> — identifier for a nonterminal between angle brackets which represents a reference or application of the rule with the name *Nonterminal*.
2. "*character string*" — one or more characters between quotation marks that represents a terminal or literal. Characters in *character string* are matched against the input stream during parsing.
3. *sequence* — nonterminals and terminals (items 1 and 2) listed left-to-right that represent a pattern to be matched against the input stream.
4. "" — quotation marks with no characters in between. This is the empty string that always matches without changing the position of the input stream.

Finally, an *ordered choice* is a left-to-right list of *single definitions* with the definitions separated by a slash, "/". Normalized grammars are minimal in nature. Syntactic operations and forms such as parenthesis "()" for grouping, question mark "?" for optional, star "*" for repetition, and plus "+" for one-or-more are all reduced to a representation using normalized rules. For example, repetition can be handled as follows:

repE ::= <E>* ≡ repE ::= <E> <repE> / "" (6)

This grammar is small, yet sufficiently powerful for our exploration of left recursion. This grammar will now be used to examine the various parts that form a left recursion.

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

3.2 Anatomy of Left Recursion

Examine the rules in (4) again where rule x is applied to parsing the input stream "3-2-1". Rule x immediately applies rule expr which then makes the left recursive reference back to rule x. Every left recursion forms a loop like this during application of the rules without consuming any input. The following are useful terms for the various parts of these loops formed by rules like (4).

- A *head* is the first rule application in the left recursion loop. The initial application of rule x in (4) is a left recursion *head*.
- A *tail* is the rule reference back to the *head*. In this example, the first alternative definition in rule expr that begins with <x> is the *tail*.
- The *chain* is the sequence of rules and rule alternatives that are active when the *tail* references back to the *head*. For this example, the *chain* is rule x to its <expr> alternative, rule expr and its first alternative <x> "-" <num>, finally back to rule x. A *chain* represents a single alternative without the pending ordered choices. Therefore, the ordered choice with second alternative <num> for rule x is not part of this *chain*.

As described earlier in this paper, left recursion creates an infinite loop. If we, however, continue to parse looking at other alternatives, then the second choice in rule expr would be attempted next. The <num> alternative matches "3" in the input stream and is returned as the result for rule x. This initial parse for the *head* rule is called the *seed*.

Now that we have an initial parse result, *seed*, for rule x, the left recursion *chain* that was abandoned can be attempted again at the original input position. This time the *seed* parse is used, and the <x> "-" <num> alternative in rule expr succeeds matching "3-2" which becomes the new parse result for rule <x>. The same *chain* is then tried again with the "3-2" parse for x. This time <x> "-" <num> matches "3-2-1". Each of these attempts to match the *chain* (or *chains*) associated with the *head* is called a *growth cycle*. If a *growth cycle* is successful, the left recursive parse result for the *head* is enlarged. In fact, successful matches must grow the parse result by consuming more input else the left recursion is terminated. In the example, attempting the *chain* for rule x again fails to match any more characters from the input stream. The left recursion, therefore, returns "3-2-1" as the result. Requiring that each growth cycle consume more input, avoids the infinite loop condition that rules like the following would create.

loopAgain ::= <loopAgain> / <seed> (7)

Rule (7) invokes itself without looking for any input beyond what is matched by <seed>.

The terms *head*, *chain*, *tail*, *seed* and *growth cycle* defined in this section will be used to describe the left recursion parsing solutions presented in the remainder of this paper. In the next subsection, the different kinds of left recursive rules and rule sets are presented.

3.3 Types of Left Recursions

Rule (1), as stated above, is a *direct* left recursive rule, while the rules in (4) form a set of *indirect* left recursive rules. This subsection will examine these in more detail and present the different kinds of left recursion that must be handled by any valid left recursive packrat parsing solution.

Rule (1) is an instance of *direct* left recursion. A *direct* left recursion involves only one rule and has a form as follows: [1]

x ::= <x> R_1 / <x> R_2 /…/<x> R_n/ S (8)

The i^{th} left recursive simple definition is represented by the left recursive reference, <x>, followed by the rest of the simple definition R_i. In (4) R_1 is "-" <num>. The S is a simple definition or ordered choice that matches the seed and is therefore not left recursive with rule x. In (4) the S is <num>. In general, a *direct* recursive rule can repeat the form in (8) any number of times as follows:

$$\begin{aligned} x ::= &<x>\ R_1\ /\ldots/\ <x>\ R_2\ /\ldots/<x>\ R_i/\ S_1 \\ &/\ <x>\ R_{i+1}\ /\ldots/\ <x>\ R_{j+2}\ /\ldots/<x>\ R_j/\ S_2 \\ &\ldots \\ &/\ <x>\ R_{k+1}\ /\ldots/\ <x>\ R_{k+2}\ /\ldots/<x>\ R_n/\ S_m \end{aligned} \tag{9}$$

The *direct* left recursion form in (8) typically occurs in grammars while the general form in (9) does not occur.

An *indirect* left recursion is more complicated than *direct* left recursion. Consider this simple example:

$$\begin{aligned} x_1 ::= &<x_2>\ R_1\ /\ S_1 \\ x_2 ::= &<x_3>\ R_2\ /\ S_2 \\ &\ldots \\ x_n ::= &<x_1>\ R_n/\ S_n \end{aligned} \tag{10}$$

The rule set in (10) forms a single chain of *indirect* left recursive rules with rule x_1 as the head. Rule (4) creates a two rule chain of this form.

Each of the rules x_i in (10) can be expanded by adding multiple *indirect* left recursive alternatives. These alternatives create separate *indirect* chains with tails referencing back to the same head rule. We call this expansion *diverging* since it adds additional chains to the left recursion. Rule set (11) is a simple example of diverging.

```
x ::= <y> '2' / <z> '3' / '1'
y ::= <x>                                (11)
z ::= <x>
```

The left recursive rule chains do not need to be completely separate but can have the same rule(s) on more than one chain. Consider the following example:

```
x ::= <y> '2' / <y> '3' / '1'            (12)
y ::= <x>
```

The two left recursive chains in (12) both have rule y. This is referred to as *merging*. The two chains in (12) share a common path starting a rule y that makes the tail reference to rule x. The collection of chains in these indirect recursions can merge to a single rule and then diverge, separate out again, creating a variety of multiple paths from the head to each tail.

The kinds of left recursions described so far have all had a single head. Left recursion in general (unlike *direct* left recursion) can have more than one rule. Each rule that is active (currently being applied) can be referenced again creating an additional head rule. Consider the following rule set.

$$\begin{aligned} x ::= &<y>\ R_1 \\ y ::= &<z>\ R_2 \\ z ::= &<y>\ R_3\ /\ M\ /\ S \end{aligned} \tag{13}$$

The rules in (13) create a chain with rule y as the head. If M is an alternative of the form <z> R_4 then rule z becomes a second head. We call this an *inner* left recursion because the first application of rule z occurs after the first application of head rule y. If M is an alternative of the form <x> R_4 then rule x becomes the second head. This type of left recursion we referred to as an *outer* left recursion because the first application of rule x occurs before the first application of head rule y. Both *inner* and *outer* left recursions are collectively referred to as *mutual* left recursive rules. The difference between *inner* and *outer* left recursion is significant because packrat parsing relies on ordered choice.

An often used form of left recursion is where the seed for one rule set is derived with another *nested* left recursive rule set. This can be shown with the following rules.

```
term   ::= <term> '+' <factor> / <factor>
factor ::= <factor> '*' <num> / <num>    (14)
```

The rule factor is a *direct* left recursion that finds the seed for term. The rule term is also a *direct* left recursion. The left recursion of rule factor is nested within the left recursion of rule term. Unlike *mutual* left recursion, the *nested* left recursion of the rule factor is completed before any growth cycles using rule term are attempted.

This section examined the various kinds of left recursion and defined the following terms to refer to them: *direct*, *indirect*, *diverging*, *merging*, *mutual (inner* or *outer*), and *nested*.

4. Previous Solution

This section presents an overview and analysis of the left recursion solution from our previous paper [13]. This solution modifies the memorization mechanism in packrat parsing. These changes are shown in the Appendix. The following is the basic idea behind this solution.

- By initially returning a FAIL for any left recursive reference, the parser can search to find a seed parse (if one exists) for the head rule.
- The head rule is then reapplied to grow the left recursion using the seed. Any previous left recursive tail receives the seed as its value.
- The head rule is reapplied using the result of the last growth cycle for left recursive tail references. This growth cycle is repeated as long as the parse succeeds by consuming more input.

This relatively simple idea is difficult to implement in packrat parsers because the memorization table obstructs the reapplication of rules involved in the left recursion. The previous solution overcame this problem by maintaining a rule invocation stack. This stack was used to determine the head rule and collect the rules involved in the left recursive chains. Once and only once during each growth cycle, a rule involved in the left recursion was allowed to reevaluate its parse result.

A separate growth loop to reevaluate the head rule was also implemented. Another disadvantage in this solution is the reevaluation of the head rule causes parsing work already done to be repeated during each growth cycle. The seed may also be reparsed when the left recursion growth cycle is terminated. The next section presents a solution that overcomes these difficulties.

5. Improved Solution

The key insight behind the improved left recursion solution presented in this section is saving (in the order they occur) a list of the left recursive chains. Each chain is a single path (without pending ordered choices) back to the head rule. By saving each chain at the point the tail is attempting to recall a parse from the memo table, the processing can be continued once a result such as the seed or a growth cycle parse becomes available. **Figure 2** shows the improved solution for supporting left recursion in a packrat parser. To detect left recursion and construct these chains, the following classes are added.

```
APPLY-RULE(R,P)
  let m = MEMO(R,P)
  if m = NIL
    then let e = new MEMOENTRY(
              new LRFAIL(NIL,thisContext),P)
         MEMO(R,P) ← e
         e.ans ← EVAL(R.body)
         e.pos ← Pos
         if e is HEAD then  ► Grow LR
           then if e.ans is LRFAIL
                  then  e.ans ← FAIL
                return APPLY-GROWRULE(e,P)
         return e.ans
  Pos ← m.pos
  if m.ans is LRFAIL  ► LR detected
    then if m.ans.heads = NIL ► New head
           then m becomes HEAD(m.ans, m.pos,{})
                m.ans.heads ← {m}
         ADDCHAIN(thisContext,m.ans.heads)
  return m.ans
```

Figure 2. The improved APPLY-RULE procedure.

LRFAIL: (*heads*: SET OF HEAD, *frame*: CONTEXT)

HEAD subclass of MEMOENTRY: (chains: Array of CHAIN)

The rules in a packrat parser are usually implemented as functions. The *frame* field in LRFAIL is a stack activation record called CONTEXT. When a rule is evaluated the activation record for the currently executing function, *thisContext,* is saved in the *frame* field. Instances of LRFAIL are used to mark currently active rules in the memo table. When a rule is invoked by a left recursive reference, the LRFAIL for the rule is retrieved from the memo table.

APPLY-RULE uses the LRFAIL to detect left recursion and create the chains associated with the left recursive head rule. The

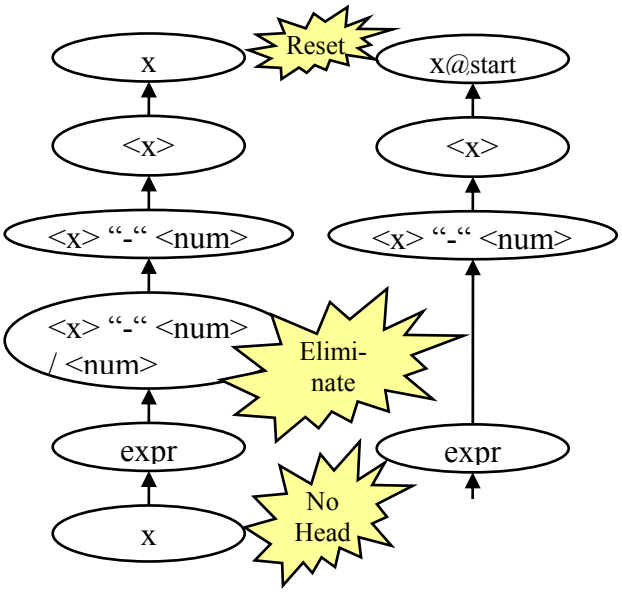

Figure 3. Example Chain Creation: the left side is the call stack, and the right side in the chain created from stack.

first time the LRFAIL has been referenced, the value of the *heads* field will be NIL. This means a new head rule has been detected. The memo table entry for the LRFAIL is then promoted to an instance of HEAD which adds a field to store the chains associated with this new head rule. Anything pointing to the old MEMOENTRY will now be pointing to the promoted HEAD instance. The *heads* field in LRFAIL is also updated to a set containing the newly promoted HEAD memo table entry. The LRFAIL now has the head context necessary to create a new left recursion chain and the memo table entry for the head where the chain will be saved.

APPLY-RULE then calls ADDCHAIN to find and add the chain for the left recursion. The activation record for the tail and the set of HEAD memo table entries are passed to ADDCHAIN. A copy of the chain—represented as a linked list of CONTEXT—is added to an array in the chains field of the HEAD memo table entry. As the parsing continues the LRFAIL is returned as the parse result and stored in the memo table for other rules in the left recursion chain. When encountered elsewhere by the parser, LRFAIL is treated as a FAIL. If more than one LRFAIL is discovered by an ordered choice the heads from the various LRFAIL instances are collected in a new instance of LRFAIL.

Every time a tail makes a left recursive reference back to the head (or to the rule on a previously detected chain) a LRFAIL is found in the memo table. A new chain is then added to the *chains* field in the HEAD memo table entry corresponding to the head rule for the chain. **Figure 3** illustrates the creation of the chain for the example set of rules (4). This is accomplished by copying the current context for the tail along with the other context records back to just before the context for the head rule. The *heads* field in the LRFAIL is a set of HEAD memo table entries. Each set element has an instance of LRFAIL as its current *ans* field value whose *frame* field is the activation record for that HEAD memo table entry. The head is the first context from this set encountered while traversing the activation record chain starting from the tail. The context records for any ordered choices are not included, and the tail rule context copy is reset to the function entry point. The chains are only collected for the head rule. By collecting them in the order encountered, the left to right processing for packrat parsing is preserved.

The *chains* for each rule head form a *grow rule.* Each *grow rule* is a repeating ordered choice between the chains found in the *chains* field. As long as one of the chains results in a parse for the rule head which consumes more input, the result is saved in the memo table. The input position is then restored to the start of the left recursion, and the whole process is repeated. The behavior of evaluating a *grow rule* is equivalent to the following:

```
APPLY-GROWRULE(M,P)
  do ► Left Repetition while improvement
     Pos ← P
     for each c in M.chains ► Ordered Choice
        do let ans = RESUME(c) ► on Chain
           if ans ≠ FAIL then  break
           Pos ← P
     if Pos <= M.pos ► No improvement
        then Pos ← M.pos
             return M.ans
     M.ans ← ans
     M.pos ← Pos
```

(15)

Each chain c is evaluated to find a parse result that succeeds. A successful parse that consumes more input than the previous result is saved and the chains are attempted again. The input position has been reset to the start of the left recursion and the

new parse result is used for the next cycle. Once the left recursion parse fails to grow, the last parse result is returned.

The loop presented in (15) is to make explicit the behavior of the *grow rule*. The implementation is actually a special grammar rule synthesized during the parse. Consider the following syntax for representing a left recursive rule.

$$x ::= S \; *> \; <x> \; R_1 \; / \; <x> \; R_2 \; / \ldots / <x> \; R \qquad (16)$$

The binary operator *> determines a seed from the left side expression S. Then repeatedly evaluates the right hand expression with the ordered choices of items <x> R_i. Each repetition is performed starting from the same input position rule x began its parse. The grow rule is the expression in (16) after the seed S. The solution in **Figure 2** evaluates a rule definition to construct a non-left recursive seed parse and the *grow rule* over the left recursive chains. The grow rule is then evaluated with the seed as its initial parse.

Consider how the improved solution works with the rules (4) on the input "3-2-1". The initial applications of rules x and expr places instances of LRFAIL in the memo table along with the activation record for each respective rule. When the x rule is referenced left recursively it is made a head rule by promoting its memo table entry to an instance of HEAD. The chain depicted in **Figure 3** is then added to the newly promoted HEAD for rule x. The LRFAIL for rule x also contains a set with the new HEAD in the heads field. The expr rule then attempts its second alternative which matches '3'. Using '3' as the seed parse, the *grow rule* with the single chain from **Figure 3** is evaluated. The first grow cycle picks up the seed '3' for rule x and then matches '-2' to form the parse of '3-2'. The *grow rule* repeats matching '3-2-1'. The *grow rule* is tried once again, but this time the rule fails. The left recursion ends, and returns the last successful match.

This solution eliminates the disadvantages of our previous solutions discussed in Section 4. The resulting solution is smaller and simpler, regaining the elegance that characterizes packrat parsing. Detecting left recursive rules and creating the *grow rules* can be performed before the parser is used. This eliminates repeating this activity for each position where a left recursive rule is applied. Parsing then becomes recognizing the seed parse and applying the *grow rule* for left recursion. The presentation of this optimization, however, is beyond the scope of this paper.

6. Java *Primary* and Other Grammars

To validate this way of supporting left recursion, the solution presented was implemented and tested in *Context-free Attributed Transformations (CAT)* a parser generator that supports PEG and CFG representations. The implementation was tested using a set of grammars covering the various kinds of left recursive grammars described in Section 3.3 . All these kinds of left recursion except the mutual inner and outer forms occur in practical grammars to implement languages. The effort to implement the improved solution was ten times easier then the previous solution. Once the implementation was tested using a collection of test cases consisting of small example grammars, the new CAT parser was used on more practical grammars. The first of these practical languages was a typical arithmetic expression grammar with terms of addition and subtraction, and factors of multiplication and division over numbers.

Next, testing was done on the Java Primary rule previously described as follows: "…a grammar that closely mirrors Java's *Primary* rule, as found in chapter 15 of the Java Language Specification [6]. Because of its heavily mutually left-recursive nature, Primary cannot be supported directly by conventional packrat parsers [9]."[13] Finally, the CAT implementation with the left recursion support presented in this paper was used to compile a new version of CAT itself. The grammar for CAT utilizes left recursive rules in several places within the language. Examples of the test grammars used are in **Figure 5**, and the Java Primary rule grammar is shown in **Figure 4**. The parse time performance is presented in the next section.

```
► Java Primary
Primary ::= <PrimaryNoNewArray>
PrimaryNoNewArray ::= <ClassInstanceCreationExpression>
        / <MethodInvocation> / <FieldAccess>
        / <ArrayAccess> / "this"
ClassInstanceCreationExpression
        ::= "new" <ClassOrInterfaceType> "()"
        / <Primary> "." "new" <Identifier> "()"
MethodInvocation ::= <Primary> "." < MethodName > "()"
        / <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"
input ► this      AST► 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)
```

Figure 4. Java's *Primary* expressions: Java *Primary* expressions with corresponding parse trees as s-expressions.

7. Performance

Packrat parsers guarantee linear parse times by computing the resulting a rule in constant time. The result for a rule at a particular position within the input stream is only calculated once. Any subsequent applications of the rule at the same position use the previously memoized result. Implementing repetition, '*', directly rather than using a repetition rule like (7) violates this assumption, thus making it possible for some grammars to yield super-linear parse times. This occurs for non-typical uses of repetition as shown by the following example:

$$\begin{aligned} &x ::= <y> \; "2" \; / \; "1" \; <x> \; / \; "" \\ &y ::= "1" \; * \end{aligned} \qquad (17)$$

When (17) is applied to a string of '1's the initial parse of y consumes all input, but rule x fails because it finds no '2' for the first choice. The second choice for rule x consumes a single '1' and repeats the application of x at input position 1. Rule y has no memoized result at position 1 even though it has previously parsed the series of input characters and memoized them at position 0. This reapplying a rule in the middle of a repetition forces a rescanning of the input stream. Therefore, (17) accepts strings of zero or more '1's in $O(n^2)$ time.

Left recursion is a kind of repetition as seen in (3). Similar super-linear parse times can result with non-typical uses such as

```
x ::= <y> “2” / “1” <x> / “”
y ::= <y> “1” / “1”                    (18)
```

which again accepts strings of zero or more ‘1’s in $O(n^2)$ time. Repetition is restored to a linear time guarantee by using a rule that memorizes the intermediate results as in (7). This transforms the rules in (17) to the following:

```
x ::= <y> “2” / “1” <x> / “”
y ::= “1” <y>  / “”                    (19)
```

The initial parse of the input stream creates a memoized result for rule y at all the input positions for a string of ‘1’s. The following is a similar rule rewrite of (18) that restores its linear parse time.

```
x ::= <y> “2” / “1” <x> / “”
y ::= <y> <y> / “1”                    (20)
```

Again, the initial parse of the input stream creates a memorized result for rule y at all the input positions for a string of ‘1’s. Transforming (17) to (19) and (18) to (20) results in grammars that recognize the same language but all produce different parse trees. With (20) the left associativity of (18) is lost, and additional nodes for rule y are introduced. It is possible to rewrite the parse trees to compensate for the transforms. Implementing repetition or left recursion to memoize these intermediate results, however, increases memory use and penalizes typical usage of these syntax structures. Therefore, we see little justification for using these implementation strategies to avoid super-linear parse times with non-typical rules that are not found in practical grammars [2, 13].

Repeating performance testing conducted on the previous solution confirmed the linear parse time behavior of the improved approach. These results are shown in **Figure 7** and **Figure 6**.

In order to compare the performance of packrat parser modifications presented in this paper the following two rules were used.

```
rr ::= “1” <rr> / “1”
lr ::= <lr> “1” / “1”                  (21)
```

The rr rule is right-recursive, while rule lr is left-recursive. Both recognize a string of one or more ‘1’s, but they generate different parse trees. Rule rr creates a right associative parse tree, while lr

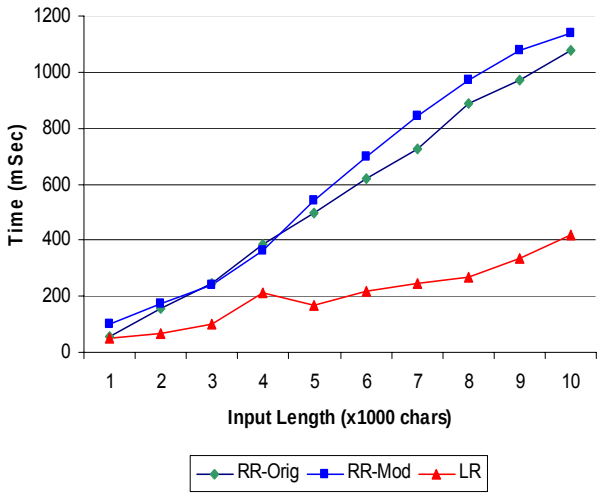

Figure 7. Performance Comparison: RR-Orig shows rr in a traditional packrat parser. RR-Mod and LR show rr and lr, respectively in the packrat parser with the improved left recursion support.

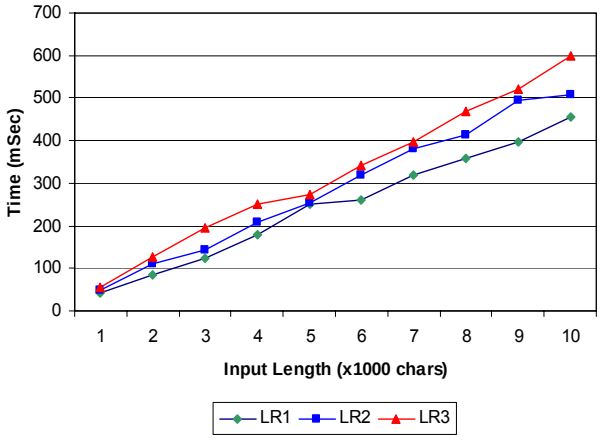

Figure 6. Left recursion parse times.

creates a left associate parse tree. These parse trees generated for an input string are the same size. The rules in (21) were used to recognize strings of ‘1’s with lengths from of 1,000 to 10,000 in increments of 1,000. **Figure 7** shows the performance parse times for rule rr with the original packrat parsing in **Figure 1** (*RR-Orig*) and with the modified packrat parser with left recursion support in **Figure 2** (*RR-Mod*). The lr rule parse time is of course only using the modified packrat parser in **Figure 2** (*LR*). **Figure 7** shows the modifications to support left recursion do not introduce significant overhead for non-left-recursive rules. The recognition times for *RR-Mod* are only slightly slower then *RR-Org*. The modified packrat parser supporting left recursion exhibits linear parse time as seen by the results in *LR*. Using left recursion can actually improve parse times. The results of *LR* are consistently better than either *RR-Orig* or *RR-Mod*. This difference is likely due to the reduced stack space used by left recursion.

The parse time results shown in **Figure 6** were obtained using the following simple rules.

```
LR1 ::= <xLR1> “1” / “1”
  xLR1 ::= <LR1>
LR2 ::= <xLR2> “1” / “1”
  xLR2 ::= <yLR2>
  yLR2 ::= <LR2>                       (22)
LR3 ::= <xLR3> “1” / “1”
  xLR3 ::= <yLR3>
  yLR3 ::= <zLR3>
  zLR3 ::= <LR3>
```

The rules LR1, LR2 and LR3 are indirect left recursions with indirect rule depths of one, two and three respectively. Both left recursive solutions produce about the same parse time results with the same linear slopes. Performance tests on the testing grammars in **Figure 5** and **Figure 4** all yield similar linear parse time results.

8. Related Work

As described in Section 1, some packrat parsers including *Pappy* [2] and *Rats!* [7], support direct left-recursive rules by rewrite transformations to perform left recursion elimination. Without support for indirect left recursion, implementing grammars such as Java requires careful analysis of the grammar and rewriting of rules. The modifications to the grammar make it no longer obvious whether the resulting parsers are equivalent. This uncertainty makes it difficult to verify if the parser does indeed recognize the language for which it was intended.

Transformational approaches such as these also present other difficulties. Syntactic and semantic predicates may complicate rewriting and global analysis works against modular framework such as *Rats!* where sub-parsers can override rules. The approach described here works well with syntactic and semantic predicates as well as modular parsing.

Frost and Haliz [5] presented a technique to support left recursion in top-down parsers which involved limiting the depth of the infinite recursion based on the input stream length. It is not always possible to know the length of the input stream when the parser begins accepting input. Interactive input from a read-eval-print loops or network socket are two such examples. This approach like others we have encountered is less efficient.

Johnson [8] described a technique based on memorization and Continuation-Passing Style (CPS) for implementing top-down parsers that support left recursion and polynomial parse times. This technique was developed for a CFG and relies heavily on the non-determinism of the CFG choice operator. The approach we presented eliminates choice from each parse chain and supports the ordered choice required for a packrat parser.

We previously developed a technique for left recursion support in an earlier version of *CAT* which only supported CFG rules. That technique memorized only the head rule parse when left recursion was detected and repeatedly re-evaluated that head rule to greedily grow a let recursive result. This solution suffers the same performance problems found in recursive descent parsers.

9. Conclusions and Future Work

We have described an improved left recursion support for packrat parsing which preserves the linear parse times while supporting backtracking and unlimited look-ahead for practical grammars. The new approach was easier to implement and understand during testing, and parses the same grammars with slightly better parse times. This improved approach has regained most if not all of the simplicity that was lost in our previous effort to support left recursion.

Packrat parsing was originally developed by Bryan Ford to support languages represented in a PEG. This work extends the class of grammars supported by including left recursion. In future work, we plan to further extend the grammars supported by efficient parsing techniques and approaches for combining PEG and CGF rules in a single grammar formalism. We are also interested in applying this approach to the implementation of Domain Specific Languages (DSL). Included in this effort is how to modularize languages in a framework where languages can be composed from a collection of small reusable grammars.

Acknowledgments

Thanks to Alex Warth and Todd Millstein co-authors of the previous paper [13] and the anonymous reviewers for their constructive comments and feedback on this work.

Appendix

This appendix presents the original left recursion solution published in our previous paper [13].

```
MEMO : (RULE,POS) → MEMOENTRY

MEMOENTRY: (ans: AST or LR, pos: POSITION)

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

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

HEADS : POSITION → HEAD

APPLY-RULE(R,P)
    let m = RECALL(R,P)
    if m = NIL
        then
            let lr = new LR(FAIL,R,NIL,LRStack)
            LRStack ← lr
            let e ← new MEMOENTRY(lr,P)
            MEMO(R,P) ← e
            let ans = EVAL(R.body)
            LRStack ← LRStack.next
            e.pos ← Pos
            if lr.head ≠ NIL
              then lr.seed ←  ans
                  return LR-ANSWER(R,P,m)
              else e.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

RECALL(R,P)
    let m = MEMO(R,P)
    let h = HEADS(P)
    if h = NIL
        then return m
    if m = NIL and R ∉ {h.head} U h.involvedSet
        then return new MEMOENTRY(FAIL,P)
    if R ∈ h.evalSet
        then h.evalSet ← h.evalSet \ {R}
            let ans = EVAL(R.body)
            m.ans ← ans
            m.pos ← Pos
    return m

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)
```

```
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 U {s.rule}
           s ← s.next

GROW-LR(R,P,M,H)
    HEADS(P) ← H
    while TRUE
        do
           Pos ← P
           H.evalSet ← COPY(H.involvedSet)
           let ans = EVAL(R.body)
           if ans = FAIL or Pos ≤ M.pos
              then break
           M.ans ← ans
           M.pos ← Pos
    HEADS(P) ← NIL
    Pos ← M.pos
    return M.ans
```

References

[1] Aho, A. V., Sethi, R., and Ullman, J. D. 1986 *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc.

[2] Ford, B. 20002 Packrat parsing: a practical linear-time algorithm with backtracking. Master's thesis, Massachusetts Institute of Technology, September 2002.

[3] Ford, B. 2002. Packrat parsing: simple, powerful, lazy, linear time, functional pearl. In *Proceedings of the Seventh ACM SIGPLAN international Conference on Functional Programming* (Pittsburgh, PA, USA, October 04 - 06, 2002). ICFP '02. ACM, New York, NY, 36-47. DOI= http://doi.acm.org/10.1145/581478.581483

[4] Ford, B. 2004. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Venice, Italy, January 14 - 16, 2004). POPL '04. ACM, New York, NY, 111-122. DOI= http://doi.acm.org/10.1145/964001.964011

[5] Frost, R. A. and Hafiz, R. 2006. A new top-down parsing algorithm to accommodate ambiguity and left recursion in polynomial time. *SIGPLAN Not.* 41, 5 (May. 2006), 46-54. DOI= http://doi.acm.org/10.1145/1149982.1149988

[6] Gosling, J., Joy, B., Steele, G., and Bracha, G. 2005 *Java(Tm) Language Specification, the (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional.

[7] Grimm, R. 2006. Better extensibility through modular syntax. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada, June 11 - 14, 2006). PLDI '06. ACM, New York, NY, 38-51. DOI= http://doi.acm.org/10.1145/1133981.1133987

[8] Johnson, M. 1995. Memoization in top-down parsing. *Comput. Linguist.* 21, 3 (September 1995), 405-417.

[9] Redziejowski, R. 2007 Parsing expression grammar as a primitive recursive-descent parser with backtracking. *Fundamenta Informaticae 79*, 3-4 (September 2007), 513-524.

[10] Van Wyk, E. R. and Schwerdfeger, A. C. 2007. Context-aware scanning for parsing extensible languages. In *Proceedings of the 6th international Conference on Generative Programming and Component Engineering* (Salzburg, Austria, October 01 - 03, 2007). GPCE '07. ACM, New York, NY, 63-72. DOI= http://doi.acm.org/10.1145/1289971.1289983

[11] Visser, E. 1997 Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, 1997.

[12] Warth, A. and Piumarta, I. 2007. OMeta: an object-oriented language for pattern matching. In *Proceedings of the 2007 Symposium on Dynamic Languages* (Montreal, Quebec, Canada, October 22 - 22, 2007). DLS '07. ACM, New York, NY, 11-19. DOI= http://doi.acm.org/10.1145/1297081.1297086

[13] Warth, A., Douglass, J. R., and Millstein, T. 2008. Packrat parsers can support left recursion. In *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (San Francisco, California, USA, January 07 - 08, 2008). PEPM '08. ACM, New York, NY, 103-110. DOI= http://doi.acm.org/10.1145/1328408.1328424