

## 3.5 PRACTICAL ISSUES

Even with automatic parser generators, the compiler writer must manage several issues to produce a robust, efficient parser for a real programming language. This section addresses several issues that arise in practice.

### 3.5.1 Error Recovery

Programmers often compile code that contains syntax errors. In fact, compilers are widely accepted as the fastest way to discover such errors. In this application, the compiler must find as many syntax errors as possible in a single attempt at parsing the code. This requires attention to the parser's behavior in error states.

All of the parsers shown in this chapter have the same behavior when they encounter a syntax error: they report the problem and halt. This behavior prevents the compiler from wasting time trying to translate an incorrect program. However, it ensures that the compiler finds at most one syntax error per compilation. Such a compiler would make finding all the syntax errors in a file of program text a potentially long and painful process.

A parser should find as many syntax errors as possible in each compilation. This requires a mechanism that lets the parser recover from an error by moving to a state where it can continue parsing. A common way of achieving this is to select one or more words that the parser can use to synchronize the input with its internal state. When the parser encounters an error, it discards input symbols until it finds a synchronizing word and then resets its internal state to one consistent with the synchronizing word.

In an Algol-like language, with semicolons as statement separators, the semicolon is often used as a synchronizing word. When an error occurs, the parser calls the scanner repeatedly until it finds a semicolon. It then changes state to one that would have resulted from successful recognition of a complete statement, rather than an error.

In a recursive-descent parser, the code can simply discard words until it finds a semicolon. At that point, it can return control to the point where the routine that parses statements reports success. This may involve manipulating the runtime stack or using a nonlocal jump like C's `setjmp` and `longjmp`.

In an LR(1) parser, this kind of resynchronization is more complex. The parser discards input until it finds a semicolon. Next, it scans backward down the parse stack until it finds a state  $s$  such that `Goto[s, Statement]` is a valid, nonerror entry. The first such state on the stack represents the statement that

contains the error. The error recovery routine then discards entries on the stack above that state, pushes the state  $\text{Goto}[s, \text{Statement}]$  onto the stack and resumes normal parsing.

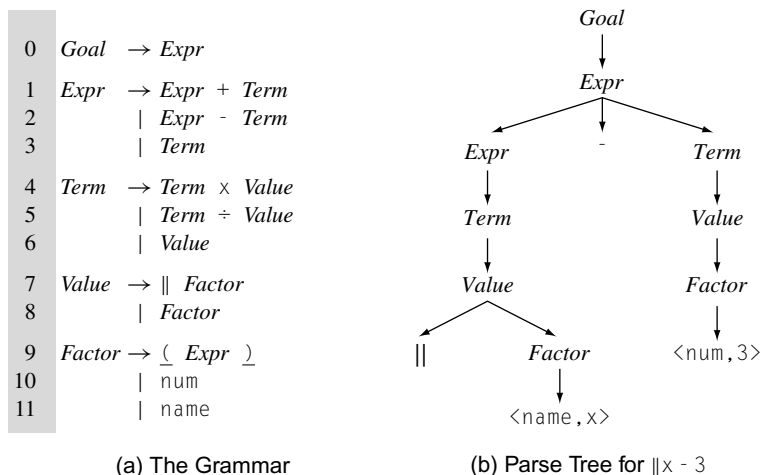
In a table-driven parser, either LL(1) or LR(1), the compiler needs a way of telling the parser generator where to synchronize. This can be done using error productions—a production whose right-hand side includes a reserved word that indicates an error synchronization point and one or more synchronizing tokens. With such a construct, the parser generator can construct error-recovery routines that implement the desired behavior.

Of course, the error-recovery routines should take steps to ensure that the compiler does not try to generate and optimize code for a syntactically invalid program. This requires simple handshaking between the error-recovery apparatus and the high-level driver that invokes the various parts of the compiler.

### 3.5.2 Unary Operators

The classic expression grammar includes only binary operators. Algebraic notation, however, includes unary operators, such as unary minus and absolute value. Other unary operators arise in programming languages, including autoincrement, autodecrement, address-of, dereference, boolean complement, and typecasts. Adding such operators to the expression grammar requires some care.

Consider adding a unary absolute-value operator,  $\|$ , to the classic expression grammar. Absolute value should have higher precedence than either  $\times$  or  $\div$ .



■ FIGURE 3.27 Adding Unary Absolute Value to the Classic Expression Grammar.

However, it needs a lower precedence than *Factor* to force evaluation of parenthetical expressions before application of  $\|$ . One way to write this grammar is shown in Figure 3.27. With these additions, the grammar is still LR(1). It lets the programmer form the absolute value of a number, an identifier, or a parenthesized expression.

Figure 3.27b shows the parse tree for the string  $\|x - 3$ . It correctly shows that the code must evaluate  $\|x$  before performing the subtraction. The grammar does not allow the programmer to write  $\|\|x$ , as that makes little mathematical sense. It does, however, allow  $\|( \|x )$ , which makes as little sense as  $\|\|x$ .

The inability to write  $\|\|x$  hardly limits the expressiveness of the language. With other unary operators, however, the issue seems more serious. For example, a C programmer might need to write  $**p$  to dereference a variable declared as `char **p`; We can add a dereference production for *Value* as well:  $Value \rightarrow * Value$ . The resulting grammar is still an LR(1) grammar, even if we replace the  $x$  operator in  $Term \rightarrow Term \times Value$  with  $*$ , overloading the operator “ $*$ ” in the way that C does. This same approach works for unary minus.

### 3.5.3 Handling Context-Sensitive Ambiguity

Using one word to represent two different meanings can create a syntactic ambiguity. One example of this problem arose in the definitions of several early programming languages, including FORTRAN, PL/I, and Ada. These languages used parentheses to enclose both the subscript expressions of an array reference and the argument list of a subroutine or function. Given a textual reference, such as `fee(i, j)`, the compiler cannot tell if `fee` is a two-dimensional array or a procedure that must be invoked. Differentiating between these two cases requires knowledge of `fee`’s declared type. This information is not syntactically obvious. The scanner undoubtedly classifies `fee` as a name in either case. A function call and an array reference can appear in many of the same situations.

Neither of these constructs appears in the classic expression grammar. We can add productions that derive them from *Factor*.

$$\begin{array}{l}
 \textit{Factor} \qquad \qquad \rightarrow \textit{FunctionReference} \\
 \qquad \qquad \qquad \qquad | \textit{ArrayReference} \\
 \qquad \qquad \qquad \qquad | ( \textit{Expr} ) \\
 \qquad \qquad \qquad \qquad | \textit{num} \\
 \qquad \qquad \qquad \qquad | \textit{name} \\
 \textit{FunctionReference} \rightarrow \textit{name} ( \textit{ArgList} ) \\
 \textit{ArrayReference} \quad \rightarrow \textit{name} ( \textit{ArgList} )
 \end{array}$$

Since the last two productions have identical right-hand sides, this grammar is ambiguous, which creates a reduce-reduce conflict in an LR(1) table builder.

Resolving this ambiguity requires extra-syntactic knowledge. In a recursive-descent parser, the compiler writer can combine the code for *FunctionReference* and *ArrayReference* and add the extra code required to check the name's declared type. In a table-driven parser built with a parser generator, the solution must work within the framework provided by the tools.

Two different approaches have been used to solve this problem. The compiler writer can rewrite the grammar to combine both the function invocation and the array reference into a single production. In this scheme, the issue is deferred until a later step in translation, when it can be resolved with information from the declarations. The parser must construct a representation that preserves all the information needed by either resolution; the later step will then rewrite the reference to its appropriate form as an array reference or as a function invocation.

Alternatively, the scanner can classify identifiers based on their declared types, rather than their microsyntactic properties. This classification requires some hand-shaking between the scanner and the parser; the coordination is not hard to arrange as long as the language has a define-before-use rule. Since the declaration is parsed before the use occurs, the parser can make its internal symbol table available to the scanner to resolve identifiers into distinct classes, such as `variable-name` and `function-name`. The relevant productions become:

$$\begin{aligned} \textit{FunctionReference} &\rightarrow \text{function-name } ( \textit{ArgList} ) \\ \textit{ArrayReference} &\rightarrow \text{variable-name } ( \textit{ArgList} ) \end{aligned}$$

Rewritten in this way, the grammar is unambiguous. Since the scanner returns a distinct syntactic category in each case, the parser can distinguish the two cases.

### 3.5.4 Left versus Right Recursion

As we have seen, top-down parsers need right-recursive grammars rather than left-recursive ones. Bottom-up parsers can accommodate either left or right recursion. Thus, the compiler writer must choose between left and right recursion in writing the grammar for a bottom-up parser. Several factors play into this decision.

## Stack Depth

In general, left recursion can lead to smaller stack depths. Consider two alternate grammars for a simple list construct, shown in Figures 3.28a and 3.28b. (Notice the similarity to the *SheepNoise* grammar.) Using these grammars to produce a five-element list leads to the derivations shown in Figures 3.28c and 3.28d, respectively. An LR(1) parser would construct these sequences in reverse. Thus, if we read the derivation from the bottom line to the top line, we can follow the parsers's actions with each grammar.

1. *Left-recursive grammar* This grammar shifts  $\text{elt}_1$  onto its stack and immediately reduces it to *List*. Next, it shifts  $\text{elt}_2$  onto the stack and reduces it to *List*. It proceeds until it has shifted each of the five  $\text{elt}_i$ s onto the stack and reduced them to *List*. Thus, the stack reaches a maximum depth of two and an average depth of  $\frac{10}{6} = 1\frac{2}{3}$ .
2. *Right-recursive grammar* This version shifts all five  $\text{elt}_i$ s onto its stack. Next, it reduces  $\text{elt}_5$  to *List* using rule two, and the remaining

$$\begin{array}{l} \text{List} \rightarrow \text{List elt} \\ \quad | \text{elt} \end{array}$$

(a) Left-Recursive Grammar

$$\begin{array}{l} \text{List} \rightarrow \text{elt List} \\ \quad | \text{elt} \end{array}$$

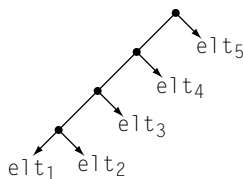
(b) Right-Recursive Grammar

```
List
List elt5
List elt4 elt5
List elt3 elt4 elt5
List elt2 elt3 elt4 elt5
elt1 elt2 elt3 elt4 elt5
```

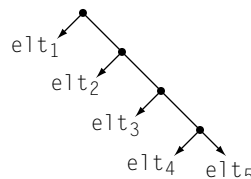
(c) Derivation with Left Recursion

```
List
elt1 List
elt1 elt2 List
elt1 elt2 elt3 List
elt1 elt2 elt3 elt4 List
elt1 elt2 elt3 elt4
elt5 List
```

(d) Derivation with Right Recursion



(e) AST with Left Recursion



(f) AST with Right Recursion

■ FIGURE 3.28 Left- and Right-Recursive List Grammars.

$elt_1$ s using rule one. Thus, its maximum stack depth will be five and its average will be  $\frac{20}{6} = 3\frac{1}{3}$ .

The right-recursive grammar requires more stack space; its maximum stack depth is bounded only by the length of the list. In contrast, the maximum stack depth with the left-recursive grammar depends on the grammar rather than the input stream.

For short lists, this is not a problem. If, however, the list represents the statement list in a long run of straight-line code, it might have hundreds of elements. In this case, the difference in space can be dramatic. If all other issues are equal, the smaller stack height is an advantage.

### Associativity

Left recursion naturally produces left associativity, and right recursion naturally produces right associativity. In some cases, the order of evaluation makes a difference. Consider the abstract syntax trees (ASTs) for the two five-element lists, shown in Figures 3.28e and 3.28f. The left-recursive grammar reduces  $elt_1$  to a *List*, then reduces *List*  $elt_2$ , and so on. This produces the AST shown on the left. Similarly, the right-recursive grammar produces the AST shown on the right.

For a list, neither of these orders is obviously incorrect, although the right-recursive AST may seem more natural. Consider, however, the result if we replace the list constructor with arithmetic operations, as in the grammars

$$\begin{array}{l}
 Expr \rightarrow Expr + Operand \\
 | Expr - Operand \\
 | Operand
 \end{array}
 \qquad
 \begin{array}{l}
 Expr \rightarrow Operand + Expr \\
 | Operand - Expr \\
 | Operand
 \end{array}$$

For the string  $x_1 + x_2 + x_3 + x_4 + x_5$  the left-recursive grammar implies a left-to-right evaluation order, while the right-recursive grammar implies a right-to-left evaluation order. With some number systems, such as floating-point arithmetic, these two evaluation orders can produce different results.

Since the mantissa of a floating-point number is small relative to the range of the exponent, addition can become an identity operation with two numbers that are far apart in magnitude. If, for example,  $x_4$  is much smaller than  $x_5$ , the processor may compute  $x_4 + x_5 = x_5$ . With well-chosen values, this effect can cascade and yield different answers from left-to-right and right-to-left evaluations.

Similarly, if any of the terms in the expression is a function call, then the order of evaluation may be important. If the function call changes the value

#### Abstract syntax tree

An AST is a contraction of the parse tree. See Section 5.2.1 on page 227.

of a variable in the expression, then changing the evaluation order might change the result.

In a string with subtractions, such as  $x_1 - x_2 + x_3$ , changing the evaluation order can produce incorrect results. Left associativity evaluates, in a postorder tree walk, to  $(x_1 - x_2) + x_3$ , the expected result. Right associativity, on the other hand, implies an evaluation order of  $x_1 - (x_2 + x_3)$ . The compiler must, of course, preserve the evaluation order dictated by the language definition. The compiler writer can either write the expression grammar so that it produces the desired order or take care to generate the intermediate representation to reflect the correct order and associativity, as described in Section 4.5.2.

#### SECTION REVIEW

Building a compiler involves more than just transcribing the grammar from some language definition. In writing down the grammar, many choices arise that have an impact on both the function and the utility of the resulting compiler. This section dealt with a variety of issues, ranging from how to perform error recovery through the tradeoff between left recursion and right recursion.

#### Review Questions

1. The programming language C uses square brackets to indicate an array subscript and parentheses to indicate a procedure or function argument list. How does this simplify the construction of a parser for C?
2. The grammar for unary absolute value introduced a new terminal symbol as the unary operator. Consider adding a unary minus to the classic expression grammar. Does the fact that the same terminal symbol occurs as either a unary minus or a binary minus introduce complications? Justify your answer.

### 3.6 ADVANCED TOPICS

To build a satisfactory parser, the compiler writer must understand the basics of engineering a grammar and a parser. Given a working parser, there are often ways of improving its performance. This section looks at two specific issues in parser construction. First, we examine transformations on the grammar that reduce the length of a derivation to produce a faster parse. These