

Semantic Analysis

4.5 Space Management for Attributes

A compiler that does not build an explicit parse tree requires some other mechanism to allocate, deallocate, and refer to storage space for attributes. In the two subsections below we consider attribute space management for bottom-up and top-down parsers, respectively. For bottom-up parsers the principal challenge is where to put the inherited attributes of symbols that have not yet been seen, and thus have no record in the parse stack. For top-down parsers this challenge does not arise, but we must go to a bit more effort to retain space for symbols that have already been parsed, and we must choose whether to manage this space automatically or to give some of the burden to the writer of action routines.

4.5.1 Bottom-Up Evaluation

EXAMPLE 4.19

Stack trace for bottom-up parse, with action routines

Figure ©4.17 shows a trace of the parse and attribute stack for $(1 + 3) * 2$, using the attribute grammar of Figure 4.1. For the sake of clarity, we show a single, combined stack for the parser and attribute evaluator, and we omit the CFSM state numbers.

It is easy to evaluate the attributes of symbols in this grammar, because the grammar is S-attributed. In an automatically generated parser, such as those produced by yacc/bison, the attribute rules associated with the productions of the grammar in Figure 4.1 would constitute action routines, to be executed when their productions are recognized. For yacc/bison, they would be written in C, with “pseudostructs” to name the attribute records of the symbols in each production. Attributes of the left-hand side symbol would be accessed as fields of the pseudostruct `$$`. Attributes of right-hand side symbols would be accessed as fields of the pseudostructs `$1`, `$2`, etc. To get from line 9 to line 10, for example, in the trace of Figure ©4.17, we would use an action routine version of the first rule of the grammar in Figure 4.1: `$$val = $1val + $3val`. ■

1. (
2. (1
3. (F₁
4. (T₁
5. (E₁
6. (E₁ +
7. (E₁ + 3
8. (E₁ + F₃
9. (E₁ + T₃
10. (E₄
11. (E₄)
12. F₄
13. T₄
14. T₄ *
15. T₄ * 2
16. T₄ * F₂
17. T₈
18. E₈

Figure 4.17 Parse/attribute stack trace for $(1 + 3) * 2$, using the grammar of Figure 4.1. Subscripts represent val attributes; they are not meant to distinguish among instances of a symbol.

When a bottom-up action routine is executed, the attribute records for symbols on the right-hand side of the production can be found in the top few entries of the attribute stack. The attribute record for the symbol on the left-hand side of the production (i.e., $$$$) will not yet lie in the stack: it is the task of the action routine to initialize this record. After the action routine completes, the parser pops the right-hand side records off the attribute stack and replaces them with $$$$. In yacc/bison, if no action routine is specified for a given production, the default action is to “copy” $\$1$ into $$$$. Since $$$$ will occupy the same location, once pushed, that $\$1$ occupied before being popped, this “copy” can be effected without doing any work.

Inherited Attributes

EXAMPLE 4.20

Finding inherited attributes in “buried” records

Unfortunately, it is not always easy to write an S-attributed grammar. A simple example in which inherited attributes are desirable arises in C or Fortran-style variable declarations, in which a type name precedes the list of variable names:

$$\begin{aligned} dec &\rightarrow type\ id_list \\ id_list &\rightarrow id \\ id_list &\rightarrow id_list\ ,\ id \end{aligned}$$

Let us assume that *type* has a synthesized attribute *tp* that contains a pointer to the symbol table entry for the type in question. Ideally, we should like to pass this attribute into *id_list* as an inherited attribute, so that we may enter each newly declared identifier into the symbol table, complete with type indication, as it is

encountered. When we recognize the production $id_list \rightarrow id$, we know that the top record on the attribute stack will be the one for id . But we know more than this: the next record down must be the one for $type$. To find the type of the new entry to be placed in the symbol table, we may safely inspect this “buried” record. Though it does not belong to a symbol of the current production, we can count on its presence because there is no other way to reach the $id_list \rightarrow id$ production.

Now what about the id in $id_list \rightarrow id_list, id$? This time the top three records on the attribute stack will be for the right-hand symbols id , $,$, and id_list . Immediately below them, however, we can still count on finding the entry for $type$, waiting for the id_list to be completed so that dec can be recognized. Using nonpositive indices for pseudostructs below the current production, we can write action routines as follows:

```

dec  $\rightarrow$  type id_list
id_list  $\rightarrow$  id { declare_id ($1.name, $0.tp) }
id_list  $\rightarrow$  id_list , id { declare_id ($3.name, $0.tp) }

```

Records deeper in the attribute stack could be accessed as $\$-1$, $\$-2$, and so on. While id_list appears in two places in this grammar fragment, both occurrences are guaranteed to lie above a $type$ record in the attribute stack, the first because it lies next to $type$ in a right-hand side, and the second by induction, because it is the beginning of the yield of the first. ■

Unfortunately, there are grammars in which a symbol that needs inherited attributes occurs in productions in which the underlying symbols are not the same. We can still handle inherited attributes in such cases, but only by modifying the underlying context-free grammar. An example can be found in languages like Perl, in which the meaning of an expression (and of the identifiers and operators within it) depends on the *context* in which that expression appears. Some Perl contexts expect arrays. Others expect numbers, strings, or Booleans. To correctly analyze an expression, we must pass the expectations of the context into the expression subtree as inherited attributes. Here is a grammar fragment that captures the problem:

```

stmt  $\rightarrow$  id := expr
       $\rightarrow$  ...
       $\rightarrow$  if expr then stmt
expr  $\rightarrow$  ...

```

Within the production for $expr$, the parser doesn’t know whether the surrounding context is an assignment or the condition of an *if* statement. If it is a condition, then the expected type of the expression is Boolean. If it is an assignment, then the expected type is that of the identifier on the assignment’s left-hand side. This identifier can be found two records below the current production in the attribute stack. ■

EXAMPLE 4.21

Grammar fragment
requiring context

EXAMPLE 4.22

Semantic hooks for context

Semantic Hooks

To allow these cases to be treated uniformly, we can add *semantic hook*, or “marker” symbols to the grammar. Semantic hooks generate ϵ , and thus do not alter the language defined by the grammar; their only purpose is to hold inherited attributes.

$$\begin{aligned} stmt &\longrightarrow id := A \text{ expr} \\ &\longrightarrow \dots \\ &\longrightarrow \text{if } B \text{ expr then } stmt \\ A &\longrightarrow \epsilon \{ \$$.tp := \$-1.tp \} \\ B &\longrightarrow \epsilon \{ \$$.tp := \text{Boolean} \} \\ \text{expr} &\longrightarrow \dots \{ \text{if } \$0.tp = \text{Boolean then } \dots \} \end{aligned}$$

Since the epsilon production for a semantic hook can provide an action routine, it is tempting to think of semantic hooks as a general technique to insert action routines in the middle of bottom-up productions. Unfortunately this is not the case: semantic hooks can be used only in places where the parser can be sure that it is in a given production. Placing a semantic hook anywhere else will break the “LR-ness” of the grammar, causing the parser generator to reject the modified grammar. Consider the following example:

EXAMPLE 4.23

Semantic hooks that break an LR CFG

1. $stmt \longrightarrow L.val := \text{expr}$
2. $\longrightarrow id \text{ args}$
3. $L.val \longrightarrow id \text{ quals}$
4. $quals \longrightarrow quals . id$
5. $\longrightarrow quals (\text{expr_list})$
6. $\longrightarrow \epsilon$
7. $\text{args} \longrightarrow (\text{expr_list})$
8. $\longrightarrow \epsilon$

An *l-value* in this grammar is a “qualified” identifier: an identifier followed by optional array subscript and record field qualifiers.¹ We have assumed that the language follows the notation of Fortran and Ada, in which parentheses delimit both procedure call arguments and array subscripts. In the case of procedure calls, it would be natural to want an action routine to pass the symbol-table index of the subroutine into the argument list as an inherited attribute, so that it can be used to check the number and types of arguments:

$$\begin{aligned} stmt &\longrightarrow id \ A \ \text{args} \\ A &\longrightarrow \epsilon \{ \$$.proc_index := \text{lookup} (\$0.name) \} \end{aligned}$$

¹ In general, an l-value in a programming language is anything to which a value can be assigned (i.e., anything that can appear on the left-hand side of an assignment). From a low-level point of view, this is basically an address. An r-value is anything that can appear on the right-hand side of an assignment. From a low level point of view, this is a value that can be stored at an address. We will discuss l-values and r-values further in Section 6.1.2.

If we try this, however, we will run into trouble, because the procedure call

```
foo(1, 2, 3);
```

and the array element assignment

```
foo(1, 2, 3) := 4;
```

begin with the same sequence of tokens. Until it sees the token after the closing parenthesis, the parser cannot tell whether it is working on production 1 or production 2. The presence of *A* in production 2 will therefore lead to a shift-reduce conflict; after seeing an *id*, the parser will not know whether to recognize *A* or shift (. ■

Left Corners

In general, the right-hand side of a production in a context-free grammar is said to consist of the *left corner* and the *trailing part*. In the left corner we cannot be sure which production we are parsing; in the trailing part the production is uniquely determined. In an LL(1) grammar, the left corner is always empty. In an LR(1) grammar, it can consist of up to the entire right-hand side. Semantic hooks can safely be inserted in the trailing part of a production, but not in the left corner. Yacc/bison recognizes this fact explicitly by allowing action routines to be embedded in right-hand sides. It automatically converts the production

$$S \longrightarrow \alpha \{ \text{your code here} \} \beta$$

to

$$S \longrightarrow \alpha A \beta$$

$$A \longrightarrow \epsilon \{ \text{your code here} \}$$

for some new, distinct symbol *A*. If the action routine is not in the trailing part, the resulting grammar will not be LALR(1), and yacc/bison will produce an error message. ■

EXAMPLE 4.24

Action routines in the trailing part

EXAMPLE 4.25

Left factoring in lieu of semantic hooks

In our procedure call and array subscript example, we cannot place a semantic hook before the *args* of production 2 because this location is in the left corner. If we wish to look up a procedure name in the symbol table before we parse the arguments, we will need to combine the productions for statements that can begin with an identifier, in a manner reminiscent of the *left factoring* discussed in Section 2.3.2:

$$stmt \longrightarrow id \ A \ \text{quals} \ assign_opt$$

$$A \longrightarrow \epsilon \{ \$$.id.index := lookup (\$0.name) \}$$

$$quals \longrightarrow quals \ . \ id$$

$$\longrightarrow quals \ (\ expr_list \)$$

$$\longrightarrow \epsilon$$

$$assign_opt \longrightarrow := \ expr$$

$$\longrightarrow \epsilon$$

This change eliminates the shift-reduce conflict, but at the expense of combining the entire grammar subtrees for procedure call arguments and array subscripts. To use the modified grammar we shall have to write action routines for *quals* that work for both kinds of constructs, and this can be a major nuisance. Users of LR-family parser generators often find that there is a tension between the desire for grammar clarity and parsability on the one hand, and the need for semantic hooks to set inherited attributes on the other. ■

4.5.2 Top-Down Evaluation

Top-down parsers, as discussed in Chapter 2, come in two principal varieties: recursive descent and table driven. Attribute management in recursive descent parsers is almost trivial: inherited attributes of symbol *foo* take the form of parameters passed into the parsing routine named *foo*; synthesized attributes are the return parameters. These synthesized attributes can then be passed as inherited attributes to symbols later in the current production, or returned as synthesized attributes of the current left-hand side.

Attribute space management for automatically generated top-down parsers is somewhat more complex. Because they allow action routines at arbitrary locations in a right-hand side, top-down parsers avoid the need to modify the grammar in order to insert semantic hooks. (Of course, because they must have empty left corners, top-down grammars can be harder to write in the first place.) Because the parse stack describes the future, instead of the past, we cannot employ an attribute stack that simply mirrors the parse stack. Our two principal options are to equip the parser with a (more complicated) algorithm for automatic space management, or to require action routines to manage space explicitly.

Automatic Management

Automatic management of attribute space for top-down parsing is more complicated than it is for bottom-up parsing. It is also more space intensive. We can still use an attribute stack, but it has to contain all of the symbols in all of the productions between the root of the (hypothetical) parse tree and the current point in the parse. All of the right-hand side symbols of a given production are adjacent in the stack; the left-hand side is buried in the right-hand side of a deeper (closer to the root) production.

EXAMPLE 4.26

Operation of an LL attribute stack

Figure 4.18 contains an LL(1) grammar for constant expressions, with action routines. Figure 4.19 uses this grammar to trace the operation of a top-down attribute stack on the sample input $(1 + 3) * 2$. The left-hand column shows the parse stack. The right-hand column shows the attribute stack. Three global pointers index into the attribute stack. One (shown as an “arrow-boxed” L in the trace) identifies the record in the attribute stack that holds the attributes of the left-hand side symbol of current production. The second (shown as an arrow-boxed R in the trace) identifies the first symbol on the right-hand side of the production. L and R allow the action routines to find the attributes of the symbols

$$\begin{aligned}
E &\longrightarrow T \{ TT.st := T.val \}^1 \quad TT \{ E.val := TT.val \}^2 \\
TT_1 &\longrightarrow + T \{ TT_2.st := TT_1.st + T.val \}^3 \quad TT_2 \{ TT_1.val := TT_2.val \}^4 \\
TT_1 &\longrightarrow - T \{ TT_2.st := TT_1.st - T.val \}^5 \quad TT_2 \{ TT_1.val := TT_2.val \}^6 \\
TT &\longrightarrow \epsilon \{ TT.val := TT.st \}^7 \\
T &\longrightarrow F \{ FT.st := F.val \}^8 \quad FT \{ T.val := FT.val \}^9 \\
FT_1 &\longrightarrow * F \{ FT_2.st := FT_1.st \times F.val \}^{10} \quad FT_2 \{ FT_1.val := FT_2.val \}^{11} \\
FT_1 &\longrightarrow / F \{ FT_2.st := FT_1.st \div F.val \}^{12} \quad FT_2 \{ FT_1.val := FT_2.val \}^{13} \\
FT &\longrightarrow \epsilon \{ FT.val := FT.st \}^{14} \\
F_1 &\longrightarrow - F_2 \{ F_1.val := - F_2.val \}^{15} \\
F &\longrightarrow (E) \{ F.val := E.val \}^{16} \\
F &\longrightarrow \text{const} \{ F.val := C.val \}^{17}
\end{aligned}$$

Figure 4.18 LL(1) grammar for constant expressions, with action routines. The boldface superscripts are for reference in Figure 4.19.

of the current production. The third pointer (shown as an arrow-boxed N in the trace) identifies the first symbol within the right-hand side that has not yet been completely parsed. It allows the parser to update L correctly when a production is predicted.

At any given time, the attribute stack contains all symbols of all productions on the path between the root of the parse tree and the symbol currently at the top of the parse stack. Figure 4.20 identifies these symbols graphically at the point in Figure 4.19 immediately above the eight elided lines. Symbols to the left in the parse tree have already been reclaimed; those to the right have yet to be allocated.

At start-up, the attribute stack contains a record for the goal symbol, pointed at by N. When we push the right-hand side of a predicted production onto the parse stack, we add an “end-of-production” marker, represented by a colon in the trace. At the same time, we push records for the right-hand-side symbols onto the attribute stack. (These are *added* to the attribute stack; they do not replace the left-hand side.) Prior to pushing these entries, we save the current L and R pointers in another stack (not shown). We then set L to the old N, and make R and N point to the newly pushed right-hand side.

When we see an action symbol at the top of the parse stack (shown in the trace as a small bold number), we pop it and execute the corresponding action routine. When we match a terminal at the top of the parse stack, we pop it and move N forward one record in the attribute stack. When we see an end-of-production marker at the top of the parse stack, we pop it, set N to the attribute record following the one currently pointed at by L, pop everything from R forward off of the attribute stack, and restore the most recently saved values of L and R. ■

It should be emphasized that while the trace is long and tedious, its complexity is completely hidden from the writer of action routines. Once the space management routines are integrated with the driver for a top-down parser generator, all the compiler writer sees is the grammar of Figure 4.18. In comparing Figures 4.17

E \$	$\boxed{N} E_?$
T1TT2:\$	$\boxed{L} E_? \boxed{R} \boxed{N} T_? TT_{?,?}$
F8FT9:1TT2:\$	$E_? \boxed{L} T_? TT_{?,?} \boxed{R} \boxed{N} F_? FT_{?,?}$
(E) 16:8FT9:1TT2:\$	$E_? T_? TT_{?,?} \boxed{L} F_? FT_{?,?} \boxed{R} \boxed{N} (E_?)$
E) 16:8FT9:1TT2:\$	$E_? T_? TT_{?,?} \boxed{L} F_? FT_{?,?} \boxed{R} (\boxed{N} E_?)$
T1TT2:) 16:8FT9:1TT2:\$	$E_? T_? TT_{?,?} F_? FT_{?,?} (\boxed{L} E_?) \boxed{R} \boxed{N} T_? TT_{?,?}$
F8FT9:1TT2:) 16:8FT9:1TT2:\$	$E_? T_? TT_{?,?} F_? FT_{?,?} (E_?) \boxed{L} T_? TT_{?,?} \boxed{R} \boxed{N} F_? FT_{?,?}$
C17:8FT9:1TT2:) 16:8FT9:1TT2:\$	$E_? T_? TT_{?,?} F_? FT_{?,?} (E_?) T_? TT_{?,?} \boxed{L} F_? FT_{?,?} \boxed{R} \boxed{N} C_1$
17:8FT9:1TT2:) 16:8FT9:1TT2:\$	$E_? T_? TT_{?,?} F_? FT_{?,?} (E_?) T_? TT_{?,?} \boxed{L} F_? FT_{?,?} \boxed{R} C_1 \boxed{N}$
:8FT9:1TT2:) 16:8FT9:1TT2:\$	$E_? T_? TT_{?,?} F_? FT_{?,?} (E_?) T_? TT_{?,?} \boxed{L} F_1 FT_{?,?} \boxed{R} C_1 \boxed{N}$
8FT9:1TT2:) 16:8FT9:1TT2:\$	$E_? T_? TT_{?,?} F_? FT_{?,?} (E_?) \boxed{L} T_? TT_{?,?} \boxed{R} F_1 \boxed{N} FT_{?,?}$
FT9:1TT2:) 16:8FT9:1TT2:\$	$E_? T_? TT_{?,?} F_? FT_{?,?} (E_?) \boxed{L} T_? TT_{?,?} \boxed{R} F_1 \boxed{N} FT_{1,?}$
14:9:1TT2:) 16:8FT9:1TT2:\$	$E_? T_? TT_{?,?} F_? FT_{?,?} (E_?) T_? TT_{?,?} F_1 \boxed{L} FT_{1,?} \boxed{R} \boxed{N}$
:9:1TT2:) 16:8FT9:1TT2:\$	$E_? T_? TT_{?,?} F_? FT_{?,?} (E_?) T_? TT_{?,?} F_1 \boxed{L} FT_{1,1} \boxed{R} \boxed{N}$
9:1TT2:) 16:8FT9:1TT2:\$	$E_? T_? TT_{?,?} F_? FT_{?,?} (E_?) \boxed{L} T_? TT_{?,?} \boxed{R} F_1 FT_{1,1} \boxed{N}$
:1TT2:) 16:8FT9:1TT2:\$	$E_? T_? TT_{?,?} F_? FT_{?,?} (E_?) \boxed{L} T_1 TT_{?,?} \boxed{R} F_1 FT_{1,1} \boxed{N}$
1TT2:) 16:8FT9:1TT2:\$	$E_? T_? TT_{?,?} F_? FT_{?,?} (\boxed{L} E_?) \boxed{R} T_1 \boxed{N} TT_{?,?}$
TT2:) 16:8FT9:1TT2:\$	$E_? T_? TT_{?,?} F_? FT_{?,?} (\boxed{L} E_?) \boxed{R} T_1 \boxed{N} TT_{1,?}$
+T3TT4:2:) 16:8FT9:1TT2:\$	$E_? T_? TT_{?,?} F_? FT_{?,?} (E_?) T_1 \boxed{L} TT_{1,?} \boxed{R} \boxed{N} + T_? TT_{?,?}$
T3TT4:2:) 16:8FT9:1TT2:\$	$E_? T_? TT_{?,?} F_? FT_{?,?} (E_?) T_1 \boxed{L} TT_{1,?} \boxed{R} + \boxed{N} T_? TT_{?,?}$
F8FT9:3TT4:2:) 16:8FT9:1TT2:\$	$E_? T_? TT_{?,?} F_? FT_{?,?} (E_?) T_1 TT_{1,?} + \boxed{L} T_? TT_{?,?} \boxed{R} \boxed{N} F_? FT_{?,?}$
C17:8FT9:3TT4:2:) 16:8FT9:1TT2:\$	$E_? T_? TT_{?,?} F_? FT_{?,?} (E_?) T_1 TT_{1,?} + T_? TT_{?,?} \boxed{L} F_? FT_{?,?} \boxed{R} \boxed{N} C_3$
(eight lines omitted)	
3TT4:2:) 16:8FT9:1TT2:\$	$E_? T_? TT_{?,?} F_? FT_{?,?} (E_?) T_1 \boxed{L} TT_{1,?} \boxed{R} + T_3 \boxed{N} TT_{?,?}$
TT4:2:) 16:8FT9:1TT2:\$	$E_? T_? TT_{?,?} F_? FT_{?,?} (E_?) T_1 \boxed{L} TT_{1,?} \boxed{R} + T_3 \boxed{N} TT_{4,?}$
7:4:2:) 16:8FT9:1TT2:\$	$E_? T_? TT_{?,?} F_? FT_{?,?} (E_?) T_1 TT_{1,?} + T_3 \boxed{L} TT_{4,?} \boxed{R} \boxed{N}$
:4:2:) 16:8FT9:1TT2:\$	$E_? T_? TT_{?,?} F_? FT_{?,?} (E_?) T_1 TT_{1,?} + T_3 \boxed{L} TT_{4,4} \boxed{R} \boxed{N}$
4:2:) 16:8FT9:1TT2:\$	$E_? T_? TT_{?,?} F_? FT_{?,?} (E_?) T_1 \boxed{L} TT_{1,?} \boxed{R} + T_3 TT_{4,4} \boxed{N}$
:2:) 16:8FT9:1TT2:\$	$E_? T_? TT_{?,?} F_? FT_{?,?} (E_?) T_1 \boxed{L} TT_{1,4} \boxed{R} + T_3 TT_{4,4} \boxed{N}$
2:) 16:8FT9:1TT2:\$	$E_? T_? TT_{?,?} F_? FT_{?,?} (\boxed{L} E_?) \boxed{R} T_1 TT_{1,4} \boxed{N}$
:) 16:8FT9:1TT2:\$	$E_? T_? TT_{?,?} F_? FT_{?,?} (\boxed{L} E_4) \boxed{R} T_1 TT_{1,4} \boxed{N}$
) 16:8FT9:1TT2:\$	$E_? T_? TT_{?,?} \boxed{L} F_? FT_{?,?} \boxed{R} (E_4 \boxed{N})$
16:8FT9:1TT2:\$	$E_? T_? TT_{?,?} \boxed{L} F_? FT_{?,?} \boxed{R} (E_4) \boxed{N}$
:8FT9:1TT2:\$	$E_? T_? TT_{?,?} \boxed{L} F_4 FT_{?,?} \boxed{R} (E_4) \boxed{N}$
8FT9:1TT2:\$	$E_? \boxed{L} T_? TT_{?,?} \boxed{R} F_4 \boxed{N} FT_{?,?}$
FT9:1TT2:\$	$E_? \boxed{L} T_? TT_{?,?} \boxed{R} F_4 \boxed{N} FT_{4,?}$
*F10FT11:9:1TT2:\$	$E_? T_? TT_{?,?} F_4 \boxed{L} FT_{4,?} \boxed{R} \boxed{N} * F_? FT_{?,?}$
F10FT11:9:1TT2:\$	$E_? T_? TT_{?,?} F_4 \boxed{L} FT_{4,?} \boxed{R} * \boxed{N} F_? FT_{?,?}$
C17:10FT11:9:1TT2:\$	$E_? T_? TT_{?,?} F_4 FT_{4,?} * \boxed{L} F_? FT_{?,?} \boxed{R} \boxed{N} C_2$
17:10FT11:9:1TT2:\$	$E_? T_? TT_{?,?} F_4 FT_{4,?} * \boxed{L} F_? FT_{?,?} \boxed{R} C_2 \boxed{N}$
:10FT11:9:1TT2:\$	$E_? T_? TT_{?,?} F_4 FT_{4,?} * \boxed{L} F_2 FT_{?,?} \boxed{R} C_2 \boxed{N}$
10FT11:9:1TT2:\$	$E_? T_? TT_{?,?} F_4 \boxed{L} FT_{4,?} \boxed{R} * F_2 \boxed{N} FT_{?,?}$
FT11:9:1TT2:\$	$E_? T_? TT_{?,?} F_4 \boxed{L} FT_{4,?} \boxed{R} * F_2 \boxed{N} FT_{8,?}$
(six lines omitted)	
1TT2:\$	$\boxed{L} E_? \boxed{R} T_8 \boxed{N} TT_{?,?}$
TT2:\$	$\boxed{L} E_? \boxed{R} T_8 \boxed{N} TT_{8,?}$
7:2:\$	$E_? T_8 \boxed{L} TT_{8,?} \boxed{R} \boxed{N}$
:2:\$	$E_? T_8 \boxed{L} TT_{8,8} \boxed{R} \boxed{N}$
2:\$	$\boxed{L} E_? \boxed{R} T_8 TT_{8,8} \boxed{N}$
: \$	$\boxed{L} E_8 \boxed{R} T_8 TT_{8,8} \boxed{N}$
\$	$E_8 \boxed{N}$

Figure 4.19 Trace of the parse stack (left) and attribute stack (right) for $(1 + 3) * 2$, using the grammar (and action routine numbers) of Figure ©4.18. Subscripts in the attribute stack indicate the values of attributes. For symbols with two attributes, st comes first.

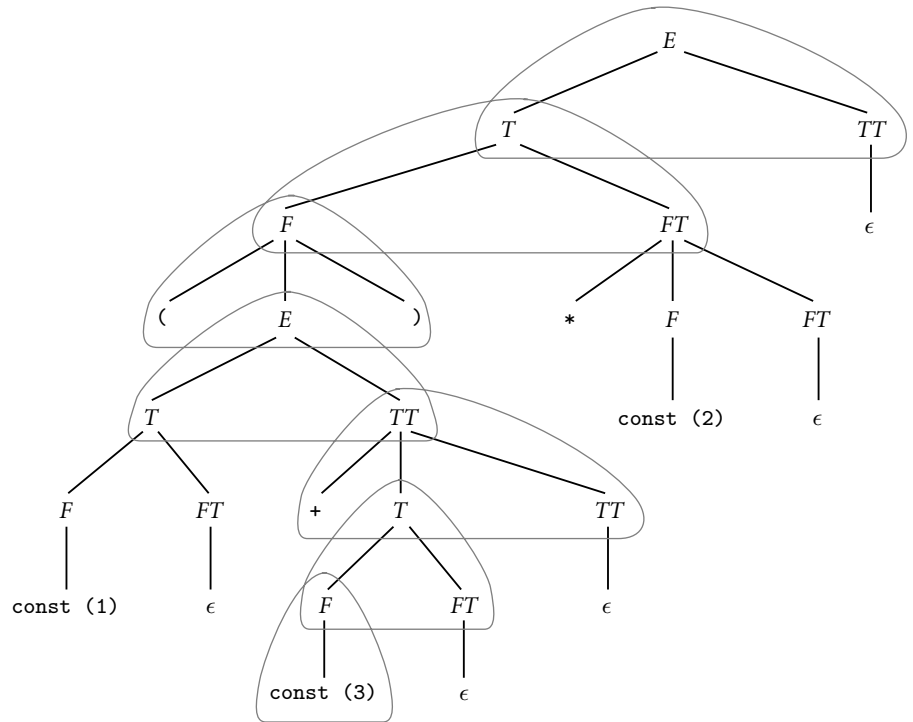


Figure 4.20 Productions with symbols currently in the attribute stack during a parse of $(1 + 3) * 2$ (using the grammar of Figure ©4.18), at the point where we are about to parse the 3. In Figure ©4.19, this point corresponds to the line immediately above the eight elided lines.

and ©4.19, one should also note that reduction and execution of a production's action routine are shown as a single step in the LR trace; they are shown separately in the LL trace, making that trace appear more complex than it really is.

Ad Hoc Management

One drawback of automatic space management for top-down grammars is the frequency with which the compiler writer must specify copy routines. Of the 17 action routines in Figure 4.9 or ©4.18, 12 simply move information from one place to another. The time required to execute these routines can be minimized by copying pointers, rather than large records, but compiler writers may still consider the copies a nuisance.

EXAMPLE 4.27

Ad hoc management of a semantic stack

An alternative is to manage space explicitly within the action routines, pushing and popping an ad hoc *semantic stack* only when information is generated or consumed. Using this technique, we can replace the action routines of Figure 4.9 with the simpler version shown in Figure ©4.21. Variable `cur_tok` is assumed to contain the synthesized attributes of the most recently matched token. The semantic stack contains pointers to syntax tree nodes. The `push_leaf` routine creates a node for a

$$\begin{aligned}
E &\longrightarrow T \ TT \\
TT &\longrightarrow + \ T \ \{ \text{bin_op } (" + ") \} \ TT \\
TT &\longrightarrow - \ T \ \{ \text{bin_op } (" - ") \} \ TT \\
TT &\longrightarrow \epsilon \\
T &\longrightarrow F \ FT \\
FT &\longrightarrow * \ F \ \{ \text{bin_op } (" \times ") \} \ FT \\
FT &\longrightarrow / \ F \ \{ \text{bin_op } (" \div ") \} \ FT \\
FT &\longrightarrow \epsilon \\
F &\longrightarrow - \ F \ \{ \text{un_op } (" + / - ") \} \\
F &\longrightarrow (\ E \) \\
F &\longrightarrow \text{const} \ \{ \text{push_leaf } (\text{cur_tok.val}) \}
\end{aligned}$$

Figure 4.21 Ad hoc management of attribute space in an LL(1) grammar to build a syntax tree.

specified constant and pushes a pointer to it onto the semantic stack. The `un_op` routine pops the top pointer off the stack, makes it the child of a newly created node for the specified unary operator, and pushes a pointer to that node back on the stack. The `bin_op` routine pops the top *two* pointers off the semantic stack and pushes a pointer to a newly created node for the specified binary operator. When the parse of E is completed, a pointer to a syntax tree describing its yield will be found in the top-most record on the semantic stack. ■

The advantage of ad hoc space management is clearly the smaller number of rules and the elimination of the inherited attributes used to represent left context. The disadvantage is that the compiler writer must be aware of what is in the semantic stack at all times, and must remember to push and pop it when appropriate.

One further advantage of an ad hoc semantic stack is that it allows action routines to push or pop an arbitrary number of records. With automatic space management, the number of records that can be seen by any one routine is limited by the number of symbols in the current production. The difference is particularly important in the case of productions that generate lists. In Section 4.5.1 we saw an SLR(1) grammar for declarations in the style of C and Fortran, in which the type name precedes the list of identifiers. Here is an LL(1) grammar fragment for a language in the style of Pascal and Ada, in which the variables precede the type:

EXAMPLE 4.28

Processing lists with an attribute stack

$$\begin{aligned}
dec &\longrightarrow id_list : type \\
id_list &\longrightarrow id \ id_list_tail \\
id_list_tail &\longrightarrow , \ id_list \\
&\longrightarrow \epsilon
\end{aligned}$$

Without resorting to non-L-attributed flow (see Exercise 4.26), we cannot pass the declared type into `id_list` as an inherited attribute. Instead, we must save up the list of identifiers and enter them into the symbol table *en masse* when the

type is finally encountered. With automatic management of space for attributes, the action routines would look something like this:

```

dec → id_list : type { declare_vars(id_list.chain, type.tp) }
id_list → id id_list_tail { id_list.chain := append(id.name, id_list_tail.chain) }
id_list_tail → , id_list { id_list_tail.chain := id_list.chain }
           → ε { id_list_tail.chain := null }

```

EXAMPLE 4.29

Processing lists with a semantic stack

With ad hoc management of space, we can get by without the linked list:

```

dec → { push(marker) }
      id_list : type
      { pop(tp)
        pop(name)
        while name ≠ marker
          declare_var(name, tp)
          pop(name) }
id_list → id { push(cur_tok.name) } id_list_tail
id_list_tail → , id_list
           → ε

```

Neither automatic nor ad hoc management of attribute space in top-down parsers is clearly superior to the other. The ad hoc approach eliminates the need for many copy rules and inherited attributes, and is consequently somewhat more time and space efficient. It also allows lists to be embedded in the semantic stack. On the other hand, it requires that the programmer who writes the action routines be continually aware of what is in the stack and why, in order to push and pop it appropriately. In the final analysis, the choice is mainly a matter of taste.

✓ CHECK YOUR UNDERSTANDING

17. Explain how to manage space for synthesized attributes in a bottom-up parser.
18. Explain how to manage space for inherited attributes in a bottom-up parser.
19. Define *left corner* and *trailing part*.
20. Under what circumstances can an action routine be embedded in the right-hand side of a production in a bottom-up parser? Equivalently, under what circumstances can a marker symbol be embedded in a right-hand side without rendering the grammar non-LR?
21. Summarize the tradeoffs between automatic and ad hoc management of space for attributes in a top-down parser.
22. At any given point in a top-down parse, which symbols will have attribute records in an automatically managed attribute stack?

