

# MODULE - 2

## Chapter 3

# Lexical Analysis

In this chapter we show how to construct a lexical analyzer. To implement a lexical analyzer by hand, it helps to start with a diagram or other description for the lexemes of each token. We can then write code to identify each occurrence of each lexeme on the input and to return information about the token identified.

We can also produce a lexical analyzer automatically by specifying the lexeme patterns to a *lexical-analyzer generator* and compiling those patterns into code that functions as a lexical analyzer. This approach makes it easier to modify a lexical analyzer, since we have only to rewrite the affected patterns, not the entire program. It also speeds up the process of implementing the lexical analyzer, since the programmer specifies the software at the very high level of patterns and relies on the generator to produce the detailed code. We shall introduce in Section 3.5 a lexical-analyzer generator called *Lex* (or *Flex* in a more recent embodiment).

We begin the study of lexical-analyzer generators by introducing regular expressions, a convenient notation for specifying lexeme patterns. We show how this notation can be transformed, first into nondeterministic automata and then into deterministic automata. The latter two notations can be used as input to a “driver,” that is, code which simulates these automata and uses them as a guide to determining the next token. This driver and the specification of the automaton form the nucleus of the lexical analyzer.

### 3.1 The Role of the Lexical Analyzer

As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program. The stream of tokens is sent to the parser for syntax analysis. It is common for the lexical analyzer to interact with the symbol table as well. When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table. In some cases, information regarding the

kind of identifier may be read from the symbol table by the lexical analyzer to assist it in determining the proper token it must pass to the parser.

These interactions are suggested in Fig. 3.1. Commonly, the interaction is implemented by having the parser call the lexical analyzer. The call, suggested by the *getNextToken* command, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next token, which it returns to the parser.

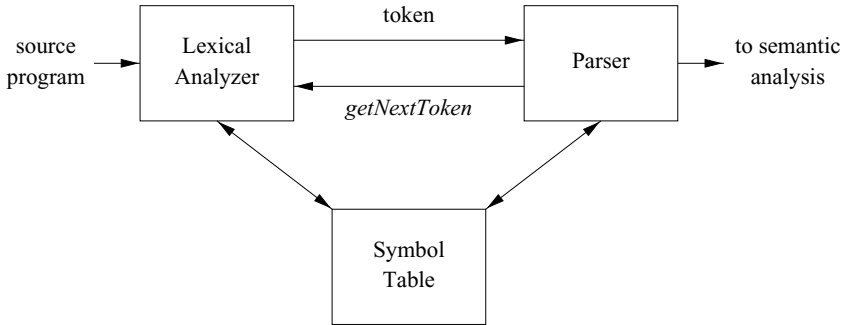


Figure 3.1: Interactions between the lexical analyzer and the parser

Since the lexical analyzer is the part of the compiler that reads the source text, it may perform certain other tasks besides identification of lexemes. One such task is stripping out comments and *whitespace* (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input). Another task is correlating error messages generated by the compiler with the source program. For instance, the lexical analyzer may keep track of the number of newline characters seen, so it can associate a line number with each error message. In some compilers, the lexical analyzer makes a copy of the source program with the error messages inserted at the appropriate positions. If the source program uses a macro-preprocessor, the expansion of macros may also be performed by the lexical analyzer.

Sometimes, lexical analyzers are divided into a cascade of two processes:

- a) *Scanning* consists of the simple processes that do not require tokenization of the input, such as deletion of comments and compaction of consecutive whitespace characters into one.
- b) *Lexical analysis* proper is the more complex portion, which produces tokens from the output of the scanner.

### 3.1.1 Lexical Analysis Versus Parsing

There are a number of reasons why the analysis portion of a compiler is normally separated into lexical analysis and parsing (syntax analysis) phases.

1. Simplicity of design is the most important consideration. The separation of lexical and syntactic analysis often allows us to simplify at least one of these tasks. For example, a parser that had to deal with comments and whitespace as syntactic units would be considerably more complex than one that can assume comments and whitespace have already been removed by the lexical analyzer. If we are designing a new language, separating lexical and syntactic concerns can lead to a cleaner overall language design.
2. Compiler efficiency is improved. A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing. In addition, specialized buffering techniques for reading input characters can speed up the compiler significantly.
3. Compiler portability is enhanced. Input-device-specific peculiarities can be restricted to the lexical analyzer.

### 3.1.2 Tokens, Patterns, and Lexemes

When discussing lexical analysis, we use three related but distinct terms:

- A *token* is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes. In what follows, we shall generally write the name of a token in boldface. We will often refer to a token by its token name.
- A *pattern* is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is *matched* by many strings.
- A *lexeme* is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

**Example 3.1 :** Figure 3.2 gives some typical tokens, their informally described patterns, and some sample lexemes. To see how these concepts are used in practice, in the C statement

```
printf("Total = %d\n", score);
```

both `printf` and `score` are lexemes matching the pattern for token **id**, and `"Total = %d\n"` is a lexeme matching **literal**. □

In many programming languages, the following classes cover most or all of the tokens:

| TOKEN             | INFORMAL DESCRIPTION                                                                                                     | SAMPLE LEXEMES                                               |
|-------------------|--------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------|
| <b>if</b>         | characters <code>i</code> , <code>f</code>                                                                               | <code>if</code>                                              |
| <b>else</b>       | characters <code>e</code> , <code>l</code> , <code>s</code> , <code>e</code>                                             | <code>else</code>                                            |
| <b>comparison</b> | <code>&lt;</code> or <code>&gt;</code> or <code>&lt;=</code> or <code>&gt;=</code> or <code>==</code> or <code>!=</code> | <code>&lt;=</code> , <code>!=</code>                         |
| <b>id</b>         | letter followed by letters and digits                                                                                    | <code>pi</code> , <code>score</code> , <code>D2</code>       |
| <b>number</b>     | any numeric constant                                                                                                     | <code>3.14159</code> , <code>0</code> , <code>6.02e23</code> |
| <b>literal</b>    | anything but <code>"</code> , surrounded by <code>"</code> 's                                                            | <code>"core dumped"</code>                                   |

Figure 3.2: Examples of tokens

1. One token for each keyword. The pattern for a keyword is the same as the keyword itself.
2. Tokens for the operators, either individually or in classes such as the token **comparison** mentioned in Fig. 3.2.
3. One token representing all identifiers.
4. One or more tokens representing constants, such as numbers and literal strings.
5. Tokens for each punctuation symbol, such as left and right parentheses, comma, and semicolon.

### 3.1.3 Attributes for Tokens

When more than one lexeme can match a pattern, the lexical analyzer must provide the subsequent compiler phases additional information about the particular lexeme that matched. For example, the pattern for token **number** matches both 0 and 1, but it is extremely important for the code generator to know which lexeme was found in the source program. Thus, in many cases the lexical analyzer returns to the parser not only a token name, but an attribute value that describes the lexeme represented by the token; the token name influences parsing decisions, while the attribute value influences translation of tokens after the parse.

We shall assume that tokens have at most one associated attribute, although this attribute may have a structure that combines several pieces of information. The most important example is the token **id**, where we need to associate with the token a great deal of information. Normally, information about an identifier — e.g., its lexeme, its type, and the location at which it is first found (in case an error message about that identifier must be issued) — is kept in the symbol table. Thus, the appropriate attribute value for an identifier is a pointer to the symbol-table entry for that identifier.

### Tricky Problems When Recognizing Tokens

Usually, given the pattern describing the lexemes of a token, it is relatively simple to recognize matching lexemes when they occur on the input. However, in some languages it is not immediately apparent when we have seen an instance of a lexeme corresponding to a token. The following example is taken from Fortran, in the fixed-format still allowed in Fortran 90. In the statement

```
D0 5 I = 1.25
```

it is not apparent that the first lexeme is D05I, an instance of the identifier token, until we see the dot following the 1. Note that blanks in fixed-format Fortran are ignored (an archaic convention). Had we seen a comma instead of the dot, we would have had a do-statement

```
D0 5 I = 1,25
```

in which the first lexeme is the keyword D0.

**Example 3.2:** The token names and associated attribute values for the Fortran statement

```
E = M * C ** 2
```

are written below as a sequence of pairs.

```
<id, pointer to symbol-table entry for E>
<assign_op>
<id, pointer to symbol-table entry for M>
<mult_op>
<id, pointer to symbol-table entry for C>
<exp_op>
<number, integer value 2>
```

Note that in certain pairs, especially operators, punctuation, and keywords, there is no need for an attribute value. In this example, the token **number** has been given an integer-valued attribute. In practice, a typical compiler would instead store a character-valued string representing the constant and use as an attribute value for **number** a pointer to that string. □

#### 3.1.4 Lexical Errors

It is hard for a lexical analyzer to tell, without the aid of other components, that there is a source-code error. For instance, if the string **fi** is encountered for the first time in a C program in the context:

```
fi ( a == f(x)) ...
```

a lexical analyzer cannot tell whether `fi` is a misspelling of the keyword `if` or an undeclared function identifier. Since `fi` is a valid lexeme for the token `id`, the lexical analyzer must return the token `id` to the parser and let some other phase of the compiler — probably the parser in this case — handle an error due to transposition of the letters.

However, suppose a situation arises in which the lexical analyzer is unable to proceed because none of the patterns for tokens matches any prefix of the remaining input. The simplest recovery strategy is “panic mode” recovery. We delete successive characters from the remaining input, until the lexical analyzer can find a well-formed token at the beginning of what input is left. This recovery technique may confuse the parser, but in an interactive computing environment it may be quite adequate.

Other possible error-recovery actions are:

1. Delete one character from the remaining input.
2. Insert a missing character into the remaining input.
3. Replace a character by another character.
4. Transpose two adjacent characters.

Transformations like these may be tried in an attempt to repair the input. The simplest such strategy is to see whether a prefix of the remaining input can be transformed into a valid lexeme by a single transformation. This strategy makes sense, since in practice most lexical errors involve a single character. A more general correction strategy is to find the smallest number of transformations needed to convert the source program into one that consists only of valid lexemes, but this approach is considered too expensive in practice to be worth the effort.

### 3.1.5 Exercises for Section 3.1

**Exercise 3.1.1:** Divide the following C++ program:

```
float limitedSquare(x) float x; {
    /* returns x-squared, but never more than 100 */
    return (x<=-10.0 || x>=10.0)?100:x*x;
}
```

into appropriate lexemes, using the discussion of Section 3.1.2 as a guide. Which lexemes should get associated lexical values? What should those values be?

**! Exercise 3.1.2:** Tagged languages like HTML or XML are different from conventional programming languages in that the punctuation (tags) are either very numerous (as in HTML) or a user-definable set (as in XML). Further, tags can often have parameters. Suggest how to divide the following HTML document:

```
Here is a photo of <B>my house</B>:
<P><IMG SRC = "house.gif"><BR>
See <A HREF = "morePix.html">More Pictures</A> if you
liked that one.<P>
```

into appropriate lexemes. Which lexemes should get associated lexical values, and what should those values be?

## 3.2 Input Buffering

Before discussing the problem of recognizing lexemes in the input, let us examine some ways that the simple but important task of reading the source program can be speeded. This task is made difficult by the fact that we often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme. The box on “Tricky Problems When Recognizing Tokens” in Section 3.1 gave an extreme example, but there are many situations where we need to look at least one additional character ahead. For instance, we cannot be sure we’ve seen the end of an identifier until we see a character that is not a letter or digit, and therefore is not part of the lexeme for **id**. In C, single-character operators like `-`, `=`, or `<` could also be the beginning of a two-character operator like `->`, `==`, or `<=`. Thus, we shall introduce a two-buffer scheme that handles large lookaheads safely. We then consider an improvement involving “sentinels” that saves time checking for the ends of buffers.

### 3.2.1 Buffer Pairs

Because of the amount of time taken to process characters and the large number of characters that must be processed during the compilation of a large source program, specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character. An important scheme involves two buffers that are alternately reloaded, as suggested in Fig. 3.3.

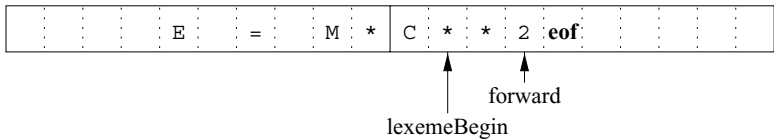


Figure 3.3: Using a pair of input buffers

Each buffer is of the same size  $N$ , and  $N$  is usually the size of a disk block, e.g., 4096 bytes. Using one system read command we can read  $N$  characters into a buffer, rather than using one system call per character. If fewer than  $N$  characters remain in the input file, then a special character, represented by **eof**,

marks the end of the source file and is different from any possible character of the source program.

Two pointers to the input are maintained:

1. Pointer **lexemeBegin**, marks the beginning of the current lexeme, whose extent we are attempting to determine.
2. Pointer **forward** scans ahead until a pattern match is found; the exact strategy whereby this determination is made will be covered in the balance of this chapter.

Once the next lexeme is determined, **forward** is set to the character at its right end. Then, after the lexeme is recorded as an attribute value of a token returned to the parser, **lexemeBegin** is set to the character immediately after the lexeme just found. In Fig. 3.3, we see **forward** has passed the end of the next lexeme, **\*\*** (the Fortran exponentiation operator), and must be retracted one position to its left.

Advancing **forward** requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move **forward** to the beginning of the newly loaded buffer. As long as we never need to look so far ahead of the actual lexeme that the sum of the lexeme's length plus the distance we look ahead is greater than  $N$ , we shall never overwrite the lexeme in its buffer before determining it.

### 3.2.2 Sentinels

If we use the scheme of Section 3.2.1 as described, we must check, each time we advance **forward**, that we have not moved off one of the buffers; if we do, then we must also reload the other buffer. Thus, for each character read, we make two tests: one for the end of the buffer, and one to determine what character is read (the latter may be a multiway branch). We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a *sentinel* character at the end. The sentinel is a special character that cannot be part of the source program, and a natural choice is the character **eof**.

Figure 3.4 shows the same arrangement as Fig. 3.3, but with the sentinels added. Note that **eof** retains its use as a marker for the end of the entire input. Any **eof** that appears other than at the end of a buffer means that the input is at an end. Figure 3.5 summarizes the algorithm for advancing **forward**. Notice how the first test, which can be part of a multiway branch based on the character pointed to by **forward**, is the only test we make, except in the case where we actually are at the end of a buffer or the end of the input.

## 3.3 Specification of Tokens

Regular expressions are an important notation for specifying lexeme patterns. While they cannot express all possible patterns, they are very effective in spec-

### Can We Run Out of Buffer Space?

In most modern languages, lexemes are short, and one or two characters of lookahead is sufficient. Thus a buffer size  $N$  in the thousands is ample, and the double-buffer scheme of Section 3.2.1 works without problem. However, there are some risks. For example, if character strings can be very long, extending over many lines, then we could face the possibility that a lexeme is longer than  $N$ . To avoid problems with long character strings, we can treat them as a concatenation of components, one from each line over which the string is written. For instance, in Java it is conventional to represent long strings by writing a piece on each line and concatenating pieces with a `+` operator at the end of each piece.

A more difficult problem occurs when arbitrarily long lookahead may be needed. For example, some languages like PL/I do not treat keywords as *reserved*; that is, you can use identifiers with the same name as a keyword like `DECLARE`. If the lexical analyzer is presented with text of a PL/I program that begins `DECLARE ( ARG1, ARG2, ...` it cannot be sure whether `DECLARE` is a keyword, and `ARG1` and so on are variables being declared, or whether `DECLARE` is a procedure name with its arguments. For this reason, modern languages tend to reserve their keywords. However, if not, one can treat a keyword like `DECLARE` as an ambiguous identifier, and let the parser resolve the issue, perhaps in conjunction with symbol-table lookup.

ifying those types of patterns that we actually need for tokens. In this section we shall study the formal notation for regular expressions, and in Section 3.5 we shall see how these expressions are used in a lexical-analyzer generator. Then, Section 3.7 shows how to build the lexical analyzer by converting regular expressions to automata that perform the recognition of the specified tokens.

#### 3.3.1 Strings and Languages

An *alphabet* is any finite set of symbols. Typical examples of symbols are letters, digits, and punctuation. The set  $\{0, 1\}$  is the *binary alphabet*. ASCII is an important example of an alphabet; it is used in many software systems. Uni-

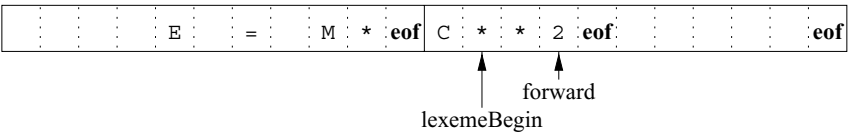


Figure 3.4: Sentinels at the end of each buffer

```

switch ( *forward++ ) {
    case eof:
        if (forward is at end of first buffer ) {
            reload second buffer;
            forward = beginning of second buffer;
        }
        else if (forward is at end of second buffer ) {
            reload first buffer;
            forward = beginning of first buffer;
        }
        else /* eof within a buffer marks the end of input */
            terminate lexical analysis;
        break;
    Cases for the other characters
}

```

Figure 3.5: Lookahead code with sentinels

### Implementing Multiway Branches

We might imagine that the switch in Fig. 3.5 requires many steps to execute, and that placing the case **eof** first is not a wise choice. Actually, it doesn't matter in what order we list the cases for each character. In practice, a multiway branch depending on the input character is made in one step by jumping to an address found in an array of addresses, indexed by characters.

code, which includes approximately 100,000 characters from alphabets around the world, is another important example of an alphabet.

A *string* over an alphabet is a finite sequence of symbols drawn from that alphabet. In language theory, the terms “sentence” and “word” are often used as synonyms for “string.” The length of a string  $s$ , usually written  $|s|$ , is the number of occurrences of symbols in  $s$ . For example, **banana** is a string of length six. The *empty string*, denoted  $\epsilon$ , is the string of length zero.

A *language* is any countable set of strings over some fixed alphabet. This definition is very broad. Abstract languages like  $\emptyset$ , the *empty set*, or  $\{\epsilon\}$ , the set containing only the empty string, are languages under this definition. So too are the set of all syntactically well-formed C programs and the set of all grammatically correct English sentences, although the latter two languages are difficult to specify exactly. Note that the definition of “language” does not require that any meaning be ascribed to the strings in the language. Methods for defining the “meaning” of strings are discussed in Chapter 5.

### Terms for Parts of Strings

The following string-related terms are commonly used:

1. A *prefix* of string  $s$  is any string obtained by removing zero or more symbols from the end of  $s$ . For example, **ban**, **banana**, and  $\epsilon$  are prefixes of **banana**.
2. A *suffix* of string  $s$  is any string obtained by removing zero or more symbols from the beginning of  $s$ . For example, **nana**, **banana**, and  $\epsilon$  are suffixes of **banana**.
3. A *substring* of  $s$  is obtained by deleting any prefix and any suffix from  $s$ . For instance, **banana**, **nan**, and  $\epsilon$  are substrings of **banana**.
4. The *proper* prefixes, suffixes, and substrings of a string  $s$  are those, prefixes, suffixes, and substrings, respectively, of  $s$  that are not  $\epsilon$  or not equal to  $s$  itself.
5. A *subsequence* of  $s$  is any string formed by deleting zero or more not necessarily consecutive positions of  $s$ . For example, **baan** is a subsequence of **banana**.

If  $x$  and  $y$  are strings, then the *concatenation* of  $x$  and  $y$ , denoted  $xy$ , is the string formed by appending  $y$  to  $x$ . For example, if  $x = \mathbf{dog}$  and  $y = \mathbf{house}$ , then  $xy = \mathbf{doghouse}$ . The empty string is the identity under concatenation; that is, for any string  $s$ ,  $\epsilon s = s\epsilon = s$ .

If we think of concatenation as a product, we can define the “exponentiation” of strings as follows. Define  $s^0$  to be  $\epsilon$ , and for all  $i > 0$ , define  $s^i$  to be  $s^{i-1}s$ . Since  $\epsilon s = s$ , it follows that  $s^1 = s$ . Then  $s^2 = ss$ ,  $s^3 = sss$ , and so on.

### 3.3.2 Operations on Languages

In lexical analysis, the most important operations on languages are union, concatenation, and closure, which are defined formally in Fig. 3.6. Union is the familiar operation on sets. The concatenation of languages is all strings formed by taking a string from the first language and a string from the second language, in all possible ways, and concatenating them. The (*Kleene*) *closure* of a language  $L$ , denoted  $L^*$ , is the set of strings you get by concatenating  $L$  zero or more times. Note that  $L^0$ , the “concatenation of  $L$  zero times,” is defined to be  $\{\epsilon\}$ , and inductively,  $L^i$  is  $L^{i-1}L$ . Finally, the positive closure, denoted  $L^+$ , is the same as the Kleene closure, but without the term  $L^0$ . That is,  $\epsilon$  will not be in  $L^+$  unless it is in  $L$  itself.

| OPERATION                           | DEFINITION AND NOTATION                                                   |
|-------------------------------------|---------------------------------------------------------------------------|
| <i>Union</i> of $L$ and $M$         | $L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$ |
| <i>Concatenation</i> of $L$ and $M$ | $LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$     |
| <i>Kleene closure</i> of $L$        | $L^* = \bigcup_{i=0}^{\infty} L^i$                                        |
| <i>Positive closure</i> of $L$      | $L^+ = \bigcup_{i=1}^{\infty} L^i$                                        |

Figure 3.6: Definitions of operations on languages

**Example 3.3:** Let  $L$  be the set of letters  $\{A, B, \dots, Z, a, b, \dots, z\}$  and let  $D$  be the set of digits  $\{0, 1, \dots, 9\}$ . We may think of  $L$  and  $D$  in two, essentially equivalent, ways. One way is that  $L$  and  $D$  are, respectively, the alphabets of uppercase and lowercase letters and of digits. The second way is that  $L$  and  $D$  are languages, all of whose strings happen to be of length one. Here are some other languages that can be constructed from languages  $L$  and  $D$ , using the operators of Fig. 3.6:

1.  $L \cup D$  is the set of letters and digits — strictly speaking the language with 62 strings of length one, each of which strings is either one letter or one digit.
2.  $LD$  is the set of 520 strings of length two, each consisting of one letter followed by one digit.
3.  $L^4$  is the set of all 4-letter strings.
4.  $L^*$  is the set of all strings of letters, including  $\epsilon$ , the empty string.
5.  $L(L \cup D)^*$  is the set of all strings of letters and digits beginning with a letter.
6.  $D^+$  is the set of all strings of one or more digits.

□

### 3.3.3 Regular Expressions

Suppose we wanted to describe the set of valid C identifiers. It is almost exactly the language described in item (5) above; the only difference is that the underscore is included among the letters.

In Example 3.3, we were able to describe identifiers by giving names to sets of letters and digits and using the language operators union, concatenation, and closure. This process is so useful that a notation called *regular expressions* has come into common use for describing all the languages that can be built from these operators applied to the symbols of some alphabet. In this notation, if *letter\_* is established to stand for any letter or the underscore, and *digit* is

established to stand for any digit, then we could describe the language of C identifiers by:

$$\textit{letter\_} ( \textit{letter\_} \mid \textit{digit} )^*$$

The vertical bar above means union, the parentheses are used to group subexpressions, the star means “zero or more occurrences of,” and the juxtaposition of *letter\_* with the remainder of the expression signifies concatenation.

The regular expressions are built recursively out of smaller regular expressions, using the rules described below. Each regular expression  $r$  denotes a language  $L(r)$ , which is also defined recursively from the languages denoted by  $r$ 's subexpressions. Here are the rules that define the regular expressions over some alphabet  $\Sigma$  and the languages that those expressions denote.

**BASIS:** There are two rules that form the basis:

1.  $\epsilon$  is a regular expression, and  $L(\epsilon)$  is  $\{\epsilon\}$ , that is, the language whose sole member is the empty string.
2. If  $a$  is a symbol in  $\Sigma$ , then  **$a$**  is a regular expression, and  $L(\mathbf{a}) = \{a\}$ , that is, the language with one string, of length one, with  $a$  in its one position. Note that by convention, we use italics for symbols, and boldface for their corresponding regular expression.<sup>1</sup>

**INDUCTION:** There are four parts to the induction whereby larger regular expressions are built from smaller ones. Suppose  $r$  and  $s$  are regular expressions denoting languages  $L(r)$  and  $L(s)$ , respectively.

1.  $(r)|(s)$  is a regular expression denoting the language  $L(r) \cup L(s)$ .
2.  $(r)(s)$  is a regular expression denoting the language  $L(r)L(s)$ .
3.  $(r)^*$  is a regular expression denoting  $(L(r))^*$ .
4.  $(r)$  is a regular expression denoting  $L(r)$ . This last rule says that we can add additional pairs of parentheses around expressions without changing the language they denote.

As defined, regular expressions often contain unnecessary pairs of parentheses. We may drop certain pairs of parentheses if we adopt the conventions that:

- a) The unary operator  $*$  has highest precedence and is left associative.
- b) Concatenation has second highest precedence and is left associative.

---

<sup>1</sup>However, when talking about specific characters from the ASCII character set, we shall generally use teletype font for both the character and its regular expression.

c)  $|$  has lowest precedence and is left associative.

Under these conventions, for example, we may replace the regular expression  $(\mathbf{a})|((\mathbf{b})^*(\mathbf{c}))$  by  $\mathbf{a|b^*c}$ . Both expressions denote the set of strings that are either a single  $a$  or are zero or more  $b$ 's followed by one  $c$ .

**Example 3.4:** Let  $\Sigma = \{a, b\}$ .

1. The regular expression  $\mathbf{a|b}$  denotes the language  $\{a, b\}$ .
2.  $\mathbf{(a|b)(a|b)}$  denotes  $\{aa, ab, ba, bb\}$ , the language of all strings of length two over the alphabet  $\Sigma$ . Another regular expression for the same language is  $\mathbf{aa|ab|ba|bb}$ .
3.  $\mathbf{a^*}$  denotes the language consisting of all strings of zero or more  $a$ 's, that is,  $\{\epsilon, a, aa, aaa, \dots\}$ .
4.  $\mathbf{(a|b)^*}$  denotes the set of all strings consisting of zero or more instances of  $a$  or  $b$ , that is, all strings of  $a$ 's and  $b$ 's:  $\{\epsilon, a, b, aa, ab, ba, bb, aaa, \dots\}$ . Another regular expression for the same language is  $\mathbf{(a^*b^*)^*}$ .
5.  $\mathbf{a|a^*b}$  denotes the language  $\{a, b, ab, aab, aaab, \dots\}$ , that is, the string  $a$  and all strings consisting of zero or more  $a$ 's and ending in  $b$ .

□

A language that can be defined by a regular expression is called a *regular set*. If two regular expressions  $r$  and  $s$  denote the same regular set, we say they are *equivalent* and write  $r = s$ . For instance,  $\mathbf{(a|b) = (b|a)}$ . There are a number of algebraic laws for regular expressions; each law asserts that expressions of two different forms are equivalent. Figure 3.7 shows some of the algebraic laws that hold for arbitrary regular expressions  $r$ ,  $s$ , and  $t$ .

| LAW                              | DESCRIPTION                                  |
|----------------------------------|----------------------------------------------|
| $r s = s r$                      | $ $ is commutative                           |
| $r (s t) = (r s) t$              | $ $ is associative                           |
| $r(st) = (rs)t$                  | Concatenation is associative                 |
| $r(s t) = rs rt; (s t)r = sr tr$ | Concatenation distributes over $ $           |
| $\epsilon r = r\epsilon = r$     | $\epsilon$ is the identity for concatenation |
| $r^* = (r \epsilon)^*$           | $\epsilon$ is guaranteed in a closure        |
| $r^{**} = r^*$                   | $*$ is idempotent                            |

Figure 3.7: Algebraic laws for regular expressions

### 3.3.4 Regular Definitions

For notational convenience, we may wish to give names to certain regular expressions and use those names in subsequent expressions, as if the names were themselves symbols. If  $\Sigma$  is an alphabet of basic symbols, then a *regular definition* is a sequence of definitions of the form:

$$\begin{array}{lll} d_1 & \rightarrow & r_1 \\ d_2 & \rightarrow & r_2 \\ & \dots & \\ d_n & \rightarrow & r_n \end{array}$$

where:

1. Each  $d_i$  is a new symbol, not in  $\Sigma$  and not the same as any other of the  $d$ 's, and
2. Each  $r_i$  is a regular expression over the alphabet  $\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$ .

By restricting  $r_i$  to  $\Sigma$  and the previously defined  $d$ 's, we avoid recursive definitions, and we can construct a regular expression over  $\Sigma$  alone, for each  $r_i$ . We do so by first replacing uses of  $d_1$  in  $r_2$  (which cannot use any of the  $d$ 's except for  $d_1$ ), then replacing uses of  $d_1$  and  $d_2$  in  $r_3$  by  $r_1$  and (the substituted)  $r_2$ , and so on. Finally, in  $r_n$  we replace each  $d_i$ , for  $i = 1, 2, \dots, n-1$ , by the substituted version of  $r_i$ , each of which has only symbols of  $\Sigma$ .

**Example 3.5:** C identifiers are strings of letters, digits, and underscores. Here is a regular definition for the language of C identifiers. We shall conventionally use italics for the symbols defined in regular definitions.

$$\begin{array}{lll} \textit{letter\_} & \rightarrow & \textit{A} \mid \textit{B} \mid \dots \mid \textit{Z} \mid \textit{a} \mid \textit{b} \mid \dots \mid \textit{z} \mid \_ \\ \textit{digit} & \rightarrow & 0 \mid 1 \mid \dots \mid 9 \\ \textit{id} & \rightarrow & \textit{letter\_} ( \textit{letter\_} \mid \textit{digit} )^* \end{array}$$

□

**Example 3.6:** Unsigned numbers (integer or floating point) are strings such as 5280, 0.01234, 6.336E4, or 1.89E-4. The regular definition

$$\begin{array}{lll} \textit{digit} & \rightarrow & 0 \mid 1 \mid \dots \mid 9 \\ \textit{digits} & \rightarrow & \textit{digit} \textit{digit}^* \\ \textit{optionalFraction} & \rightarrow & . \textit{digits} \mid \epsilon \\ \textit{optionalExponent} & \rightarrow & ( \textit{E} ( + \mid - \mid \epsilon ) \textit{digits} ) \mid \epsilon \\ \textit{number} & \rightarrow & \textit{digits} \textit{optionalFraction} \textit{optionalExponent} \end{array}$$

is a precise specification for this set of strings. That is, an *optionalFraction* is either a decimal point (dot) followed by one or more digits, or it is missing (the empty string). An *optionalExponent*, if not missing, is the letter E followed by an optional + or − sign, followed by one or more digits. Note that at least one digit must follow the dot, so *number* does not match 1., but does match 1.0.

□

### 3.3.5 Extensions of Regular Expressions

Since Kleene introduced regular expressions with the basic operators for union, concatenation, and Kleene closure in the 1950s, many extensions have been added to regular expressions to enhance their ability to specify string patterns. Here we mention a few notational extensions that were first incorporated into Unix utilities such as **Lex** that are particularly useful in the specification lexical analyzers. The references to this chapter contain a discussion of some regular-expression variants in use today.

1. *One or more instances.* The unary, postfix operator  $^+$  represents the positive closure of a regular expression and its language. That is, if  $r$  is a regular expression, then  $(r)^+$  denotes the language  $(L(r))^+$ . The operator  $^+$  has the same precedence and associativity as the operator  $*$ . Two useful algebraic laws,  $r^* = r^+|\epsilon$  and  $r^+ = rr^* = r^*r$  relate the Kleene closure and positive closure.
2. *Zero or one instance.* The unary postfix operator  $?$  means “zero or one occurrence.” That is,  $r?$  is equivalent to  $r|\epsilon$ , or put another way,  $L(r?) = L(r) \cup \{\epsilon\}$ . The  $?$  operator has the same precedence and associativity as  $*$  and  $^+$ .
3. *Character classes.* A regular expression  $a_1|a_2|\cdots|a_n$ , where the  $a_i$ ’s are each symbols of the alphabet, can be replaced by the shorthand  $[a_1a_2\cdots a_n]$ . More importantly, when  $a_1, a_2, \dots, a_n$  form a logical sequence, e.g., consecutive uppercase letters, lowercase letters, or digits, we can replace them by  $a_1\text{-}a_n$ , that is, just the first and last separated by a hyphen. Thus, **[abc]** is shorthand for **a|b|c**, and **[a-z]** is shorthand for **a|b|...|z**.

**Example 3.7:** Using these shorthands, we can rewrite the regular definition of Example 3.5 as:

$$\begin{aligned} \text{letter\_} &\rightarrow [\text{A-Za-z}] \\ \text{digit} &\rightarrow [0-9] \\ \text{id} &\rightarrow \text{letter\_} ( \text{letter\_} | \text{digit} )^* \end{aligned}$$

The regular definition of Example 3.6 can also be simplified:

$$\begin{aligned} \text{digit} &\rightarrow [0-9] \\ \text{digits} &\rightarrow \text{digit}^+ \\ \text{number} &\rightarrow \text{digits} ( . \text{ digits} )? ( \text{E} [+-]? \text{ digits} )? \end{aligned}$$

□

### 3.3.6 Exercises for Section 3.3

**Exercise 3.3.1:** Consult the language reference manuals to determine (i) the sets of characters that form the input alphabet (excluding those that may only appear in character strings or comments), (ii) the lexical form of numerical constants, and (iii) the lexical form of identifiers, for each of the following languages: (a) C (b) C++ (c) C# (d) Fortran (e) Java (f) Lisp (g) SQL.

**! Exercise 3.3.2:** Describe the languages denoted by the following regular expressions:

a)  $a(a|b)^*a$ .

b)  $((\epsilon|a)b^*)^*$ .

c)  $(a|b)^*a(a|b)(a|b)$ .

d)  $a^*ba^*ba^*ba^*$ .

!! e)  $(aa|bb)^*((ab|ba)(aa|bb)^*(ab|ba)(aa|bb)^*)^*$ .

**Exercise 3.3.3:** In a string of length  $n$ , how many of the following are there?

a) Prefixes.

b) Suffixes.

c) Proper prefixes.

! d) Substrings.

! e) Subsequences.

**Exercise 3.3.4:** Most languages are *case sensitive*, so keywords can be written only one way, and the regular expressions describing their lexemes are very simple. However, some languages, like SQL, are *case insensitive*, so a keyword can be written either in lowercase or in uppercase, or in any mixture of cases. Thus, the SQL keyword `SELECT` can also be written `select`, `Select`, or `sElEcT`, for instance. Show how to write a regular expression for a keyword in a case-insensitive language. Illustrate the idea by writing the expression for “select” in SQL.

**! Exercise 3.3.5:** Write regular definitions for the following languages:

a) All strings of lowercase letters that contain the five vowels in order.

b) All strings of lowercase letters in which the letters are in ascending lexicographic order.

c) Comments, consisting of a string surrounded by `/*` and `*/`, without an intervening `*/`, unless it is inside double-quotes (`"`).

- !! d) All strings of digits with no repeated digits. *Hint*: Try this problem first with a few digits, such as  $\{0, 1, 2\}$ .
- !! e) All strings of digits with at most one repeated digit.
- !! f) All strings of  $a$ 's and  $b$ 's with an even number of  $a$ 's and an odd number of  $b$ 's.
- g) The set of Chess moves, in the informal notation, such as  $p-k4$  or  $kbp \times qn$ .
- !! h) All strings of  $a$ 's and  $b$ 's that do not contain the substring  $abb$ .
- i) All strings of  $a$ 's and  $b$ 's that do not contain the subsequence  $abb$ .

**Exercise 3.3.6:** Write character classes for the following sets of characters:

- a) The first ten letters (up to "j") in either upper or lower case.
- b) The lowercase consonants.
- c) The "digits" in a hexadecimal number (choose either upper or lower case for the "digits" above 9).
- d) The characters that can appear at the end of a legitimate English sentence (e.g., exclamation point).

The following exercises, up to and including Exercise 3.3.10, discuss the extended regular-expression notation from *Lex* (the lexical-analyzer generator that we shall discuss extensively in Section 3.5). The extended notation is listed in Fig. 3.8.

**Exercise 3.3.7:** Note that these regular expressions give all of the following symbols (*operator characters*) a special meaning:

\ " . ^ \$ [ ] \* + ? { } | /

Their special meaning must be turned off if they are needed to represent themselves in a character string. We can do so by quoting the character within a string of length one or more; e.g., the regular expression `"**"` matches the string `**`. We can also get the literal meaning of an operator character by preceding it by a backslash. Thus, the regular expression `\**` also matches the string `**`. Write a regular expression that matches the string `"\`.

**Exercise 3.3.8:** In *Lex*, a *complemented character class* represents any character except the ones listed in the character class. We denote a complemented class by using `^` as the first character; this symbol (caret) is not itself part of the class being complemented, unless it is listed within the class itself. Thus, `[^A-Za-z]` matches any character that is not an uppercase or lowercase letter, and `[^\^]` represents any character but the caret (or newline, since newline cannot be in any character class). Show that for every regular expression with complemented character classes, there is an equivalent regular expression without complemented character classes.

| EXPRESSION     | MATCHES                                 | EXAMPLE                |
|----------------|-----------------------------------------|------------------------|
| $c$            | the one non-operator character $c$      | <code>a</code>         |
| $\backslash c$ | character $c$ literally                 | <code>\*</code>        |
| $"s"$          | string $s$ literally                    | <code>"**"</code>      |
| $.$            | any character but newline               | <code>a.*b</code>      |
| $\wedge$       | beginning of a line                     | <code>^abc</code>      |
| $\$$           | end of a line                           | <code>abc\\$</code>    |
| $[s]$          | any one of the characters in string $s$ | <code>[abc]</code>     |
| $[\wedge s]$   | any one character not in string $s$     | <code>[^abc]</code>    |
| $r^*$          | zero or more strings matching $r$       | <code>a*</code>        |
| $r^+$          | one or more strings matching $r$        | <code>a^+</code>       |
| $r^?$          | zero or one $r$                         | <code>a^?</code>       |
| $r\{m, n\}$    | between $m$ and $n$ occurrences of $r$  | <code>a\{1, 5\}</code> |
| $r_1 r_2$      | an $r_1$ followed by an $r_2$           | <code>ab</code>        |
| $r_1 \mid r_2$ | an $r_1$ or an $r_2$                    | <code>a b</code>       |
| $(r)$          | same as $r$                             | <code>(a b)</code>     |
| $r_1/r_2$      | $r_1$ when followed by $r_2$            | <code>abc/123</code>   |

Figure 3.8: `Lex` regular expressions

**! Exercise 3.3.9:** The regular expression  $r\{m, n\}$  matches from  $m$  to  $n$  occurrences of the pattern  $r$ . For example, `a\{1, 5\}` matches a string of one to five  $a$ 's. Show that for every regular expression containing repetition operators of this form, there is an equivalent regular expression without repetition operators.

**! Exercise 3.3.10:** The operator  $\wedge$  matches the left end of a line, and  $\$$  matches the right end of a line. The operator  $\wedge$  is also used to introduce complemented character classes, but the context always makes it clear which meaning is intended. For example, `^[^aeiou]*\$` matches any complete line that does not contain a lowercase vowel.

- How do you tell which meaning of  $\wedge$  is intended?
- Can you always replace a regular expression using the  $\wedge$  and  $\$$  operators by an equivalent expression that does not use either of these operators?

**! Exercise 3.3.11:** The UNIX shell command `sh` uses the operators in Fig. 3.9 in filename expressions to describe sets of file names. For example, the filename expression `*.o` matches all file names ending in `.o`; `sort1.?` matches all file names of the form `sort1.c`, where  $c$  is any character. Show how `sh` filename

| EXPRESSION   | MATCHES                      | EXAMPLE     |
|--------------|------------------------------|-------------|
| ' <i>s</i> ' | string <i>s</i> literally    | '\'         |
| \ <i>c</i>   | character <i>c</i> literally | '\'         |
| *            | any string                   | *.o         |
| ?            | any character                | sort1.?     |
| [ <i>s</i> ] | any character in <i>s</i>    | sort1.[cso] |

Figure 3.9: Filename expressions used by the shell command `sh`

expressions can be replaced by equivalent regular expressions using only the basic union, concatenation, and closure operators.

**! Exercise 3.3.12:** SQL allows a rudimentary form of patterns in which two characters have special meaning: underscore (`_`) stands for any one character and percent-sign (`%`) stands for any string of 0 or more characters. In addition, the programmer may define any character, say *e*, to be the escape character, so an *e* preceding `_`, `%`, or another *e* gives the character that follows its literal meaning. Show how to express any SQL pattern as a regular expression, given that we know which character is the escape character.

### 3.4 Recognition of Tokens

In the previous section we learned how to express patterns using regular expressions. Now, we must study how to take the patterns for all the needed tokens and build a piece of code that examines the input string and finds a prefix that is a lexeme matching one of the patterns. Our discussion will make use of the following running example.

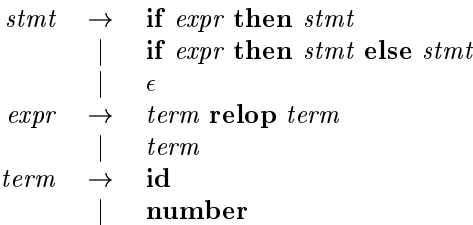


Figure 3.10: A grammar for branching statements

**Example 3.8:** The grammar fragment of Fig. 3.10 describes a simple form of branching statements and conditional expressions. This syntax is similar to that of the language Pascal, in that **then** appears explicitly after conditions.

For **relop**, we use the comparison operators of languages like Pascal or SQL, where = is “equals” and <> is “not equals,” because it presents an interesting structure of lexemes.

The terminals of the grammar, which are **if**, **then**, **else**, **relop**, **id**, and **number**, are the names of tokens as far as the lexical analyzer is concerned. The patterns for these tokens are described using regular definitions, as in Fig. 3.11. The patterns for *id* and *number* are similar to what we saw in Example 3.7.

|               |   |                                                               |
|---------------|---|---------------------------------------------------------------|
| <i>digit</i>  | → | [0-9]                                                         |
| <i>digits</i> | → | <i>digit</i> <sup>+</sup>                                     |
| <i>number</i> | → | <i>digits</i> ( . <i>digits</i> )? ( E [+-]? <i>digits</i> )? |
| <i>letter</i> | → | [A-Za-z]                                                      |
| <i>id</i>     | → | <i>letter</i> ( <i>letter</i>   <i>digit</i> )*               |
| <i>if</i>     | → | <b>if</b>                                                     |
| <i>then</i>   | → | <b>then</b>                                                   |
| <i>else</i>   | → | <b>else</b>                                                   |
| <i>relop</i>  | → | <   >   <=   >=   =   <>                                      |

Figure 3.11: Patterns for tokens of Example 3.8

For this language, the lexical analyzer will recognize the keywords **if**, **then**, and **else**, as well as lexemes that match the patterns for *relop*, *id*, and *number*. To simplify matters, we make the common assumption that keywords are also *reserved words*: that is, they are not identifiers, even though their lexemes match the pattern for identifiers.

In addition, we assign the lexical analyzer the job of stripping out whitespace, by recognizing the “token” *ws* defined by:

$$ws \rightarrow ( \text{blank} \mid \text{tab} \mid \text{newline} )^+$$

Here, **blank**, **tab**, and **newline** are abstract symbols that we use to express the ASCII characters of the same names. Token *ws* is different from the other tokens in that, when we recognize it, we do not return it to the parser, but rather restart the lexical analysis from the character that follows the whitespace. It is the following token that gets returned to the parser.

Our goal for the lexical analyzer is summarized in Fig. 3.12. That table shows, for each lexeme or family of lexemes, which token name is returned to the parser and what attribute value, as discussed in Section 3.1.3, is returned. Note that for the six relational operators, symbolic constants LT, LE, and so on are used as the attribute value, in order to indicate which instance of the token **relop** we have found. The particular operator found will influence the code that is output from the compiler. □

| LEXEMES           | TOKEN NAME    | ATTRIBUTE VALUE        |
|-------------------|---------------|------------------------|
| Any <i>ws</i>     | —             | —                      |
| <b>if</b>         | <b>if</b>     | —                      |
| <b>then</b>       | <b>then</b>   | —                      |
| <b>else</b>       | <b>else</b>   | —                      |
| Any <i>id</i>     | <b>id</b>     | Pointer to table entry |
| Any <i>number</i> | <b>number</b> | Pointer to table entry |
| <                 | <b>relop</b>  | LT                     |
| <=                | <b>relop</b>  | LE                     |
| =                 | <b>relop</b>  | EQ                     |
| <>                | <b>relop</b>  | NE                     |
| >                 | <b>relop</b>  | GT                     |
| >=                | <b>relop</b>  | GE                     |

Figure 3.12: Tokens, their patterns, and attribute values

### 3.4.1 Transition Diagrams

As an intermediate step in the construction of a lexical analyzer, we first convert patterns into stylized flowcharts, called “transition diagrams.” In this section, we perform the conversion from regular-expression patterns to transition diagrams by hand, but in Section 3.6, we shall see that there is a mechanical way to construct these diagrams from collections of regular expressions.

*Transition diagrams* have a collection of nodes or circles, called *states*. Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns. We may think of a state as summarizing all we need to know about what characters we have seen between the *lexemeBegin* pointer and the *forward* pointer (as in the situation of Fig. 3.3).

*Edges* are directed from one state of the transition diagram to another. Each edge is *labeled* by a symbol or set of symbols. If we are in some state *s*, and the next input symbol is *a*, we look for an edge out of state *s* labeled by *a* (and perhaps by other symbols, as well). If we find such an edge, we advance the *forward* pointer and enter the state of the transition diagram to which that edge leads. We shall assume that all our transition diagrams are *deterministic*, meaning that there is never more than one edge out of a given state with a given symbol among its labels. Starting in Section 3.5, we shall relax the condition of determinism, making life much easier for the designer of a lexical analyzer, although trickier for the implementer. Some important conventions about transition diagrams are:

1. Certain states are said to be *accepting*, or *final*. These states indicate that a lexeme has been found, although the actual lexeme may not consist of all positions between the *lexemeBegin* and *forward* pointers. We always

indicate an accepting state by a double circle, and if there is an action to be taken — typically returning a token and an attribute value to the parser — we shall attach that action to the accepting state.

2. In addition, if it is necessary to retract the *forward* pointer one position (i.e., the lexeme does not include the symbol that got us to the accepting state), then we shall additionally place a \* near that accepting state. In our example, it is never necessary to retract *forward* by more than one position, but if it were, we could attach any number of \*'s to the accepting state.
3. One state is designated the *start state*, or *initial state*; it is indicated by an edge, labeled “start,” entering from nowhere. The transition diagram always begins in the start state before any input symbols have been read.

**Example 3.9 :** Figure 3.13 is a transition diagram that recognizes the lexemes matching the token **relop**. We begin in state 0, the start state. If we see < as the first input symbol, then among the lexemes that match the pattern for **relop** we can only be looking at <, <>, or <=. We therefore go to state 1, and look at the next character. If it is =, then we recognize lexeme <=, enter state 2, and return the token **relop** with attribute LE, the symbolic constant representing this particular comparison operator. If in state 1 the next character is >, then instead we have lexeme <>, and enter state 3 to return an indication that the not-equals operator has been found. On any other character, the lexeme is <, and we enter state 4 to return that information. Note, however, that state 4 has a \* to indicate that we must retract the input one position.

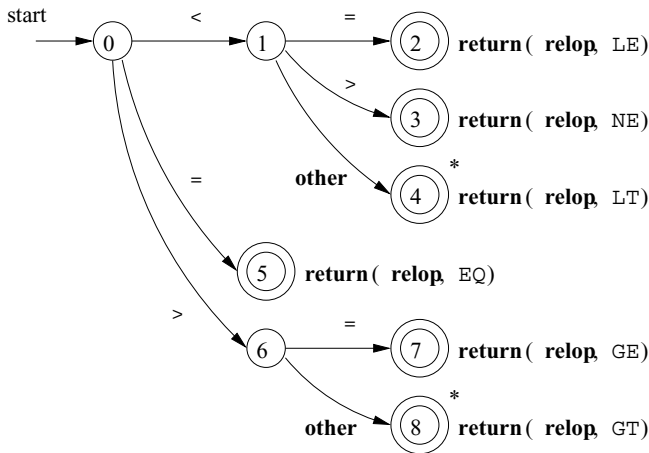


Figure 3.13: Transition diagram for **relop**

On the other hand, if in state 0 the first character we see is =, then this one character must be the lexeme. We immediately return that fact from state 5.

The remaining possibility is that the first character is `>`. Then, we must enter state 6 and decide, on the basis of the next character, whether the lexeme is `>=` (if we next see the `=` sign), or just `>` (on any other character). Note that if, in state 0, we see any character besides `<`, `=`, or `>`, we can not possibly be seeing a `relop` lexeme, so this transition diagram will not be used.  $\square$

### 3.4.2 Recognition of Reserved Words and Identifiers

Recognizing keywords and identifiers presents a problem. Usually, keywords like `if` or `then` are reserved (as they are in our running example), so they are not identifiers even though they *look* like identifiers. Thus, although we typically use a transition diagram like that of Fig. 3.14 to search for identifier lexemes, this diagram will also recognize the keywords `if`, `then`, and `else` of our running example.

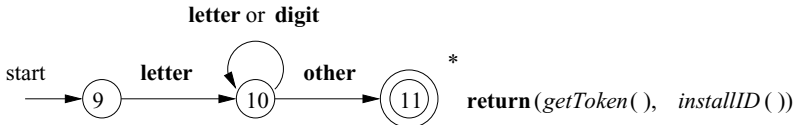


Figure 3.14: A transition diagram for `id`'s and keywords

There are two ways that we can handle reserved words that look like identifiers:

1. Install the reserved words in the symbol table initially. A field of the symbol-table entry indicates that these strings are never ordinary identifiers, and tells which token they represent. We have supposed that this method is in use in Fig. 3.14. When we find an identifier, a call to *installID* places it in the symbol table if it is not already there and returns a pointer to the symbol-table entry for the lexeme found. Of course, any identifier not in the symbol table during lexical analysis cannot be a reserved word, so its token is `id`. The function *getToken* examines the symbol table entry for the lexeme found, and returns whatever token name the symbol table says this lexeme represents — either `id` or one of the keyword tokens that was initially installed in the table.
2. Create separate transition diagrams for each keyword; an example for the keyword `then` is shown in Fig. 3.15. Note that such a transition diagram consists of states representing the situation after each successive letter of the keyword is seen, followed by a test for a “nonletter-or-digit,” i.e., any character that cannot be the continuation of an identifier. It is necessary to check that the identifier has ended, or else we would return token `then` in situations where the correct token was `id`, with a lexeme like `thennextvalue` that has `then` as a proper prefix. If we adopt this approach, then we must prioritize the tokens so that the reserved-word

tokens are recognized in preference to **id**, when the lexeme matches both patterns. We *do not* use this approach in our example, which is why the states in Fig. 3.15 are unnumbered.

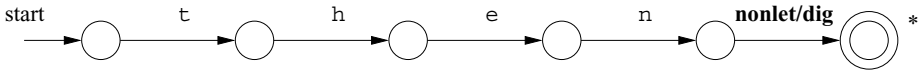


Figure 3.15: Hypothetical transition diagram for the keyword **then**

### 3.4.3 Completion of the Running Example

The transition diagram for **id**'s that we saw in Fig. 3.14 has a simple structure. Starting in state 9, it checks that the lexeme begins with a letter and goes to state 10 if so. We stay in state 10 as long as the input contains letters and digits. When we first encounter anything but a letter or digit, we go to state 11 and accept the lexeme found. Since the last character is not part of the identifier, we must retract the input one position, and as discussed in Section 3.4.2, we enter what we have found in the symbol table and determine whether we have a keyword or a true identifier.

The transition diagram for token **number** is shown in Fig. 3.16, and is so far the most complex diagram we have seen. Beginning in state 12, if we see a digit, we go to state 13. In that state, we can read any number of additional digits. However, if we see anything but a digit, dot, or E, we have seen a number in the form of an integer; 123 is an example. That case is handled by entering state 20, where we return token **number** and a pointer to a table of constants where the found lexeme is entered. These mechanics are not shown on the diagram but are analogous to the way we handled identifiers.

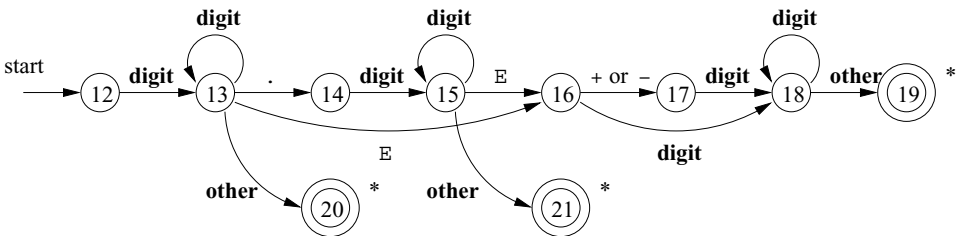


Figure 3.16: A transition diagram for unsigned numbers

If we instead see a dot in state 13, then we have an “optional fraction.” State 14 is entered, and we look for one or more additional digits; state 15 is used for that purpose. If we see an E, then we have an “optional exponent,” whose recognition is the job of states 16 through 19. Should we, in state 15, instead see anything but E or a digit, then we have come to the end of the fraction, there is no exponent, and we return the lexeme found, via state 21.

The final transition diagram, shown in Fig. 3.17, is for whitespace. In that diagram, we look for one or more “whitespace” characters, represented by **delim** in that diagram — typically these characters would be blank, tab, newline, and perhaps other characters that are not considered by the language design to be part of any token.

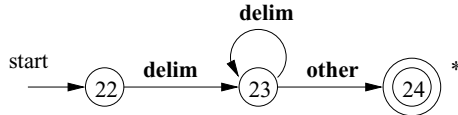


Figure 3.17: A transition diagram for whitespace

Note that in state 24, we have found a block of consecutive whitespace characters, followed by a nonwhitespace character. We retract the input to begin at the nonwhitespace, but we do not return to the parser. Rather, we must restart the process of lexical analysis after the whitespace.

### 3.4.4 Architecture of a Transition-Diagram-Based Lexical Analyzer

There are several ways that a collection of transition diagrams can be used to build a lexical analyzer. Regardless of the overall strategy, each state is represented by a piece of code. We may imagine a variable **state** holding the number of the current state for a transition diagram. A switch based on the value of **state** takes us to code for each of the possible states, where we find the action of that state. Often, the code for a state is itself a switch statement or multiway branch that determines the next state by reading and examining the next input character.

**Example 3.10 :** In Fig. 3.18 we see a sketch of `getRelop()`, a C++ function whose job is to simulate the transition diagram of Fig. 3.13 and return an object of type **TOKEN**, that is, a pair consisting of the token name (which must be **relop** in this case) and an attribute value (the code for one of the six comparison operators in this case). `getRelop()` first creates a new object **retToken** and initializes its first component to **RELOP**, the symbolic code for token **relop**.

We see the typical behavior of a state in case 0, the case where the current state is 0. A function `nextChar()` obtains the next character from the input and assigns it to local variable *c*. We then check *c* for the three characters we expect to find, making the state transition dictated by the transition diagram of Fig. 3.13 in each case. For example, if the next input character is `=`, we go to state 5.

If the next input character is not one that can begin a comparison operator, then a function `fail()` is called. What `fail()` does depends on the global error-recovery strategy of the lexical analyzer. It should reset the **forward** pointer to **lexemeBegin**, in order to allow another transition diagram to be applied to

```

TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}

```

Figure 3.18: Sketch of implementation of **relop** transition diagram

the true beginning of the unprocessed input. It might then change the value of **state** to be the start state for another transition diagram, which will search for another token. Alternatively, if there is no other transition diagram that remains unused, **fail()** could initiate an error-correction phase that will try to repair the input and find a lexeme, as discussed in Section 3.1.4.

We also show the action for state 8 in Fig. 3.18. Because state 8 bears a \*, we must retract the input pointer one position (i.e., put *c* back on the input stream). That task is accomplished by the function **retract()**. Since state 8 represents the recognition of lexeme >, we set the second component of the returned object, which we suppose is named **attribute**, to GT, the code for this operator. □

To place the simulation of one transition diagram in perspective, let us consider the ways code like Fig. 3.18 could fit into the entire lexical analyzer.

1. We could arrange for the transition diagrams for each token to be tried sequentially. Then, the function **fail()** of Example 3.10 resets the pointer **forward** and starts the next transition diagram, each time it is called. This method allows us to use transition diagrams for the individual keywords, like the one suggested in Fig. 3.15. We have only to use these before we use the diagram for **id**, in order for the keywords to be reserved words.

2. We could run the various transition diagrams “in parallel,” feeding the next input character to all of them and allowing each one to make whatever transitions it required. If we use this strategy, we must be careful to resolve the case where one diagram finds a lexeme that matches its pattern, while one or more other diagrams are still able to process input. The normal strategy is to take the longest prefix of the input that matches any pattern. That rule allows us to prefer identifier **thenext** to keyword **then**, or the operator **->** to **-**, for example.
3. The preferred approach, and the one we shall take up in the following sections, is to combine all the transition diagrams into one. We allow the transition diagram to read input until there is no possible next state, and then take the longest lexeme that matched any pattern, as we discussed in item (2) above. In our running example, this combination is easy, because no two tokens can start with the same character; i.e., the first character immediately tells us which token we are looking for. Thus, we could simply combine states 0, 9, 12, and 22 into one start state, leaving other transitions intact. However, in general, the problem of combining transition diagrams for several tokens is more complex, as we shall see shortly.

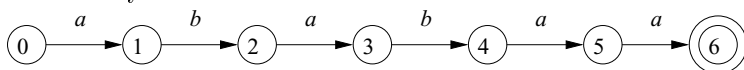
### 3.4.5 Exercises for Section 3.4

**Exercise 3.4.1:** Provide transition diagrams to recognize the same languages as each of the regular expressions in Exercise 3.3.2.

**Exercise 3.4.2:** Provide transition diagrams to recognize the same languages as each of the regular expressions in Exercise 3.3.5.

The following exercises, up to Exercise 3.4.12, introduce the Aho-Corasick algorithm for recognizing a collection of keywords in a text string in time proportional to the length of the text and the sum of the length of the keywords. This algorithm uses a special form of transition diagram called a *trie*. A trie is a tree-structured transition diagram with distinct labels on the edges leading from a node to its children. Leaves of the trie represent recognized keywords.

Knuth, Morris, and Pratt presented an algorithm for recognizing a single keyword  $b_1b_2 \cdots b_n$  in a text string. Here the trie is a transition diagram with  $n + 1$  states, 0 through  $n$ . State 0 is the initial state, and state  $n$  represents acceptance, that is, discovery of the keyword. From each state  $s$  from 0 through  $n - 1$ , there is a transition to state  $s + 1$ , labeled by symbol  $b_{s+1}$ . For example, the trie for the keyword **ababaa** is:



In order to process text strings rapidly and search those strings for a keyword, it is useful to define, for keyword  $b_1b_2 \cdots b_n$  and position  $s$  in that keyword (corresponding to state  $s$  of its trie), a *failure function*,  $f(s)$ , computed as in

Fig. 3.19. The objective is that  $b_1b_2 \cdots b_{f(s)}$  is the longest proper prefix of  $b_1b_2 \cdots b_s$  that is also a suffix of  $b_1b_2 \cdots b_s$ . The reason  $f(s)$  is important is that if we are trying to match a text string for  $b_1b_2 \cdots b_n$ , and we have matched the first  $s$  positions, but we then fail (i.e., the next position of the text string does not hold  $b_{s+1}$ ), then  $f(s)$  is the longest prefix of  $b_1b_2 \cdots b_n$  that could possibly match the text string up to the point we are at. Of course, the next character of the text string must be  $b_{f(s)+1}$ , or else we still have problems and must consider a yet shorter prefix, which will be  $b_{f(f(s))}$ .

```

1)   $t = 0$ ;
2)   $f(1) = 0$ ;
3)  for ( $s = 1$ ;  $s < n$ ;  $s++$ ) {
4)      while ( $t > 0 \ \&\& \ b_{s+1} \neq b_{t+1}$ )  $t = f(t)$ ;
5)      if ( $b_{s+1} == b_{t+1}$ ) {
6)           $t = t + 1$ ;
7)           $f(s + 1) = t$ ;
8)      }
9)  }

```

Figure 3.19: Algorithm to compute the failure function for keyword  $b_1b_2 \cdots b_n$

As an example, the failure function for the trie constructed above for **ababaa** is:

| $s$    | 1 | 2 | 3 | 4 | 5 | 6 |
|--------|---|---|---|---|---|---|
| $f(s)$ | 0 | 0 | 1 | 2 | 3 | 1 |

For instance, states 3 and 1 represent prefixes **aba** and **a**, respectively.  $f(3) = 1$  because **a** is the longest proper prefix of **aba** that is also a suffix of **aba**. Also,  $f(2) = 0$ , because the longest proper prefix of **ab** that is also a suffix is the empty string.

**Exercise 3.4.3:** Construct the failure function for the strings:

- a) **abababaab**.
- b) **aaaaaa**.
- c) **abbaabb**.

**! Exercise 3.4.4:** Prove, by induction on  $s$ , that the algorithm of Fig. 3.19 correctly computes the failure function.

**!! Exercise 3.4.5:** Show that the assignment  $t = f(t)$  in line (4) of Fig. 3.19 is executed at most  $n$  times. Show that therefore, the entire algorithm takes only  $O(n)$  time on a keyword of length  $n$ .

Having computed the failure function for a keyword  $b_1b_2 \cdots b_n$ , we can scan a string  $a_1a_2 \cdots a_m$  in time  $O(m)$  to tell whether the keyword occurs in the string. The algorithm, shown in Fig. 3.20, slides the keyword along the string, trying to make progress by matching the next character of the keyword with the next character of the string. If it cannot do so after matching  $s$  characters, then it “slides” the keyword right  $s - f(s)$  positions, so only the first  $f(s)$  characters of the keyword are considered matched with the string.

```

1)   $s = 0$ ;
2)  for ( $i = 1$ ;  $i \leq m$ ;  $i++$ ) {
3)      while ( $s > 0$  &&  $a_i \neq b_{s+1}$ )  $s = f(s)$ ;
4)      if ( $a_i == b_{s+1}$ )  $s = s + 1$ ;
5)      if ( $s == n$ ) return “yes”;
6)  }
7)  return “no”;

```

Figure 3.20: The KMP algorithm tests whether string  $a_1a_2 \cdots a_m$  contains a single keyword  $b_1b_2 \cdots b_n$  as a substring in  $O(m + n)$  time

**Exercise 3.4.6:** Apply Algorithm KMP to test whether keyword **ababaa** is a substring of:

- a) **abababaab**.
- b) **abababbaa**.

**!! Exercise 3.4.7:** Show that the algorithm of Fig. 3.20 correctly tells whether the keyword is a substring of the given string. *Hint:* proceed by induction on  $i$ . Show that for all  $i$ , the value of  $s$  after line (4) is the length of the longest prefix of the keyword that is a suffix of  $a_1a_2 \cdots a_i$ .

**!! Exercise 3.4.8:** Show that the algorithm of Fig. 3.20 runs in time  $O(m + n)$ , assuming that function  $f$  is already computed and its values stored in an array indexed by  $s$ .

**Exercise 3.4.9:** The *Fibonacci strings* are defined as follows:

- 1.  $s_1 = \mathbf{b}$ .
- 2.  $s_2 = \mathbf{a}$ .
- 3.  $s_k = s_{k-1}s_{k-2}$  for  $k > 2$ .

For example,  $s_3 = \mathbf{ab}$ ,  $s_4 = \mathbf{aba}$ , and  $s_5 = \mathbf{abaab}$ .

- a) What is the length of  $s_n$ ?

- b) Construct the failure function for  $s_6$ .
- c) Construct the failure function for  $s_7$ .
- !! d) Show that the failure function for any  $s_n$  can be expressed by  $f(1) = f(2) = 0$ , and for  $2 < j \leq |s_n|$ ,  $f(j)$  is  $j - |s_{k-1}|$ , where  $k$  is the largest integer such that  $|s_k| \leq j + 1$ .
- !! e) In the KMP algorithm, what is the largest number of consecutive applications of the failure function, when we try to determine whether keyword  $s_k$  appears in text string  $s_{k+1}$ ?

Aho and Corasick generalized the KMP algorithm to recognize any of a set of keywords in a text string. In this case, the trie is a true tree, with branching from the root. There is one state for every string that is a prefix (not necessarily proper) of any keyword. The parent of a state corresponding to string  $b_1b_2 \cdots b_k$  is the state that corresponds to  $b_1b_2 \cdots b_{k-1}$ . A state is accepting if it corresponds to a complete keyword. For example, Fig. 3.21 shows the trie for the keywords **he**, **she**, **his**, and **hers**.

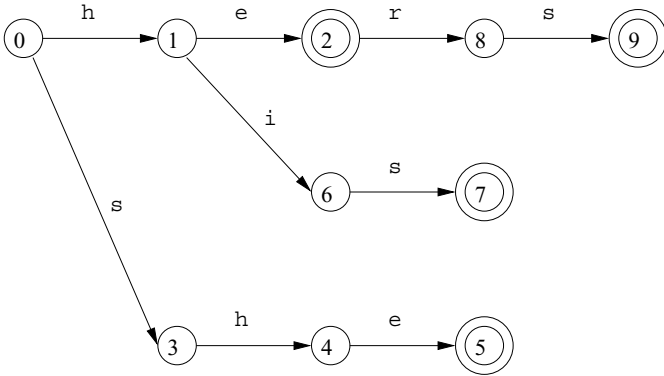


Figure 3.21: Trie for keywords **he**, **she**, **his**, **hers**

The failure function for the general trie is defined as follows. Suppose  $s$  is the state that corresponds to string  $b_1b_2 \cdots b_n$ . Then  $f(s)$  is the state that corresponds to the longest proper suffix of  $b_1b_2 \cdots b_n$  that is also a prefix of *some* keyword. For example, the failure function for the trie of Fig. 3.21 is:

| $s$    | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|--------|---|---|---|---|---|---|---|---|---|
| $f(s)$ | 0 | 0 | 0 | 1 | 2 | 0 | 3 | 0 | 3 |

**! Exercise 3.4.10:** Modify the algorithm of Fig. 3.19 to compute the failure function for general tries. *Hint:* The major difference is that we cannot simply test for equality or inequality of  $b_{s+1}$  and  $b_{t+1}$  in lines (4) and (5) of Fig. 3.19. Rather, from any state there may be several transitions out on several characters, as there are transitions on both **e** and **i** from state 1 in Fig. 3.21. Any of

those transitions could lead to a state that represents the longest suffix that is also a prefix.

**Exercise 3.4.11:** Construct the tries and compute the failure function for the following sets of keywords:

- a) `aaa`, `abaaa`, and `ababaaa`.
- b) `all`, `fall`, `fatal`, `llama`, and `lame`.
- c) `pipe`, `pet`, `item`, `temper`, and `perpetual`.

**! Exercise 3.4.12:** Show that your algorithm from Exercise 3.4.10 still runs in time that is linear in the sum of the lengths of the keywords.

## 3.5 The Lexical-Analyzer Generator Lex

In this section, we introduce a tool called *Lex*, or in a more recent implementation *Flex*, that allows one to specify a lexical analyzer by specifying regular expressions to describe patterns for tokens. The input notation for the *Lex* tool is referred to as the *Lex language* and the tool itself is the *Lex compiler*. Behind the scenes, the *Lex* compiler transforms the input patterns into a transition diagram and generates code, in a file called `lex.yy.c`, that simulates this transition diagram. The mechanics of how this translation from regular expressions to transition diagrams occurs is the subject of the next sections; here we only learn the *Lex* language.

### 3.5.1 Use of Lex

Figure 3.22 suggests how *Lex* is used. An input file, which we call `lex.l`, is written in the *Lex* language and describes the lexical analyzer to be generated. The *Lex* compiler transforms `lex.l` to a C program, in a file that is always named `lex.yy.c`. The latter file is compiled by the C compiler into a file called `a.out`, as always. The C-compiler output is a working lexical analyzer that can take a stream of input characters and produce a stream of tokens.

The normal use of the compiled C program, referred to as `a.out` in Fig. 3.22, is as a subroutine of the parser. It is a C function that returns an integer, which is a code for one of the possible token names. The attribute value, whether it be another numeric code, a pointer to the symbol table, or nothing, is placed in a global variable `yylval`,<sup>2</sup> which is shared between the lexical analyzer and parser, thereby making it simple to return both the name and an attribute value of a token.

---

<sup>2</sup>Incidentally, the `yy` that appears in `yylval` and `lex.yy.c` refers to the *Yacc* parser-generator, which we shall describe in Section 4.9, and which is commonly used in conjunction with *Lex*.

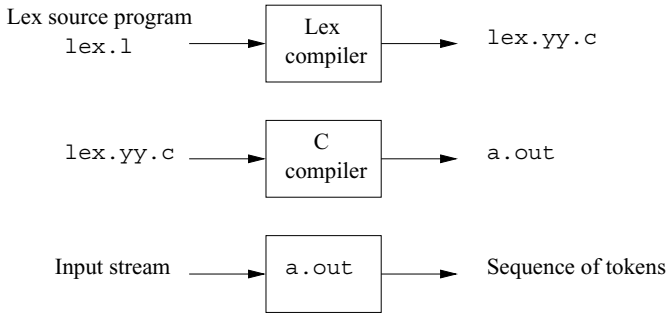


Figure 3.22: Creating a lexical analyzer with Lex

### 3.5.2 Structure of Lex Programs

A Lex program has the following form:

```

declarations
%%
translation rules
%%
auxiliary functions
  
```

The declarations section includes declarations of variables, *manifest constants* (identifiers declared to stand for a constant, e.g., the name of a token), and regular definitions, in the style of Section 3.3.4.

The translation rules each have the form

```
Pattern { Action }
```

Each pattern is a regular expression, which may use the regular definitions of the declaration section. The actions are fragments of code, typically written in C, although many variants of Lex using other languages have been created.

The third section holds whatever additional functions are used in the actions. Alternatively, these functions can be compiled separately and loaded with the lexical analyzer.

The lexical analyzer created by Lex behaves in concert with the parser as follows. When called by the parser, the lexical analyzer begins reading its remaining input, one character at a time, until it finds the longest prefix of the input that matches one of the patterns  $P_i$ . It then executes the associated action  $A_i$ . Typically,  $A_i$  will return to the parser, but if it does not (e.g., because  $P_i$  describes whitespace or comments), then the lexical analyzer proceeds to find additional lexemes, until one of the corresponding actions causes a return to the parser. The lexical analyzer returns a single value, the token name, to the parser, but uses the shared, integer variable `yylval` to pass additional information about the lexeme found, if needed.

**Example 3.11:** Figure 3.23 is a *Lex* program that recognizes the tokens of Fig. 3.12 and returns the token found. A few observations about this code will introduce us to many of the important features of *Lex*.

In the declarations section we see a pair of special brackets, `%{` and `%}`. Anything within these brackets is copied directly to the file `lex.yy.c`, and is not treated as a regular definition. It is common to place there the definitions of the manifest constants, using *C* `#define` statements to associate unique integer codes with each of the manifest constants. In our example, we have listed in a comment the names of the manifest constants, `LT`, `IF`, and so on, but have not shown them defined to be particular integers.<sup>3</sup>

Also in the declarations section is a sequence of regular definitions. These use the extended notation for regular expressions described in Section 3.3.5. Regular definitions that are used in later definitions or in the patterns of the translation rules are surrounded by curly braces. Thus, for instance, *delim* is defined to be a shorthand for the character class consisting of the blank, the tab, and the newline; the latter two are represented, as in all UNIX commands, by backslash followed by `t` or `n`, respectively. Then, *ws* is defined to be one or more delimiters, by the regular expression `{delim}+`.

Notice that in the definition of *id* and *number*, parentheses are used as grouping metasympols and do not stand for themselves. In contrast, `E` in the definition of *number* stands for itself. If we wish to use one of the *Lex* metasympols, such as any of the parentheses, `+`, `*`, or `?`, to stand for themselves, we may precede them with a backslash. For instance, we see `\.` in the definition of *number*, to represent the dot, since that character is a metasympol representing “any character,” as usual in UNIX regular expressions.

In the auxiliary-function section, we see two such functions, `installID()` and `installNum()`. Like the portion of the declaration section that appears between `%{...%}`, everything in the auxiliary section is copied directly to file `lex.yy.c`, but may be used in the actions.

Finally, let us examine some of the patterns and rules in the middle section of Fig. 3.23. First, *ws*, an identifier declared in the first section, has an associated empty action. If we find whitespace, we do not return to the parser, but look for another lexeme. The second token has the simple regular expression pattern `if`. Should we see the two letters `if` on the input, and they are not followed by another letter or digit (which would cause the lexical analyzer to find a longer prefix of the input matching the pattern for *id*), then the lexical analyzer consumes these two letters from the input and returns the token name `IF`, that is, the integer for which the manifest constant `IF` stands. Keywords `then` and `else` are treated similarly.

The fifth token has the pattern defined by *id*. Note that, although keywords like `if` match this pattern as well as an earlier pattern, *Lex* chooses whichever

---

<sup>3</sup>If *Lex* is used along with *Yacc*, then it would be normal to define the manifest constants in the *Yacc* program and use them without definition in the *Lex* program. Since `lex.yy.c` is compiled with the *Yacc* output, the constants thus will be available to the actions in the *Lex* program.

```

%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
}%

/* regular definitions */
delim      [ \t\n]
ws          {delim}+
letter      [A-Za-z]
digit       [0-9]
id          {letter}{letter}{digit}*
number      {digit}+(\.{digit})?(E[+-]?{digit})?

%%

{ws}        { /* no action and no return */}
if           {return(IF);}
then         {return(THEN);}
else         {return(ELSE);}
{id}         {yyval = (int) installID(); return(ID);}
{number}     {yyval = (int) installNum(); return(NUMBER);}
"<"         {yyval = LT; return(RELOP);}
"<="        {yyval = LE; return(RELOP);}
"="          {yyval = EQ; return(RELOP);}
"<>"        {yyval = NE; return(RELOP);}
">"         {yyval = GT; return(RELOP);}
">="        {yyval = GE; return(RELOP);}

%%

int installID() { /* function to install the lexeme, whose
                  first character is pointed to by yytext,
                  and whose length is yyleng, into the
                  symbol table and return a pointer
                  thereto */
}

int installNum() { /* similar to installID, but puts numer-
                   ical constants into a separate table */
}

```

Figure 3.23: Lex program for the tokens of Fig. 3.12

pattern is listed first in situations where the longest matching prefix matches two or more patterns. The action taken when *id* is matched is threefold:

1. Function `installID()` is called to place the lexeme found in the symbol table.
2. This function returns a pointer to the symbol table, which is placed in global variable `yylval`, where it can be used by the parser or a later component of the compiler. Note that `installID()` has available to it two variables that are set automatically by the lexical analyzer that `Lex` generates:
  - (a) `yyltext` is a pointer to the beginning of the lexeme, analogous to `lexemeBegin` in Fig. 3.3.
  - (b) `yyleng` is the length of the lexeme found.
3. The token name `ID` is returned to the parser.

The action taken when a lexeme matching the pattern *number* is similar, using the auxiliary function `installNum()`. □

### 3.5.3 Conflict Resolution in Lex

We have alluded to the two rules that `Lex` uses to decide on the proper lexeme to select, when several prefixes of the input match one or more patterns:

1. Always prefer a longer prefix to a shorter prefix.
2. If the longest possible prefix matches two or more patterns, prefer the pattern listed first in the `Lex` program.

**Example 3.12:** The first rule tells us to continue reading letters and digits to find the longest prefix of these characters to group as an identifier. It also tells us to treat `<=` as a single lexeme, rather than selecting `<` as one lexeme and `=` as the next lexeme. The second rule makes keywords reserved, if we list the keywords before `id` in the program. For instance, if `then` is determined to be the longest prefix of the input that matches any pattern, and the pattern `then` precedes `{id}`, as it does in Fig. 3.23, then the token `THEN` is returned, rather than `ID`. □

### 3.5.4 The Lookahead Operator

`Lex` automatically reads one character ahead of the last character that forms the selected lexeme, and then retracts the input so only the lexeme itself is consumed from the input. However, sometimes, we want a certain pattern to be matched to the input only when it is followed by a certain other characters. If so, we may use the slash in a pattern to indicate the end of the part of the

pattern that matches the lexeme. What follows / is additional pattern that must be matched before we can decide that the token in question was seen, but what matches this second pattern is not part of the lexeme.

**Example 3.13:** In Fortran and some other languages, keywords are not reserved. That situation creates problems, such as a statement

```
IF(I,J) = 3
```

where IF is the name of an array, not a keyword. This statement contrasts with statements of the form

```
IF( condition ) THEN ...
```

where IF is a keyword. Fortunately, we can be sure that the keyword IF is always followed by a left parenthesis, some text — the condition — that may contain parentheses, a right parenthesis and a letter. Thus, we could write a Lex rule for the keyword IF like:

```
IF / \( .* \) {letter}
```

This rule says that the pattern the lexeme matches is just the two letters IF. The slash says that additional pattern follows but does not match the lexeme. In this pattern, the first character is the left parentheses. Since that character is a Lex metasymbol, it must be preceded by a backslash to indicate that it has its literal meaning. The dot and star match “any string without a newline.” Note that the dot is a Lex metasymbol meaning “any character except newline.” It is followed by a right parenthesis, again with a backslash to give that character its literal meaning. The additional pattern is followed by the symbol *letter*, which is a regular definition representing the character class of all letters.

Note that in order for this pattern to be foolproof, we must preprocess the input to delete whitespace. We have in the pattern neither provision for whitespace, nor can we deal with the possibility that the condition extends over lines, since the dot will not match a newline character.

For instance, suppose this pattern is asked to match a prefix of input:

```
IF(A<(B+C)*D)THEN...
```

the first two characters match IF, the next character matches \(), the next nine characters match .\*, and the next two match \) and *letter*. Note the fact that the first right parenthesis (after C) is not followed by a letter is irrelevant; we only need to find some way of matching the input to the pattern. We conclude that the letters IF constitute the lexeme, and they are an instance of token **if**.

□

### 3.5.5 Exercises for Section 3.5

**Exercise 3.5.1:** Describe how to make the following modifications to the `Lex` program of Fig. 3.23:

- a) Add the keyword `while`.
- b) Change the comparison operators to be the C operators of that kind.
- c) Allow the underscore (`_`) as an additional letter.
- ! d) Add a new pattern with token `STRING`. The pattern consists of a double-quote (`"`), any string of characters and a final double-quote. However, if a double-quote appears in the string, it must be escaped by preceding it with a backslash (`\`), and therefore a backslash in the string must be represented by two backslashes. The lexical value, which is the string without the surrounding double-quotes, and with backslashes used to escape a character removed. Strings are to be installed in a table of strings.

**Exercise 3.5.2:** Write a `Lex` program that copies a file, replacing each non-empty sequence of white space by a single blank.

**Exercise 3.5.3:** Write a `Lex` program that copies a C program, replacing each instance of the keyword `float` by `double`.

! **Exercise 3.5.4:** Write a `Lex` program that converts a file to “Pig latin.” Specifically, assume the file is a sequence of words (groups of letters) separated by whitespace. Every time you encounter a word:

1. If the first letter is a consonant, move it to the end of the word and then add `ay`.
2. If the first letter is a vowel, just add `ay` to the end of the word.

All nonletters are copied intact to the output.

! **Exercise 3.5.5:** In SQL, keywords and identifiers are case-insensitive. Write a `Lex` program that recognizes the keywords `SELECT`, `FROM`, and `WHERE` (in any combination of capital and lower-case letters), and token `ID`, which for the purposes of this exercise you may take to be any sequence of letters and digits, beginning with a letter. You need not install identifiers in a symbol table, but tell how the “install” function would differ from that described for case-sensitive identifiers as in Fig. 3.23.

## Chapter 4

# Syntax Analysis

This chapter is devoted to parsing methods that are typically used in compilers. We first present the basic concepts, then techniques suitable for hand implementation, and finally algorithms that have been used in automated tools. Since programs may contain syntactic errors, we discuss extensions of the parsing methods for recovery from common errors.

By design, every programming language has precise rules that prescribe the syntactic structure of well-formed programs. In C, for example, a program is made up of functions, a function out of declarations and statements, a statement out of expressions, and so on. The syntax of programming language constructs can be specified by context-free grammars or BNF (Backus-Naur Form) notation, introduced in Section 2.2. Grammars offer significant benefits for both language designers and compiler writers.

- A grammar gives a precise, yet easy-to-understand, syntactic specification of a programming language.
- From certain classes of grammars, we can construct automatically an efficient parser that determines the syntactic structure of a source program. As a side benefit, the parser-construction process can reveal syntactic ambiguities and trouble spots that might have slipped through the initial design phase of a language.
- The structure imparted to a language by a properly designed grammar is useful for translating source programs into correct object code and for detecting errors.
- A grammar allows a language to be evolved or developed iteratively, by adding new constructs to perform new tasks. These new constructs can be integrated more easily into an implementation that follows the grammatical structure of the language.

## 4.1 Introduction

In this section, we examine the way the parser fits into a typical compiler. We then look at typical grammars for arithmetic expressions. Grammars for expressions suffice for illustrating the essence of parsing, since parsing techniques for expressions carry over to most programming constructs. This section ends with a discussion of error handling, since the parser must respond gracefully to finding that its input cannot be generated by its grammar.

### 4.1.1 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 of token names can be generated by the grammar for the source language. We expect the parser to report any syntax errors in an intelligible fashion and to recover from commonly occurring errors to continue processing the remainder of the program. Conceptually, for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing. In fact, the parse tree need not be constructed explicitly, since checking and translation actions can be interspersed with parsing, as we shall see. Thus, the parser and the rest of the front end could well be implemented by a single module.

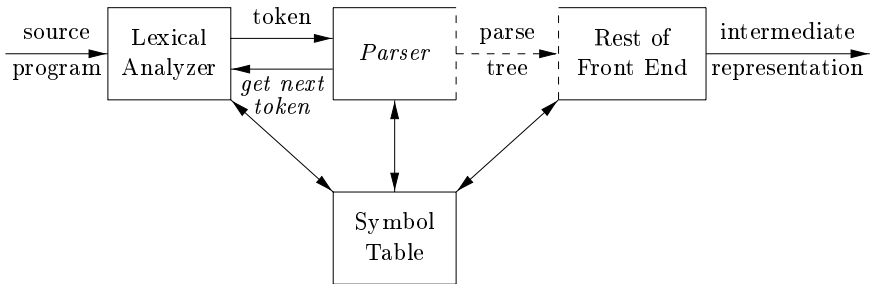


Figure 4.1: Position of parser in compiler model

There are three general types of parsers for grammars: universal, top-down, and bottom-up. Universal parsing methods such as the Cocke-Younger-Kasami algorithm and Earley's algorithm can parse any grammar (see the bibliographic notes). These general methods are, however, too inefficient to use in production compilers.

The methods commonly used in compilers can be classified as being either top-down or bottom-up. As implied by their names, top-down methods build parse trees from the top (root) to the bottom (leaves), while bottom-up methods start from the leaves and work their way up to the root. In either case, 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 for subclasses of grammars, but several of these classes, particularly, LL and LR grammars, are expressive enough to describe most of the syntactic constructs in modern programming languages. Parsers implemented by hand often use LL grammars; for example, the predictive-parsing approach of Section 2.4.2 works for LL grammars. Parsers for the larger class of LR grammars are usually constructed using automated tools.

In this chapter, we assume that the output of the parser is some representation of the parse tree for the stream of tokens that comes from 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. We have lumped all of these activities into the “rest of the front end” box in Fig. 4.1. These activities will be covered in detail in subsequent chapters.

### 4.1.2 Representative Grammars

Some of the grammars that will be examined in this chapter are presented here for ease of reference. Constructs that begin with keywords like **while** or **int**, are relatively easy to parse, because the keyword guides the choice of the grammar production that must be applied to match the input. We therefore concentrate on expressions, which present more of a challenge, because of the associativity and precedence of operators.

Associativity and precedence are captured in the following grammar, which is similar to ones used in Chapter 2 for describing expressions, terms, and factors.  $E$  represents expressions consisting of terms separated by  $+$  signs,  $T$  represents terms consisting of factors separated by  $*$  signs, and  $F$  represents factors that can be either parenthesized expressions or identifiers:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow ( E ) \mid \text{id} \end{aligned} \tag{4.1}$$

Expression grammar (4.1) belongs to the class of LR grammars that are suitable for bottom-up parsing. This grammar can be adapted to handle additional operators and additional levels of precedence. However, it cannot be used for top-down parsing because it is left recursive.

The following non-left-recursive variant of the expression grammar (4.1) will be used for top-down parsing:

$$\begin{aligned} E &\rightarrow T E' \\ E' &\rightarrow + T E' \mid \epsilon \\ T &\rightarrow F T' \\ T' &\rightarrow * F T' \mid \epsilon \\ F &\rightarrow ( E ) \mid \text{id} \end{aligned} \tag{4.2}$$

The following grammar treats  $+$  and  $*$  alike, so it is useful for illustrating techniques for handling ambiguities during parsing:

$$E \rightarrow E + E \mid E * E \mid ( E ) \mid \text{id} \quad (4.3)$$

Here,  $E$  represents expressions of all types. Grammar (4.3) permits more than one parse tree for expressions like  $a + b * c$ .

### 4.1.3 Syntax Error Handling

The remainder of this section considers 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 in connection with specific parsing methods.

If a compiler had to process only correct programs, its design and implementation would be simplified greatly. However, a compiler is expected to assist the programmer in locating and tracking down errors that inevitably creep into programs, despite the programmer's best efforts. Strikingly, few languages have been designed with error handling in mind, even though errors are so commonplace. Our civilization would be radically different if spoken languages had the same requirements for syntactic accuracy as computer languages. Most programming language specifications do not describe how a compiler should respond to errors; error handling 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 handling of errors.

Common programming errors can occur at many different levels.

- *Lexical* errors include misspellings of identifiers, keywords, or operators — e.g., the use of an identifier `ellipseSize` instead of `ellipseSize` — and missing quotes around text intended as a string.
- *Syntactic* errors include misplaced semicolons or extra or missing braces; that is, “{” or “}.” As another example, in C or Java, the appearance of a `case` statement without an enclosing `switch` is a syntactic error (however, this situation is usually allowed by the parser and caught later in the processing, as the compiler attempts to generate code).
- *Semantic* errors include type mismatches between operators and operands, e.g., the return of a value in a Java method with result type `void`.
- *Logical* errors can be anything from incorrect reasoning on the part of the programmer to the use in a C program of the assignment operator `=` instead of the comparison operator `==`. The program containing `=` may be well formed; however, it may not reflect the programmer's intent.

The precision of parsing methods allows syntactic errors to be detected very efficiently. Several parsing methods, such as the LL and LR methods, detect

an error as soon as possible; that is, when the stream of tokens from the lexical analyzer cannot be parsed further according to the grammar for the language. More precisely, they have the *viable-prefix property*, meaning that they detect that an error has occurred as soon as they see a prefix of the input that cannot be completed to form a string in the language.

Another reason for emphasizing error recovery during parsing is that many errors appear syntactic, whatever their cause, and are exposed when parsing cannot continue. A few semantic errors, such as type mismatches, can also be detected efficiently; however, accurate detection of semantic and logical errors at compile time is in general a difficult task.

The error handler in a parser has goals that are simple to state but challenging to realize:

- Report the presence of errors clearly and accurately.
- Recover from each error quickly enough to detect subsequent errors.
- Add minimal overhead to the processing of correct programs.

Fortunately, common errors are simple ones, and a relatively straightforward error-handling mechanism often suffices.

How should an error handler report the presence of an error? At the very least, it must 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 is to print the offending line with a pointer to the position at which an error is detected.

#### 4.1.4 Error-Recovery Strategies

Once an error is detected, how should the parser recover? Although no strategy has proven itself universally acceptable, a few methods have broad applicability. The simplest approach is for the parser to quit with an informative error message when it detects the first error. Additional errors are often uncovered if the parser can restore itself to a state where processing of the input can continue with reasonable hopes that the further processing will provide meaningful diagnostic information. If errors pile up, it is better for the compiler to give up after exceeding some error limit than to produce an annoying avalanche of “spurious” errors.

The balance of this section is devoted to the following recovery strategies: panic-mode, phrase-level, error-productions, and global-correction.

##### Panic-Mode Recovery

With this method, on discovering an error, the parser discards input symbols one at a time until one of a designated set of *synchronizing tokens* is found. The synchronizing tokens are usually delimiters, such as semicolon or `}`, whose role in the source program is clear and unambiguous. The compiler designer

must select the synchronizing tokens appropriate for the source language. While panic-mode correction often skips a considerable amount of input without checking it for additional errors, it has the advantage of simplicity, and, unlike some methods to be considered later, is guaranteed not to go into an infinite loop.

### Phrase-Level Recovery

On discovering an error, a parser may perform local correction on the remaining input; that is, it may replace a prefix of the remaining input by some string that allows the parser to continue. A typical local correction is to replace a comma by a semicolon, delete an extraneous semicolon, or insert a missing semicolon. The choice of the local correction is left to the compiler designer. Of course, we must be careful to choose replacements that do not lead to infinite loops, as would be the case, for example, if we always inserted something on the input ahead of the current input symbol.

Phrase-level replacement has been used in several error-repairing compilers, as it can correct any input string. Its major drawback is the difficulty it has in coping with situations in which the actual error has occurred before the point of detection.

### Error Productions

By anticipating common errors that might be encountered, we can augment the grammar for the language at hand with productions that generate the erroneous constructs. A parser constructed from a grammar augmented by these error productions detects the anticipated errors when an error production is used during parsing. The parser can then generate appropriate error diagnostics about the erroneous construct that has been recognized in the input.

### Global Correction

Ideally, we would like a compiler to make as few changes as possible in processing an incorrect input string. There are algorithms for choosing a minimal sequence of changes to obtain a globally least-cost correction. Given an incorrect input string  $x$  and grammar  $G$ , these algorithms will find a parse tree for a related string  $y$ , such that the number of insertions, deletions, and changes of tokens required to transform  $x$  into  $y$  is as small as possible. Unfortunately, these methods are in general too costly to implement in terms of time and space, so these techniques are currently only of theoretical interest.

Do note that a closest correct program may not be what the programmer had in mind. Nevertheless, the notion of least-cost correction provides a yardstick for evaluating error-recovery techniques, and has been used for finding optimal replacement strings for phrase-level recovery.

## 4.2 Context-Free Grammars

Grammars were introduced in Section 2.2 to systematically describe the syntax of programming language constructs like expressions and statements. Using a syntactic variable *stmt* to denote statements and variable *expr* to denote expressions, the production

$$stmt \rightarrow \text{if} ( expr ) stmt \text{ else } stmt \quad (4.4)$$

specifies the structure of this form of conditional statement. Other productions then define precisely what an *expr* is and what else a *stmt* can be.

This section reviews the definition of a context-free grammar and introduces terminology for talking about parsing. In particular, the notion of derivations is very helpful for discussing the order in which productions are applied during parsing.

### 4.2.1 The Formal Definition of a Context-Free Grammar

From Section 2.2, a context-free grammar (grammar for short) consists of terminals, nonterminals, a start symbol, and productions.

1. *Terminals* are the basic symbols from which strings are formed. The term “token name” is a synonym for “terminal” and frequently we will use the word “token” for terminal when it is clear that we are talking about just the token name. We assume that the terminals are the first components of the tokens output by the lexical analyzer. In (4.4), the terminals are the keywords **if** and **else** and the symbols “(” and “).”
2. *Nonterminals* are syntactic variables that denote sets of strings. In (4.4), *stmt* and *expr* are nonterminals. The sets of strings denoted by nonterminals help define the language generated by the grammar. Nonterminals impose a hierarchical structure on the language that is key to syntax analysis and translation.
3. In a grammar, one nonterminal is distinguished as the *start symbol*, and the set of strings it denotes is the language generated by the grammar. Conventionally, the productions for the start symbol are listed first.
4. The productions of a grammar specify the manner in which the terminals and nonterminals can be combined to form strings. Each *production* consists of:
  - (a) A nonterminal called the *head* or *left side* of the production; this production defines some of the strings denoted by the head.
  - (b) The symbol  $\rightarrow$ . Sometimes  $::=$  has been used in place of the arrow.
  - (c) A *body* or *right side* consisting of zero or more terminals and nonterminals. The components of the body describe one way in which strings of the nonterminal at the head can be constructed.

**Example 4.5 :** The grammar in Fig. 4.2 defines simple arithmetic expressions. In this grammar, the terminal symbols are

**id + - \* / ( )**

The nonterminal symbols are *expression*, *term* and *factor*, and *expression* is the start symbol    $\square$

|                   |               |                                 |
|-------------------|---------------|---------------------------------|
| <i>expression</i> | $\rightarrow$ | <i>expression</i> + <i>term</i> |
| <i>expression</i> | $\rightarrow$ | <i>expression</i> - <i>term</i> |
| <i>expression</i> | $\rightarrow$ | <i>term</i>                     |
| <i>term</i>       | $\rightarrow$ | <i>term</i> * <i>factor</i>     |
| <i>term</i>       | $\rightarrow$ | <i>term</i> / <i>factor</i>     |
| <i>term</i>       | $\rightarrow$ | <i>factor</i>                   |
| <i>factor</i>     | $\rightarrow$ | ( <i>expression</i> )           |
| <i>factor</i>     | $\rightarrow$ | <b>id</b>                       |

Figure 4.2: Grammar for simple arithmetic expressions

## 4.2.2 Notational Conventions

To avoid always having to state that “these are the terminals,” “these are the nonterminals,” and so on, the following notational conventions for grammars will be used throughout the remainder of this book.

1. These symbols are terminals:
  - (a) Lowercase letters early in the alphabet, such as *a*, *b*, *c*.
  - (b) Operator symbols such as +, \*, and so on.
  - (c) Punctuation symbols such as parentheses, comma, and so on.
  - (d) The digits 0, 1, . . . , 9.
  - (e) Boldface strings such as **id** or **if**, each of which represents a single terminal symbol.
2. These symbols are nonterminals:
  - (a) Uppercase letters early in the alphabet, such as *A*, *B*, *C*.
  - (b) The letter *S*, which, when it appears, is usually the start symbol.
  - (c) Lowercase, italic names such as *expr* or *stmt*.
  - (d) When discussing programming constructs, uppercase letters may be used to represent nonterminals for the constructs. For example, nonterminals for expressions, terms, and factors are often represented by *E*, *T*, and *F*, respectively.

3. Uppercase letters late in the alphabet, such as  $X, Y, Z$ , represent *grammar symbols*; that is, either nonterminals or terminals.
4. Lowercase letters late in the alphabet, chiefly  $u, v, \dots, z$ , represent (possibly empty) strings of terminals.
5. Lowercase Greek letters,  $\alpha, \beta, \gamma$  for example, represent (possibly empty) strings of grammar symbols. Thus, a generic production can be written as  $A \rightarrow \alpha$ , where  $A$  is the head and  $\alpha$  the body.
6. A set of productions  $A \rightarrow \alpha_1, A \rightarrow \alpha_2, \dots, A \rightarrow \alpha_k$  with a common head  $A$  (call them *A-productions*), may be written  $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_k$ . Call  $\alpha_1, \alpha_2, \dots, \alpha_k$  the *alternatives* for  $A$ .
7. Unless stated otherwise, the head of the first production is the start symbol.

**Example 4.6:** Using these conventions, the grammar of Example 4.5 can be rewritten concisely as

$$\begin{aligned} E &\rightarrow E + T \mid E - T \mid T \\ T &\rightarrow T * F \mid T / F \mid F \\ F &\rightarrow ( E ) \mid \text{id} \end{aligned}$$

The notational conventions tell us that  $E, T$ , and  $F$  are nonterminals, with  $E$  the start symbol. The remaining symbols are terminals.  $\square$

### 4.2.3 Derivations

The construction of a parse tree can be made precise by taking a derivational view, in which productions are treated as rewriting rules. Beginning with the start symbol, each rewriting step replaces a nonterminal by the body of one of its productions. This derivational view corresponds to the top-down construction of a parse tree, but the precision afforded by derivations will be especially helpful when bottom-up parsing is discussed. As we shall see, bottom-up parsing is related to a class of derivations known as “rightmost” derivations, in which the rightmost nonterminal is rewritten at each step.

For example, consider the following grammar, with a single nonterminal  $E$ , which adds a production  $E \rightarrow - E$  to the grammar (4.3):

$$E \rightarrow E + E \mid E * E \mid - E \mid ( E ) \mid \text{id} \quad (4.7)$$

The production  $E \rightarrow - E$  signifies that if  $E$  denotes an expression, then  $- E$  must also denote an expression. The replacement of a single  $E$  by  $- E$  will be described by writing

$$E \Rightarrow -E$$

which is read, “ $E$  derives  $-E$ .” The production  $E \rightarrow ( E )$  can be applied to replace any instance of  $E$  in any string of grammar symbols by  $(E)$ , e.g.,  $E * E \Rightarrow (E) * E$  or  $E * E \Rightarrow E * (E)$ . We can take a single  $E$  and repeatedly apply productions in any order to get a sequence of replacements. For example,

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\mathbf{id})$$

We call such a sequence of replacements a *derivation* of  $-(\mathbf{id})$  from  $E$ . This derivation provides a proof that the string  $-(\mathbf{id})$  is one particular instance of an expression.

For a general definition of derivation, consider a nonterminal  $A$  in the middle of a sequence of grammar symbols, as in  $\alpha A \beta$ , where  $\alpha$  and  $\beta$  are arbitrary strings of grammar symbols. Suppose  $A \rightarrow \gamma$  is a production. Then, we write  $\alpha A \beta \Rightarrow \alpha \gamma \beta$ . The symbol  $\Rightarrow$  means, “derives in one step.” When a sequence of derivation steps  $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n$  rewrites  $\alpha_1$  to  $\alpha_n$ , we say  $\alpha_1$  *derives*  $\alpha_n$ . Often, we wish to say, “derives in zero or more steps.” For this purpose, we can use the symbol  $\stackrel{*}{\Rightarrow}$ . Thus,

1.  $\alpha \stackrel{*}{\Rightarrow} \alpha$ , for any string  $\alpha$ , and
2. If  $\alpha \stackrel{*}{\Rightarrow} \beta$  and  $\beta \Rightarrow \gamma$ , then  $\alpha \stackrel{*}{\Rightarrow} \gamma$ .

Likewise,  $\stackrel{+}{\Rightarrow}$  means, “derives in one or more steps.”

If  $S \stackrel{*}{\Rightarrow} \alpha$ , where  $S$  is the start symbol of a grammar  $G$ , we say that  $\alpha$  is a *sentential form* of  $G$ . Note that a sentential form may contain both terminals and nonterminals, and may be empty. A *sentence* of  $G$  is a sentential form with no nonterminals. The *language generated by* a grammar is its set of sentences. Thus, a string of terminals  $w$  is in  $L(G)$ , the language generated by  $G$ , if and only if  $w$  is a sentence of  $G$  (or  $S \stackrel{*}{\Rightarrow} w$ ). A language that can be generated by a grammar is said to be a *context-free language*. If two grammars generate the same language, the grammars are said to be *equivalent*.

The string  $-(\mathbf{id} + \mathbf{id})$  is a sentence of grammar (4.7) because there is a derivation

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(\mathbf{id} + E) \Rightarrow -(\mathbf{id} + \mathbf{id}) \quad (4.8)$$

The strings  $E, -E, -(E), \dots, -(\mathbf{id} + \mathbf{id})$  are all sentential forms of this grammar. We write  $E \stackrel{*}{\Rightarrow} -(\mathbf{id} + \mathbf{id})$  to indicate that  $-(\mathbf{id} + \mathbf{id})$  can be derived from  $E$ .

At each step in a derivation, there are two choices to be made. We need to choose which nonterminal to replace, and having made this choice, we must pick a production with that nonterminal as head. For example, the following alternative derivation of  $-(\mathbf{id} + \mathbf{id})$  differs from derivation (4.8) in the last two steps:

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(E + \mathbf{id}) \Rightarrow -(\mathbf{id} + \mathbf{id}) \quad (4.9)$$

Each nonterminal is replaced by the same body in the two derivations, but the order of replacements is different.

To understand how parsers work, we shall consider derivations in which the nonterminal to be replaced at each step is chosen as follows:

1. In *leftmost* derivations, the leftmost nonterminal in each sentential is always chosen. If  $\alpha \Rightarrow \beta$  is a step in which the leftmost nonterminal in  $\alpha$  is replaced, we write  $\alpha \underset{lm}{\Rightarrow} \beta$ .
2. In *rightmost* derivations, the rightmost nonterminal is always chosen; we write  $\alpha \underset{rm}{\Rightarrow} \beta$  in this case.

Derivation (4.8) is leftmost, so it can be rewritten as

$$E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E + E) \underset{lm}{\Rightarrow} -(\mathbf{id} + E) \underset{lm}{\Rightarrow} -(\mathbf{id} + \mathbf{id})$$

Note that (4.9) is a rightmost derivation.

Using our notational conventions, every leftmost step can be written as  $wA\gamma \underset{lm}{\Rightarrow} w\delta\gamma$ , where  $w$  consists of terminals only,  $A \rightarrow \delta$  is the production applied, and  $\gamma$  is a string of grammar symbols. To emphasize that  $\alpha$  derives  $\beta$  by a leftmost derivation, we write  $\alpha \underset{lm}{\overset{*}{\Rightarrow}} \beta$ . If  $S \underset{lm}{\overset{*}{\Rightarrow}} \alpha$ , then we say that  $\alpha$  is a *left-sentential form* of the grammar at hand.

Analogous definitions hold for rightmost derivations. Rightmost derivations are sometimes called *canonical* derivations.

#### 4.2.4 Parse Trees and Derivations

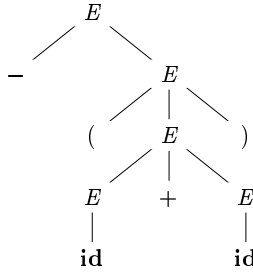
A parse tree is a graphical representation of a derivation that filters out the order in which productions are applied to replace nonterminals. Each interior node of a parse tree represents the application of a production. The interior node is labeled with the nonterminal  $A$  in the head of the production; the children of the node are labeled, from left to right, by the symbols in the body of the production by which this  $A$  was replaced during the derivation.

For example, the parse tree for  $-(\mathbf{id} + \mathbf{id})$  in Fig. 4.3, results from the derivation (4.8) as well as derivation (4.9).

The leaves of a parse tree are labeled by nonterminals or terminals and, read from left to right, constitute a sentential form, called the *yield* or *frontier* of the tree.

To see the relationship between derivations and parse trees, consider any derivation  $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n$ , where  $\alpha_1$  is a single nonterminal  $A$ . For each sentential form  $\alpha_i$  in the derivation, we can construct a parse tree whose yield is  $\alpha_i$ . The process is an induction on  $i$ .

**BASIS:** The tree for  $\alpha_1 = A$  is a single node labeled  $A$ .

Figure 4.3: Parse tree for  $-(\text{id} + \text{id})$ 

**INDUCTION:** Suppose we already have constructed a parse tree with yield  $\alpha_{i-1} = X_1X_2 \cdots X_k$  (note that according to our notational conventions, each grammar symbol  $X_i$  is either a nonterminal or a terminal). Suppose  $\alpha_i$  is derived from  $\alpha_{i-1}$  by replacing  $X_j$ , a nonterminal, by  $\beta = Y_1Y_2 \cdots Y_m$ . That is, at the  $i$ th step of the derivation, production  $X_j \rightarrow \beta$  is applied to  $\alpha_{i-1}$  to derive  $\alpha_i = X_1X_2 \cdots X_{j-1}\beta X_{j+1} \cdots X_k$ .

To model this step of the derivation, find the  $j$ th non- $\epsilon$  leaf from the left in the current parse tree. This leaf is labeled  $X_j$ . Give this leaf  $m$  children, labeled  $Y_1, Y_2, \dots, Y_m$ , from the left. As a special case, if  $m = 0$ , then  $\beta = \epsilon$ , and we give the  $j$ th leaf one child labeled  $\epsilon$ .

**Example 4.10:** The sequence of parse trees constructed from the derivation (4.8) is shown in Fig. 4.4. In the first step of the derivation,  $E \Rightarrow -E$ . To model this step, add two children, labeled  $-$  and  $E$ , to the root  $E$  of the initial tree. The result is the second tree.

In the second step of the derivation,  $-E \Rightarrow -(E)$ . Consequently, add three children, labeled  $($ ,  $E$ , and  $)$ , to the leaf labeled  $E$  of the second tree, to obtain the third tree with yield  $-(E)$ . Continuing in this fashion we obtain the complete parse tree as the sixth tree.  $\square$

Since a parse tree ignores variations in the order in which symbols in sentential forms are replaced, there is a many-to-one relationship between derivations and parse trees. For example, both derivations (4.8) and (4.9), are associated with the same final parse tree of Fig. 4.4.

In what follows, we shall frequently parse by producing a leftmost or a rightmost derivation, since there is a one-to-one relationship between parse trees and either leftmost or rightmost derivations. Both leftmost and rightmost derivations pick a particular order for replacing symbols in sentential forms, so they too filter out variations in the order. It is not hard to show that every parse tree has associated with it a unique leftmost and a unique rightmost derivation.

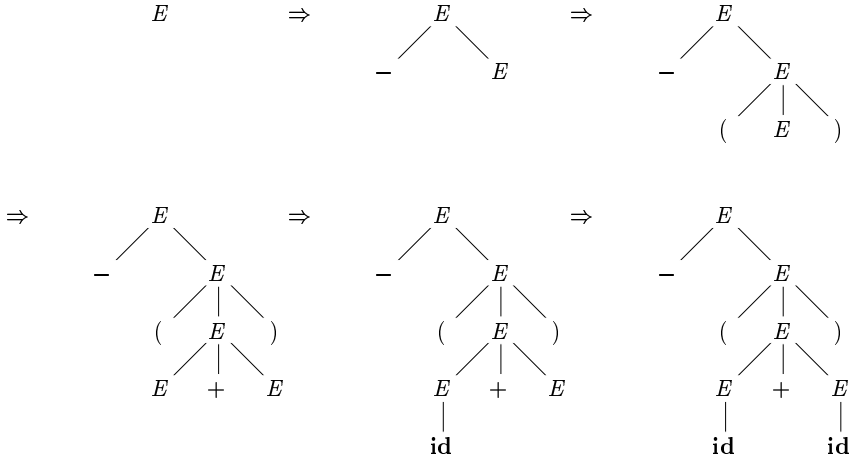


Figure 4.4: Sequence of parse trees for derivation (4.8)

### 4.2.5 Ambiguity

From Section 2.2.4, a grammar that produces more than one parse tree for some sentence is said to be *ambiguous*. Put another way, an ambiguous grammar is one that produces more than one leftmost derivation or more than one rightmost derivation for the same sentence.

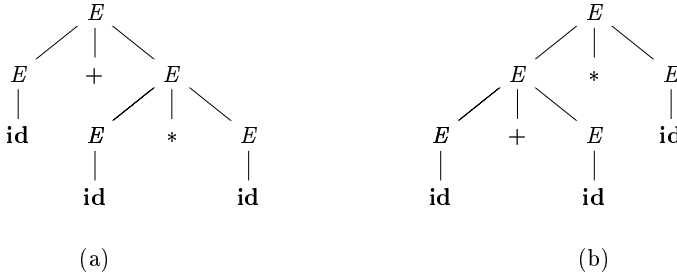
**Example 4.11 :** The arithmetic expression grammar (4.3) permits two distinct leftmost derivations for the sentence **id + id \* id**:

$$\begin{array}{ll}
 E & \Rightarrow E + E & E & \Rightarrow E * E \\
 & \Rightarrow \mathbf{id} + E & & \Rightarrow E + E * E \\
 & \Rightarrow \mathbf{id} + E * E & & \Rightarrow \mathbf{id} + E * E \\
 & \Rightarrow \mathbf{id} + \mathbf{id} * E & & \Rightarrow \mathbf{id} + \mathbf{id} * E \\
 & \Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id} & & \Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}
 \end{array}$$

The corresponding parse trees appear in Fig. 4.5.

Note that the parse tree of Fig. 4.5(a) reflects the commonly assumed precedence of + and \*, while the tree of Fig. 4.5(b) does not. That is, it is customary to treat operator \* as having higher precedence than +, corresponding to the fact that we would normally evaluate an expression like  $a + b * c$  as  $a + (b * c)$ , rather than as  $(a + b) * c$ .  $\square$

For most parsers, it is desirable that the grammar be made unambiguous, for if it is not, we cannot uniquely determine which parse tree to select for a sentence. In other cases, it is convenient to use carefully chosen ambiguous grammars, together with *disambiguating rules* that “throw away” undesirable parse trees, leaving only one tree for each sentence.

Figure 4.5: Two parse trees for **id+id\*id**

### 4.2.6 Verifying the Language Generated by a Grammar

Although compiler designers rarely do so for a complete programming-language grammar, it is useful to be able to reason that a given set of productions generates a particular language. Troublesome constructs can be studied by writing a concise, abstract grammar and studying the language that it generates. We shall construct such a grammar for conditional statements below.

A proof that a grammar  $G$  generates a language  $L$  has two parts: show that every string generated by  $G$  is in  $L$ , and conversely that every string in  $L$  can indeed be generated by  $G$ .

**Example 4.12:** Consider the following grammar:

$$S \rightarrow ( S ) S \mid \epsilon \quad (4.13)$$

It may not be initially apparent, but this simple grammar generates all strings of balanced parentheses, and only such strings. To see why, we shall show first that every sentence derivable from  $S$  is balanced, and then that every balanced string is derivable from  $S$ . To show that every sentence derivable from  $S$  is balanced, we use an inductive proof on the number of steps  $n$  in a derivation.

**BASIS:** The basis is  $n = 1$ . The only string of terminals derivable from  $S$  in one step is the empty string, which surely is balanced.

**INDUCTION:** Now assume that all derivations of fewer than  $n$  steps produce balanced sentences, and consider a leftmost derivation of exactly  $n$  steps. Such a derivation must be of the form

$$S \xRightarrow{lm} (S)S \xRightarrow{lm}^* (x)S \xRightarrow{lm}^* (x)y$$

The derivations of  $x$  and  $y$  from  $S$  take fewer than  $n$  steps, so by the inductive hypothesis  $x$  and  $y$  are balanced. Therefore, the string  $(x)y$  must be balanced. That is, it has an equal number of left and right parentheses, and every prefix has at least as many left parentheses as right.

Having thus shown that any string derivable from  $S$  is balanced, we must next show that every balanced string is derivable from  $S$ . To do so, use induction on the length of a string.

**BASIS:** If the string is of length 0, it must be  $\epsilon$ , which is balanced.

**INDUCTION:** First, observe that every balanced string has even length. Assume that every balanced string of length less than  $2n$  is derivable from  $S$ , and consider a balanced string  $w$  of length  $2n$ ,  $n \geq 1$ . Surely  $w$  begins with a left parenthesis. Let  $(x)$  be the shortest nonempty prefix of  $w$  having an equal number of left and right parentheses. Then  $w$  can be written as  $w = (x)y$  where both  $x$  and  $y$  are balanced. Since  $x$  and  $y$  are of length less than  $2n$ , they are derivable from  $S$  by the inductive hypothesis. Thus, we can find a derivation of the form

$$S \Rightarrow (S)S \xRightarrow{*} (x)S \xRightarrow{*} (x)y$$

proving that  $w = (x)y$  is also derivable from  $S$ .  $\square$

### 4.2.7 Context-Free Grammars Versus Regular Expressions

Before leaving this section on grammars and their properties, we establish that grammars are a more powerful notation than regular expressions. Every construct that can be described by a regular expression can be described by a grammar, but not vice-versa. Alternatively, every regular language is a context-free language, but not vice-versa.

For example, the regular expression  $(a|b)^*abb$  and the grammar

$$\begin{array}{lll} A_0 & \rightarrow & aA_0 \mid bA_0 \mid aA_1 \\ A_1 & \rightarrow & bA_2 \\ A_2 & \rightarrow & bA_3 \\ A_3 & \rightarrow & \epsilon \end{array}$$

describe the same language, the set of strings of  $a$ 's and  $b$ 's ending in  $abb$ .

We can construct mechanically a grammar to recognize the same language as a nondeterministic finite automaton (NFA). The grammar above was constructed from the NFA in Fig. 3.24 using the following construction:

1. For each state  $i$  of the NFA, create a nonterminal  $A_i$ .
2. If state  $i$  has a transition to state  $j$  on input  $a$ , add the production  $A_i \rightarrow aA_j$ . If state  $i$  goes to state  $j$  on input  $\epsilon$ , add the production  $A_i \rightarrow A_j$ .
3. If  $i$  is an accepting state, add  $A_i \rightarrow \epsilon$ .
4. If  $i$  is the start state, make  $A_i$  be the start symbol of the grammar.

On the other hand, the language  $L = \{a^n b^n \mid n \geq 1\}$  with an equal number of  $a$ 's and  $b$ 's is a prototypical example of a language that can be described by a grammar but not by a regular expression. To see why, suppose  $L$  were the language defined by some regular expression. We could construct a DFA  $D$  with a finite number of states, say  $k$ , to accept  $L$ . Since  $D$  has only  $k$  states, for an input beginning with more than  $k$   $a$ 's,  $D$  must enter some state twice, say  $s_i$ , as in Fig. 4.6. Suppose that the path from  $s_i$  back to itself is labeled with a sequence  $a^{j-i}$ . Since  $a^i b^i$  is in the language, there must be a path labeled  $b^i$  from  $s_i$  to an accepting state  $f$ . But, then there is also a path from the initial state  $s_0$  through  $s_i$  to  $f$  labeled  $a^j b^i$ , as shown in Fig. 4.6. Thus,  $D$  also accepts  $a^j b^i$ , which is not in the language, contradicting the assumption that  $L$  is the language accepted by  $D$ .

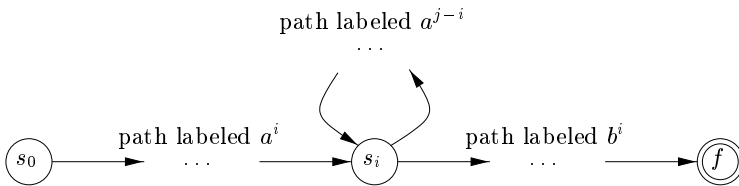


Figure 4.6: DFA  $D$  accepting both  $a^i b^i$  and  $a^j b^i$ .

Colloquially, we say that “finite automata cannot count,” meaning that a finite automaton cannot accept a language like  $\{a^n b^n \mid n \geq 1\}$  that would require it to keep count of the number of  $a$ 's before it sees the  $b$ 's. Likewise, “a grammar can count two items but not three,” as we shall see when we consider non-context-free language constructs in Section 4.3.5.

## 4.2.8 Exercises for Section 4.2

**Exercise 4.2.1:** Consider the context-free grammar:

$$S \rightarrow S S + \mid S S * \mid a$$

and the string  $aa + a*$ .

- Give a leftmost derivation for the string.
- Give a rightmost derivation for the string.
- Give a parse tree for the string.

! d) Is the grammar ambiguous or unambiguous? Justify your answer.

! e) Describe the language generated by this grammar.

**Exercise 4.2.2:** Repeat Exercise 4.2.1 for each of the following grammars and strings:

- a)  $S \rightarrow 0 S 1 \mid 0 1$  with string 000111.
- b)  $S \rightarrow + S S \mid * S S \mid a$  with string  $+ * aaa$ .
- ! c)  $S \rightarrow S ( S ) S \mid \epsilon$  with string  $((())())$ .
- ! d)  $S \rightarrow S + S \mid S S \mid ( S ) \mid S * \mid a$  with string  $(a + a) * a$ .
- ! e)  $S \rightarrow ( L ) \mid a$  and  $L \rightarrow L , S \mid S$  with string  $((a, a), a, (a))$ .
- !! f)  $S \rightarrow a S b S \mid b S a S \mid \epsilon$  with string  $aabbab$ .
- ! g) The following grammar for boolean expressions:

$$\begin{aligned}
 bexpr &\rightarrow bexpr \textbf{ or } bterm \mid bterm \\
 bterm &\rightarrow bterm \textbf{ and } bfactor \mid bfactor \\
 bfactor &\rightarrow \textbf{ not } bfactor \mid ( bexpr ) \mid \textbf{ true } \mid \textbf{ false }
 \end{aligned}$$

**Exercise 4.2.3:** Design grammars for the following languages:

- a) The set of all strings of 0s and 1s such that every 0 is immediately followed by at least one 1.
- ! b) The set of all strings of 0s and 1s that are *palindromes*; that is, the string reads the same backward as forward.
- ! c) The set of all strings of 0s and 1s with an equal number of 0s and 1s.
- !! d) The set of all strings of 0s and 1s with an unequal number of 0s and 1s.
- ! e) The set of all strings of 0s and 1s in which 011 does not appear as a substring.
- !! f) The set of all strings of 0s and 1s of the form  $xy$ , where  $x \neq y$  and  $x$  and  $y$  are of the same length.
- ! **Exercise 4.2.4:** There is an extended grammar notation in common use. In this notation, square and curly braces in production bodies are metasymbols (like  $\rightarrow$  or  $\mid$ ) with the following meanings:

- i) Square braces around a grammar symbol or symbols denotes that these constructs are optional. Thus, production  $A \rightarrow X [Y] Z$  has the same effect as the two productions  $A \rightarrow X Y Z$  and  $A \rightarrow X Z$ .
- ii) Curly braces around a grammar symbol or symbols says that these symbols may be repeated any number of times, including zero times. Thus,  $A \rightarrow X \{Y Z\}$  has the same effect as the infinite sequence of productions  $A \rightarrow X$ ,  $A \rightarrow X Y Z$ ,  $A \rightarrow X Y Z Y Z$ , and so on.

Show that these two extensions do not add power to grammars; that is, any language that can be generated by a grammar with these extensions can be generated by a grammar without the extensions.

**Exercise 4.2.5:** Use the braces described in Exercise 4.2.4 to simplify the following grammar for statement blocks and conditional statements:

$$\begin{array}{ll} stmt & \rightarrow \text{if } expr \text{ then } stmt \text{ else } stmt \\ & \quad | \text{if } stmt \text{ then } stmt \\ & \quad | \text{begin } stmtList \text{ end} \\ stmtList & \rightarrow stmt ; stmtList \mid stmt \end{array}$$

**! Exercise 4.2.6:** Extend the idea of Exercise 4.2.4 to allow any regular expression of grammar symbols in the body of a production. Show that this extension does not allow grammars to define any new languages.

**! Exercise 4.2.7:** A grammar symbol  $X$  (terminal or nonterminal) is *useless* if there is no derivation of the form  $S \xRightarrow{*} wXy \xRightarrow{*} wxy$ . That is,  $X$  can never appear in the derivation of any sentence.

- Give an algorithm to eliminate from a grammar all productions containing useless symbols.
- Apply your algorithm to the grammar:

$$\begin{array}{ll} S & \rightarrow 0 \mid A \\ A & \rightarrow AB \\ B & \rightarrow 1 \end{array}$$

**Exercise 4.2.8:** The grammar in Fig. 4.7 generates declarations for a single numerical identifier; these declarations involve four different, independent properties of numbers.

$$\begin{array}{ll} stmt & \rightarrow \text{declare id } optionList \\ optionList & \rightarrow optionList \text{ option } \mid \epsilon \\ option & \rightarrow mode \mid scale \mid precision \mid base \\ mode & \rightarrow \text{real} \mid \text{complex} \\ scale & \rightarrow \text{fixed} \mid \text{floating} \\ precision & \rightarrow \text{single} \mid \text{double} \\ base & \rightarrow \text{binary} \mid \text{decimal} \end{array}$$

Figure 4.7: A grammar for multi-attribute declarations

- Generalize the grammar of Fig. 4.7 by allowing  $n$  options  $A_i$ , for some fixed  $n$  and for  $i = 1, 2, \dots, n$ , where  $A_i$  can be either  $a_i$  or  $b_i$ . Your grammar should use only  $O(n)$  grammar symbols and have a total length of productions that is  $O(n)$ .

- ! b) The grammar of Fig. 4.7 and its generalization in part (a) allow declarations that are contradictory and/or redundant, such as:

```
declare foo real fixed real floating
```

We could insist that the syntax of the language forbid such declarations; that is, every declaration generated by the grammar has exactly one value for each of the  $n$  options. If we do, then for any fixed  $n$  there is only a finite number of legal declarations. The language of legal declarations thus has a grammar (and also a regular expression), as any finite language does. The obvious grammar, in which the start symbol has a production for every legal declaration has  $n!$  productions and a total production length of  $O(n \times n!)$ . You must do better: a total production length that is  $O(n2^n)$ .

- !! c) Show that any grammar for part (b) must have a total production length of at least  $2^n$ .
- d) What does part (c) say about the feasibility of enforcing nonredundancy and noncontradiction among options in declarations via the syntax of the programming language?

## 4.3 Writing a Grammar

Grammars are capable of describing most, but not all, of the syntax of programming languages. For instance, the requirement that identifiers be declared before they are used, cannot be described by a context-free grammar. Therefore, the sequences of tokens accepted by a parser form a superset of the programming language; subsequent phases of the compiler must analyze the output of the parser to ensure compliance with rules that are not checked by the parser.

This section begins with a discussion of how to divide work between a lexical analyzer and a parser. We then consider several transformations that could be applied to get a grammar more suitable for parsing. One technique can eliminate ambiguity in the grammar, and other techniques — left-recursion elimination and left factoring — are useful for rewriting grammars so they become suitable for top-down parsing. We conclude this section by considering some programming language constructs that cannot be described by any grammar.

### 4.3.1 Lexical Versus Syntactic Analysis

As we observed in Section 4.2.7, everything that can be described by a regular expression can also be described by a grammar. We may therefore reasonably ask: “Why use regular expressions to define the lexical syntax of a language?” There are several reasons.

1. Separating the syntactic structure of a language into lexical and non-lexical parts provides a convenient way of modularizing the front end of a compiler into two manageable-sized components.
2. The lexical rules of a language are frequently quite simple, and to describe them we do not need a notation as powerful as grammars.
3. Regular expressions generally provide a more concise and easier-to-understand notation for tokens than grammars.
4. More efficient lexical analyzers can be constructed automatically from regular expressions than from arbitrary grammars.

There are no firm guidelines as to what to put into the lexical rules, as opposed to the syntactic rules. Regular expressions are most useful for describing the structure of constructs such as identifiers, constants, keywords, and white space. Grammars, on the other hand, are most useful for describing nested structures such as balanced parentheses, matching begin-end's, corresponding if-then-else's, and so on. These nested structures cannot be described by regular expressions.

### 4.3.2 Eliminating Ambiguity

Sometimes an ambiguous grammar can be rewritten to eliminate the ambiguity. As an example, we shall eliminate the ambiguity from the following “dangling-else” grammar:

$$\begin{array}{ll}
 stmt & \rightarrow \text{if } expr \text{ then } stmt \\
 & | \text{if } expr \text{ then } stmt \text{ else } stmt \\
 & | \text{other}
 \end{array} \tag{4.14}$$

Here “**other**” stands for any other statement. According to this grammar, the compound conditional statement

**if  $E_1$  then  $S_1$  else if  $E_2$  then  $S_2$  else  $S_3$**

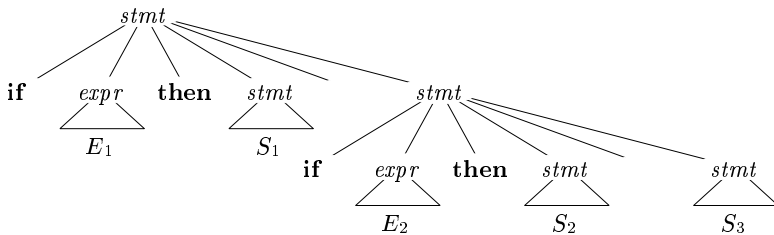


Figure 4.8: Parse tree for a conditional statement

has the parse tree shown in Fig. 4.8.<sup>1</sup> Grammar (4.14) is ambiguous since the string

$$\text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2 \quad (4.15)$$

has the two parse trees shown in Fig. 4.9.

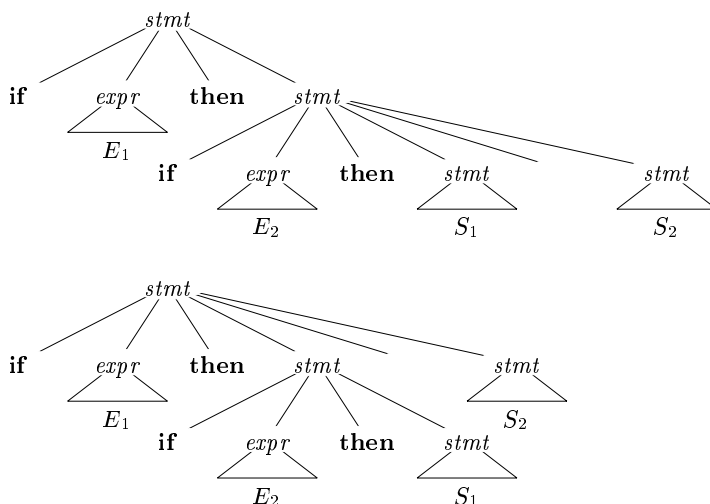


Figure 4.9: Two parse trees for an ambiguous sentence

In all programming languages with conditional statements of this form, the first parse tree is preferred. The general rule is, “Match each `if` with the closest unmatched **then**.”<sup>2</sup> This disambiguating rule can theoretically be incorporated directly into a grammar, but in practice it is rarely built into the productions.

**Example 4.16:** We can rewrite the dangling-else grammar (4.14) as the following unambiguous grammar. The idea is that a statement appearing between a **then** and an **else** must be “matched”; that is, the interior statement must not end with an unmatched or open **then**. A matched statement is either an **if-then-else** statement containing no open statements or it is any other kind of unconditional statement. Thus, we may use the grammar in Fig. 4.10. This grammar generates the same strings as the dangling-else grammar (4.14), but it allows only one parsing for string (4.15); namely, the one that associates each **else** with the closest previous unmatched **then**.  $\square$

<sup>1</sup>The subscripts on  $E$  and  $S$  are just to distinguish different occurrences of the same nonterminal, and do not imply distinct nonterminals.

<sup>2</sup> We should note that *C* and its derivatives are included in this class. Even though the *C* family of languages do not use the keyword **then**, its role is played by the closing parenthesis for the condition that follows **if**.

|                     |   |                                                                                       |
|---------------------|---|---------------------------------------------------------------------------------------|
| <i>stmt</i>         | → | <i>matched_stmt</i>                                                                   |
|                     |   | <i>open_stmt</i>                                                                      |
| <i>matched_stmt</i> | → | <b>if</b> <i>expr</i> <b>then</b> <i>matched_stmt</i> <b>else</b> <i>matched_stmt</i> |
|                     |   | <b>other</b>                                                                          |
| <i>open_stmt</i>    | → | <b>if</b> <i>expr</i> <b>then</b> <i>stmt</i>                                         |
|                     |   | <b>if</b> <i>expr</i> <b>then</b> <i>matched_stmt</i> <b>else</b> <i>open_stmt</i>    |

Figure 4.10: Unambiguous grammar for if-then-else statements

### 4.3.3 Elimination of Left Recursion

A grammar is *left recursive* if it has a nonterminal  $A$  such that there is a derivation  $A \xRightarrow{+} A\alpha$  for some string  $\alpha$ . Top-down parsing methods cannot handle left-recursive grammars, so a transformation is needed to eliminate left recursion. In Section 2.4.5, we discussed *immediate left recursion*, where there is a production of the form  $A \rightarrow A\alpha$ . Here, we study the general case. In Section 2.4.5, we showed how the left-recursive pair of productions  $A \rightarrow A\alpha \mid \beta$  could be replaced by the non-left-recursive productions:

$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \mid \epsilon \end{aligned}$$

without changing the strings derivable from  $A$ . This rule by itself suffices for many grammars.

**Example 4.17:** The non-left-recursive expression grammar (4.2), repeated here,

$$\begin{aligned} E &\rightarrow T E' \\ E' &\rightarrow + T E' \mid \epsilon \\ T &\rightarrow F T' \\ T' &\rightarrow * F T' \mid \epsilon \\ F &\rightarrow ( E ) \mid \text{id} \end{aligned}$$

is obtained by eliminating immediate left recursion from the expression grammar (4.1). The left-recursive pair of productions  $E \rightarrow E + T \mid T$  are replaced by  $E \rightarrow T E'$  and  $E' \rightarrow + T E' \mid \epsilon$ . The new productions for  $T$  and  $T'$  are obtained similarly by eliminating immediate left recursion.  $\square$

Immediate left recursion can be eliminated by the following technique, which works for any number of  $A$ -productions. First, group the productions as

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

where no  $\beta_i$  begins with an  $A$ . Then, replace the  $A$ -productions by

$$\begin{array}{l} A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A' \\ A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon \end{array}$$

The nonterminal  $A$  generates the same strings as before but is no longer left recursive. This procedure eliminates all left recursion from the  $A$  and  $A'$  productions (provided no  $\alpha_i$  is  $\epsilon$ ), but it does not eliminate left recursion involving derivations of two or more steps. For example, consider the grammar

$$\begin{array}{l} S \rightarrow A a \mid b \\ A \rightarrow A c \mid S d \mid \epsilon \end{array} \quad (4.18)$$

The nonterminal  $S$  is left recursive because  $S \Rightarrow Aa \Rightarrow Sda$ , but it is not immediately left recursive.

Algorithm 4.19, below, systematically eliminates left recursion from a grammar. It is guaranteed to work if the grammar has no cycles (derivations of the form  $A \stackrel{+}{\Rightarrow} A$ ) or  $\epsilon$ -productions (productions of the form  $A \rightarrow \epsilon$ ). Cycles can be eliminated systematically from a grammar, as can  $\epsilon$ -productions (see Exercises 4.4.6 and 4.4.7).

**Algorithm 4.19:** Eliminating left recursion.

**INPUT:** Grammar  $G$  with no cycles or  $\epsilon$ -productions.

**OUTPUT:** An equivalent grammar with no left recursion.

**METHOD:** Apply the algorithm in Fig. 4.11 to  $G$ . Note that the resulting non-left-recursive grammar may have  $\epsilon$ -productions.  $\square$

- 1) arrange the nonterminals in some order  $A_1, A_2, \dots, A_n$ .
- 2) **for** ( each  $i$  from 1 to  $n$  ) {
- 3)     **for** ( each  $j$  from 1 to  $i - 1$  ) {
- 4)         replace each production of the form  $A_i \rightarrow A_j \gamma$  by the  
               productions  $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \cdots \mid \delta_k \gamma$ , where  
                $A_j \rightarrow \delta_1 \mid \delta_2 \mid \cdots \mid \delta_k$  are all current  $A_j$ -productions
- 5)     }
- 6)     eliminate the immediate left recursion among the  $A_i$ -productions
- 7) } }

Figure 4.11: Algorithm to eliminate left recursion from a grammar

The procedure in Fig. 4.11 works as follows. In the first iteration for  $i = 1$ , the outer for-loop of lines (2) through (7) eliminates any immediate left recursion among  $A_1$ -productions. Any remaining  $A_1$  productions of the form  $A_1 \rightarrow A_l \alpha$  must therefore have  $l > 1$ . After the  $i - 1$ st iteration of the outer for-loop, all nonterminals  $A_k$ , where  $k < i$ , are “cleaned”; that is, any production  $A_k \rightarrow A_l \alpha$ , must have  $l > k$ . As a result, on the  $i$ th iteration, the inner loop

of lines (3) through (5) progressively raises the lower limit in any production  $A_i \rightarrow A_m \alpha$ , until we have  $m \geq i$ . Then, eliminating immediate left recursion for the  $A_i$  productions at line (6) forces  $m$  to be greater than  $i$ .

**Example 4.20:** Let us apply Algorithm 4.19 to the grammar (4.18). Technically, the algorithm is not guaranteed to work, because of the  $\epsilon$ -production, but in this case, the production  $A \rightarrow \epsilon$  turns out to be harmless.

We order the nonterminals  $S, A$ . There is no immediate left recursion among the  $S$ -productions, so nothing happens during the outer loop for  $i = 1$ . For  $i = 2$ , we substitute for  $S$  in  $A \rightarrow S d$  to obtain the following  $A$ -productions.

$$A \rightarrow A c \mid A a d \mid b d \mid \epsilon$$

Eliminating the immediate left recursion among these  $A$ -productions yields the following grammar.

$$\begin{aligned} S &\rightarrow A a \mid b \\ A &\rightarrow b d A' \mid A' \\ A' &\rightarrow c A' \mid a d A' \mid \epsilon \end{aligned}$$

□

### 4.3.4 Left Factoring

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive, or top-down, parsing. When the choice between two alternative  $A$ -productions is not clear, we may be able to rewrite the productions to defer the decision until enough of the input has been seen that we can make the right choice.

For example, if we have the two productions

$$\begin{aligned} stmt &\rightarrow \text{if } expr \text{ then } stmt \text{ else } stmt \\ &\mid \text{if } expr \text{ then } stmt \end{aligned}$$

on seeing the input **if**, we cannot immediately tell which production to choose to expand  $stmt$ . In general, if  $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$  are two  $A$ -productions, and the input begins with a nonempty string derived from  $\alpha$ , we do not know whether to expand  $A$  to  $\alpha\beta_1$  or  $\alpha\beta_2$ . However, we may defer the decision by expanding  $A$  to  $\alpha A'$ . Then, after seeing the input derived from  $\alpha$ , we expand  $A'$  to  $\beta_1$  or to  $\beta_2$ . That is, left-factored, the original productions become

$$\begin{aligned} A &\rightarrow \alpha A' \\ A' &\rightarrow \beta_1 \mid \beta_2 \end{aligned}$$

**Algorithm 4.21:** Left factoring a grammar.

**INPUT:** Grammar  $G$ .

**OUTPUT:** An equivalent left-factored grammar.

**METHOD:** For each nonterminal  $A$ , find the longest prefix  $\alpha$  common to two or more of its alternatives. If  $\alpha \neq \epsilon$  — i.e., there is a nontrivial common prefix — replace all of the  $A$ -productions  $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$ , where  $\gamma$  represents all alternatives that do not begin with  $\alpha$ , by

$$\begin{aligned} A &\rightarrow \alpha A' \mid \gamma \\ A' &\rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n \end{aligned}$$

Here  $A'$  is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix.  $\square$

**Example 4.22:** The following grammar abstracts the “dangling-else” problem:

$$\begin{aligned} S &\rightarrow i E t S \mid i E t S e S \mid a \\ E &\rightarrow b \end{aligned} \tag{4.23}$$

Here,  $i$ ,  $t$ , and  $e$  stand for **if**, **then**, and **else**;  $E$  and  $S$  stand for “conditional expression” and “statement.” Left-factored, this grammar becomes:

$$\begin{aligned} S &\rightarrow i E t S S' \mid a \\ S' &\rightarrow e S \mid \epsilon \\ E &\rightarrow b \end{aligned} \tag{4.24}$$

Thus, we may expand  $S$  to  $iEtSS'$  on input  $i$ , and wait until  $iEtS$  has been seen to decide whether to expand  $S'$  to  $eS$  or to  $\epsilon$ . Of course, these grammars are both ambiguous, and on input  $e$ , it will not be clear which alternative for  $S'$  should be chosen. Example 4.33 discusses a way out of this dilemma.  $\square$

### 4.3.5 Non-Context-Free Language Constructs

A few syntactic constructs found in typical programming languages cannot be specified using grammars alone. Here, we consider two of these constructs, using simple abstract languages to illustrate the difficulties.

**Example 4.25:** The language in this example abstracts the problem of checking that identifiers are declared before they are used in a program. The language consists of strings of the form  $wcw$ , where the first  $w$  represents the declaration of an identifier  $w$ ,  $c$  represents an intervening program fragment, and the second  $w$  represents the use of the identifier.

The abstract language is  $L_1 = \{wcw \mid w \text{ is in } (\mathbf{a|b})^*\}$ .  $L_1$  consists of all words composed of a repeated string of  $a$ ’s and  $b$ ’s separated by  $c$ , such as  $aabcaab$ . While it is beyond the scope of this book to prove it, the non-context-freeness of  $L_1$  directly implies the non-context-freeness of programming languages like C and Java, which require declaration of identifiers before their use and which allow identifiers of arbitrary length.

For this reason, a grammar for C or Java does not distinguish among identifiers that are different character strings. Instead, all identifiers are represented

by a token such as **id** in the grammar. In a compiler for such a language, the semantic-analysis phase checks that identifiers are declared before they are used.  $\square$

**Example 4.26:** The non-context-free language in this example abstracts the problem of checking that the number of formal parameters in the declaration of a function agrees with the number of actual parameters in a use of the function. The language consists of strings of the form  $a^n b^m c^n d^m$ . (Recall  $a^n$  means  $a$  written  $n$  times.) Here  $a^n$  and  $b^m$  could represent the formal-parameter lists of two functions declared to have  $n$  and  $m$  arguments, respectively, while  $c^n$  and  $d^m$  represent the actual-parameter lists in calls to these two functions.

The abstract language is  $L_2 = \{a^n b^m c^n d^m \mid n \geq 1 \text{ and } m \geq 1\}$ . That is,  $L_2$  consists of strings in the language generated by the regular expression  $\mathbf{a^*b^*c^*d^*}$  such that the number of  $a$ 's and  $c$ 's are equal and the number of  $b$ 's and  $d$ 's are equal. This language is not context free.

Again, the typical syntax of function declarations and uses does not concern itself with counting the number of parameters. For example, a function call in C-like language might be specified by

$$\begin{array}{ll} \text{stmt} & \rightarrow \text{id ( expr\_list )} \\ \text{expr\_list} & \rightarrow \text{expr\_list , expr} \\ & \mid \text{expr} \end{array}$$

with suitable productions for *expr*. Checking that the number of parameters in a call is correct is usually done during the semantic-analysis phase.  $\square$

### 4.3.6 Exercises for Section 4.3

**Exercise 4.3.1:** The following is a grammar for regular expressions over symbols  $a$  and  $b$  only, using  $+$  in place of  $|$  for union, to avoid conflict with the use of vertical bar as a metasyMBOL in grammars:

$$\begin{array}{ll} \text{rexpr} & \rightarrow \text{rexpr} + \text{rterm} \mid \text{rterm} \\ \text{rterm} & \rightarrow \text{rterm} \text{ rfactor} \mid \text{rfactor} \\ \text{rfactor} & \rightarrow \text{rfactor} * \mid \text{rprimary} \\ \text{rprimary} & \rightarrow \mathbf{a} \mid \mathbf{b} \end{array}$$

- Left factor this grammar.
- Does left factoring make the grammar suitable for top-down parsing?
- In addition to left factoring, eliminate left recursion from the original grammar.
- Is the resulting grammar suitable for top-down parsing?

**Exercise 4.3.2:** Repeat Exercise 4.3.1 on the following grammars:

- a) The grammar of Exercise 4.2.1.
- b) The grammar of Exercise 4.2.2(a).
- c) The grammar of Exercise 4.2.2(c).
- d) The grammar of Exercise 4.2.2(e).
- e) The grammar of Exercise 4.2.2(g).

**! Exercise 4.3.3:** The following grammar is proposed to remove the “dangling-else ambiguity” discussed in Section 4.3.2:

$$\begin{array}{ll}
 stmt & \rightarrow \text{if } expr \text{ then } stmt \\
 & \quad | \quad matchedStmt \\
 matchedStmt & \rightarrow \text{if } expr \text{ then } matchedStmt \text{ else } stmt \\
 & \quad | \quad \text{other}
 \end{array}$$

Show that this grammar is still ambiguous.