CHAPTER
7

Language Translation Principles

TABLE OF CONTENTS

You are now multilingual because you understand at least four languages—English, C, Pep/9 assembly language, and machine language. The first is a natural language, and the other three are artificial languages.

The fundamental question of computer science

Keeping that in mind, let's turn to the fundamental question of computer science, which is: What can be automated? We use computers to automate everything from writing payroll checks to correcting spelling errors in manuscripts. Although computer science has been moderately successful in automating the translation of natural languages, say from German to English, it has been quite successful in translating artificial languages. You have already learned how to translate between the three artificial languages of C, Pep/9 assembly language, and machine language. Compilers and assemblers automate this translation process for artificial languages.

Automatic translation

Because each level of a computer system has its own artificial language, the automatic translation between these languages is at the very heart of computer science. Computer scientists have developed a rich body of theory about artificial languages and the automation of the translation process. This chapter introduces the theory and shows how it applies to the translation of C and Pep/9 assembly language.

Syntax and semantics

Two attributes of an artificial language are its syntax and semantics. A computer language's *syntax* is the set of rules that a program listing must obey to be declared a valid program of the language. Its *semantics* is the meaning or logic behind the valid program. Operationally, a syntactically correct program will be successfully translated by a translator program. The semantics of the language determine the result produced by the translated program when the object program is executed.

The part of an automatic translator that compares the source program with the language's syntax is called the *parser*. The part that assigns meaning to the source program is called the *code generator*. Most computer science theory applies to the syntactic rather than the semantic part of the translation process.

Three common techniques to describe a language's syntax are

Techniques to specify syntax

- Grammars
- Finite-state machines
- Regular expressions

This chapter introduces grammars and finite-state machines. It shows how to construct a software finite-state machine to aid in the parsing process. The last section shows a complete program, including code generation, that automatically translates between two languages. Space limitations preclude a presentation of regular expressions.

7.1 Languages, Grammars, and Parsing

Every language has an alphabet. Formally, an *alphabet* is a finite, nonempty set of characters. For example, the C alphabet is the nonempty set

The C alphabet

```
{  a,  b,  c,  d,  e,  f,  g,  h,  i,  j,  k,  l,  m,  n,
   o,  p,  q,  r,  s,  t,  u,  v,  w,  x,  y,  z,  A,  B,
   C,  D,  E,  F,  G,  H,  I,  J,  K,  L,  M,  N,  O,  P,
   Q,  R,  S,  T,  U,  V,  W,  X,  Y,  Z,  0,  1,  2,  3,
   4,  5,  6,  7,  8,  9,  +,  -,  *,  /,  =,  <,  >,  [,
   ],  (,  ),  {,  },  .,  ,,  :,  ;,  &,  !,  %,  ',  ",
   _,  \,  #,  ?,  ^,  |,  ~}
```

The alphabet for Pep/9 assembly language is similar except for the punctuation characters, as shown in the following set:

The Pep/9 assembly language alphabet

```
{  a,  b,  c,  d,  e,  f,  g,  h,  i,  j,  k,  l,  m,  n,
   o,  p,  q,  r,  s,  t,  u,  v,  w,  x,  y,  z,  A,  B,
   C,  D,  E,  F,  G,  H,  I,  J,  K,  L,  M,  N,  O,  P,
   Q,  R,  S,  T,  U,  V,  W,  X,  Y,  Z,  0,  1,  2,  3,
   4,  5,  6,  7,  8,  9,  \,  .,  ,,  :,  ;,  ',  "}
```

Another example of an alphabet is the alphabet for the language of real numbers, not in scientific notation. It is the set

The alphabet for real numbers

```
{  0,  1,  2,  3,  4,  5,  6,  7,  8,  9,  +,  -,  .  }
```

Concatenation

Concatenation

An abstract data type is a set of possible values together with a set of operations on the values. Notice that an alphabet is a set of values. The pertinent operation on this set of values is *concatenation*, which is simply the joining of two or more characters to form a string. An example from the C alphabet is the concatenation of `!` and `=` to form the string `!=`. In the Pep/9 assembly alphabet, you can concatenate `0` and `x` to make `0x`, the prefix of a hexadecimal constant. And in the language of real numbers, you can concatenate `-`, `2`, `3`, `.`, and `7` to make `-23.7`.

Concatenation applies not only to individual characters in an alphabet to construct a string, but also to strings concatenated to construct longer strings. From the C alphabet, you can concatenate `void`, `printBar`, and `(int n)` to produce the procedure heading

```
void printBar(int n)
```

The length of a string is the number of characters in the string. The string `void` has a length of four. The string of length zero, called the *empty string*, is denoted by the Greek letter ε to distinguish it from the English characters in an alphabet. Its concatenation properties are

The empty string

$$\varepsilon x = x\varepsilon = x$$

where x is a string. The empty string is useful for describing syntax rules.

Identity elements

In mathematics terminology, ε is the identity element for the concatenation operation. In general, an *identity element*, i, for an operation is one that does not change a value, x, when x is operated on by i.

Example 7.1 One is the identity element for multiplication because

$$1 \cdot x = x \cdot 1 = x$$

and true is the identity element for the AND operation because

$$\text{true AND } q = q \text{ AND true} = q. \quad ■$$

Languages

The closure of an alphabet

If T is an alphabet, the closure of T, denoted T^*, is the set of all possible strings formed by concatenating elements from T. T^* is extremely large. For example, if T is the set of characters and punctuation marks of the English alphabet, T^* includes all the sentences in the collected works of Shakespeare, in the English Bible, and in all the English encyclopedias ever published. It includes all strings of those characters ever printed in all the libraries in all the world throughout history, and then some.

Not only does it include all those meaningful strings, it includes meaningless ones as well. Here are some elements of T^* for the English alphabet:

```
To be or not to be, that is the question.
Go fly a kite.
Here over highly toward?
alkeu jfoj ,9nm20mfq23jk l?x!jeo
```

Some elements of T^* where T is the alphabet of the language for real numbers are

```
-2894.01
24
+78.3.80
--234---
6
```

You can easily construct many other elements of T^* with the two alphabets just mentioned. Because strings can be infinitely long, the closure of any alphabet has an infinite number of elements.

What is a language? In the examples of T^* that were just presented, some of the strings are in the language and some are not. In the English example, the first two strings are valid English sentences; that is, they are in the language. The last two strings are not in the language. A *language* is a subset of the closure of its alphabet. Of the infinite number of strings you can construct from concatenating strings of characters from its alphabet, only some will be in the language.

The definition of a language

Example 7.2 Consider the following two elements of T^*, where T is the alphabet for the C language:

```
#include <stdio.h>
int main() {
   printf("Valid");
   return 0;
}
```

```
#include <stdio.h>
int main(); {
   printf("Valid");
   return 0;
}
```

The first element of T^* is in the C language, but the second is not because it has a syntax error. ■

Grammars

To define a language, you need a way to specify which of the many elements of T^* are in the language and which are not. A *grammar* is a system that specifies how you can concatenate the characters of alphabet T to form a legal string in a language. Formally, a grammar contains four parts:

The four parts of a grammar

- N, a nonterminal alphabet
- T, a terminal alphabet
- P, a set of rules of production
- S, the start symbol, which is an element of N

An element from the nonterminal alphabet, N, represents a string of characters from the terminal alphabet, T. A nonterminal symbol is frequently enclosed in angle brackets, `<>`. You see the terminal symbols when you read the language. You do not see the nonterminal symbols. The rules of production use the nonterminals to describe the structure of the language, which may not be readily apparent when you read the language.

Example 7.3 In the C grammar, the nonterminal <compound-statement> might represent the following group of terminals:

```
{
  int i;
  scanf("%d", &i);
  i++;
  printf("%d", i);
}
```

The listing of a C program always contains terminals, never nonterminals. You would never see a C listing like this:

```
#include <stdio.h>
main()
<compound-statement>
```

The nonterminal symbol, <compound-statement>, is useful for describing the structure of a C program. ▮

Every grammar has a special nonterminal called the *start symbol*, *S*. Notice that *N* is a set, but *S* is not. *S* is one of the elements of set *N*. The start symbol, along with the rules of production, *P*, enables you to decide whether a string of terminals is a valid sentence in the language. If, starting from *S*, you can generate the string of terminals using the rules of production, then the string is a valid sentence.

A Grammar for C Identifiers

The grammar in FIGURE 7.1 specifies a C identifier. Even though a C identifier can use any uppercase or lowercase letter or digit, to keep the example small, this grammar permits only the letters `a`, `b`, and `c` and the digits `1`, `2`, and `3`. You know the rules for constructing an identifier. The first character must be a letter, and the remaining characters, if any, can be letters or digits in any combination.

This grammar has three nonterminals, namely, <identifier>, <letter>, and <digit>. The start symbol is <identifier>, one of the elements from the set of nonterminals.

FIGURE 7.1
A grammar for C identifiers.

N = { <identifier> , <letter> , <digit> }
T = { `a` , `b` , `c` , `1` , `2` , `3` }
P = the productions

1. <identifier> → <letter>
2. <identifier> → <identifier> <letter>
3. <identifier> → <identifier> <digit>
4. <letter> → `a`
5. <letter> → `b`
6. <letter> → `c`
7. <digit> → `1`
8. <digit> → `2`
9. <digit> → `3`

S = <identifier>

Productions

The rules of production are of the form

$$A \rightarrow w$$

where A is a nonterminal and w is a string of terminals and nonterminals. The symbol → means "produces." You should read production rule number 3 in Figure 7.1 as, "An identifier produces an identifier followed by a digit."

Derivations

The grammar specifies the language by a process called a *derivation*. To derive a valid sentence in the language, you begin with the start symbol and substitute for nonterminals from the rules of production until you get a string of terminals. The following is a derivation of the identifier `cab3` from this grammar. The symbol ⇒ means "derives in one step."

<identifier>	⇒ <identifier> <digit>	Rule 3
	⇒ <identifier> `3`	Rule 9
	⇒ <identifier> <letter> `3`	Rule 2
	⇒ <identifier> `b 3`	Rule 5
	⇒ <identifier> <letter> `b 3`	Rule 2
	⇒ <identifier> `a b 3`	Rule 4
	⇒ <letter> `a b 3`	Rule 1
	⇒ `c a b 3`	Rule 6

Next to each derivation step is the production rule on which the substitution is based. For example, Rule 2,

$$\text{<identifier>} \rightarrow \text{<identifier>} \text{ <letter>}$$

was used to substitute for <identifier> in the derivation step

$$\text{<identifier> } \mathtt{3} \Rightarrow \text{<identifier> <letter> } \mathtt{3}$$

You should read this derivation step as "Identifier followed by 3 derives in one step identifier followed by letter followed by 3."

Analogous to the closure operation on an alphabet is the closure of the derivation operation. The symbol $\Rightarrow^*$ means "derives in zero or more steps." You can summarize the previous eight derivation steps as

$$\text{<identifier>} \Rightarrow^* \mathtt{c\ a\ b\ 3}$$

This derivation proves that `cab3` is a valid identifier because it can be derived from the start symbol, <identifier>. A language specified by a grammar consists of all the strings derivable from the start symbol using the rules of production. The grammar provides an operational test for membership in the language. If it is impossible to derive a string, the string is not in the language.

A Grammar for Signed Integers

The grammar in FIGURE 7.2 defines the language of signed integers, where `d` represents a decimal digit. The start symbol is I, which stands for *integer*. F is the first character, which is an optional sign, and M is the magnitude.

Sometimes the rules of production are not numbered and are combined on one line to conserve space on the printed page. You can write the rules of production for this grammar as

$$\begin{aligned} &\mathtt{I} \rightarrow \mathtt{FM} \\ &\mathtt{F} \rightarrow \mathtt{+} \mid \mathtt{-} \mid \varepsilon \\ &\mathtt{M} \rightarrow \mathtt{d} \mid \mathtt{dM} \end{aligned}$$

FIGURE 7.2
A grammar for signed integers.

$N = \{ \mathrm{I}, \mathrm{F}, \mathrm{M} \}$
$T = \{ +, -, \mathtt{d} \}$
$P =$ the productions
1. I → FM
2. F → +
3. F → -
4. F → ε
5. M → dM
6. M → d

$S = \mathrm{I}$

where the vertical bar, |, is the alternation operator and is read as "or." Read the last line as "M produces `d`, or `d` followed by M."

Here are some derivations of valid signed integers in this grammar:

```
I ⇒ FM          I ⇒ FM          I ⇒ FM
  ⇒ FdM           ⇒ FdM           ⇒ FdM
  ⇒ FddM          ⇒ Fdd           ⇒ FddM
  ⇒ Fddd          ⇒ dd            ⇒ FdddM
  ⇒ -ddd                          ⇒ Fdddd
                                  ⇒ +dddd
```

Note how the last step of the second derivation uses the empty string to derive `dd` from `Fdd`. It uses the production F → ε and the fact that `εd = d`.

This production rule with the empty string is a convenient way to express the fact that a positive or negative sign in front of the magnitude is optional.

Some illegal strings from this grammar are `ddd+`, `+-ddd`, and `ddd+dd`. Try to derive these strings from the grammar to convince yourself that they are not in the language. Can you informally prove from the rules of production that each of these strings is not in the language?

Recursive productions

The productions in both of the sample grammars have recursive rules in which a nonterminal is defined in terms of itself. Rule 3 of Figure 7.1 defines an <identifier> in terms of an <identifier> as

<identifier> → <identifier> <digit>

and Rule 5 of Figure 7.2 defines M in terms of M as

M → `d`M

Recursive rules produce languages with an infinite number of legal sentences. To derive an identifier, you can keep substituting <identifier> <digit> for <identifier> as long as you like to produce an arbitrarily long identifier.

As in all recursive definitions, there must be an escape hatch to provide the basis for the definition. Otherwise, the sequence of substitutions for the nonterminal could never stop. The rule M → `d` provides the basis for M in Figure 7.2.

A Context-Sensitive Grammar

The production rules for the previous grammars always contain a single nonterminal on the left side. The grammar in FIGURE 7.3 has some production rules with both a terminal and nonterminal on the left side.

Here is a derivation of a string of terminals with this grammar:

A ⇒ `a`ABC	Rule 1
⇒ `aa`ABCBC	Rule 1
⇒ `aaab`CBCBC	Rule 2
⇒ `aaab`BCCBC	Rule 3
⇒ `aaab`BCBCC	Rule 3
⇒ `aaab`BBCCC	Rule 3
⇒ `aaabb`BCCC	Rule 4
⇒ `aaabbb`CCC	Rule 4
⇒ `aaabbbc`CC	Rule 5
⇒ `aaabbbcc`C	Rule 6
⇒ `aaabbbccc`	Rule 6

An example of a substitution in this derivation is using Rule 5 in the step

`aaabbb`CCC ⇒ `aaabbbc`CC

FIGURE 7.3
A context-sensitive grammar.

N = { A , B , C }
T = { `a` , `b` , `c` }
P = the productions
1. A → `a`ABC
2. A → `ab`C
3. CB → BC
4. `b`B → `bb`
5. `b`C → `bc`
6. `c`C → `cc`

S = A

Rule 5 says that you can substitute `c` for C, but only if the C has a `b` to its left.

In the English language, to quote a phrase out of context means to quote it without regard to the other phrases that surround it. Rule 5 is an example of a context-sensitive rule. It does not permit the substitution of C by `c` unless C is in the proper context—namely, immediately to the right of a `b`.

Context-sensitive grammars

Loosely speaking, a *context-sensitive grammar* is one in which the production rules may contain more than just a single nonterminal on the left side. In contrast, grammars that are restricted to a single nonterminal on the left side of every production rule are called *context-free*. (The precise theoretical definitions of *context-sensitive* and *context-free grammars* are more restrictive than these definitions. For the sake of simplicity, this chapter uses the previous definitions, although you should be aware that a more rigorous description of the theory would not define them as we have here.)

Some other examples of valid strings in the language specified by this grammar are `abc`, `aabbcc`, and `aaaabbbbcccc`. Two examples of invalid strings are `aabc` and `cba`. You should derive these valid strings and also try to derive the invalid strings to prove their invalidity to yourself. Some experimentation with the rules should convince you that the language is the set of strings that begins with one or more `a`'s, followed by an equal number of `b`'s, followed by the same number of `c`'s. Mathematically, this language, L, can be written

$$L = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n > 0\}$$

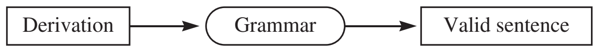
which you should read as "The language L is the set of strings $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ such that n is greater than 0." The notation $\mathtt{a}^n$ means the concatenation of n `a`'s.

The Parsing Problem

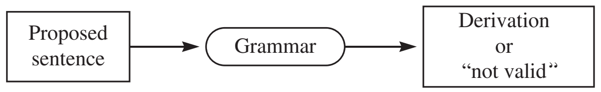
Deriving valid strings from a grammar is fairly straightforward. You can arbitrarily pick some nonterminal on the right side of the current intermediate string and select rules for the substitution repeatedly until you get a string of terminals. Such random derivations can give you many sample strings from the language.

An automatic translator, however, has a more difficult task. You give a translator a string of terminals that is supposed to be a valid sentence in an artificial language. Before the translator can produce the object code, it must determine whether the string of terminals is indeed valid. The only way to determine whether a string is valid is to derive it from the start symbol of the grammar. The translator must attempt such a derivation. If it succeeds, it knows the string is a valid sentence. The problem of determining whether a given string of terminal characters is valid for a specific grammar is called *parsing* and is illustrated schematically in FIGURE 7.4.

FIGURE 7.4
The difference between deriving an arbitrary sentence and parsing a proposed sentence.



(a) Deriving a valid sentence.



(b) The parsing problem.

Parsing a given string is more difficult than deriving an arbitrary valid string. The parsing problem is a form of searching. The parsing algorithm must search for just the right sequence of substitutions to derive the proposed string. Not only must it find the derivation if the proposed string is valid, but it must also admit the possibility that the proposed string may not be valid. If you look for a lost diamond ring in your room and do not find it, that does not mean the ring is not in your room. It may simply mean that you did not look in the right place. Similarly, if you try to find a derivation for a proposed string and do not find it, how do you know that such a derivation does not exist? A translator must be able to prove that no derivation exists if the proposed string is not valid.

A Grammar for Expressions

To see some of the difficulty a parser may encounter, consider FIGURE 7.5, which shows a grammar that describes an arithmetic infix expression. Suppose you are given the string of terminals

```
( a * a ) + a
```

and the production rules of this grammar, and are asked to parse the proposed string. The correct parse is

$E \Rightarrow E + T$	Rule 1
$\Rightarrow T + T$	Rule 2
$\Rightarrow F + T$	Rule 4
$\Rightarrow (E) + T$	Rule 5
$\Rightarrow (T) + T$	Rule 2
$\Rightarrow (T * F) + T$	Rule 3
$\Rightarrow (F * F) + T$	Rule 4

FIGURE 7.5
A grammar for expressions. Nonterminal E represents the expression. T represents a term and F a factor in the expression.

$N = \{ E, T, F \}$
$T = \{ +, *, (,), a \}$
$P =$ the productions
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow a$

$S = E$

⇒ (a * F) + T	Rule 6
⇒ (a * a) + T	Rule 6
⇒ (a * a) + F	Rule 4
⇒ (a * a) + a	Rule 6

The reason this could be difficult is that you might make a bad decision early in the parse that looks plausible at the time but that leads to a dead end. For example, you might spot the "`(`" in the string that you were given and choose Rule 5 immediately. Your attempted parse might be

E ⇒ T	Rule 2
⇒ F	Rule 4
⇒ (E)	Rule 5
⇒ (T)	Rule 2
⇒ (T * F)	Rule 3
⇒ (F * F)	Rule 4
⇒ (a * F)	Rule 6
⇒ (a * a)	Rule 6

Until now, you have seemingly made progress toward your goal of parsing the original expression because the intermediate string looks more like the original string at each successive step of the derivation. Unfortunately, now you are stuck because there is no way to get the `+ a` part of the original string.

After reaching this dead end, you may be tempted to conclude that the proposed string is invalid, but that would be a mistake. Just because you cannot find a derivation does not mean that such a derivation does not exist.

One interesting aspect of a parse is that it can be represented as a tree. The start symbol is the root of the tree. Each interior node of the tree is a nonterminal, and each leaf is a terminal. The children of an interior node are the symbols from the right side of the production rule substituted for the parent node in the derivation. The tree is called a *syntax tree*, for obvious reasons. FIGURE 7.6 shows the syntax tree for `(a * a) + a` with the grammar in Figure 7.5, and FIGURE 7.7 shows it for `dd` with the grammar in Figure 7.2.

FIGURE 7.6 The syntax tree for the parse of `(a * a) + a` in Figure 7.5.

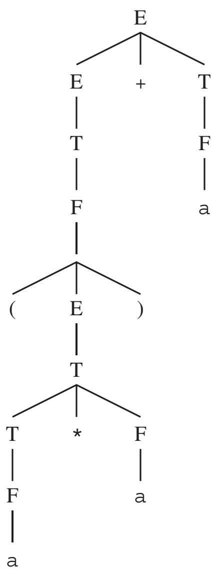
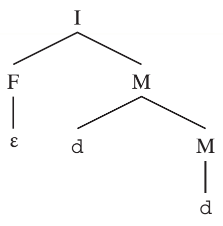


FIGURE 7.7 The syntax tree for the parse of `dd` in Figure 7.2.



A C Subset Grammar

The rules of production for the grammar in FIGURE 7.8 specify a small subset of the C language. The only primitive types in this language are integer and character. The language has no provision for constant or type declarations and does not permit reference parameters. It also omits `switch` and `for` statements. Despite these limitations, it gives an idea of how the syntax for a real language is formally defined.

FIGURE 7.8
A grammar for a subset of the C language.

```
<translation-unit> →
    <external-declaration>
    | <translation-unit> <external-declaration>
<external-declaration> →
    <function-definition>
    | <declaration>
<function-definition> →
    <type-specifier> <identifier> ( <parameter-list> ) <compound-statement>
    | <identifier> ( <parameter-list> ) <compound-statement>
<declaration> → <type-specifier> <declarator-list> ;
<type-specifier> → void | char | int
<declarator-list> →
    <identifier>
    | <declarator-list> , <identifier>
<parameter-list> →
    ε
    | <parameter-declaration>
    | <parameter-list> , <parameter-declaration>
<parameter-declaration> → <type-specifier> <identifier>
<compound-statement> → { <declaration-list> <statement-list> }
<declaration-list> →
    ε
    | <declaration>
    | <declaration-list> <declaration>
<statement-list> →
    ε
    | <statement>
    | <statement-list> <statement>
<statement> →
    <compound-statement>
    | <expression-statement>
    | <selection-statement>
    | <iteration-statement>
<expression-statement> → <expression> ;
<selection-statement> →
    if ( <expression> ) <statement>
```

(*continues*)

FIGURE 7.8
A grammar for a subset of the C language. (*continued*)

```
    | if ( <expression> ) <statement> else <statement>
<iteration-statement> →
    while ( <expression> ) <statement>
    | do <statement> while ( <expression> ) ;
<expression> →
    <relational-expression>
    | <identifier> = <expression>
<relational-expression> →
    <additive-expression>
    | <relational-expression> < <additive-expression>
    | <relational-expression> > <additive-expression>
    | <relational-expression> <= <additive-expression>
    | <relational-expression> >= <additive-expression>
<additive-expression> →
    <multiplicative-expression>
    | <additive-expression> + <multiplicative-expression>
    | <additive-expression> - <multiplicative-expression>
<multiplicative-expression> →
    <unary-expression>
    | <multiplicative-expression> * <unary-expression>
    | <multiplicative-expression> / <unary-expression>
<unary-expression> →
    <primary-expression>
    | <identifier> ( <argument-expression-list> )
<primary-expression> →
    <identifier>
    | <constant>
    | ( <expression> )
<argument-expression-list> →
    <expression>
    | <argument-expression-list> , <expression>
<constant> →
    <integer-constant>
    | <character-constant>
<integer-constant> →
    <digit>
    | <integer-constant> <digit>
```

```
<character-constant> → ' <letter> '
<identifier> →
    <letter>
    | <identifier> <letter>
    | <identifier> <digit>
<letter> →
    a | b | c | d | e | f | g | h | i | j | k | l | m |
    n | o | p | q | r | s | t | u | v | w | x | y | z |
    A | B | C | D | E | F | G | H | I | J | K | L | M |
    N | O | P | Q | R | S | T | U | V | W | X | Y | Z
<digit> →
    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

The nonterminals for this grammar are enclosed in angle brackets, <>. Any symbol not in brackets is in the terminal alphabet and may literally appear in a C program listing. The start symbol for this grammar is the nonterminal <translation-unit>.

The specification of a programming language by the rules of production of its grammar is called *Backus Naur Form*, abbreviated *BNF*. In BNF, the production symbol → is sometimes written `::=`. The Algol 60 language, designed in 1960, popularized BNF.

Backus Naur Form (BNF)

The following example of a parse with this grammar shows that

```
while ( a <= 9 )
    S1;
```

is a valid <statement>, assuming that *S1* is a valid <expression>. The parse consists of the derivation in FIGURE 7.9.

FIGURE 7.10 shows the corresponding syntax tree for this parse. The nonterminal <statement> is the root of the tree because the purpose of the parse is to show that the string is a valid <statement>.

With this example in mind, consider the task of a C compiler. The compiler has programmed into it a set of production rules similar to the rules of Figure 7.8. A programmer submits a text file containing the source program, a long string of terminals, to the compiler. First, the compiler must determine whether the string of terminal characters represents a valid C translation unit. If the string is a valid <translation-unit>, then the compiler must generate the corresponding object code in a lower-level language. If it is not, the compiler must issue an appropriate syntax error.

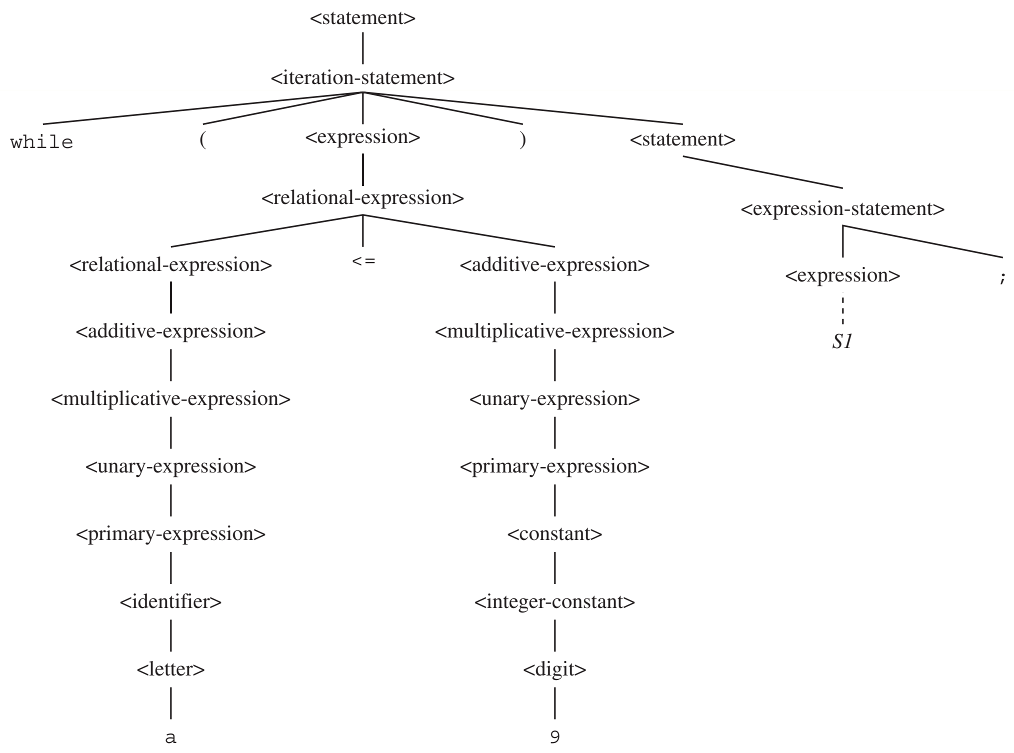
FIGURE 7.9
The derivation of nonterminal <statement> `while ( a <= 9 )` *S1*`;` for the grammar in Figure 7.8.

```
<statement>
    ⇒ <iteration-statement>
    ⇒ while ( <expression> ) <statement>
    ⇒ while ( <relational-expression> ) <statement>
    ⇒ while ( <relational-expression> <= <additive-expression> ) <statement>
    ⇒ while ( <additive-expression> <= <additive-expression> ) <statement>
    ⇒ while ( <multiplicative-expression> <= <additive-expression> ) <statement>
    ⇒ while ( <unary-expression> <= <additive-expression> ) <statement>
    ⇒ while ( <primary-expression> <= <additive-expression> ) <statement>
    ⇒ while ( <identifier> <= <additive-expression> ) <statement>
    ⇒ while ( <letter> <= <additive-expression> ) <statement>
    ⇒ while ( a <= <additive-expression> ) <statement>
    ⇒ while ( a <= <multiplicative-expression> ) <statement>
    ⇒ while ( a <= <unary-expression> ) <statement>
    ⇒ while ( a <= <primary-expression> ) <statement>
    ⇒ while ( a <= <constant> ) <statement>
    ⇒ while ( a <= <integer-constant> ) <statement>
    ⇒ while ( a <= <digit> ) <statement>
    ⇒ while ( a <= 9 ) <statement>
    ⇒ while ( a <= 9 ) <expression-statement>
    ⇒ while ( a <= 9 ) <expression> ;
    ⇒* while ( a <= 9 ) S1;
```

There are literally hundreds of rules of production in the standard C grammar. Imagine what a job the C compiler has, sorting through those rules every time you submit a program to it! Fortunately, computer science theory has developed to the point where parsing is not difficult for a compiler. When designed using the theory, C compilers can parse a program in a way that guarantees they will correctly decide which production to use for the substitution at every step of the derivation. If their parsing algorithm does not find the derivation of <translation-unit> to match the source, they can prove that such a derivation does not exist and that the proposed source program must have a syntax error.

Code generation is more difficult than parsing for compilers. The reason is that the object code must run on a specific machine produced by a specific manufacturer. Because every manufacturer's machine has a different architecture with different instruction sets, code-generation techniques

FIGURE 7.10
The syntax tree for a parse of nonterminal <statement> `while (a <= 9)` *S1*`;` for the grammar in Figure 7.9.



for one machine may not be appropriate for another. A single, standard von Neumann architecture based on theoretical concepts does not exist. Consequently, not as much theory for code generation has been developed to guide compiler designers in their compiler construction efforts.

Context Sensitivity of C

C has a context-free grammar.

It appears from Figure 7.8 that the C language is context-free. Every production rule has only a single nonterminal on the left side. This is in contrast to a context-sensitive grammar, which can have more than a single nonterminal on the left, as in Figure 7.3. Appearances are deceiving. Even though the grammar for this subset of C, as well as the full standard C language, is context-free, the language itself has some context-sensitive aspects.

Consider the grammar in Figure 7.3. How do its rules of production guarantee that the number of `c`'s at the end of a string must equal the number of `a`'s at the beginning of the string? Rules 1 and 2 guarantee that for each `a` generated, exactly one C will be generated. Rule 3 lets the C commute to the right of B. Finally, Rule 5 lets you substitute `c` for C in the context of having a `b` to the left of C. The language could not be specified by a context-free grammar because it needs Rules 3 and 5 to get the C's to the end of the string.

C is not a context-free language.

There are context-sensitive aspects of the C language that Figure 7.8 does not specify. For example, the definition of <parameter-list> allows any number of formal parameters, and the definition of <argument-expression-list> allows any number of actual parameters. You could write a C program containing a procedure with three formal parameters and a procedure call with two actual parameters that is derivable from <translation-unit> with the grammar in Figure 7.8. If you try to compile the program, however, the compiler will declare a syntax error.

The fact that the number of formal parameters must equal the number of actual parameters in C is similar to the fact that the number of `a`'s at the beginning of the string must equal the number of `c`'s at the end of the string in the language defined by the grammar in Figure 7.3. The only way to put that restriction in C's grammar would be to include many complicated, context-sensitive rules. It is easier for the compiler to parse the program with a context-free grammar and check for any violations after the parse—usually with the help of its symbol table—that the grammar cannot specify.

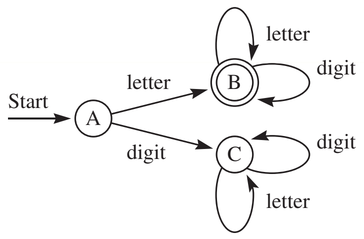
7.2 Finite-State Machines

Finite-state machines (FSMs) are another way to specify the syntax of a sentence in a language. In diagram form, an FSM is a finite set of states represented by circles called *nodes* and transitions between the states represented by *arcs* between the circles. Each arc begins at one state, ends at another, and contains an arrowhead at the ending state. Each arc is also labeled with a character from the terminal alphabet of the language.

One state of the FSM is designated as the start state and at least one, possibly more, is designated a final state. On a diagram, the start state has an incoming arrow and a final state is indicated by a double circle.

Mathematically, such a collection of nodes connected by arcs is called a *graph*. When the arcs are directed, as they are in an FSM, the structure is called a *directed graph* or *digraph*.

FIGURE 7.11
An FSM to parse an identifier.



An FSM to Parse an Identifier

FIGURE 7.11 shows an FSM that parses an identifier as defined by the grammar in Figure 7.1. The set of states is {A, B, C}. A is the start state, and B is the final state. There is a transition from A to B on a letter, from A to C on a digit, from B to B on a letter or a digit, and from C to C on a letter or a digit.

To use the FSM, imagine that the input string is written on a piece of paper tape. Start in the start state, and scan the characters on the input tape from left to right. Each time you scan the next character on the tape, make a transition to another state of the FSM. Use only the transition that is allowed by the arc corresponding to the character you have just scanned. After scanning all the input characters, if you are in a final state, the characters are a valid identifier. Otherwise they are not.

Example 7.4 To parse the string `cab3`, you would make the following transitions:

Current state: A	Input: `cab3`	Scan `c` and go to B.
Current state: B	Input: `ab3`	Scan `a` and go to B.
Current state: B	Input: `b3`	Scan `b` and go to B.
Current state: B	Input: `3`	Scan `3` and go to B.
Current state: B	Input:	Check for final state.

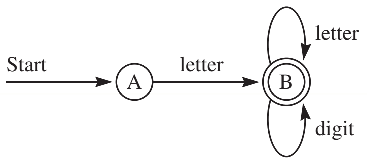
Because there is no more input and the last state is B, a final state, `cab3` is a valid identifier.

You can also represent an FSM by its state transition table. FIGURE 7.12 is the state transition table for the FSM of Figure 7.11. The table lists the next state reached by the transition from a given current state on a given input symbol.

FIGURE 7.12
The state transition table for the FSM of Figure 7.11.

Current State	Next State	
	Letter	Digit
→ A	B	C
Ⓑ	B	B
C	C	C

FIGURE 7.13
The FSM of Figure 7.11 without the failure state.



Simplified FSMs

It is often convenient to simplify the diagram for an FSM by eliminating the state whose sole purpose is to provide transitions for illegal input characters. State C in this machine is such a state. If the first character is a digit, the string will not be a valid identifier, regardless of the following characters. State C acts like a failure state. Once you make a transition to C, you can never make a transition to another state, and you know the input string eventually will be declared invalid. FIGURE 7.13 shows the simplified FSM of Figure 7.11 without the failure state.

When you parse a string with this simplified machine, you will not be able to make a transition when you encounter an illegal character in the input string. There are two ways to detect an illegal sentence in a simplified FSM:

- You may run out of input and not be in a final state.
- You may be in some state, and the next input character does not correspond to any of the transitions from that state.

FIGURE 7.14 is the corresponding state transition table for Figure 7.13. The state transition table for a simplified machine has no entry for a missing transition. Note that this table has no entry under the digit column for the current state of A. The remaining machines in this chapter are written in simplified form.

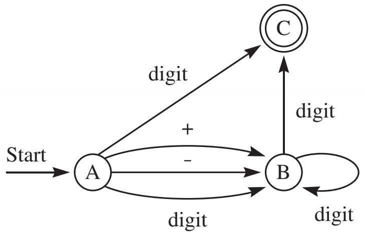
FIGURE 7.14
The state transition table for the FSM of Figure 7.13.

Current State	Next State	
	Letter	Digit
→A	B	
Ⓑ	B	B

Nondeterministic FSMs

When you parse a sentence using a grammar, frequently you must choose between several production rules for substitution in a derivation step. Similarly, nondeterministic FSMs require you to decide between more than one transition when parsing the input string. FIGURE 7.15 is a nondeterministic FSM to parse a signed integer. It is nondeterministic because there is at least one state that has more than one transition from it on the same character. For example, state A has a transition to both B and C on a digit. There is also some nondeterminism at state B because,

FIGURE 7.15
A nondeterministic FSM to parse a signed integer.



given that the next input character is a digit, a transition both to B and to C is possible.

Example 7.5 You must make the following decisions to parse `+203` with this nondeterministic FSM:

Current state: A	Input: `+203`	Scan `+` and go to B.
Current state: B	Input: `203`	Scan `2` and go to B.
Current state: B	Input: `03`	Scan `0` and go to B.
Current state: B	Input: `3`	Scan `3` and go to C.
Current state: C	Input:	Check for final state.

Because there is no more input and you are in the final state C, you have proven that the input string `+203` is a valid signed integer. ■

When parsing with rules of production, you run the risk of making an incorrect choice early in the parse. You may reach a dead end where no substitution will get your intermediate string of terminals and nonterminals closer to the given string. Just because you reach such a dead end does not necessarily mean that the string is invalid. All invalid strings will produce dead ends in an attempted parse. But even valid strings have the potential for producing dead ends if you make a wrong decision early in the derivation.

The same principle applies with nondeterministic FSMs. With the machine of Figure 7.15, if you are in the start state, A, and the next input character is `7`, you must choose between the transitions to B and to C. Suppose you choose the transition to C and then find that there is another input character to scan. Because there are no transitions from C, you have reached a dead end in your attempted parse. You must conclude, therefore, that either the input string was invalid—or it was valid and you made an incorrect choice at an earlier point.

FIGURE 7.16
The state transition table for the FSM of Figure 7.15.

Current State	Next State		
	+	–	Digit
→ A	B	B	B, C
B			B, C
Ⓒ			

FIGURE 7.16 is the state transition table for the machine of Figure 7.15. The nondeterminism is evident from the multiple entries (B, C) in the digit column. They represent a choice that must be made when attempting a parse.

Machines with Empty Transitions

In the same way that it is convenient to incorporate the empty string into production rules, it is sometimes convenient to construct FSMs with transitions on the empty string. Such transitions are called *empty transitions*. FIGURE 7.17 is an FSM that corresponds closely to the grammar in Figure 7.2 to parse a signed integer, and FIGURE 7.18 is its state transition table.

In Figure 7.17, F is the state after the first character, and M is the magnitude state analogous to the F and M nonterminals of the grammar. In the same way that a sign can be +, -, or neither, the transition from I to F can be on +, -, or ε.

FIGURE 7.17
An FSM with an empty transition to parse a signed integer.

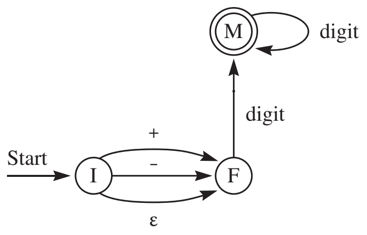


FIGURE 7.18
The state transition table for the FSM of Figure 7.17.

Current State	Next State			
	+	–	Digit	ε
→ I	F	F		F
F			M	
Ⓜ			M	

Example 7.6 To parse `32` requires the following decisions:

Current state: I	Input: `32`	Scan ε and go to F.
Current state: F	Input: `32`	Scan `3` and go to M.
Current state: M	Input: `2`	Scan `2` and go to M.
Current state: M	Input:	Check for final state.

The transition from I to F on ε does not consume an input character. When you are in state I, you can do one of three things: (a) scan + and go to F, (b) scan - and go to F, or (c) scan nothing (that is, the empty string) and go to F. ■

Machines with empty transitions are considered nondeterministic.

Machines with empty transitions are always considered nondeterministic. In Example 7.6, the nondeterminism comes from the decision you must make when you are in state I and the next character is +. You must decide whether to go from I to F on + or from I to F on ε. These are different transitions because they leave you with different input strings, even though they are transitions to the same state.

Given an FSM with empty transitions, it is always possible to transform it to an equivalent machine without the empty transitions. There are two steps in the algorithm to eliminate an empty transition.

The algorithm to remove an empty transition

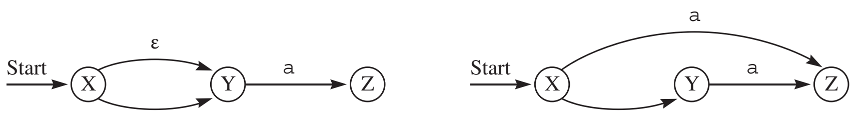
- Given a transition from p to q on ε, for every transition from q to r on `a`, add a transition from p to r on `a`.
- If q is a final state, make p a final state.

This algorithm follows from the concatenation property of ε:

$$\varepsilon a = a$$

Example 7.7 FIGURE 7.19 shows how to remove an empty transition from the machine in part (a), resulting in the equivalent machine in part (b). Because there is a transition from state X to state Y on ε, and from state Y

FIGURE 7.19 Removing an empty transition.



(a) The original FSM.

(b) The equivalent FSM without an empty transition.

to state Z on `a`, you can eliminate the empty transition if you construct a transition from state X to state Z on `a`. If you are in X, you might just as well go to Z directly on `a`. The state and remaining input will be the same as if you went from X to Z via Y on ε. ■

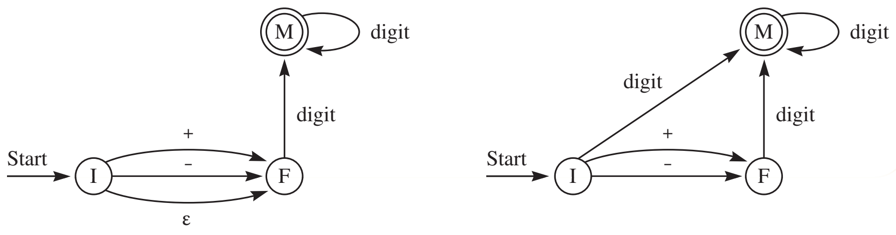
Example 7.8 FIGURE 7.20 shows this transformation on the FSM of Figure 7.17. The empty transition from I to F is replaced by the transition from I to M on digit, because there is a transition from F to M on digit. ■

In Example 7.8, there is only one transition from F to M, so the empty transition from I to F is replaced by only one transition from I to M. If an FSM has more than one transition from the destination state of the empty transition, you must add more than one transition when you eliminate the empty transition.

Example 7.9 To eliminate the empty transition from W to X in FIGURE 7.21(a), you need to replace it with two transitions, one from W to Y on `a` and one from W to Z on `b`. In this example, because X is a final state in Figure 7.21(a), W becomes a final state in the equivalent machine of Figure 7.21(b) in accordance with the second step of the algorithm. ■

Removing the empty transition from Figure 7.17 produced a deterministic machine. In general, however, removing all the empty transitions does not guarantee that the FSM is deterministic. Even though

FIGURE 7.20 Removing the empty transition from the FSM of Figure 7.17.



(a) The original FSM.

(b) The empty transition removed.

all machines with empty transitions are nondeterministic, an FSM with no empty transitions may still be nondeterministic. Figure 7.15 is such a machine, for example.

The advantage of a deterministic FSM

Given the choice, you are always better off parsing with a deterministic rather than a nondeterministic FSM. With a deterministic machine, there is no possibility of making a wrong choice with a valid input string and terminating in a dead end. If you ever terminate at a dead end, you can conclude with certainty that the input string is invalid.

Computer scientists have been able to prove that for every nondeterministic FSM there is an equivalent deterministic FSM. That is, there is a deterministic machine that recognizes exactly the same language. Unfortunately, the proof of this useful result is beyond the scope of this text. The proof consists of a recipe that tells how to construct an equivalent deterministic machine from the nondeterministic one.

Multiple Token Recognizers

The definition of a token

A *token* is a string of terminal characters that has meaning as a group. The characters usually correspond to some nonterminal in a language's grammar. For example, consider the Pep/9 assembly language statement

```
mask: .WORD 0x00FF
```

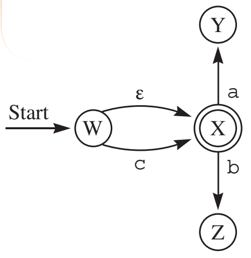
The tokens in this statement are `mask:`, `.WORD`, and `0x00FF`. Each is a set of characters from the assembly language alphabet and has meaning as a group. Their individual meanings are a symbol definition, a dot command, and a hexadecimal constant, respectively.

To a certain extent, the particular grouping of characters that you choose to form one token is arbitrary. For example, you could choose the string of characters `0x` and `00FF` to be separate tokens, `0x` for the prefix and `00FF` for the value. You would normally choose the characters of a token to be those that make the implementation of the FSM as simple as possible.

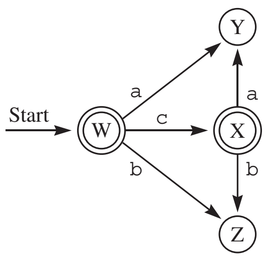
A common use of an FSM in a translator is to detect the tokens in the source string. Consider the assembler's job when confronted with this source line. Suppose the assembler has already determined that `mask:` is a symbol definition and `.WORD` is a dot command. It knows that either a decimal or hexadecimal constant can follow the dot command, so it must be programmed to accept either. It needs an FSM that recognizes both.

FIGURE 7.22(a) shows two machines for parsing a hexadecimal constant and an unsigned integer. D is the final state in the first machine, and F is the final state in the second machine for the unsigned integer. A hexadecimal constant is the digit `0`, followed by lowercase `x` or uppercase `X`, followed by one or more hexdigits, which are `0`..`9`, or `a`..`f`, or `A`..`F`. In the second machine, a digit is `0`..`9`.

FIGURE 7.21
Removing an empty transition.

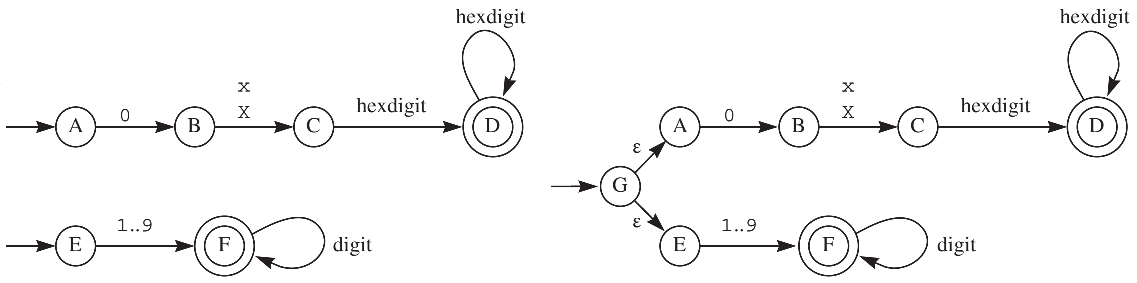


(a) The original FSM.



(b) The equivalent FSM without an empty transition.

FIGURE 7.22
Combining two machines to construct one FSM that recognizes both tokens.



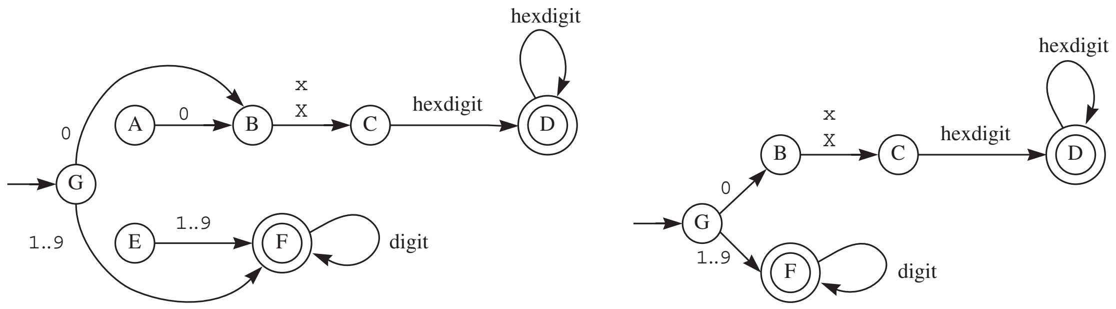
(a) Separate machines for a hexadecimal constant and an unsigned decimal integer.

(b) One nondeterministic FSM that recognizes a hexadecimal constant or an unsigned integer token.

To construct an FSM that will recognize both the hexadecimal constant and the unsigned integer, draw a new start state for the combined machine, state G in Figure 7.22(b). Then draw empty transitions from the new start state to the start state of each individual machine—in this example, from G to A and G to E. The result is one nondeterministic FSM that will recognize either token. The final state on termination tells you what token you have recognized. After the parse, if you terminate in state D, you have detected a hexadecimal constant, and if you terminate in state F, you have detected an unsigned integer.

To get the machine into a more useful form, you should eliminate the empty transitions. **FIGURE 7.23(a)** shows removal of the empty transitions for the FSM of Figure 7.22(b). After their removal, states A and E are

FIGURE 7.23
Transforming the FSM of Figure 7.22(b).



(a) Removing the empty transitions.

(b) Removing the inaccessible states.

inaccessible; that is, you can never reach them starting from the start state, regardless of the input string. Consequently, they can never affect the parse and can be eliminated from the machine, as shown in Figure 7.23(b).

As another example of when the translator needs to recognize multiple tokens, consider the assembler's job when confronted with the following two source lines:

```
NOTE: LDWA this,d ;comment 1
      NOTA        ;comment 2
```

The first token on the first line is a symbol definition. The first token on the second line is a mnemonic for a unary instruction. At the beginning of each line, the translator needs an FSM to recognize a symbol definition, which is in the form of an identifier followed immediately by a colon, or a mnemonic, which is in the form of an identifier. FIGURE 7.24 shows the appropriate multiple-token FSM.

In the first line, this machine makes the following transitions:

A to B on `N`
B to B on `O`
B to B on `T`
B to B on `E`
B to C on `:`

after which the translator halts in final state C and therefore has detected a symbol definition. In the second line, it makes the transitions

A to B on `N`
B to B on `O`
B to B on `T`
B to B on `A`

Because the next input character is not a colon, the FSM does not make the transition to state C. The translator halts in final state B and therefore has detected an identifier.

Grammars Versus FSMs

Grammars and FSMs are not equivalent in power. Of the two, grammars are more powerful than FSMs. That is, there are some languages whose syntax rules are so complex that, even though they can be specified with a grammar, they cannot be specified with an FSM. On the other hand, any language whose syntax rules are simple enough to be specified by an FSM can also be specified by a grammar.

Figure 7.1 is the grammar for an identifier, and Figure 7.13 is the FSM for an identifier. The rules for forming a valid identifier are that the first

FIGURE 7.24
An FSM to parse a Pep/9 assembly language identifier or symbol definition.

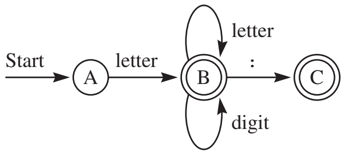
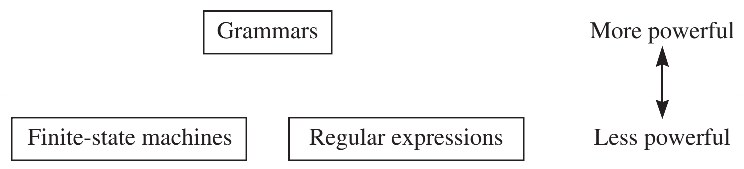


FIGURE 7.25
The power of grammars, FSMs, and regular expressions.



character must be a letter and the remaining characters must be letters or digits. These rules are so simple that an identifier can be specified by either a grammar or an FSM.

Figure 7.5 is a grammar for an expression. The language of expressions is so complex that it is mathematically impossible to specify an FSM that can parse an expression. The problem with FSMs for expressions is that you can have unlimited nested parentheses. Once the FSM scans a left parenthesis, it must transition to a state knowing that it is nested one level deep. If it scans another left parenthesis, it must transition to a state knowing that it is now nested two levels deep. If it then scans a right parenthesis, it must transition back to a state representing one level deep. It continues scanning left and right parentheses, transitioning to appropriate states for each level of nesting. To detect a valid expression, the final states must be ones with no nesting.

There is no mathematical limit in the grammar to the nesting level of an expression. Therefore, to construct an equivalent FSM, there would be no limit to the number of states. However, an FSM must have a finite number of states. Therefore, it is impossible to specify an FSM for an expression.

Although a description of regular expressions is beyond the scope of this text, how powerful are they? It turns out that for every regular expression there is an equivalent FSM, and for every FSM there is an equivalent regular expression. Consequently, FSMs and regular expressions are equal in power and are both less powerful than grammars. FIGURE 7.25 shows the power relationship between the three methods for specifying the syntax of a language.

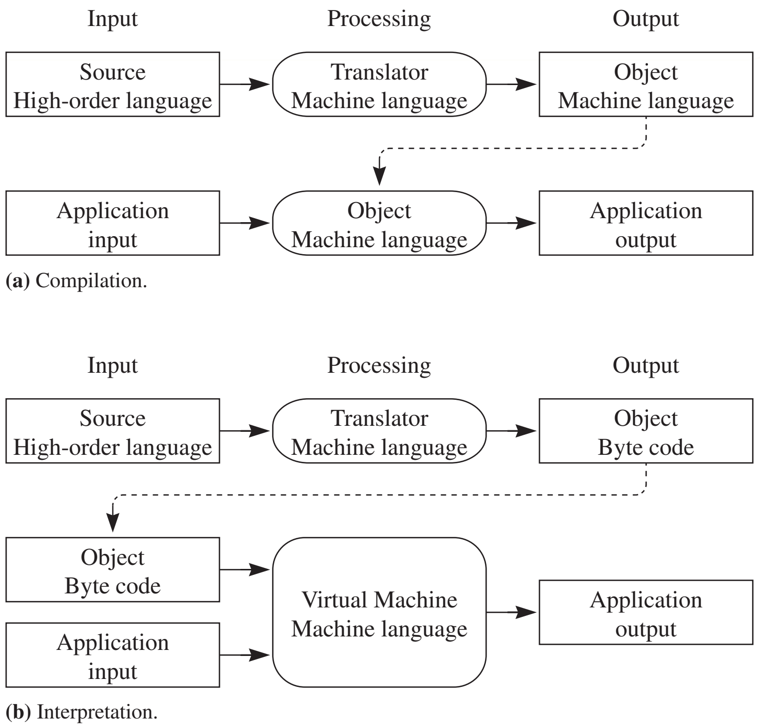
7.3 Implementing Finite-State Machines

The remainder of this chapter shows how language translators convert a source program into an object program. It uses the Java language rather than C to illustrate the translation techniques. The syntax of the Java language is similar to that of C, and it has the advantage of being object-oriented. Java provides an extensive library of graphical user interface (GUI) elements for input and output. The programs in this chapter get their input as a string of terminal characters from a single input window and send the results of the translation to the standard output window. The GUI programming details are not shown but are available with the software for this text.

Java itself is an interpreted language based on the Java Virtual Machine (JVM). FIGURE 7.26 shows the difference between a compiled language and an interpreted language. Part (a) shows the translation process for a compiled language like C. Every run in the computation process executes a machine language program with input and output. In the first run, a C compiler converts the source code in a high-level language to the object code in machine language. In the second run, the machine language object code executes, processing the application input and producing the application output.

Part (b) shows the translation process for an interpreted language like Java and Pep/9, both of which are based on virtual machines. In the first

FIGURE 7.26
The difference between compilation and interpretation.



run, the object code is byte code instead of machine language. In the second run, the object code does not execute directly. Instead, the virtual machine executes with two sources of input, the object byte code from the first run and the application input.

Advantage of interpretation

Advantages of interpretation include fast compilation time and ease of portability. It is faster to compile into byte code because byte code is at a higher level of abstraction than machine code and thus easier to translate. Figure 2.3 shows how a compiled language like C achieves its platform independence. The language maintainers must have a compiler for every platform. With an interpreted language like Java, the same compiler works for all platforms. The language maintainers need only to provide a virtual machine for every platform, a simpler task than providing separate compilers.

Disadvantage of interpretation

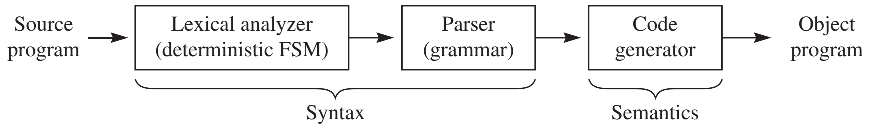
A disadvantage of interpretation is slow execution speed compared to compilation. During execution time, the application is not executing directly. Instead, the virtual machine is executing. This extra layer of abstraction provided by the virtual machine during run time makes execution of interpreted programs generally slower than execution of equivalent compiled programs.

The Compilation Process

The syntax of a programming language is usually specified by a formal grammar, which forms the basis of the parsing algorithm for the translator. Rather than specifying all the syntax, as the grammar in Figure 7.8 does, the formal grammar frequently specifies an upper level of abstraction and leaves the lower level to be specified by regular expressions or FSMs.

FIGURE 7.27 shows the steps in a typical compilation process. The low-level syntax analysis is called *lexical analysis*, and the high-level syntax analysis is called *parsing*. (This is a more specialized meaning of the word *parse*. It is sometimes used in a more general sense to include all syntax analysis.) In most translators for artificial languages, the lexical analyzer is based on a deterministic FSM whose input is a string of characters. The parser is usually based on a grammar whose input is the sequence of tokens taken from the lexical analyzer.

FIGURE 7.27
Steps in the compilation process.



Each stage in the compilation process takes its input from the previous stage and sends its output to the following stage. The input and output of each stage are as follows:

- The input of the lexical analyzer is a string of symbols from the terminal alphabet of the source program.
- The output of the lexical analyzer and input of the parser is a stream of tokens.
- The output of the parser and input of the code generator is the syntax tree of the parse and/or the source program written in an internal low-level language.
- The output of the code generator is the object program.

A nonterminal symbol for the lexical analyzer acts like a terminal symbol for the parser. A common example of such a symbol is an identifier. The FSM has individual letters and digits as its terminal alphabet, and it inputs a string of them as it makes its state transitions. If the string `abc3` is input, the FSM declares that an identifier has been detected and passes that information on to the parser. The parser uses <identifier> as a terminal symbol in its parse of the sentence from the language.

The state variable

An algorithm that implements an FSM has an enumerated variable called the *state variable* whose possible values correspond to the possible states of the FSM. The algorithm initializes the state variable to the start state of the machine and gets the string of terminal characters one at a time in a loop. Each character causes a change of state. There are two common implementation techniques:

The two FSM implementation techniques

- Table-lookup
- Direct-code

They differ in the way that the state variable gets its next value. The table-lookup technique stores the state transition table and looks up the next state based on the current state and input character. The direct-code technique tests the current state and input character in the code itself and assigns the next state to the state variable directly.

A Table-Lookup Parser

The program in FIGURE 7.28 implements the FSM of Figure 7.11 with the table-lookup technique. Variable FSM, a two-dimensional array of integers, is the state transition table shown in Figure 7.12. The program classifies each input character as a letter or digit. Because B is the final state, it declares that the input string is a valid identifier if the state on termination of the loop is B.

FIGURE 7.28
Implementation of the FSM of Figure 7.11 with the table-lookup technique.

```
public static boolean isAlpha(char ch) {
   return ('a' <= ch && ch <= 'z') || ('A' <= ch && ch <= 'Z');
}

// States
static final int S_A = 0;
static final int S_B = 1;
static final int S_C = 2;
// Alphabet
static final int T_LETTER = 0;
static final int T_DIGIT = 1;
// State transition table
static final int[][] FSM = {
   {S_B, S_C},
   {S_B, S_B},
   {S_C, S_C}
};

public void actionPerformed(ActionEvent event) {
   String line = textField.getText();
   char ch;
   int FSMChar;
   int state = S_A;
   for (int i = 0; i < line.length(); i++) {
      ch = line.charAt(i);
      FSMChar = isAlpha(ch) ? T_LETTER : T_DIGIT;
      state = FSM[state][FSMChar];
   }
   if (state == S_B) {
      System.out.printf("%s is a valid identifier.\n", line);
   } else {
      System.out.printf("%s is not a valid identifier.\n", line);
   }
}
```

Input/Output

```
Enter a string of letters and digits: cab3
cab3 is a valid identifier.
```

Input/Output

```
Enter a string of letters and digits: 3cab
3cab is not a valid identifier.
```

The input and output shown in the figure are a reflection of the GUI widget in the complete program, not shown here. The input comes from a dialog box with the label "Enter a string of letters and digits:" above a text input field. When the user clicks the Parse button, that event triggers execution of function `actionPerformed()`. The type `String` in the program is the Java immutable string type. Function `getText()` retrieves the user input from the text input field and gives it to variable `line`. The `for` loop processes the terminal characters one at a time with the `charAt()` function. It looks up the next state in the FSM table from the current state and the current input.

The program assumes that the user will enter only letters and digits. If the user enters some other character, it will detect the character as a digit. For example, if the user enters `cab#`, the program will detect it as a valid identifier even though it is not. A problem for the student at the end of this chapter suggests an improved FSM and corresponding implementation.

A Direct-Code Parser

The program in FIGURE 7.29 uses the direct-code technique to parse an integer. It is an implementation of the FSM of Figure 7.20(b). Function `actionPerformed()` allows the user to enter any string of characters. If the string is not a valid integer, `parseNum` will set attribute `valid` to false and the program will issue an error message. Otherwise, `valid` will be true and `number` will be the correct integer value entered.

FIGURE 7.29
Implementation of the FSM of Figure 7.20(b) with the direct code technique.

```
public void actionPerformed(ActionEvent event) {
   String line = textField.getText();
   Parser parser = new Parser();
   parser.parseNum(line);
   if (parser.getValid()) {
      System.out.printf("Number = %d\n", parser.getNumber());
   } else {
      System.out.printf("Invalid entry.\n");
   }
}

public enum State {
   S_I, S_F, S_M, S_STOP
}
```

(continues)

FIGURE 7.29
Implementation of the FSM of Figure 7.20(b) with the direct code technique. (*continued*)

```
public class Parser {

   private boolean valid = false;
   private int number = 0;
   public boolean getValid() {
      return valid;
   }

   public int getNumber() {
      return number;
   }

   public boolean isDigit(char ch) {
      return ('0' <= ch) && (ch <= '9');
   }

   public void parseNum(String line) {
      line = line + '\n';
      int lineIndex = 0;
      char nextChar;
      int sign = +1;
      valid = true;
      State state = State.S_I;
      do {
         nextChar = line.charAt(lineIndex++);
         switch (state) {
            case S_I:
               if (nextChar == '+') {
                  sign = +1;
                  state = State.S_F;
               } else if (nextChar == '-') {
                  sign = -1;
                  state = State.S_F;
               } else if (isDigit(nextChar)) {
                  sign = +1;
                  number = nextChar - '0';
                  state = State.S_M;
               } else {
                  valid = false;
               }
               break;
            case S_F:
               if (isDigit(nextChar)) {
                  number = nextChar - '0';
                  state = State.S_M;
               } else {
                  valid = false;
               }
               break;
            case S_M:
               if (isDigit(nextChar)) {
                  number = 10 * number + nextChar - '0';
               } else if (nextChar == '\n') {
                  number = sign * number;
                  state = State.S_STOP;
               } else {
                  valid = false;
               }
               break;
         }
      } while ((state != State.S_STOP) && valid);
   }
}
```

Input/Output

```
Enter a number: q
Invalid entry.
```

Input/Output

```
Enter a number: -58
Number = -58
```

Although the program is shown as one listing, it is actually fragments from three different files. The software in this chapter follows the Java coding convention of one file per class with the name of the file the same name as the class. For example, class `Parser` is in a separate file named `Parser.java`. The `State` class is also in a separate file.

Function `parseNum()` installs a newline character as a sentinel, regardless of how many or few characters the user enters. If the user enters no characters in the dialog box and simply presses the Parse button, `parseNum()` will install the newline character at `line[0]`.

The procedure has a local enumerated variable called `state`, whose possible values are `S_I`, `S_F`, or `S_M`, corresponding to the states I, F, and M of the FSM in Figure 7.20(b). An additional state called `S_STOP` is for terminating the loop. The function initializes `valid` to true and `state` to the start state, `S_I`.

A `do` loop simulates the transitions in the FSM, which is the direct-code technique. A single `switch` statement determines the current state, and a single nested `if` statement within each case processes the next character. Assignment statements in the code change the state variable directly.

In a simplified FSM, there are two ways to stop—either you run out of input or you reach a state with no transitions from it on the next character, in which case the string is not valid. Corresponding to these termination conditions, there are two ways to quit the `do` loop—when the input sentinel is reached in a final state or when the string is discovered to be invalid.

The body of a `do` loop always executes at least once. Nevertheless, the code executes correctly even if the Parse button is pressed with an empty text input field. `parseNum()` installs the newline character in `line[0]`. It initializes `state` to I, enters the `do` loop, and immediately sets `nextChar` to the newline character. Then `valid` gets false, and the loop terminates correctly.

In addition to determining whether the string is valid, `parseNum()` converts the string of characters to the proper integer value. If the first character is + or a digit, it sets `sign` to +1. If the first character is `-`, it sets `sign` to −1. The first digit detected sets `number` to its proper value in state I or F. Its value is maintained correctly in state M each time a succeeding digit is detected. The magnitude is multiplied by the `sign` when the loop terminates with a valid number.

Integrating semantic actions with syntactic actions

The computation of the correct integer value is a semantic action, and the state assignment is a syntax action. It is easy with the direct-code technique to integrate the semantic processing with the syntactic processing because there is a distinct place in the syntax code to include the required semantic processing. For example, you know in state I that if the character is `-`, `sign` must be set to −1. It is easy to determine where to include that assignment in the syntax code.

If the user enters leading spaces before a legal string of digits, the FSM will declare the string invalid. The next program shows how to correct this deficiency.

An Input Buffer Class

The following two programs use the same technique to get characters from the input stream. Instead of duplicating the code for the input processing in each program, this section shows an implementation of an input buffer class that both programs use. It is stored in a separate file named `InBuffer.java` and is included in each program. FIGURE 7.30 shows the implementation of the input buffer.

As shown in the following two programs, the FSM function sometimes detects a character from the input stream that terminates the current token, yet will be required from the input stream in a subsequent call to the function. Conceptually, the function must push the character back into the

FIGURE 7.30
The input buffer class included in the programs of Figures 7.32 and 7.35.

```
public class InBuffer {

   private String inString;
   private String line;
   private int lineIndex;

   public InBuffer(String string) {
      inString = string + "\n\n";
      // To guarantee inString.length() == 0 eventually
   }

   public void getLine() {
      int i = inString.indexOf('\n');
      line = inString.substring(0, i + 1);
      inString = inString.substring(i + 1);
      lineIndex = 0;
   }

   public boolean inputRemains() {
      return inString.length() != 0;
   }

   public char advanceInput() {
      return line.charAt(lineIndex++);
   }

   public void backUpInput() {
      lineIndex--;
   }
}
```

input stream so it will be retrieved on the subsequent call. `backUpInput()` provides that operation on the buffer class. Although the FSM function needs to access characters from the input buffer, it does not access the attributes of the buffer directly. Only the procedures `getLine()`, `inputRemains()`, `advanceInput()`, and `backUpInput()` access the buffer. The reason for this design is to provide the FSM function with a more convenient abstract structure of the input stream.

The remaining two programs in this chapter use multiline input. The constructor for the buffer appends two newline characters to input `string` and stores the result in `inString`. Lines are separated by the newline character `\n`. Function `getLine()` deletes the first line from `inString` using the newline character as the separater and puts it in `line`. Appending two newline characters at the beginning guarantees that the last line deleted from `inString` will have length 0.

A Multiple-Token Parser

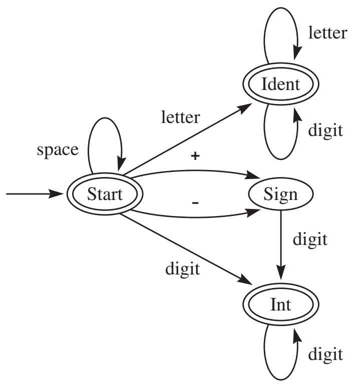
If the parser of a C compiler is analyzing the string

```
total =
```

it knows that the next nonterminal could be an identifier such as `amount` or an integer such as `100`. Because it does not know which token to expect, it calls an FSM that can recognize either, as in FIGURE 7.31.

The state labeled *Ident* is a final state for detecting the identifier token. Int is the final state for detecting an integer. The transition from Start to Start is on the space character. It allows for leading spaces before either token. If

FIGURE 7.31
The FSM of a program that recognizes identifiers and integers.



the only characters left to scan are trailing spaces at the end of a line, the FSM procedure will return the empty token. That is why the start state is also a final state.

FIGURE 7.32 shows two input/output runs from a program that implements the multiple-token recognizer of Figure 7.31. The first run has an input of two lines, the first line with five nonempty tokens and the second line with six nonempty tokens. Here is an explanation of the first run of Figure 7.32.

The machine starts in the start state and scans the first terminal, `H`. That takes it to the Ident state. The following terminals, `e`, `r`, and `e`, make transitions to the same state. The next terminal is a space. There is no transition from state Ident on the terminal space. Because the machine is in the final state for identifiers, it concludes that an identifier has been scanned. It puts the space terminal, which it could not use in this state, back into the input for use as the first terminal for the next token. It then declares that an identifier has been scanned.

The machine starts over in the start state. It uses the leftover space to make a transition to Start. A few more spaces produce a few more transitions

FIGURE 7.32
The input/output of a program that recognizes identifiers and integers.

Input
```
Here is A47 48B
   C-49 ALongIdentifier +50 D16-51
```
Output
```
Identifier = Here
Identifier = is
Identifier = A47
Integer    = 48
Identifier = B
Empty token
Identifier = C
Integer    = -49
Identifier = ALongIdentifier
Integer    = 50
Identifier = D16
Integer    = -51
Empty token
```
(a) First run.

Input
```
Here is A47+ 48B
   C+49

ALongIdentifier
```
Output
```
Identifier = Here
Identifier = is
Identifier = A47
Syntax error
Identifier = C
Integer    = 49
Empty token
Empty token
Identifier = ALongIdentifier
Empty token
```
(b) Second run.

to Start, after which the `i` and `s` characters produce the recognition of a second identifier, as shown in the sample output. Similarly, `A47` is recognized as an identifier.

For the next token, the initial `4` sends the machine into the Integer state. The `8` makes the transition to the same state. Now the machine inputs the `B`. There is no transition from state Integer on the terminal `B`. Because the machine is in the final state for integers, it concludes that an integer has been scanned. It puts the `B` terminal, which it could not use in this state, back into the input for use as the first terminal for the next token. It then declares that an integer has been scanned. Notice that `B` is detected as an identifier the next time around.

The machine continues recognizing tokens until it gets to the end of the line, at which point it recognizes the empty token. It will recognize the empty token whether or not there are trailing spaces in the input because the buffer appends two newline characters to the input string.

The second sample input shows how the machine handles a string of characters that contains a syntax error. After recognizing `Here`, `is`, and `A47`, on the next call, the FSM gets the + and goes to state Sign. Because the next character is space, and there is no transition from Sign on space, the FSM returns the invalid token.

Like all multiple-token recognizers, this machine operates on the following design principle:

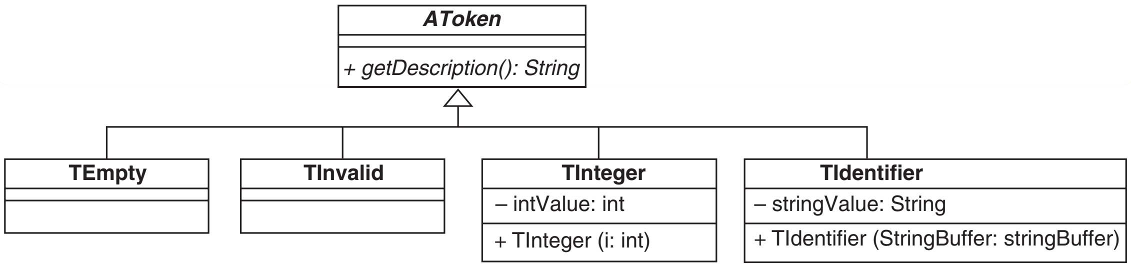
A design principle for multiple-token recognizers

› You can never fail once you reach a final state. Instead, if the final state does not have a transition from it on the terminal just input, you have recognized a token and should back up the input. The character will then be available as the first terminal for the next token.

The machine handles an empty line (or a line with only spaces) correctly, returning the empty token on the first call.

FIGURE 7.33 is a Unified Modeling Language (UML) diagram of the class structure of a token. `AToken` is an abstract token with no attributes and one public abstract operation, `getDescription()`. The plus sign in front of the

FIGURE 7.33
The UML diagram of the class structure of `AToken`.



operations is the UML notation for public access. The open triangle is the UML symbol for inheritance; Figure 7.33 shows that the concrete classes `TEmpty`, `TInvalid`, `TInteger`, and `TIdentifier` inherit from `AToken`. The UML convention is to show abstract class names and methods in a slanted font.

Each of the concrete classes must implement the abstract methods they inherit from their superclass. Method `getDescription()` returns a string for the output shown in Figure 7.32. In addition to the inherited methods, class `TInteger` has a private attribute `intValue`, which stores the integer value detected by the parser, and a public constructor. The minus sign in front of the attribute is the UML symbol for private access. Class `TIdentifier` has a similar attribute of type `String` and its own constructor. Its constructor has a formal parameter of type `StringBuffer`, which is the Java mutable string type.

FIGURE 7.34 shows the corresponding Java implementation of the token class structure of Figure 7.33. It is a collection of code fragments from five separate files; to save space, it does not show the `@Override` annotation.

FIGURE 7.34
A Java implementation of class `AToken` in Figure 7.33.

```
abstract public class AToken {
   public abstract String getDescription();
}

public class TEmpty extends AToken {
   public String getDescription() {
      return "Empty token";
   }
}

public class TInvalid extends AToken {
   public String getDescription() {
      return "Syntax error";
   }
}

public class TInteger extends AToken {
   private final int intValue;
   public TInteger(int i) {
      intValue = i;
   }
```

(continues)

FIGURE 7.34
A Java implementation of class `AToken` in Figure 7.33. (*continued*)

```
   public String getDescription() {
      return String.format("Integer    = %d", intValue);
   }
}

public class TIdentifier extends AToken {
   private final String stringValue;
   public TIdentifier(StringBuffer stringBuffer) {
      stringValue = new String(stringBuffer);
   }
   public String getDescription() {
      return String.format("Identifier = %s", stringValue);
   }
}
```

FIGURE 7.35 is the direct-code implementation of the FSM of Figure 7.31. It is a collection of code fragments from three Java class files. The constructor for class `Tokenizer` sets `b` to the buffer that has been loaded with the input string. Method `getToken()` returns an abstract token whose dynamic type will be one of the concrete classes `TEmpty`, `TInvalid`, `TInteger`, or `TIdentifier`.

The Java `StringBuffer` class is used for efficiency. Method `getToken()` maintains a local string value, which is mutable, for processing

FIGURE 7.35
A Java implementation of the FSM of Figure 7.31.

```
public class Util {
   public static boolean isDigit(char ch) {
      return ('0' <= ch) && (ch <= '9');
   }
   public static boolean isAlpha(char ch) {
      return (('a' <= ch) && (ch <= 'z') || ('A' <= ch) && (ch <= 'Z'));
   }
}
```

```
public enum LexState {
   LS_START, LS_IDENT, LS_SIGN, LS_INTEGER, LS_STOP
}

public class Tokenizer {
   private final InBuffer b;
   public Tokenizer(InBuffer inBuffer) {
      b = inBuffer;
   }
   public AToken getToken() {
      char nextChar;
      StringBuffer localStringValue = new StringBuffer("");
      int localIntValue = 0;
      int sign = +1;
      AToken aToken = new TEmpty();
      LexState state = LexState.LS_START;
      do {
         nextChar = b.advanceInput();
         switch (state) {
            case LS_START:
               if (Util.isAlpha(nextChar)) {
                  localStringValue.append(nextChar);
                  state = LexState.LS_IDENT;
               } else if (nextChar == '-') {
                  sign = -1;
                  state = LexState.LS_SIGN;
               } else if (nextChar == '+') {
                  sign = +1;
                  state = LexState.LS_SIGN;
               } else if (Util.isDigit(nextChar)) {
                  localIntValue = nextChar - '0';
                  state = LexState.LS_INTEGER;
               } else if (nextChar == '\n') {
                  state = LexState.LS_STOP;
               } else if (nextChar != ' ') {
                  aToken = new TInvalid();
               }
               break;
```

(continues)

FIGURE 7.35
A Java implementation of the FSM of Figure 7.31. (*continued*)

```
            case LS_IDENT:
               if (Util.isAlpha(nextChar) || Util.isDigit(nextChar)) {
                  localStringValue.append(nextChar);
               } else {
                  b.backUpInput();
                  aToken = new TIdentifier(localStringValue);
                  state = LexState.LS_STOP;
               }
               break;
            case LS_SIGN:
               if (Util.isDigit(nextChar)) {
                  localIntValue = nextChar - '0';
                  state = LexState.LS_INTEGER;
               } else {
                  aToken = new TInvalid();
               }
               break;
            case LS_INTEGER:
               if (Util.isDigit(nextChar)) {
                  localIntValue = 10 * localIntValue + nextChar - '0';
               } else {
                  b.backUpInput();
                  aToken = new TInteger(sign * localIntValue);
                  state = LexState.LS_STOP;
               }
               break;
         }
      } while ((state != LexState.LS_STOP) && !(aToken instanceof TInvalid));
      return aToken;
   }
}
```

identifiers. The `append()` method mutates the string by appending `nextChar` to it. This is more efficient that appending `nextChar` to a copy of the local string value.

FIGURE 7.36 shows function `actionPerformed()`. It has a single abstract token `aToken`. The outer `while` loop executes once for each line of input, and the inner `do` loop executes once for each token in the line.

FIGURE 7.36
The `actionPerformed()` method for the tokenizer of Figure 7.35.

```
public void actionPerformed(ActionEvent event) {
   InBuffer inBuffer = new InBuffer(textArea.getText());
   Tokenizer t = new Tokenizer(inBuffer);
   AToken aToken;
   inBuffer.getLine();
   while (inBuffer.inputRemains()) {
      do {
         aToken = t.getToken();
         System.out.println(aToken.getDescription());
      } while (!(aToken instanceof TEmpty) && !(aToken instanceof TInvalid));
      inBuffer.getLine();
   }
}
```

The output relies on polymorphic dispatch to display the tokens that are detected. That is, the main program does not explicitly test the dynamic type of the token to choose how to output its value. It simply uses its abstract token to invoke the `getDescription()` method.

7.4 Code Generation

To *translate* is to transform a string of characters from some input alphabet to another string of characters from some output alphabet. The typical phases in such a translation are lexical analysis, parsing, and code generation. This section consists of a program that translates from one language to another. It illustrates all three phases of a simple automatic translator.

A Language Translator

FIGURE 7.37 shows the input/output of the translator. The input is the source and the output is the object code and a formatted program listing. The source and object languages are line oriented, as are assembly languages.

The source language has the syntax of C function calls, and the object language has the syntax of assignment statements with the assignment operator `<-`. A sample statement from the input language is

```
set (Time, 15)
```

FIGURE 7.37
The input/output of a program that translates from one language to another.

Input
```
set (Time, 15)
set (   Accel, 3)
set (TSquared   , Time)
    MUL ( TSquared, Time)
set ( Position, TSquared)
mul (Position, Accel)
dIV(Position,2)
stop
end
```

Output
```
Object code:
Time <- 15
Accel <- 3
TSquared <- Time
TSquared <- TSquared * Time
Position <- TSquared
Position <- Position * Accel
Position <- Position / 2
stop

Program listing:
set (Time, 15)
set (Accel, 3)
set (TSquared, Time)
mul (TSquared, Time)
set (Position, TSquared)
mul (Position, Accel)
div (Position, 2)
stop
end
```

(a) First run.

Input
```
set (Alpha,, 123)
set (Alpha)
sit (Alpha, 123)
set, (Alpha)
mul (Alpha, Beta
set (123, Alpha)
neg (Alpha, Beta)
set (Alpha, 123) x
```

Output
```
9 errors were detected.

Program listing:
ERROR: Second argument not an identifier or integer.
ERROR: Comma expected after first argument.
ERROR: Line must begin with function identifier.
ERROR: Left parenthesis expected after function.
ERROR: Right parenthesis expected after argument.
ERROR: First argument not an identifier.
ERROR: Right parenthesis expected after argument.
ERROR: Illegal trailing character.

ERROR: Missing "end" sentinel.
```

(b) Second run.

The corresponding object statement is

```
Time <- 15
```

The word `set` is reserved in the source language. The other reserved words are `add`, `sub`, `mul`, `div`, `neg`, `abs`, and `end`. `Time` is a user-defined identifier. Identifiers follow the same rules as in the C language. Integers, such as `15` in the previous example, also follow the C syntax.

The `set` procedure takes two arguments, separated by a comma and surrounded by parentheses. The first argument must be an identifier, but the second can be an identifier or an integer constant.

Another example of a translation is

```
mul (TSquared, Time)
```

which is written in the object language as

```
TSquared <- TSquared * Time
```

As with the `set` procedure, the first argument of a `mul` procedure call must be an identifier. To translate the `mul` statement, the translator must duplicate its first argument, which appears on both sides of the assignment operator.

The other procedure calls are similar, except for `neg` and `abs`, which take a single argument. For `neg`, the translator prefixes the argument with a dash character on the right side of the assignment operator. For `abs`, the translator encloses the argument in vertical bars. For example, the source statements

```
neg (Alpha)
abs (Beta)
```

are translated to

```
Alpha <- -Alpha
Beta <- |Beta|
```

The reserved word `end` is the sentinel for the translator. It generates no code and corresponds to `.END` in Pep/9 assembly language. Any number of spaces can occur anywhere in a source line, except within an identifier or integer.

The translator must not crash if syntax errors occur in the input stream. In Figure 7.37, there is also a run that shows a source file full of errors. The program generates appropriate error messages in the source listing to help the user find the bugs in the source program. If the translator detects any errors, it suppresses the object code output.

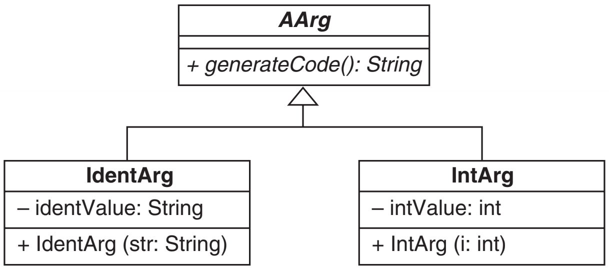
This program is based on a two-stage analysis of the syntax, as shown in Figure 7.27. Instead of using a grammar to specify the parsing problem as indicated in the figure, however, the structure of this source language is simple enough for the parser to be based on an FSM.

The complete Java project for the translator has 26 classes and 26 associated `.java` files. FIGURE 7.38 is the start of a listing of code fragments from the program that produces the output of Figure 7.37. The

FIGURE 7.38
The lookup maps for the translator program.

```
public enum Mnemon {
   M_ADD, M_SUB, M_MUL, M_DIV, M_NEG, M_ABS, M_SET, M_STOP, M_END
}

public final class Maps {

   public static final Map<String, Mnemon> unaryMnemonTable;
   public static final Map<String, Mnemon> nonUnaryMnemonTable;
   public static final Map<Mnemon, String> mnemonStringTable;

   static {
      unaryMnemonTable = new HashMap<>();
      unaryMnemonTable.put("stop", Mnemon.M_STOP);
      unaryMnemonTable.put("end", Mnemon.M_END);

      nonUnaryMnemonTable = new HashMap<>();
      nonUnaryMnemonTable.put("neg", Mnemon.M_NEG);
      nonUnaryMnemonTable.put("abs", Mnemon.M_ABS);
      nonUnaryMnemonTable.put("add", Mnemon.M_ADD);
      nonUnaryMnemonTable.put("sub", Mnemon.M_SUB);
      nonUnaryMnemonTable.put("mul", Mnemon.M_MUL);
      nonUnaryMnemonTable.put("div", Mnemon.M_DIV);
      nonUnaryMnemonTable.put("set", Mnemon.M_SET);

      mnemonStringTable = new EnumMap<>(Mnemon.class);
      mnemonStringTable.put(Mnemon.M_NEG, "neg");
      mnemonStringTable.put(Mnemon.M_ABS, "abs");
      mnemonStringTable.put(Mnemon.M_ADD, "add");
      mnemonStringTable.put(Mnemon.M_SUB, "sub");
      mnemonStringTable.put(Mnemon.M_MUL, "mul");
      mnemonStringTable.put(Mnemon.M_DIV, "div");
      mnemonStringTable.put(Mnemon.M_SET, "set");
      mnemonStringTable.put(Mnemon.M_STOP, "stop");
      mnemonStringTable.put(Mnemon.M_END, "end");
   }
}
```

FIGURE 7.39
The UML diagram of the class structure of `AArg`.



program listing continues in the following figures. Figure 7.38 shows the setup of three Java maps used by the translator. The first two maps, one for unary instructions and one for nonunary instructions, take the string representation of a reserved word as the key and return the enumerated mnemonic representation. The third table uses enumerated mnemonic values as the key to look up the string symbol to place in the generated code. The maps use the lowercase string representation of the source code reserved word.

FIGURE 7.39 is the UML diagram of an abstract argument, and FIGURE 7.40 is its Java implementation. Because an argument in the source code can be either an identifier or an integer, the program stores a general argument as an `AArg`, which at run time is either an `IdentArg` or an `IntArg`. Class `AArg` defines the abstract method `generateCode()`, which contributes to code generation when the value of an argument must be output.

FIGURE 7.41 is the UML diagram of an abstract token, and FIGURE 7.42 is a partial listing of its Java implementation. The implementations of `TLeftParen`, `TRightParen`, `TEmpty`, and `TInvalid` are identical to the implementation of `TComma` and are not shown in the figure. This structure of a token is similar to the one in Figure 7.33 of the previous section. Classes `TIdentifier` and `TInteger` have getter methods to retrieve the values of their attributes.

The lexical analyzer returns an identifier when it encounters a reserved word and when it encounters an argument. When it encounters a reserved word, the parser needs to look up the word in the mnemonic map. It uses `getStringValue()` to get the identifier value from the token.

FIGURE 7.43 is the UML diagram of the abstract code class `ACode`, and FIGURE 7.44 is a complete listing of its Java implementation. An object of class

FIGURE 7.40
The Java implementation of class `AArg` in Figure 7.39.

```
abstract public class AArg {
   abstract public String generateCode();
}

public class IdentArg extends AArg {
   private final String identValue;
   public IdentArg(String str) {
      identValue = str;
   }
   public String generateCode() {
      return identValue;
   }
}

public class IntArg extends AArg {
   private final int intValue;
   public IntArg(int i) {
      intValue = i;
   }
   public String generateCode() {
      return String.format("%d", intValue);
   }
}
```

FIGURE 7.41
The UML diagram of the class structure of `AToken`.

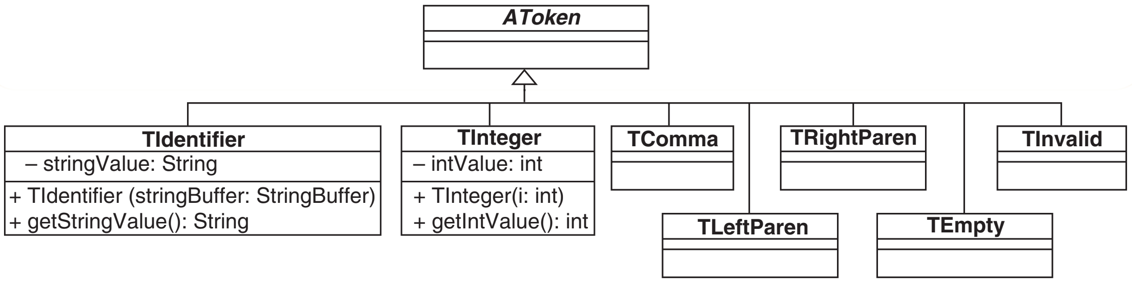


FIGURE 7.42
The Java implementation of class `AToken` in Figure 7.41.

```
abstract public class AToken {
}

public class TIdentifier extends AToken {
   private final String stringValue;
   public TIdentifier(StringBuffer stringBuffer) {
      stringValue = new String(stringBuffer);
   }
   public String getStringValue() {
      return stringValue;
   }
}

public class TInteger extends AToken {
   private final int intValue;
   public TInteger(int i) {
      intValue = i;
   }
   public int getIntValue() {
      return intValue;
   }
}

public class TComma extends AToken {
}
```

`ACode` represents one line of source code and its corresponding object code. Execution of method `generateCode()` returns a string representation of the object code for that line, and execution of `genereateListing()` returns a string representation of the formatted source code for that line. Consequently, a code object must contain all the data it needs to output the source code and object code for that line.

For example, Figure 7.43 shows that an object of class `TwoArgInstr` has two attributes, `firstArg`, which is an abstract argument, and `secondArg`, also an abstract argument. In addition, it has enumerated `mnemonic`. Consider the last line of input from Figure 7.37(a):

```
dIV(Position,2)
```

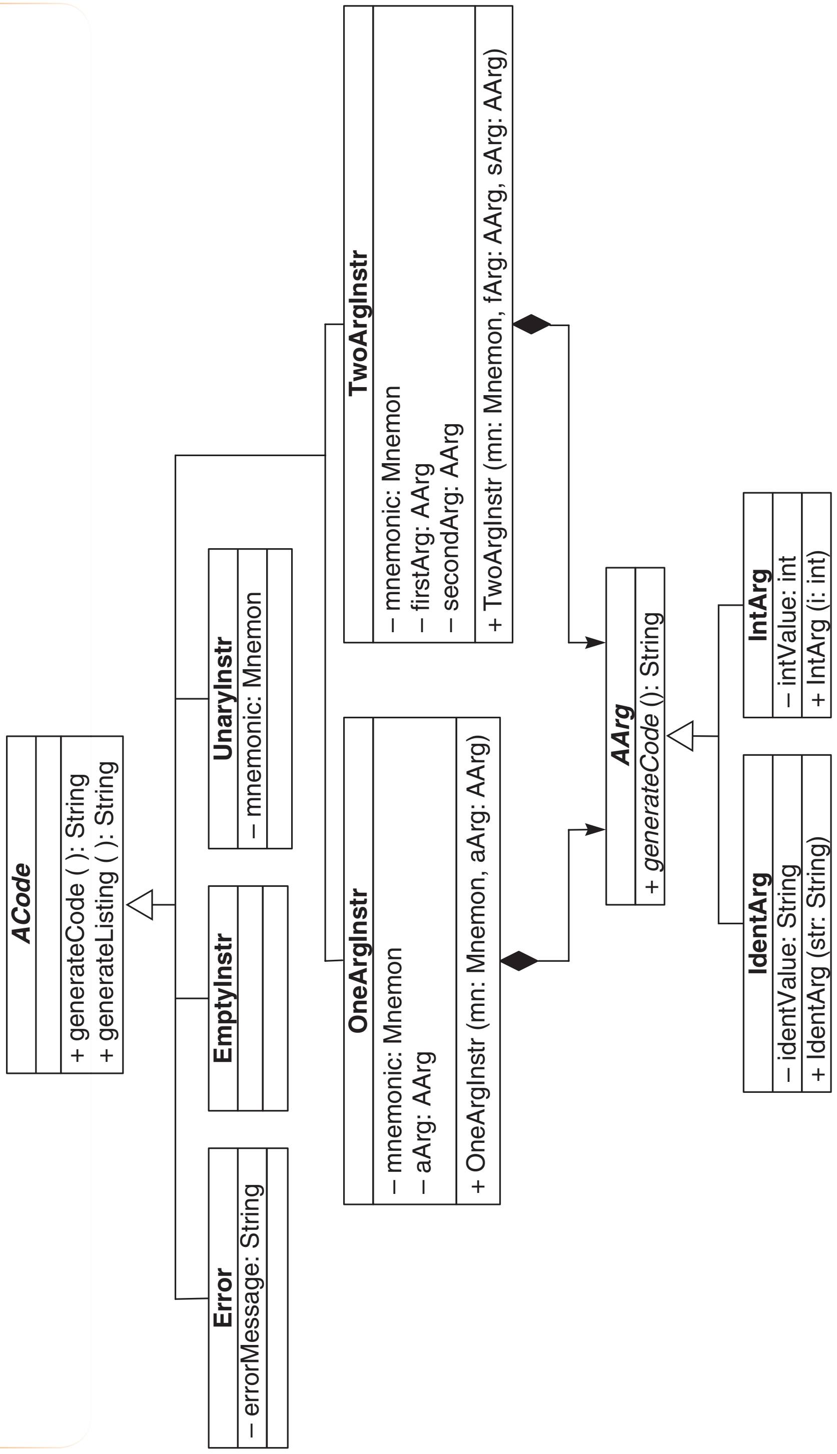


FIGURE 7.43
The UML diagram of the class structure of ACode.

FIGURE 7.44
The Java implementation of class `ACode` in Figure 7.43.

```
abstract public class ACode {
   abstract public String generateCode();
   abstract public String generateListing();
}

public class Error extends ACode {
   private final String errorMessage;
   public Error(String errMessage) {
      errorMessage = errMessage;
   }
   public String generateListing() {
      return "ERROR: " + errorMessage + "\n";
   }
   public String generateCode() {
      return "";
   }
}

public class EmptyInstr extends ACode {
   // For an empty source line.
   public String generateListing() {
      return "\n";
   }
   public String generateCode() {
      return "";
   }
}

public class UnaryInstr extends ACode {
   private final Mnemon mnemonic;
   public UnaryInstr(Mnemon mn) {
      mnemonic = mn;
   }
   public String generateListing() {
      return Maps.mnemonStringTable.get(mnemonic) + "\n";
   }
```

(*continues*)

FIGURE 7.44
The Java implementation of class `ACode` in Figure 7.43. (*continued*)

```
   public String generateCode() {
      switch (mnemonic) {
         case M_STOP:
            return "stop\n";
         case M_END:
            return "";
         default:
            return ""; // Should not occur.
      }
   }
}

public class OneArgInstr extends ACode {
   private final Mnemon mnemonic;
   private final AArg aArg;
   public OneArgInstr(Mnemon mn, AArg aArg) {
      mnemonic = mn;
      this.aArg = aArg;
   }
   public String generateListing() {
      return String.format("%s (%s)\n",
              Maps.mnemonStringTable.get(mnemonic),
              aArg.generateCode());
   }
   public String generateCode() {
      switch (mnemonic) {
         case M_ABS:
            return String.format("%s <- |%s|\n",
                    aArg.generateCode(),
                    aArg.generateCode());
         case M_NEG:
            return String.format("%s <- -%s\n",
                    aArg.generateCode(),
                    aArg.generateCode());
         default:
            return ""; // Should not occur.
      }
   }
}
```

```
public class TwoArgInstr extends ACode {
   private final Mnemon mnemonic;
   private final AArg firstArg;
   private final AArg secondArg;
   public TwoArgInstr(Mnemon mn, AArg fArg, AArg sArg) {
      mnemonic = mn;
      firstArg = fArg;
      secondArg = sArg;
   }
   public String generateListing() {
      return String.format("%s (%s, %s)\n",
              Maps.mnemonStringTable.get(mnemonic),
              firstArg.generateCode(),
              secondArg.generateCode());
   }
   public String generateCode() {
      switch (mnemonic) {
         case M_SET:
            return String.format("%s <- %s\n",
                    firstArg.generateCode(),
                    secondArg.generateCode());
         case M_ADD:
            return String.format("%s <- %s + %s\n",
                    firstArg.generateCode(),
                    firstArg.generateCode(),
                    secondArg.generateCode());
         case M_SUB:
            return String.format("%s <- %s - %s\n",
                    firstArg.generateCode(),
                    firstArg.generateCode(),
                    secondArg.generateCode());
         case M_MUL:
            return String.format("%s <- %s * %s\n",
                    firstArg.generateCode(),
                    firstArg.generateCode(),
                    secondArg.generateCode());
```

(continues)

FIGURE 7.44
The Java implementation of class `ACode` in Figure 7.43. (*continued*)

```
            case M_DIV:
               return String.format("%s <- %s / %s\n",
                       firstArg.generateCode(),
                       firstArg.generateCode(),
                       secondArg.generateCode());
            default:
               return ""; // Should not occur.
         }
      }
   }
```

The code object would have `M_DIV` for `mnemonic`, `firstArg` would be an `IdentArg` with an `identValue` with string value "Position," and `secondArg` would be an `IntArg` with an `intValue` of 2.

The concrete code classes contain the methods for generating both the object code listing and the formatted source code listing. For the preceding source line, the translator generates an object of class `TwoArgInstr` during the parse phase. It sets the attributes `mnemonic`, `firstArg`, and `secondArg` as described above. The following code in `generateListing()` returns the string for the formatted listing:

```
public String generateListing() {
   return String.format("%s (%s, %s)\n",
           Maps.mnemonStringTable.get(mnemonic),
           firstArg.generateCode(),
           secondArg.generateCode());
}
```

It uses `mnemonic` as the key to the map for looking up the string representation of the reserved word `div`. Then, it invokes `generateCode()` for the first and second arguments and formats them within parentheses. The result is the string

```
div (Position, 2)
```

formatted in the standard style.

The following code in `generateCode()` returns the string for the object code:

```
case M_MUL:
   return String.format("%s <- %s * %s\n",
          firstArg.generateCode(),
          firstArg.generateCode(),
          secondArg.generateCode());
```

The result is the string

```
Position <- Position / 2
```

The first argument occurs twice in the object code, once on the left side and once on the right side of the assignment operator.

The UML symbol for class composition is the solid diamond touching the `OneArgInstr` class box and the `TwoArgInstr` class box in Figure 7.43. The meaning of class composition is "has a" as opposed to the meaning of inheritance, which is "is a." A `OneArgInstr` object "is a" `ACode` object, and a `OneArgInstr` object "has a" `AArg` object.

FIGURE 7.45 is a partial listing of the tokenizer class. Function `getToken()` works like the `getToken()` function in Figure 7.35 except

FIGURE 7.45
The lexical analyzer.

```
public enum LexState {
   LS_START, LS_IDENT, LS_SIGN, LS_INTEGER, LS_STOP
}

public class Tokenizer {
   private final InBuffer b;
   public Tokenizer(InBuffer inBuffer) {
      b = inBuffer;
   }
```

```
public class Tokenizer {
   private final InBuffer b;
   public Tokenizer(InBuffer inBuffer) {
      b = inBuffer;
   }
```

(continues)

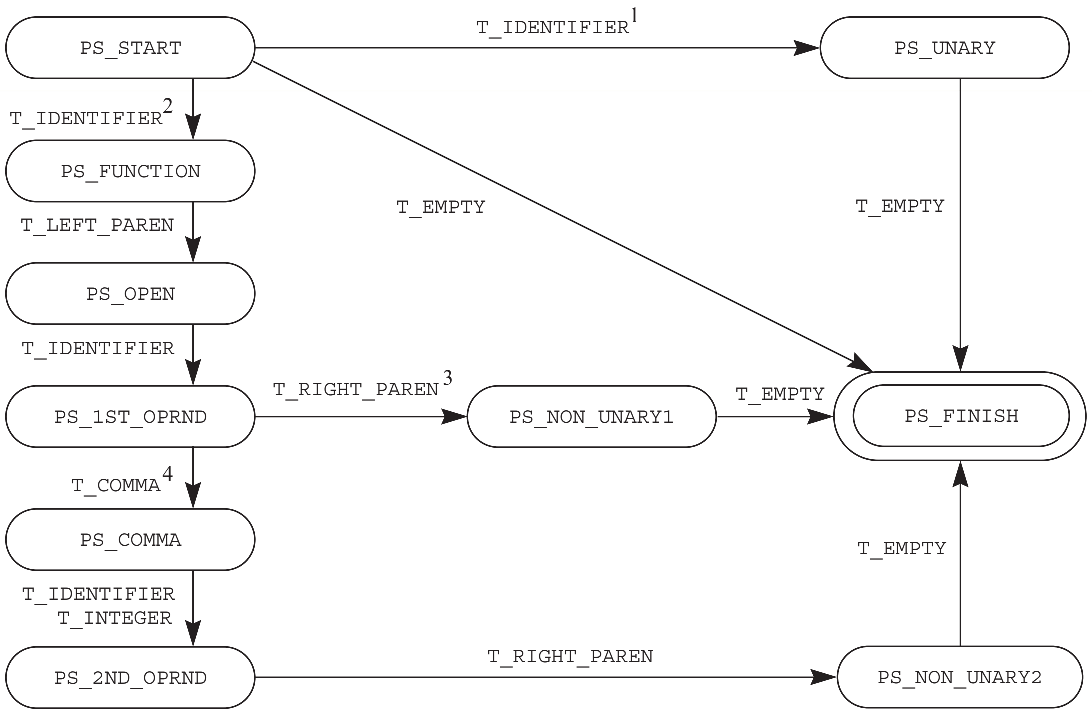
FIGURE 7.45
The lexical analyzer. (*continued*)

```
   public AToken getToken() {
      char nextChar;
      StringBuffer localStringValue = new StringBuffer("");
      int localIntValue = 0;
      int sign = +1;
      AToken aToken = new TEmpty();
      LexState state = LexState.LS_START;
      do {
         nextChar = b.advanceInput();
         switch (state) {
            case LS_START:
               if (Util.isAlpha(nextChar)) {
                  localStringValue.append(nextChar);
                  state = LexState.LS_IDENT;
               } else if (nextChar == '-') {
...
            case LS_INTEGER:
               if (Util.isDigit(nextChar)) {
                  localIntValue = 10 * localIntValue + nextChar - '0';
               } else {
                  b.backUpInput();
                  aToken = new TInteger(localIntValue);
                  state = LexState.LS_STOP;
               }
               break;
         }
      } while ((state != LexState.LS_STOP) && !(aToken instanceof TInvalid));
      return aToken;
   }
}
```

that it detects one of the seven tokens in Figure 7.41. As before, `aToken` is an abstract token returned by the function. Its dynamic type can be any of the seven concrete subclasses of `aToken`.

FIGURE 7.46 shows a deterministic FSM that describes the source language. The transitions of the machine are on the tokens from the lexical analyzer, indicated in the figure by the words that begin with `T`, as in

FIGURE 7.46
The FSM for the parser `processSourceLine` of Figure 7.47.



Note 1: Only the identifiers `stop` and `end`.
Note 2: Only the identifiers `set, add, sub, mul, div, neg,` and `abs`.
Note 3: Only for mnemonics `M_NEG` and `M_ABS`.
Note 4: Only for mnemonics `M_SET, M_ADD, M_SUB,`and `M_MUL, M_DIV`.

Figure 7.41. The final state `PS_FINISH` can be reached only by input of token `T_EMPTY`. The transition from `PS_START` to `PS_FINISH` will occur if there is a blank line or if there is a line that contains only spaces. The terminal strings `end` and `stop` are the only identifiers that make the transition from `PS_START` to `PS_UNARY`. The identifiers that correspond to the other reserved words—`set`, `add`, `sub`, `mul`, `div`, `neg`, and `abs`—make the transition from `PS_START` to `PS_FUNCTION`. All other identifiers are invalid when detected in the `PS_START` state.

FIGURE 7.47 is a partial listing of the translator that implements the FSM of Figure 7.46. Class `Translator` has two methods, private method `parseLine()` and public method `translate()`, which calls `parseLine()` in a loop that executes once per source line.

FIGURE 7.47
A partial listing of the translator that implements the FSM of Figure 7.46.

```
public enum ParseState {
   PS_START, PS_UNARY, PS_FUNCTION, PS_OPEN, PS_1ST_OPRND, PS_NONUNARY1,
   PS_COMMA, PS_2ND_OPRND, PS_NON_UNARY2, PS_FINISH
}

public class Translator {
   private final InBuffer b;
   private Tokenizer t;
   private ACode aCode;
   public Translator(InBuffer inBuffer) {
      b = inBuffer;
   }
   // Sets aCode and returns boolean true if end statement is processed.
   private boolean parseLine() {
      boolean terminate = false;
      AArg localFirstArg = new IntArg(0);
      AArg localSecondArg;
      Mnemon localMnemon = Mnemon.M_END; // Useless initialization
      AToken aToken;
      aCode = new EmptyInstr();
      ParseState state = ParseState.PS_START;
      do {
         aToken = t.getToken();
         switch (state) {
            case PS_START:
               if (aToken instanceof TIdentifier) {
                  TIdentifier localTIdentifier = (TIdentifier) aToken;
                  String tempStr = localTIdentifier.getStringValue();
                  if (Maps.unaryMnemonTable.containsKey(
                        tempStr.toLowerCase())) {
                     localMnemon = Maps.unaryMnemonTable.get(
                           tempStr.toLowerCase());
                     aCode = new UnaryInstr(localMnemon);
                     terminate = localMnemon == Mnemon.M_END;
                     state = ParseState.PS_UNARY;
                  } else if (Maps.nonUnaryMnemonTable.containsKey(
                          tempStr.toLowerCase())) {
                     localMnemon = Maps.nonUnaryMnemonTable.get(
                             tempStr.toLowerCase());
                     state = ParseState.PS_FUNCTION;
                  } else {
                     aCode = new Error(
                             "Line must begin with function identifier.");
                  }
               } else if (aToken instanceof TEmpty) {
                  aCode = new EmptyInstr();
                  state = ParseState.PS_FINISH;
               } else {
                  aCode = new Error(
                          "Line must begin with function identifier.");
               }
               break;
...
            case PS_COMMA:
               if (aToken instanceof TIdentifier) {
                  TIdentifier localTIdentifier = (TIdentifier) aToken;
                  localSecondArg = new IdentArg(
                          localTIdentifier.getStringValue());
                  aCode = new TwoArgInstr(
                          localMnemon, localFirstArg, localSecondArg);
                  state = ParseState.PS_2ND_OPRND;
               } else if (aToken instanceof TInteger) {
                  TInteger localTInteger = (TInteger) aToken;
                  localSecondArg = new IntArg(
                          localTInteger.getIntValue());
                  aCode = new TwoArgInstr(
                          localMnemon, localFirstArg, localSecondArg);
                  state = ParseState.PS_2ND_OPRND;
               } else {
                  aCode = new Error(
                         "Second argument not an identifier or integer.");
               }
               break;
```

(continues)

FIGURE 7.47
A partial listing of the translator that implements the FSM of Figure 7.46. (*continued*)

```
...
            case PS_NON_UNARY2:
               if (aToken instanceof TEmpty) {
                  state = ParseState.PS_FINISH;
               } else {
                  aCode = new Error("Illegal trailing character.");
               }
               break;
         }
      } while (state != ParseState.PS_FINISH && !(aCode instanceof Error));
      return terminate;
   }

   public void translate() {
      ArrayList<ACode> codeTable = new ArrayList<>();
      int numErrors = 0;
      t = new Tokenizer(b);
      boolean terminateWithEnd = false;
      b.getLine();
      while (b.inputRemains() && !terminateWithEnd) {
         terminateWithEnd = parseLine(); // Sets aCode and returns boolean.
         codeTable.add(aCode);
         if (aCode instanceof Error) {
            numErrors++;
         }
         b.getLine();
      }
      if (!terminateWithEnd) {
         aCode = new Error("Missing \"end\" sentinel.");
         codeTable.add(aCode);
         numErrors++;
      }
      if (numErrors == 0) {
         System.out.printf("Object code:\n");
         for (int i = 0; i < codeTable.size(); i++) {
            System.out.printf("%s", codeTable.get(i).generateCode());
         }
      }
      if (numErrors == 1) {
         System.out.printf("One error was detected.\n");
      } else if (numErrors > 1) {
         System.out.printf("%d errors were detected.\n", numErrors);
      }
      System.out.printf("\nProgram listing:\n");
      for (int i = 0; i < codeTable.size(); i++) {
         System.out.printf("%s", codeTable.get(i).generateListing());
      }
   }
}
```

The first line in `translate()` instantiates a code table as a list of abstract code objects. It maintains a count of the number of errors detected and instantiates a tokenizer, passing the input buffer for its constructor. It also maintains a Boolean flag initialized to false that is set to true when the `end` token is detected. It calls the `getLine()` method of the buffer to get the first line of the source. To reestablish the loop invariant, it calls `getLine()` as the last statement in the body of the loop. The loop continues executing as long as input remains in the buffer and the Boolean flag remains false.

The first statement in the `while` loop calls `parseLine()`, which returns true when it detects the end token. As a side effect, it sets the `aCode` attribute of the translator class to the concrete code object that it constructs in the parse. `translate()` stores this concrete code object in its code table. It also increments the error count if the code is an `Error` object. The remaining code in `translate()` outputs the object code and formatted source listing by looping through the code table and invoking `generateCode()` and `generateListing()` for each code object.

The structure of `parseLine()` in Figure 7.47 is identical to the structure of `getToken()` in Figure 7.45 because both functions implement an FSM. Both functions have a state variable named `state` and have a `do` loop that terminates when a sentinel is detected or when an error occurs. The first statement in the `getToken()` loop is

```
nextChar = b.advanceInput();
```

which gets the next terminal character from the buffer. This loop for the lexical analyzer scans enough terminal characters to comprise a single token. The first statement in the `parseLine()` loop is

```
aToken = t.getToken();
```

which gets the next token. This loop for the parser scans enough tokens to comprise a single source line. It is doing the same processing as the lexical analyzer loop but at a higer level of abstraction. A nonterminal symbol for the lexical analyzer acts like a terminal symbol for the parser.

Figure 7.47 shows code fragments of the FSM for the parser in the case of `PS_START`, `PS_COMMA`, and `PS_NON_UNARY2`. Code for the other cases is similar. In the case of `PS_START`, the parser expects either an identifier or the empty token. If it detects an identifier, it checks the maps for unary and nonunary instructions using the string it gets from the token as the key. If the map has an entry for the key, it retrieves the corresponding mnemonic from the map. It stores the retrieved mnemonic in a local variable that it uses for the rest of the parse.

If it detects a unary instruction, it has all the information necessary to instantiate a concrete code object, which it gives to `aCode` with the statement

```
aCode = new UnaryInstr(localMnemon);
```

This is the side effect referred to earlier. If it detects `M_END`, it sets the termination flag to true, which eventually terminates the loop.

If it detects a nonunary instruction, it does not have all the information necessary to instantiate a concrete code object. It simply stores the local mnemonic to be used later and sets the next state to `PS_FUNCTION` using the direct-code technique.

In the case of `PS_COMMA`, the parser has detected the comma token and expects the second argument, which can be an identifier or an integer. If it is an identifier, it instantiates a new second argument object with the statement

```
localSecondArg = new IdentArg(
    localTIdentifier.getStringValue());
```

The constructor for the argument requires a string, which the parser has gotten from the token. The parser has previously instantiated the first argument the same way. Now it can instantiate the code object using the local mnemonic and the two arguments with the statement

```
aCode = new TwoArgInstr(
    localMnemon, localFirstArg, localSecondArg);
```

The code is similar if the second argument is an integer. In both instances, the state variable is set to `PS_SECOND_OPRND` in accordance with the FSM of Figure 7.46.

FIGURE 7.48 is a complete listing of the `actionPerformed()` function that invokes the translator. It is a simple three-step process. The first statement instantiates the input buffer with the source code string that the user enters in the input dialog box. The second statement instantiates

FIGURE 7.48
The `actionPerformed()` function for the translator that produces the output of Figure 7.37.

```
public void actionPerformed(ActionEvent event) {
   InBuffer inBuffer = new InBuffer(textArea.getText());
   Translator tr = new Translator(inBuffer);
   tr.translate();
}
```

the translator, passing it the input buffer in its constructor so it will have access to the source code. The third statement invokes the void function `translate()`.

The functions that perform the three phases of the automatic translation are as follows:

The three translation phases of the program

- Lexical analyzer: `getToken()`
- Parser: `parseLine()`
- Code generator: `generateCode()`

The lexical analyzer takes as input the stream of terminal characters from the input buffer and provides as output the resulting stream of tokens for the parser. The translator calls the parser for each line of source code, and the parser calls the lexical analyzer. In general, the output of the parser and input of the code generator is the syntax tree of the parse and/or the source program written in an internal low-level language. In this translator, the output of the parser and input of the code generator is only the source program written in an internal low-level language. This low-level language is the list of code objects stored in `codeTable`. After the parse is complete, `translate()` generates the code by iterating through the code table and calling the code generation function for each code object.

Parser Characteristics

Rather than define the syntax of the source language with the FSM of Figure 7.46, you could define it with a grammar. A formal grammar for the source language would have a simple structure. For example, a production rule for a `set` statement might be

<set-statement> → `set ( `<identifier>` , `<argument>` )`

where <argument> would be defined in another production rule as <identifier> or <integer>. Unlike in C, this grammar would contain no recursive definitions.

The simple nature of the source syntax allows the parsing of this language to be based on a deterministic FSM. Parsers for most programming languages cannot be this simple. Although it is common for lexical analyzers to be based on FSMs, it is rare that a parser can also be based on an FSM. In practice, most languages are too complex for such a technique to be possible.

Parsers are usually not based on an FSM.

Because the production rules of a real grammar invariably contain many recursive definitions, the parsing algorithm itself may contain recursive procedures that reflect the recursion of the grammar. Such an algorithm is called a *recursive descent parser*.

Regardless of the complexity of the source language or the parsing technique of the translator, the relationship of the parser to the lexical analyzer in a translation program is always the same. The parser is at a higher level of abstraction than the lexical analyzer. The lexical analyzer scans the characters and recognizes tokens, which it passes to the parser. The parser scans the tokens and produces a syntax tree and/or the source program written in an internal low-level language. The code generator uses the syntax tree and/or the low-level translation to produce the object code.

Chapter Summary

The fundamental question of computer science is: What can be automated? The automatic translation of artificial languages is at the heart of computer science. Each artificial language has an alphabet. The closure of a set, T^*, is the set of all possible strings formed by concatenating elements from T. A language is a subset of the closure of its alphabet. A grammar describes the syntax of a language and has four parts: a nonterminal alphabet, a terminal alphabet, a set of rules of production, and a start symbol. Derivation is the process by which a grammar determines a valid sentence in the language. To derive a sentence in the language, you begin with the start symbol and substitute production rules until you get a string of terminals. The parsing problem is to determine the substitution sequence to match a given string of terminals. There are hundreds of rules of production in the standard C grammar. A context-free grammar is one that restricts the left side of all the production rules to contain a single nonterminal. Although the C grammar is context-free, certain aspects of the language are context-sensitive.

A finite-state machine (FSM) also describes the syntax of a language. It consists of a set of states and transitions between the states. Each transition is marked with an input terminal symbol. One state is the start state, and at least one, possibly more, is the final state. A nondeterministic FSM may have more than one transition from a given state on one input terminal symbol. A sentence is valid if, starting at the start state, you can make a sequence of transitions dictated by the symbols in the sentence and end in a final state.

Two software implementation techniques of FSMs are the table-lookup technique and the direct-code technique. Both techniques contain loops that are controlled by a state variable, which is initialized to the start state. Each execution of the loop corresponds to a transition in the FSM. In the table-lookup technique, the transitions are assigned from a two-dimensional transition table. In the direct-code technique, the transitions are assigned with selection statements in the body of the loop.

The three translation phases of an automatic translator are the lexical analyzer, the parser, and the code generator. The input of the lexical analyzer is a stream of terminal symbols in the source program. The output of the lexical analyzer, which is the input to the parser, is a stream of tokens. The output of the parser, which is the input of the code generator, is an abstract syntax tree and/or the source program written in an internal low-level language. For most high-level languages, the lexical analyzer is based on an FSM and the parser is based on a context-free grammar. The code generator is highly dependent on the nature of the object language.

Exercises

Section 7.1

*__1.__ What is the fundamental question of computer science?

2. What is the identity element for the addition operation on integers? What is the identity element for the OR operation on Booleans?

3. Derive the following strings with the grammar of Figure 7.1 and draw the corresponding syntax tree:

*(a) `abc123` (b) `a1b2c3` (c) `a321bc`

4. Derive the following strings with the grammar of Figure 7.2 and draw the corresponding syntax tree:

*(a) `-d` (b) `+ddd` (c) `d`

5. Derive the following strings with the grammar of Figure 7.3:

*(a) `abc` (b) `aabbcc`

6. For each of the following strings, state whether it can be derived from the rules of the grammar of Figure 7.5. If it can, draw the corresponding syntax tree:

*__(a)__ `a + ( a )` **(b)** `a * ( + a )` **(c)** `a * ( a + a )`
(d) `a * ( a + a ) * a` **(e)** `a + ( - a )` **(f)** `( ( ( a ) ) )`

7. For the grammar of Figure 7.8, draw the syntax tree for <statement> from the following strings, assuming that *S1*, *S2*, *S3*, *S4*, *C1*, and *C2* are valid <expression>s:

*__(a)__

```
{ if ( C1 )
    S1 ;
  S2 ;
}
```

(b)

```
{ if ( C1 )
      if ( C2 )
          S1 ;
      else
          S2 ;
    S3 ;
}
```

(c)

```
{ if ( C1 )
      if ( C2 )
          S1 ;
        else
          S2 ;
      else
        S3 ;
      S4 ;
}
```

(d)

```
{ S1 ;
    while ( C1 )
    { if ( C2 )
          S2 ;
        S3 ;
      }
}
```

8. For the grammar of Figure 7.8, draw the syntax tree for <statement> from the following strings, assuming that `alpha`, `beta`, and `gamma` are valid <identifier>s and `1` and `24` are valid <constant>s:

*__(a)__ `alpha = 1 ;`
(b) `alpha = alpha + 1 ;`
(c) `alpha = (beta * 1) ;`
(d) `alpha = ((beta + 1) * (gamma + 24)) ;`
(e) `alpha (beta) ;`
(f) `alpha (beta, 24) ;`

9. For the grammar of Figure 7.8, draw the syntax tree for <translation-unit> from the following string, assuming that `alpha`, `beta`, `gamma`, and `main` are valid <identifier>s and *C1*, *S1*, and *S2* are <expression>s:

```
int main()
{ int gamma;
  alpha (gamma);
  if (C1)
      S1;
  else
      S2;
}
```

10. The question this exercise poses is "Can two different grammars produce the same language?" The grammars in FIGURE 7.49 and FIGURE 7.50 are not the same because they have different nonterminal sets and different production rules. Experiment with these two grammars by deriving some terminal strings. From your experiments, describe the languages produced by these grammars. Is it possible to derive a valid string of terminals with the grammar in Figure 7.49 that is not in 7.50 or vice versa? Prove your conjecture.

FIGURE 7.49
A grammar for Exercise 10.

$N = \{ A, B \}$
$T = \{ 0, 1 \}$
$P =$ the productions
1. $A \rightarrow 0\ B$
2. $B \rightarrow 1\ 0\ B$
3. $B \rightarrow \varepsilon$

$S = A$

FIGURE 7.50
Another grammar for Exercise 10.

$N = \{ C \}$
$T = \{ 0, 1 \}$
$P =$ the productions
1. $C \rightarrow C\ 1 0$
2. $C \rightarrow 0$

$S = C$

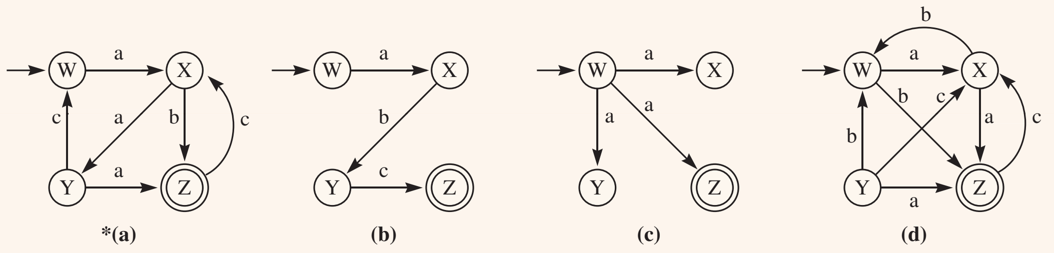
Section 7.2

11. For each of the machines shown in FIGURE 7.51, (1) state whether the FSM is deterministic or nondeterministic, and (2) identify any states that are inaccessible.

12. Remove the empty transitions to produce the equivalent machine for each of the FSMs in FIGURE 7.52.

13. Draw a deterministic FSM that recognizes strings of 1's and 0's specified by each of the following criteria. Each FSM should reject any characters

FIGURE 7.51
The FSMs for Exercise 11.



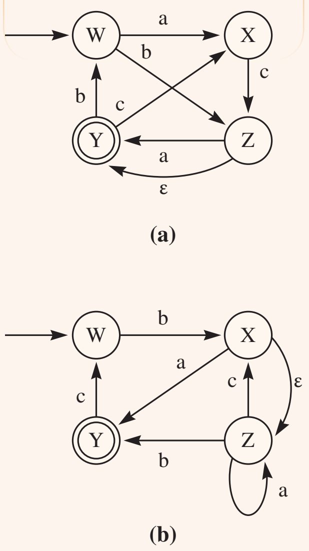


FIGURE 7.52 The FSMs for Exercise 12.

that are not 0 or 1. *(a) The string of three characters, `101`. **(b)** All strings of arbitrary length that end in `101`. For example, the FSM should accept `1101` but reject `1011`. **(c)** All strings of arbitrary length that begin with `101`. For example, the FSM should accept `1010` but reject `0101`. **(d)** All strings of arbitrary length that contain a `101` at least once anywhere. For example, the FSM should accept all the strings mentioned in parts (a), (b), and (c), as well as strings such as `11100001011111100111`.

Section 7.4

14. Design a grammar that describes the source language of the translator in Figure 7.47.

Problems

Section 7.3

15. Improve the program in Figure 7.28 as suggested in the text by defining a third enumeration in `Alphabet` called `T_OTHER`, which represents a symbol that is neither a letter nor a digit.

16. Implement each FSM in Exercise 13 using the table-lookup technique of the program in Figure 7.28. Classify a character as `B_ONE`, `B_ZERO`, or `B_OTHER` in the transition table.

17. Implement each FSM in Exercise 13 using the direct-code technique of the program in Figure 7.29. Write a procedure called `parsePat()` for a parse pattern that corresponds to `parseNum()`. Do not include the attribute `number` or method `getNumber()` in class `Parser`.

18. A hexadecimal digit is '`0`'..'`9`', or '`a`'..'`f`', or '`A`'..'`F`'. A hexadecimal constant is a sequence of hexadecimal digits. Examples include `3`, `a`, `0d`, and `FF4e`. Use the direct-code technique for implementing an FSM as in the program of Figure 7.29 to parse a hexadecimal constant and convert it to a nonnegative integer. The input/output should be similar to that in the figure, with invalid input producing an error message and a valid hexadecimal input string producing the nonnegative integer value.

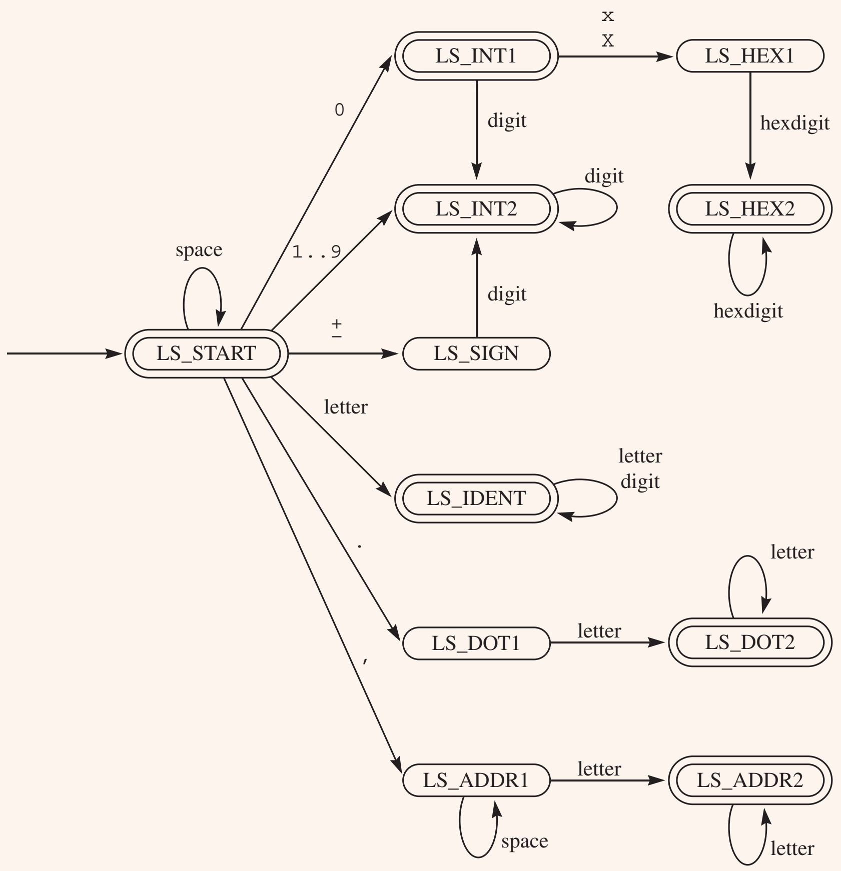
Section 7.4

19. Write an assembler for Pep/9 assembly language. Complete the following milestones in the order they are listed.

(a) Write class `Tokenizer` with method `getToken()`, to implement the FSM of FIGURE 7.53. Use class `InBuffer` from Figure 7.30. Implement method `getDescription()` for each concrete token and output the tokens with a nested `do` loop as in `actionPerformed()` of Figure 7.36.

Integers are stored in two bytes. When considered unsigned, the range is 0..65535. When considered signed, the range is −32768..32767. Your program must accept integers in the range −32768..65535. Each time you scan a decimal digit and update the total value, check it against this range. If inputting a decimal digit makes the total value go out of this range, return the invalid token.

FIGURE 7.53
The FSM for `getToken` in Problem 19(a).



Hexadecimal constants are also stored in two bytes and are never signed. The maximum value that a hexadecimal constant can have is 65535. Each time you scan a hex digit and update the total value, check its *decimal* value against this upper limit. If inputting a hex digit makes the total greater than this upper limit, return the invalid token. You should check the limit every time you scan a hexadecimal digit. Do not test that the number of hexadecimal digits is less than five, because, for example, `0x00F4B7` is valid.

Addressing modes must be stored with a Java `String` attribute as with identifiers. The parser will convert the identifiers to enumerated types by table lookup.

A common mistake is to call `advanceInput()` within the `switch` statement. Make sure you do not do that `.advanceInput()` must be called from only one place, namely as the first statement in the body of the `do` loop.

Following is an example input/output. All the tokens are valid according to the FSM. For example, there is no dot command `.beta`, nor is there an addressing mode `cat`. However, the corresponding tokens are valid. The parser detects the errors later in the translation.

Input

```
alpha .beta
   b7 0x23ab ,SfX
,i , cat
-32768 65535
```

Output

```
Identifier = alpha
Dot command = beta
Empty token
Identifier = b7
Hexadecimal constant = 9131
Addressing Mode = SfX
Empty token
Addressing Mode = i
Addressing Mode = cat
Empty token
Integer = -32768
Integer = 65535
Empty token
```

(b) Design the state transition diagram for the FSM of the Pep/9 parser that corresponds to the FSM of Figure 7.46. Assume that each transition is on one of the tokens in Figure 7.53.

(c) This phase of the project is to write the parser based on your FSM of part (b). Complete the `generateListing()` methods of the code classes, and output the formatted listing of the source program but not the object code. Here is the list of instructions your program should process:

- Unary instructions—`STOP`, `ASLA`, `ASRA`
- Nonunary instructions—`BR`, `BRLT`, `BREQ`, `BRLE`, `CPWA`, `DECI`, `DECO`, `ADDA`, `SUBA`, `STWA`, `LDWA`
- Dot commands—`.BLOCK`, `.END`
- Constants—decimal, hexadecimal

Design an abstract argument `AArg` with two subclasses for a hexadecimal constant and a decimal constant, each with an integer attribute, analogous to Figure 7.40. Design your abstract code class `ACode` analogous to the code class in Figure 7.44. The class for a nonunary mnemonic must have an abstract argument for its instruction specifier and an addressing mnemonic for its addressing mode, which must be enumerated as described in part (a). Do not combine the addressing mode enumerated types with any other enumerated type. They must be separate. Set up separate Java maps for looking up unary mnemonic identifiers, nonunary mnemonic identifiers, dot commands, and addressing modes. For your code classes, do not use a Boolean attribute to distinguish unary from nonunary instructions. Instead, have separate classes for unary and nonunary instructions.

Do not use the names `OneArgInstr` or `TwoArgInstr` from the Figure 7.44 example to describe your instructions. In Pep/9 assembly language, instructions are either unary or nonunary. Do not use the names `firstArg` or `secondArg` from the figure to describe the items that follow the mnemonic. For nonunary instructions, the items following the mnemonic are the operand specifier and the addressing mode.

If you detect an illegal addressing mode or other error, you must generate an error code object to handle the error. For example, do not use the nonunary code object to generate any error messages.

The output should conform to the standard pretty-printing format of the Pep/9 assembler when you select Format From Listing in the Edit menu. For hexadecimal constants, the `%X` format placeholder will output an integer value in hexadecimal format. Research the Java documentation for the field width and leading zero options. For strings, the `%s` format placeholder has options to either left justify or right justify in a field padded with spaces.

(d) Complete the `generateCode()` methods of your code classes to emit the hexadecimal object code for the assembly language program in a format suitable for use by the Pep/9 loader. Following is an example input/output. Your code generator should emit one line of hex pairs for each line of source code to make it easy to visually compare the object with the source.

Input

```
BR   0x0007,  i
.BLOCK 4
deci    0x2   ,d
LDWA     +2,d
AdDa -5,  i
STWA     0x0004,d
    DECO    0x04,d
STOP
.END
```

Output

```
Object code:
12 00 07
00 00 00 00
31 00 02
C1 00 02
60 FF FB
E1 00 04
39 00 04
00
zz

Program listing:
BR      0x0007
.BLOCK  4
DECI    0x0002,d
```

```
LDWA    2,d
ADDA    -5,i
STWA    0x0004,d
DECO    0x0004,d
STOP
.END
```

To get a decimal value into hex, you can use the fact that `n/256` is an eight-bit right shift of `n`. Use it to output the first byte of integer `n`. Also, `n%256` is an eight-bit remainder. Use it to output the second byte of integer `n`.

All hex digit pairs in the object code must be separated by exactly one space, no lines in the object code may contain a trailing space at the end of the line, and the entire sequence must terminate with lowercase `zz`. To test your object code, copy the hex code from the Java console, paste it into the object code pane of the Pep/9 application, and execute your program.

(e) Extend the assembler by including all 40 instructions in the Pep/9 instruction set.

(f) Extend the assembler by producing a listing that shows the object code next to the source line that produced it. Print the source line with the standard spacing conventions and uppercase and lowercase conventions of the Pep/9 assembler.

(g) Extend the assembler by permitting character constants enclosed in single quotes.

(h) Extend the assembler by permitting the dot commands `.WORD` and `.BYTE`.

(i) Extend the assembler by permitting the `.ASCII` dot command with strings enclosed in double quotes.

(j) Extend the assembler by permitting a source line to contain a comment prefixed by a semicolon. A line may contain only a comment, or a valid instruction followed by a comment.

(k) Extend the assembler by permitting symbols.