Producing a Top-Down Parse Order with Bottom-Up Parsing* Technical Report 95-378

James P. Schmeiser† David T. Barnard‡
Department of Computing and Information Science
Queen's University, Kingston, Canada

March 6, 1995

Abstract

An adaptation of the standard LR(1) and LALR(1) parsing algorithm is described such that a top-down parse order is produced rather than the standard bottom-up order.

*This work was supported by the Natural Sciences and Engineering Research Council of Canada, the Information Technology Research Centre of Ontario and Queen's University at Kingston.

†James Schmeiser is a developer in the IBM Canada Laboratory C/C++ compiler group.

‡The paper was completed while the second author was on sabbatical at INRIA (Institut National de Recherche en Informatique et en Automatique), Rocquencourt, France.

1 Introduction

There are two standard approaches to parsing: top-down and bottom-up. In a typical top-down approach, such as LL(1) parsing [3], the input is predicted and verified against the actual input. Bottom-up parsers scan input until an entire rule is seen and is then recognised. This latter approach is employed by parsers for the LR(1) [2] and LALR(1) languages [1] (see [4] for more information on LL(1), LR(1) and LALR(1) parsing).

The parse orders, that is, the order of the rules describing the parse, are different depending on whether a top-down or bottom-up parse is employed. To a compiler writer, the top-down order (a left-most canonical parse) is often preferable to a bottom-up parse order (a right-most canonical parse); unfortunately, the LL(1) languages are a proper subset of the LALR(1) languages. Rather than limiting the expressiveness of the language by using a top-down parser, a method of generating a top-down parse order using a bottom-up parser is described.

The paper is organised as follows. Following definitions, a discussion relating parse orders and tree traversals is presented that illustrates how a top-down parse order is derived during a bottom-up parse. This is followed by the algorithm. Finally, a trace of the algorithm shows the computation of the top-down parse order using an example LALR(1) grammar.

2 Definitions

A grammar G is described by a 4-tuple $G = (V_T, V_N, S, \Phi)$ where V_T and V_N are the sets of terminal symbols and nonterminal symbols, respectively. $S \in V_N$ is the starting symbol of the grammar and Φ is the set of productions (grammar rules). A grammar is said to be *reduced* if all symbols in the grammar generate some (possibly null) part of a terminal string in the language, there are no rules of the form $A \rightarrow A$ in Φ and there exists some derivation $S \stackrel{+}{\Longrightarrow} A$ for all A in V_N (see [4] for more information on reduced grammars). A grammar is unambiguous if only one derivation exists for each string in the language. It is assumed hereafter that all grammars are reduced and unambiguous.

Standard parsing terminology is assumed. As well, the symbol immediately to the right of the dot in an item is called the *desired symbol*.

3 Parse Orders

A top-down parse order corresponds to a pre-order traversal of the nodes in the parse tree. Label each non-leaf node with a singleton list containing the number of the production expanded and leaf nodes with [ ]. Concatenating the lists as they are visited in a pre-order traversal results in the top-down parse order for the input string. Consider the grammar and parse tree shown in Figure 1. The nodes are labeled with the symbols of the grammar along with the appropriate list. The pre-order traversal of the tree is $[1, 2, 3, 4, 5, 6, 7]$ which is the top-down parse of the input string $abcdfghikno$.

The bottom-up parse order corresponds to a post-order left-to-right traversal of the parse tree. For the above example, concatenating the strings during the a post-order traversal results in the bottom-up parse $[3, 5, 4, 2, 7, 6, 1]$.

1. R → M Q
2. M → E f g L
3. E → a b c d
4. L → J k
5. J → h i
6. Q → n P
7. P → o

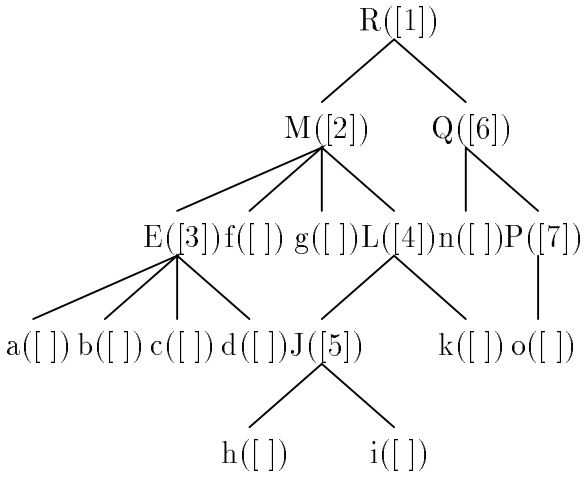

Figure 1: Example Grammar and Parse Tree

Generating a top-down parse order in a bottom-up parse reduces to concatenating the lists at each node in the order produced by a pre-order traversal during a post-order traversal of the parse tree. This is accomplished by computing at each node N (with children $N_1 \ldots N_n$) the result of the following computation.

$$list(N) = production(N) +\!\!+ list(N_1) +\!\!+ \ldots +\!\!+ list(N_n)$$

The list at the root of the tree corresponds to the list of nodes as they would be visited during a pre-order traversal of the parse tree and is the top-down parse order for the input string. Note that concatenation can be performed in constant time since the list at each node is only needed for the computation of the list at its parent.

4 Parsing Algorithm

The parsing algorithm uses the same tables as the standard algorithm, that is, an action and a goto table that are generated in the usual manner. The parsing algorithm, however, requires some modification. Rather than stacking the states of the machine as the parse proceeds, each stack element contains a state and rule list pair representing the rules that expand the desired symbol. When a transition is made on a symbol and a new pair placed on the stack, no information is known about the expansions of the new desired symbol so the rule list is empty.

When a rule is reduced, rather than outputting the rule number, the rule lists associated with the popped rule are concatenated. The rule list associated with the top-most stack pair is [ ] since reduce states have no desired symbol. As each pair is popped from the stack, its rule list is appended to the front of the rule list being constructed. When all of the appropriate pairs have been removed from the stack, the rule list of the new top of the stack is appended to the front of the list since

```
parse()
    S ← φ
    push(S, (0, [ ]))
    i ← 0
    while TRUE do
        (T, −) ← top(S)
        case action[T, input[i]] of
        ERROR:
            return -1
        SHIFT:
            push(S, (goto[T, input[i]], [ ]))
            i ← i + 1
        ACCEPT:
            (−, P) ← top(pop(S))
            return P
        REDUCE:
            P ← [ ]
            for j ← 1 to |rule| do
                (−, Q) ← pop(S)
                P ← Q ++P
            (T, Q) ← pop(S)
            push(S, (T, [rule] ++Q ++P))
            push(S, (goto[T, lhs[rule]], [ ]))
```

Figure 2: Parsing Algorithm

it represents the expansion of the left-most symbol in the recognised rule. The recognised rule is then placed at the head of the resulting list and a transition is made on the nonterminal symbol on the left-hand-side of the recognised rule. Finally, when the algorithm accepts, the entire rule list is available as the expansion of the padding symbol.

The algorithm is shown in Figure 2. It assumes the presence of the action and goto tables that are indexed by the states in the machine and the symbols in the grammar (see [4] for more information on the generation of these tables). The significant aspect of the algorithm is the way that reductions are handled.

Consider the LALR(1) example grammar and its parsing tables shown in Figure 3 which are taken from [4]. A trace of the parsing algorithm is shown in Figure 4.

Assuming that list concatenation is performed in constant time, the modified algorithm requires the same time as a typical LR(1) parsing algorithm. However, it requires more space since the original parsing algorithm can output the rules as they are recognised while the modified algorithm must store the rules until the end of the parse to do the re-ordering. As each rule is recognised, it is placed in a list and never replicated, so the algorithm requires $O(n)$ space, in addition to the normal requirements for the tables.

0. $S' \rightarrow \mathrm{S} \hookleftarrow$

1. $\mathrm{S} \rightarrow \mathrm{E} = \mathrm{E}$
2. $\quad | \quad \mathrm{f}$

3. $\mathrm{E} \rightarrow \mathrm{T}$
4. $\quad | \quad \mathrm{E} + \mathrm{T}$

5. $\mathrm{T} \rightarrow \mathrm{f}$
6. $\quad | \quad \mathrm{T} * \mathrm{f}$

	ACTION					*GOTO*						
STATE	f	$=$	$+$	$*$	$\hookleftarrow$	S	E	T	f	$=$	$+$	$*$
0	S					11	1	2	3			
1		S	S							4	5	
2		$R3$	$R3$	S	$R3$							9
3		$R5$	$R5$	$R5$	$R2$							
4	S						6	2	7			
5	S							8	7			
6			S		$R1$						5	
7		$R5$	$R5$	$R5$	$R5$							
8		$R4$	$R4$	S	$R4$							9
9	S								10			
10		$R6$	$R6$	$R6$	$R6$							
11					A							

Figure 3: Example LALR(1) Grammar and Tables

STACK	*INPUT*
$(0,[\,])$	$f = f + f * f \hookleftarrow$
$(0,[\,])(3,[\,])$	$= f + f * f \hookleftarrow$
$(0,[5])(2,[\,])$	$= f + f * f \hookleftarrow$
$(0,[3,5])(1,[\,])$	$= f + f * f \hookleftarrow$
$(0,[3,5])(1,[\,])(4,[\,])$	$f + f * f \hookleftarrow$
$(0,[3,5])(1,[\,])(4,[\,])(3,[\,])$	$+ f * f \hookleftarrow$
$(0,[3,5])(1,[\,])(4,[5])(2,[\,])$	$+ f * f \hookleftarrow$
$(0,[3,5])(1,[\,])(4,[3,5])(6,[\,])$	$+ f * f \hookleftarrow$
$(0,[3,5])(1,[\,])(4,[3,5])\ (6,[\,])(5,[\,])$	$f * f \hookleftarrow$
$(0,[3,5])(1,[\,])(4,[3,5])\ (6,[\,])(5,[\,])(7,[\,])$	$* f \hookleftarrow$
$(0,[3,5])(1,[\,])(4,[3,5])\ (6,[\,])(5,[5])(8,[\,])$	$* f \hookleftarrow$
$(0,[3,5])(1,[\,])(4,[3,5])\ (6,[\,])(5,[5])(8,[\,])(9,[\,])$	$f \hookleftarrow$
$(0,[3,5])(1,[\,])(4,[3,5])\ (6,[\,])(5,[5])(8,[\,])(9,[\,])(10,[\,])$	$\hookleftarrow$
$(0,[3,5])(1,[\,])(4,[3,5])\ (6,[\,])(5,[6,5])(8,[\,])$	$\hookleftarrow$
$(0,[3,5])(1,[\,])(4,[4,3,5,6,5])(6,[\,])$	$\hookleftarrow$
$(0,[1,3,5,4,3,5,6,5])(11,[\,])$	$\hookleftarrow$

Figure 4: Example Trace

5 Conclusions

A modification to the standard parsing algorithms for LR(1) and LALR(1) grammars that produces a top-down parse order rather than the standard bottom-up order was presented. The changes are minimal, do not increase the time required for the algorithm to run, and do not affect the table generation algorithms. The algorithm combines benefits of both top-down and bottom-up parsing by providing the more intuitive parse order of LL(1) with the increased descriptive power of LALR(1) and LR(1).

References

[1] F. L. DeRemer. *Practical Translators for LR(k) Languages.* PhD thesis, Cambridge, Massachusetts, 1969.

[2] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, November 1967.

[3] P. Lewis and R. Stearns. Syntax-directed transdution. *J. ACM*, 15(3):465–488, 1968.

[4] J.P. Tremblay and P.G. Sorenson. *The Theory and Practice of Compiler Writing.* McGraw-Hill, 1985.