CPS Recursive Ascent Parsing

Arthur Nunes-Harwitt
Nassau Community College
One Education Drive
Garden City, NY 11530
nunesa@newton.matcmp.ncc.edu

ABSTRACT

The parsing problem is ubiquitous in Computer Science. Perhaps that is part of the reason it has been used so frequently to illustrate the power of continuations and monads. It is often the case, though, that the parser examples of continuation passing style and monads are top-down parsers. I intend to show that the continuation passing style of programming naturally leads to bottom-up or recursive ascent parsers. The programming language Scheme will be used to code example parsers.

1. INTRODUCTION

Parsing is an important problem. So much so, that we currently have several powerful technologies available for parsing, such as top-down recursive descent parsing and the bottom-up LR family of automated parsing tools. Is there any reason to look beyond these techniques?

The LR automated parsing tools can generate parsers for a very large class of grammars. Nevertheless, programmers often write recursive descent parsers despite the fact that that method can handle a smaller class of grammars. Why? In a word, recursive descent parsers are *hackable*. They are easy to understand, and it is straightforward to insert semantic actions. In contrast, LR tools are black boxes that generate inscrutable code. By writing a parser using continuation passing style (CPS) it is possible to get both the benefit of being able to parse a large class of grammars and the advantage of being able to understand the parser code.

This paper is structured as follows. First I will summarize some of the previous work done on parsing. Then I will give a brief overview of monadic programming since my parser's design, while not actually making use of monads, was influenced by those ideas. After that, I will discuss CPS recursive ascent parsing and give some sample parsers to demonstrate the power of the technique.

2. REVIEW OF PARSING

$$E ::= + \; E \; E \mid n$$

Figure 1: Grammar 1

This section discusses various approaches to parsing and their respective advantages and disadvantages.

2.1 Top-Down Parsing

It is easy to write a recursive descent parser by hand; in fact, often top-down parsing is used to illustrate some technique in functional programming[11, 13, 14]. While it is possible to automate recursive descent parsing[1, 5], the intuition behind the formalism involves viewing grammar variables as procedures. Each production $A ::= B_1 \; \cdots \; B_n$ can be read, not as a rewrite rule, but rather as a method for getting A — namely, get B_1 through B_n. With this intuition, it is easy to translate a grammar into code, and it is easy to discern what a recursive descent parser is doing.

For example, the production
$A ::= B_{1,1} \; \cdots \; B_{1,m_1} \mid \cdots \mid B_{n,1} \; \cdots \; B_{n,m_n}$
corresponds with the following Scheme[7] code.

```
(define (get-A)
  (cond ((eq? (peekToken) predictor_1)
         (let ((result11 (get-B11))
               ...
               (result1m1 (get-B1m1)))
           (make-result1 result11 ... result1m1)))
        ...
        ((eq? (peekToken) predictor_n)
         (let ((resultn1 (get-Bn1))
               ...
               (resultnm1 (get-Bnmn)))
           (make-resultn resultn1 ... resultnmn)))
        (else (error "Syntax error"))))
```

When there are tokens in the token input stream that can identify or predict which rule to use, recursive descent parsing works very well. Grammar 1 (Figure 1) is an example of a grammar that poses no problem for recursive descent parsing. There are two rules, and they can be distinguished immediately because one starts with a plus sign and one starts with a number.

Grammar 2 (Figure 2) is an example of a grammar that is more difficult. Not only is there no token to predict which rule to use, but also, because of the left recursion, the parser

$$E ::= E\ E\ \texttt{+} \mid n$$

Figure 2: Grammar 2

$$\begin{array}{rcl} S & ::= & \texttt{<}\ S \mid M \\ M & ::= & \texttt{<}\ M\ \texttt{>} \mid \texttt{<>} \end{array}$$

Figure 3: Grammar 3

will go into an infinite loop continually trying to get an E. Fortunately, there is a simple transformation that eliminates the left recursion; this transformation also eliminates any confusion about which rule to pick.

Grammar 3 (Figure 3) is an even more difficult example. Here there is no simple transformation to apply. The language generated by the grammar consists of a sequence of less-than signs followed by matched nested less-than and greater-than signs. Any less-than sign could potentially start the matched portion; the parser would have to guess which less-than sign it was. Deterministic recursive descent parsing cannot handle this grammar[5].

2.2 Bottom-Up Parsing

2.2.1 LR Parsing

LR parsing[1, 5] refers to a family of bottom-up parser generator algorithms. The reason that these techniques are used to build parser generators rather than actual parsers is that it is too difficult to construct parsers manually using these techniques. LR based parsers all shift tokens from the token input stream onto the parse stack. An oracle then decides if what is on the top of the stack can be reduced using one of the productions in the grammar. The oracle is realized as a finite state machine. Differences in the techniques are determined by the way look-ahead is incorporated into the oracle; these differences affect the size of the parser generated.

LR parsing is powerful; LR based parsers can handle all of the example grammars. However, as mentioned earlier, they are very difficult to write by hand, and the parser code is even more difficult to understand. A table representing the SLR oracle for the parser for grammar 2 (Figure 2) is in figure 4. Note that the table has many states even for this simple grammar.

2.2.2 Classic Recursive Ascent Parsing

While it is possible to generate table-driven top-down parsers, it is also possible to construct procedure-based LR parsers.

	action			*goto*
state	*n*	+	$	*E*
0	*s*4			1
1	*s*4		*acc*	2
2		*s*3		
3	*r*2	*r*2	*r*2	
4	*r*1	*r*1	*r*1	

Figure 4: SLR Parser Table

$$\begin{array}{lcl}
E & ::= & \langle\ E\ \rightarrow\ .n\ \rangle \\
E & ::= & \langle\ E\ \rightarrow\ .E\ E\ \texttt{+}\ \rangle \\
\langle\ n, E\ \rightarrow\ .n\ \rangle & ::= & \langle\ E\ \rightarrow\ n.\ \rangle \\
\langle\ E, E\ \rightarrow\ .E\ E\ \texttt{+}\ \rangle & ::= & \langle\ E\ \rightarrow\ E.\ E\ \texttt{+}\ \rangle \\
\langle\ E, E\ \rightarrow\ E\ .E\ \texttt{+}\ \rangle & ::= & \langle\ E\ \rightarrow\ E\ E.\ \texttt{+}\ \rangle \\
\langle\ \texttt{+}, E\ \rightarrow\ E\ E\ .\texttt{+}\ \rangle & ::= & \langle\ E\ \rightarrow\ E\ E\ \texttt{+}.\ \rangle \\
\langle\ E\ \rightarrow\ .E\ E\ \texttt{+}\ \rangle & ::= & n\ \langle\ n, E\ \rightarrow\ .E\ E\ \texttt{+}\ \rangle \\
\langle\ E\ \rightarrow\ E\ .E\ \texttt{+}\ \rangle & ::= & n\ \langle\ n, E\ \rightarrow\ E\ .E\ \texttt{+}\ \rangle \\
\langle\ E\ \rightarrow\ E\ E\ .\texttt{+}\ \rangle & ::= & \texttt{+}\ \langle\ \texttt{+}, E\ \rightarrow\ E\ E\ .\texttt{+}\ \rangle \\
\langle\ E\ \rightarrow\ .n\ \rangle & ::= & n\ \langle\ n, E\ \rightarrow\ .n\ \rangle \\
\langle\ n, E\ \rightarrow\ .E\ E\ \texttt{+}\ \rangle & ::= & \langle\ E\ \rightarrow\ n.\ \rangle\ \langle\ E, E\ \rightarrow\ .E\ E\ \texttt{+}\ \rangle \\
\langle\ n, E\ \rightarrow\ E\ .E\ \texttt{+}\ \rangle & ::= & \langle\ E\ \rightarrow\ n.\ \rangle\ \langle\ E, E\ \rightarrow\ E\ .E\ \texttt{+}\ \rangle \\
\langle\ E, E\ \rightarrow\ .E\ E\ \texttt{+}\ \rangle & ::= & \langle\ E\ \rightarrow\ E.E\texttt{+}\ \rangle\ \langle\ E, E\ \rightarrow\ .E\ E\ \texttt{+}\ \rangle \\
\langle\ E, E\ \rightarrow\ E\ .E\ \texttt{+}\ \rangle & ::= & \langle\ E\ \rightarrow\ E.E\texttt{+}\ \rangle\ \langle\ E, E\ \rightarrow\ E\ .E\ \texttt{+}\ \rangle \\
\langle\ E\ \rightarrow\ n.\ \rangle & ::= & \epsilon \\
\langle\ E\ \rightarrow\ E\ E\ \texttt{+}.\ \rangle & ::= & \epsilon
\end{array}$$

Figure 5: Leermaker's transformation of grammar 2

This technique is known as classic *recursive ascent parsing*[2], and can sometimes make for faster parsers[8]. Unfortunately, it does not make for more readable parsers, as the procedures in the recursive ascent parser correspond exactly to the states in the LR parsing table.

2.2.3 Leermaker's Approach to Recursive Ascent

Leermaker[9] takes a very different view of recursive ascent parsing. Instead of basing the procedures on states constructed in the traditional approach to LR parser construction, he uses a complex transformation to construct a new grammar that can be parsed with a recursive descent parser. This transformation involves items, just as the construction of the oracle does in the traditional LR approach. Unfortunately, his technique suffers from the same problems as LR parsers — parsers are difficult to read and write. The transformed grammar of grammar 2 (Figure 2), which can be parsed directly by a recursive descent parser, is in figure 5.

2.2.4 Recursive Ascent-Descent Parsing

Researchers, such as Horspool[8], have been concerned about the complexity and inscrutability of LR parsers. One approach, called left-corner parsing[3, 8], involves waiting until enough bottom-up information has been amassed that it is clear which production must be reduced. This often occurs before all the variables of the production's right-hand side are pushed onto the stack. In that case, top-down techniques can be used for the rest of the right-hand side. If the bottom-up part of the parser is written using recursive ascent, this technique is called *recursive ascent-descent parsing.* This approach is powerful, and it is how I would automate CPS recursive ascent parsing. However, while Horspool is interested in parser generators, I am focusing on writing parsers by hand.

3. REVIEW OF MONADS

Although I do not use monads, the parsers presented in this paper use operators that hide certain parameters. That style was inspired by monads, and so I review that topic here.

It is well known that the CPS-transform can be used to model control[12, 6]. This transform adds a continuation parameter to each term.

$$\begin{aligned}
\overline{x} &= \lambda k.(k\ x)\\
\overline{\lambda x.M} &= \lambda k.(k\ (\lambda x.\overline{M}))\\
\overline{(MN)} &= \lambda k.(\overline{M}\ (\lambda m.(\overline{N}\ (\lambda n.((m\ n)\ k)))))\\
\overline{\mathit{call/cc}} &= \lambda k.(k\ (\lambda f.\lambda c.((f\ (\lambda v.\lambda q.(c\ v)))\ c)))
\end{aligned}$$

Moggi[10] was the first to realize that adding parameters to hold onto information necessary for a particular side-effect, such as control, can be characterized mathematically as a monad. He and other researchers[4] point out that much of the complexity and plumbing can be hidden in the monadic operators. However, since the last line involves the continuation operator *call/cc*, it is possible to hide the plumbing only by introducing another operator *callcc*= $\lambda k.(k\ (\lambda f.\lambda c.((f\ (\lambda v.\lambda q.(c\ v)))\ c)))$. Following the recent convention[13], I will use the names *return* and *bind* for the monadic operators. These operators are subject to rules that prevent them from doing anything more than plumbing. We see that if $\mathit{return} = \lambda a.\lambda k.(k\ a)$, and $\mathit{bind} = \lambda c.\lambda f.\lambda k.(c(\lambda a.((f\ a)\ k)))$, the CPS-transform can be written without the k's. In this example, $(\mathit{bind}\ W\ (\lambda w.M))$ is abbreviated $(\mathit{bind}\ (w\ W)\ M)$.

$$\begin{aligned}
\overline{x} &= (\mathit{return}\ x)\\
\overline{\lambda x.M} &= (\mathit{return}\ (\lambda x.\overline{M}))\\
\overline{(M\ N)} &= (\mathit{bind}\ (m\ \overline{M})\ (\mathit{bind}\ (n\ \overline{N})\ (m\ n)))\\
\overline{\mathit{call/cc}} &= \mathit{callcc}
\end{aligned}$$

4. CPS RECURSIVE ASCENT PARSING

So far, there have been two approaches to recursive ascent parsing. The classic approach has a procedure for each state of the parsing automaton. Leermaker's approach involves transforming a grammar G to a new grammar G', which can be parsed using a recursive descent parser. Actually, the two are related: non-terminals in Leermaker's derived grammar G' are items of the original grammar G. CPS recursive ascent parsing is different from those methods in that is does not involve computing the items of the grammar.

The name "recursive ascent," like "recursive descent," is used here simply to describe the working of the parser. Further, the intuition behind the approach is similar to the intuition of recursive descent parsing — but turned on its head. With recursive descent parsing, the parser starts with the start symbol, works its way *down* to the terminals, and then comes back up. With CPS recursive ascent, the parser starts with the terminals and continues as appropriate *upwards* until the start symbol is reached.

Every parsing procedure has the following structure.

$$(\lambda\ (s\ f)\ (\lambda\ (ts\ s_{var}\ s_{val})\ M))$$

The parameters s and f are success and failure continuations, where a continuation has the form of the inner abstraction. These parameters are visible and are used explicitly as a means of combination. The inner abstraction

```
(define (get-E s f)
  (token-stream-empty??
   f
   (test-token-stream??
    token-plus?
    (popping
     (get-E (get-E (reduce 2 E make-sum s) f) f))
    (test-token-stream??
     number?
     (shift E s)
     f))))
```

Figure 6: Parser for Grammar 1

is hidden using a monadic style. It is not possible to use an actual monad because no values are returned due to the continuation passing style. The parameter *ts* is the token stream, s_{var} is a stack of grammar variables, and s_{val} is the stack of values determined by semantic actions. The parameters s_{var} and s_{val} could be represented by a single stack, but I found it simpler to have two stacks. I occasionally refer to s_{var} simply as "the parse stack." Using the parsing operators keeps the stacks consistent with each other.

There are only a few parsing operators: *token-stream-empty??*, *test-token-stream??*, *test-parse-stack??*, *shift*, *reduce*, *reduce-type*, and *popping*. I use the convention of writing a concluding double question-mark for any operator that tests one of the hidden parameters and then calls a success or failure continuation as a result. The meaning of each operator is fairly intuitive; their definitions can be found in appendix A.

Using the same three grammars illustrated earlier, I am going to give examples in Scheme[7] that show how to write CPS recursive ascent parsers. These examples will also demonstrate that CPS recursive ascent parsers are easy to read, and that they are more powerful than recursive descent parsers. Like recursive descent parsers, CPS recursive ascent parsers will often try to 'get' a grammar variable; however, there are occasions when there is no top-down predictor token. In those cases, the parser simply shifts a value onto the parse stack, and continues to a procedure representing the fact that the parser 'has' that value on the parse stack.

Grammar 1 (Figure 1) is a simple grammar that is easily parsed using recursive descent techniques. It should also be easy to parse using CPS recursive ascent techniques, and, indeed, it is. Focusing on the terminals, the parser (Figure 6) looks for a plus sign or a number. If it finds a plus sign, it continues by getting two E's and then reducing. If it finds a number it shifts it as an E, skipping an unnecessary step.

Grammar 2 (Figure 2) is more difficult to parse. A top-down parser cannot handle the grammar until it is rewritten so that the left recursion is eliminated. While the bottom-up approaches can handle the grammar as is, they generate complicated parsers. As we have seen, even the SLR approach generates many states, and the other approaches to recursive ascent do no better.

Whereas my parser code for the previous grammar was sim-

```
(define (get-n s f)
  (token-stream-empty??
   f
   (test-token-stream??
    number?
    (shift E s)
    f)))

(define (have-E s f)
  (token-stream-empty??
   s
   (test-token-stream??
    number?
    (shift E (have-EE s f))
    f)))

(define (have-EE s f)
  (token-stream-empty??
   f
   (test-token-stream??
    token-plus?
    (popping
     (reduce 2 E make-sum (test-parse-stack??
                           E-waiting?
                           (have-EE s f)
                           (have-E s f))))
    (test-token-stream??
     number?
     (shift E (have-EE s f))
     f))))

(define (get-E s f)
  (get-n (have-E s f) f))
```

Figure 7: Parser for Grammar 2

ilar to a top-down parser, here I will start to illustrate how to write the parser in a bottom-up style that does not eliminate left recursion. The procedures for this parser(Figure 7) are little or no more complicated than those for a recursive descent parser.

First I write a procedure that gets a number from the token stream. If it finds one, it shifts it onto the parse stack as an E; otherwise, it calls its failure continuation. Then I write two more procedures. One corresponds to the case where there is one E on the parse stack. If the token stream is empty, it succeeds; if it finds a number on the token stream, it shifts the number onto the parse stack as an E, and continues to the two-E case; otherwise, it calls its failure continuation. The other procedure corresponds to the case when there are two E's on the parse stack. If it finds a plus sign on the token stream, it pops off the two E's, replaces them with a single E, and then continues to the appropriate procedure based on what is on the parse stack; if it encounters a number, it continues to have two E's on the parse stack; otherwise it calls its failure continuation. One more procedure, *get-E*, glues the pieces of the parser together.

Grammar 3 (Figure 3) cannot be parsed by a top-down parser. A top-down parser cannot pick which rule of S to use, since both start with a less-than sign. The bottom-up approaches can handle the grammar, but as this grammar is more complicated than the previous two, there are even more states to contend with.

```
(define (get-< s f)
  (token-stream-empty??
   f
   (test-token-stream??
    left-token?
    (shift < (get-< s f))
    f)))

(define (have-M s f)
  (token-stream-empty??
   s
   (test-token-stream??
    right-token?
    (popping
     (test-parse-stack??
      <M-waiting?
      (reduce 2 M make-pair2 (have-M s f))
      f))
     f)))

(define (reduce-to-start s f)
  (test-parse-stack??
   just-S?
   s
   (test-parse-stack??
    <S-waiting?
    (reduce 2 S make-seq (reduce-to-start s f))
    (test-parse-stack??
     M-waiting?
     (reduce-type S (reduce-to-start s f))
     f))))


(define (get-S s f)
  (get-<
   s
   (token-stream-empty??
    f
    (test-token-stream??
     right-token?
     (popping
      (test-parse-stack??
       <-waiting?
       (reduce 1 M make-pair1 (have-M (reduce-to-start s f) f))
       f))
     f))))
```

Figure 8: Parser for Grammar 3

Using the techniques developed to parse grammar 2, I will show that CPS recursive ascent parsing is more powerful than recursive descent. Furthermore, this parser(Figure 8) will also be short and understandable, in contrast to previous bottom-up parsing techniques.

For this grammar, I first write a procedure that shifts less-than signs onto the parse stack; if there isn't one, it calls its failure continuation. I also write a procedure that corresponds to the case where there is an M on the top of the parse stack. If it finds a greater-than sign on the token stream, the token is popped off, and the M and the less-than sign below it are reduced to an M. If the token stream is empty, it succeeds; otherwise, it calls its failure continuation. There are two more procedures necessary to make things work. A reduction continuation procedure converts an M at the top of the parse stack to an S, and then reduces a less-than sign and an S to simply an S. Finally, procedure *get-S* plugs the continuations into the appropriate places to complete the parser.

5. CONCLUSION

While it has long been possible to write parsers by hand, and to parse a large class of grammars, it has not been possible to do both. A parser with both qualities is desirable, since it is easier to understand and modify a hand written parser. CPS recursive ascent parsing provides the best of both worlds, with all the advantages of a hand written parser and the ability to parse a larger class of grammars. Questions for future research include, which error handling strategies are most appropriate, how one can get the best performance, and how these techniques can be adapted for non-deterministic parsing.

6. ACKNOWLEDGMENTS

I would like to thank Melissa Nunes-Harwitt for carefully reading previous drafts of this paper.

7. REFERENCES

[1] A. V. Aho, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1985.

[2] F. E. J. K. Aretz. On a Recursive Ascent Parser. *Information Processing Letters*, Vol. 29, No. 3, 201-206, 1988.

[3] A. J. Demers. Generalized Left Corner Parsing. *Proc. 4th Symposium on Principles of Programming Languages*, 170-182, 1977.

[4] D. Espinosa. *Semantic Lego.* Doctoral thesis, Columbia University, 1995.

[5] C. N. Fischer, R. J. LeBlanc, Jr. *Crafting a Compiler with C.* Benjamin/Cummings, 1991.

[6] D. P. Friedman, M. Wand, C. T. Haynes. *Essentials of Programming Languages – 2nd Edition.* MIT Press, 2001.

[7] R. Kelsey, W. Clinger, and J. Rees editors. Revised^5 Report on the Algorithmic Language Scheme. *ACM SIGPLAN Notices*, Vol. 33, No. 9, 26-76, 1998.

[8] R. N. Horspool. Recursive Ascent-Descent Parsing. *Journal of Computer Languages*, Vol. 18, No. 1, 1-16, 1993.

[9] R. Leermakers. *The Functional Treatment of Parsing.* Kluwer Academic Publishers, 1993.

[10] E. Moggi. Notions of Computation and Monads. *Information and Computation*, Vol. 93, 55-92, 1991.

[11] C. Okasaki. Even Higher-Order Functions for Parsing or Why Would Anyone Ever Want to Use a Sixth-Order Function? *J. of Functional Programming*, Vol. 8, No. 2, 195-199, 1998.

[12] J. C. Reynolds. The Discoveries of Continuations. *LISP AND SYMBOLIC COMPUTATION: An International Journal*, Vol. 6, 233-247,1993.

[13] J. Sobel, E. Hilsdale, R. K. Dybvig, D. P. Friedman. Abstraction and Performance from Explicit Monadic Reflection. Unpublished manuscript, 1999.

[14] P. Wadler. How to Replace Failure by a List of Successes. *Conference on Function Programming Languages and Computer Architecture*, 113-128, 1985.

Appendix A

Helper code for any CPS recursive ascent parser.

```
(define peek-token-stream car)
(define pop-token-stream cdr)
(define empty-token-stream? null?)

(define top-stack car)
(define pop-stack cdr)
(define empty-stack? null?)
(define empty-stack '())

(define-syntax popping
  (syntax-rules ()
    ((popping ?k)
     (lambda (ts ps rs)
       (?k (pop-token-stream ts) ps rs)))))

(define-syntax shift
  (syntax-rules ()
    ((shift ?parse-var ?k)
     (lambda (ts ps rs)
       (?k (pop-token-stream ts)
           (cons (quote ?parse-var) ps)
           (cons (peek-token-stream ts) rs))))))

(define (remove n L)
  (if (= n 0)
    L
    (remove (- n 1) (cdr L))))

(define (retrieve n L)
  (let loop ((n n)
             (L L)
             (a '()))
    (if (= n 0)
      a
      (loop (- n 1) (cdr L) (cons (car L) a)))))

(define-syntax reduce
  (syntax-rules ()
    ((reduce ?n ?parse-var ?action ?k)
     (lambda (ts ps rs)
       (let ((m ?n))
         (?k ts
             (cons (quote ?parse-var) (remove m ps))
             (cons (apply ?action (retrieve m rs))
                   (remove m rs))))))))

(define-syntax reduce-type
  (syntax-rules ()
    ((reduce-type ?parse-var ?k)
     (lambda (ts ps rs)
       (?k
         ts
         (cons (quote ?parse-var)
               (pop-stack ps))
         rs)))))

(define-syntax token-stream-empty??
  (syntax-rules ()
    ((token-stream-empty?? s f)
     (lambda (ts ps rs)
       (if (empty-token-stream? ts)
   (s ts ps rs)
   (f ts ps rs))))))

(define-syntax test-parse-stack??
  (syntax-rules ()
    ((test-parse-stack?? ?test? s f)
     (lambda (ts ps rs)
       (if (?test? ps)
           (s ts ps rs)
           (f ts ps rs))))))

(define-syntax test-token-stream??
  (syntax-rules ()
    ((test-token-stream?? ?test? s f)
     (lambda (ts ps rs) ;assumes non-empty stream
       (if (?test? (peek-token-stream ts))
   (s ts ps rs)
   (f ts ps rs))))))

(define (init-s ts ps rs)
  (if (empty-token-stream? ts)
      (top-stack rs)
      (init-f ts ps rs)))

(define (init-f ts ps rs)
  (if (empty-token-stream? ts)
      (begin
        (display "Unexpected end of input.")
        (newline))
      (begin
        (display "Unexpected token ")
        (display (peek-token-stream ts))
        (newline)))
  ts)
```

Appendix B

Helper code for the example parsers.

Helper code for the parser for grammar 1.

```
(define (token-plus? t) (eq? t '+))

(define (make-sum r1 r2) `(+ ,r1 ,r2))

(define (parse ts)
  ((get-E init-s init-f) ts empty-stack empty-stack))
```

Helper code for the parser for grammar 2.

```
(define (token-plus? token) (eq? token '+))

(define (E-waiting? stack)
  (and (not (empty-stack? stack))
       (not (empty-stack? (pop-stack stack)))
       (eq? (top-stack (pop-stack stack)) 'E)))


(define (make-sum r1 r2) `(+ ,r1 ,r2))

(define (parse ts)
  ((get-E init-s init-f) ts empty-stack empty-stack))
```

Helper code for the parser for grammar 3.

```
(define (left-token? token) (eq? token '<))
(define (right-token? token) (eq? token '>))

(define (<-waiting? stack)
  (and (not (empty-stack? stack))
       (eq? (top-stack stack) '<)))

(define (M-waiting? stack)
  (and (not (empty-stack? stack))
       (eq? (top-stack stack) 'M)))

(define (<M-waiting? stack)
  (if (M-waiting? stack)
      (let ((s2 (pop-stack stack)))
(and (not (empty-stack? s2))
     (eq? (top-stack s2) '<)))
      #f))

(define (<S-waiting? stack)
  (and (not (empty-stack? stack))
       (eq? (top-stack stack) 'S)
       (let ((s2 (pop-stack stack)))
 (and (not (empty-stack? s2))
      (eq? (top-stack s2) '<)))))

(define (just-S? stack)
  (and (not (empty-stack? stack))
       (eq? (top-stack stack) 'S)
       (empty-stack? (pop-stack stack))))


(define (make-pair1 token) '(< >))

(define (make-pair2 token M) `(< ,M >))

(define (make-seq left right) (list left right))

(define (parse ts)
  ((get-S init-s init-f) ts empty-stack empty-stack))
```