4.1 THE ROLE OF THE PARSER

In our compiler model, the parser obtains a string of tokens from the lexical analyzer, as shown in Fig. 4.1, and verifies that the string can be generated by the grammar for the source language. We expect the parser to report any syntax errors in an intelligible fashion. It should also recover from commonly occurring errors so that it can continue processing the remainder of its input.

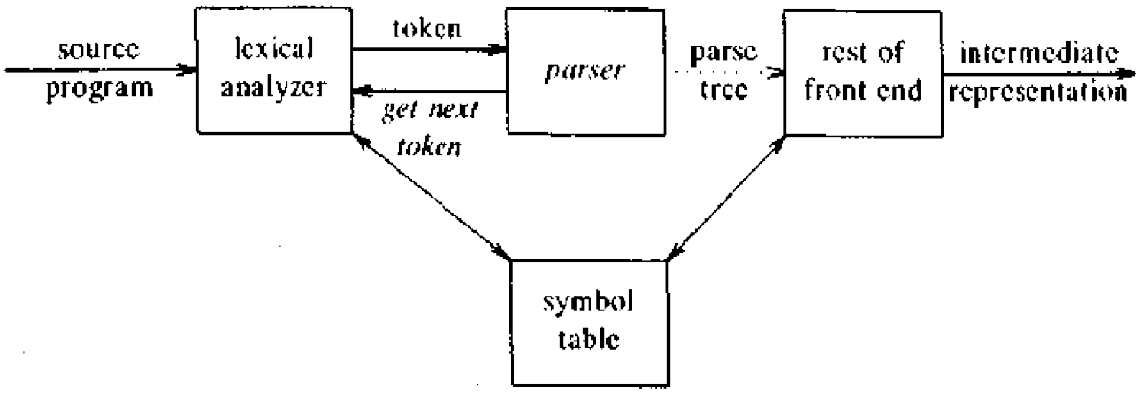

Fig. 4.1. Position of parser in compiler model.

There are three general types of parsers for grammars. Universal parsing methods such as the Cocke-Younger-Kasami algorithm and Earley's algorithm can parse any grammar (see the bibliographic notes). These methods, however, are too inefficient to use in production compilers. The methods commonly used in compilers are classified as being either top-down or bottom-up. As indicated by their names, top-down parsers build parse trees from the top (root) to the bottom (leaves), while bottom-up parsers start from the leaves and work up to the root. In both cases, the input to the parser is scanned from left to right, one symbol at a time.

The most efficient top-down and bottom-up methods work only on subclasses of grammars, but several of these subclasses, such as the LL and LR grammars, are expressive enough to describe most syntactic constructs in programming languages. Parsers implemented by hand often work with LL grammars; e.g., the approach of Section 2.4 constructs parsers for LL grammars. Parsers for the larger class of LR grammars are usually constructed by automated tools.

In this chapter, we assume the output of the parser is some representation of the parse tree for the stream of tokens produced by the lexical analyzer. In practice, there are a number of tasks that might be conducted during parsing, such as collecting information about various tokens into the symbol table, performing type checking and other kinds of semantic analysis, and generating intermediate code as in Chapter 2. We have lumped all of these activities into the "rest of front end" box in Fig. 4.1. We shall discuss these activities in detail in the next three chapters.

In the remainder of this section, we consider the nature of syntactic errors and general strategies for error recovery. Two of these strategies, called panic-mode and phrase-level recovery, are discussed in more detail together with the individual parsing methods. The implementation of each strategy calls upon the compiler writer's judgment, but we shall give some hints regarding approach.

Syntax Error Handling

If a compiler had to process only correct programs, its design and implementation would be greatly simplified. But programmers frequently write incorrect programs, and a good compiler should assist the programmer in identifying and locating errors. It is striking that although errors are so commonplace, few languages have been designed with error handling in mind. Our civilization would be radically different if spoken languages had the same requirement for syntactic accuracy as computer languages. Most programming language specifications do not describe how a compiler should respond to errors; the response is left to the compiler designer. Planning the error handling right from the start can both simplify the structure of a compiler and improve its response to errors.

We know that programs can contain errors at many different levels. For example, errors can be

- lexical, such as misspelling an identifier, keyword, or operator
- syntactic, such as an arithmetic expression with unbalanced parentheses
- semantic, such as an operator applied to an incompatible operand
- logical, such as an infinitely recursive call

Often much of the error detection and recovery in a compiler is centered around the syntax analysis phase. One reason for this is that many errors are syntactic in nature or are exposed when the stream of tokens coming from the lexical analyzer disobeys the grammatical rules defining the programming language. Another is the precision of modern parsing methods; they can detect the presence of syntactic errors in programs very efficiently. Accurately detecting the presence of semantic and logical errors at compile time is a much more difficult task. In this section, we present a few basic techniques for recovering from syntax errors; their implementation is discussed in conjunction with the parsing methods in this chapter.

The error handler in a parser has simple-to-state goals:

- It should report the presence of errors clearly and accurately.
- It should recover from each error quickly enough to be able to detect subsequent errors.
- It should not significantly slow down the processing of correct programs.

The effective realization of these goals presents difficult challenges.

Fortunately, common errors are simple ones and a relatively straightforward

error-handling mechanism often suffices. In some cases, however, an error may have occurred long before the position at which its presence is detected, and the precise nature of the error may be very difficult to deduce. In difficult cases, the error handler may have to guess what the programmer had in mind when the program was written.

Several parsing methods, such as the LL and LR methods, detect an error as soon as possible. More precisely, they have the *viable-prefix property*, meaning they detect that an error has occurred as soon as they see a prefix of the input that is not a prefix of any string in the language.

Example 4.1. To gain an appreciation of the kinds of errors that occur in practice, let us examine the errors Ripley and Druseikis [1978] found in a sample of student Pascal programs.

They discovered that errors do not occur that frequently; 60% of the programs compiled were syntactically and semantically correct. Even when errors did occur, they were quite sparse; 80% of the statements having errors had only one error, 13% had two. Finally, most errors were trivial; 90% were single token errors.

Many of the errors could be classified simply: 60% were punctuation errors, 20% operator and operand errors, 15% keyword errors, and the remaining five per cent other kinds. The bulk of the punctuation errors revolved around the incorrect use of semicolons.

For some concrete examples, consider the following Pascal program.

```
(1)     program prmax(input, output);
(2)     var
(3)         x, y: integer;
(4)     function max(i:integer; j:integer) : integer;
(5)     { return maximum of integers i and j }
(6)     begin
(7)         if i > j then max := i
(8)         else max := j
(9)     end;
(10)    begin
(11)        readln (x,y);
(12)        writeln (max(x,y))
(13)    end.
```

A common punctuation error is to use a comma in place of the semicolon in the argument list of a function declaration (e.g., using a comma in place of the first semicolon on line (4)); another is to leave out a mandatory semicolon at the end of a line (e.g., the semicolon at the end of line (4)); another is to put in an extraneous semicolon at the end of a line before an `else` (e.g., putting a semicolon at the end of line (7)).

Perhaps one reason why semicolon errors are so common is that the use of semicolons varies greatly from one language to another. In Pascal, a

semicolon is a statement separator; in PL/1 and C, it is a statement terminator. Some studies have suggested that the latter usage is less error prone (Gannon and Horning [1975]).

A typical example of an operator error is to leave out the colon from `:=`. Misspellings of keywords are usually rare, but leaving out the `i` from `writeln` would be a representative example.

Many Pascal compilers have no difficulty handling common insertion, deletion, and mutation errors. In fact, several Pascal compilers will correctly compile the above program with a common punctuation or operator error; they will issue only a warning diagnostic, pinpointing the offending construct.

However, another common type of error is much more difficult to repair correctly. This is a missing `begin` or `end` (e.g., line (9) missing). Most compilers would not try to repair this kind of error. □

How should an error handler report the presence of an error? At the very least, it should report the place in the source program where an error is detected because there is a good chance that the actual error occurred within the previous few tokens. A common strategy employed by many compilers is to print the offending line with a pointer to the position at which an error is detected. If there is a reasonable likelihood of what the error actually is, an informative, understandable diagnostic message is also included; e.g., "semicolon missing at this position."

Once an error is detected, how should the parser recover? As we shall see, there are a number of general strategies, but no one method clearly dominates. In most cases, it is not adequate for the parser to quit after detecting the first error, because subsequent processing of the input may reveal additional errors. Usually, there is some form of error recovery in which the parser attempts to restore itself to a state where processing of the input can continue with a reasonable hope that correct input will be parsed and otherwise handled correctly by the compiler.

An inadequate job of recovery may introduce an annoying avalanche of "spurious" errors, those that were not made by the programmer, but were introduced by the changes made to the parser state during error recovery. In a similar vein, syntactic error recovery may introduce spurious semantic errors that will later be detected by the semantic analysis or code generation phases. For example, in recovering from an error, the parser may skip a declaration of some variable, say `zap`. When `zap` is later encountered in expressions, there is nothing syntactically wrong, but since there is no symbol-table entry for `zap`, a message "`zap` undefined" is generated.

A conservative strategy for a compiler is to inhibit error messages that stem from errors uncovered too close together in the input stream. After discovering one syntax error, the compiler should require several tokens to be parsed successfully before permitting another error message. In some cases, there may be too many errors for the compiler to continue sensible processing. (For example, how should a Pascal compiler respond to a Fortran program as