Chapter 7

Assembling to the ISA Level

You are now multilingual because you understand at least four languages—English, C, 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 documents. 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, 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 a Pep/10 assembly language program to machine 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

Section 7.1 introduces grammars and shows how they apply to the C programming language. Section 7.2 describes finite-state machines and compares them to grammars. Section 7.3 shows how finite-state machines help to implement the first stage of a translator, which is known as the *lexical analyzer*. Section 7.4 shows how a grammar helps to implement the second stage of a translator, which is known as the *parser*. Section 7.5 shows how to implement the third stage of a translator, which is known as the *code generator*.

The lexical analyzer, parser, and code generator

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/10 assembly language is similar except for some punctuation characters, as shown in the following set:

The Pep/10 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 real number alphabet

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

Concatenation

A data type is a set of 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/10 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

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 first part of the function definition

```
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

$$true \text{ AND } q = q \text{ AND } true = q$$

■

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:

Some elements in the closure of the English alphabet

```
To be or not to be, that is the question.
My dog has fleas.
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

Some elements in the closure of the real number alphabet

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

Because strings can be infinitely long, the closure of any alphabet has an infinite number of elements.

What is a language? In the above examples of T^*, 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 definition of a grammer

- 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

Terminal and nonterminal alphabets

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 English grammar, <noun> is a nonterminal. A valid English sentence is

```
My dog has fleas.
```

You would never see the nonterminal in a sentence like this:

```
My <noun> has fleas.
```

The nonterminal symbol, <noun>, is useful for describing the structure of an English sentence. ■

Example 7.4 In the C grammar, the nonterminal <compound-statement> might represent the following group of terminals:

```
{
   int i;
   scanf("%d", &i);
   printf("%d", i);
   return 0;
}
```

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. ■

The start symbol and rules of production

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 in the language.

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,

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

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

$S =$ <identifier>

Figure 7.1 A grammar for C identifiers.

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.

The rules of production are of the form

A general rule of production

$A \to w$

where A is a nonterminal and w is a concatenation of terminals and nonterminals. The symbol $\to$ means "produces." You should read production Rule 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 $\Rightarrow$ means "derives in one step."

A derivation of `cab3`

<identifier>	$\Rightarrow$ **<identifier> <digit>**	Rule 3
	$\Rightarrow$ <identifier> **`3`**	Rule 9
	$\Rightarrow$ **<identifier> <letter>** `3`	Rule 2
	$\Rightarrow$ <identifier> **`b`** `3`	Rule 5
	$\Rightarrow$ **<identifier> <letter>** `b 3`	Rule 2
	$\Rightarrow$ <identifier> **`a`** `b 3`	Rule 4
	$\Rightarrow$ **<letter>** `a b 3`	Rule 1
	$\Rightarrow$ **`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,

<identifier> → <identifier> <letter>

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

<identifier> `3` ⇒ <identifier> <letter> `3`

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

The closure of a derivation

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

<identifier> $\Rightarrow^*$ `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

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

$S =$ I

Figure 7.2 A grammar for signed integers.

I → FM
F → `+` | `-` | ε
M → `d` | `d`M

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."

The alternation operator

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

Some derivations of signed integers

I ⇒ **FM**	I ⇒ **FM**	I ⇒ **FM**
⇒ F**`d`**M	⇒ F**`d`**M	⇒ F**`d`**
⇒ F`d`**`d`**M	⇒ F`d`**`d`**	⇒ **`+`**`d`
⇒ F`dd`**`d`**	⇒ `dd`	
⇒ **`-`**`ddd`		

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

The productions in both of the above 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 above 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
⇒ `a`**`a`AB**CBC	Rule 1
⇒ `aa`**`ab`C**CBC	Rule 2
⇒ `aaab`**BC**CBC	Rule 3
⇒ `aaab`BC**BC**C	Rule 3
⇒ `aaab`B**BC**CC	Rule 3
⇒ `aaa`**`bb`**BCCC	Rule 4
⇒ `aaab`**`bb`**CCC	Rule 4
⇒ `aaabb`**`bc`**CC	Rule 5
⇒ `aaabbb`**`cc`**C	Rule 6
⇒ `aaabbbc`**`cc`**	Rule 6

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

Figure 7.3 A context-sensitive grammar.

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

`aaabbb`CCC ⇒ `aaabbbc`CC

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, the above definitions will suffice.)

Context-free grammars

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

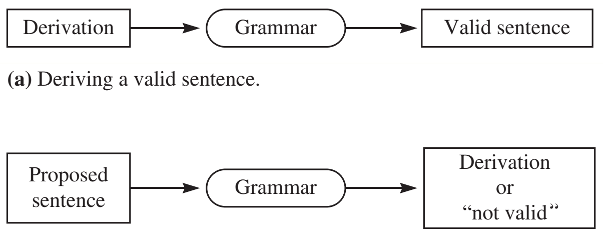
Deriving valid strings from a grammar is fairly straightforward. You can arbitrarily pick some nonterminal on the right side of the current intermediate string and select rules for the substitution repeatedly until you get a string of terminals. Such random derivations can give you many sample strings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

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

Parsing is more difficult than deriving.



(a) Deriving a valid sentence.

(b) The parsing problem.

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

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. Nonterminal E represents the expression. T represents a term and F a factor in the 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 ⇒ **E + T**	Rule 1
⇒ **T** + T	Rule 2
⇒ **F** + T	Rule 4
⇒ **(E)** + T	Rule 5
⇒ (**T**) + T	Rule 2
⇒ (**T * F**) + T	Rule 3
⇒ (**F** * F) + T	Rule 4
⇒ (**a** * F) * T	Rule 6
⇒ (a * **a**) * T	Rule 6
⇒ (a * a) * **F**	Rule 5
⇒ (a * a) * **a**	Rule 6

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

$S =$ E

Figure 7.5 A grammar for expressions.

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 `dd` with the grammar in Figure 7.2.

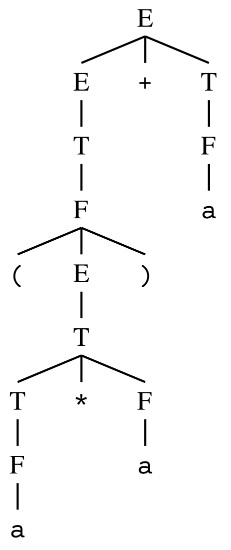


Figure 7.6 The syntax tree for the parse of `(a*a)+a`.

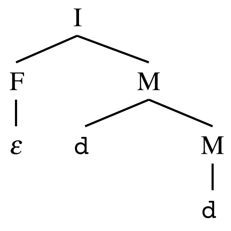


Figure 7.7 The syntax tree for the parse of `dd`.

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 constants or pointers. 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.

The start symbol for C

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>.

Backus Naur Form

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.

The following example of a parse with this grammar shows that

```
<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>
        | if ( <expression> ) <statement> else <statement>
<iteration-statement> →
        while ( <expression> ) <statement>
        | do <statement> while ( <expression> ) ;
```

Figure 7.8 A grammar for a subset of the C language. (*Continues*)

<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>

Figure 7.8 (*Continued*) A grammar for a subset of the C language.

(*Continues*)

```
<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
```

Figure 7.8 (*Continued*) A grammar for a subset of the C language.

```
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. Notice that the last step in the derivation uses the closure of the derivation operator $\Rightarrow^*$. It assumes that *S1* is some expression, for example `a+1`, that can be derived from <expression> in zero or more steps.

A dashed line in a syntax tree corresponds to the closure of a derivation operation in a derivation.

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>. Notice the dashed line that connects <expression> with *S1* and is how the closure of the derivation operation is rendered in a syntax tree.

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.

The parser for a C compiler

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 for a C compiler

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 architec-

<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*** **`;`**

Figure 7.9 The derivation to prove that `while(a<=9)` *S1* `;` is a valid <statement> for the grammar in Figure 7.8.

ture with different instruction sets, code-generation techniques 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` gener-

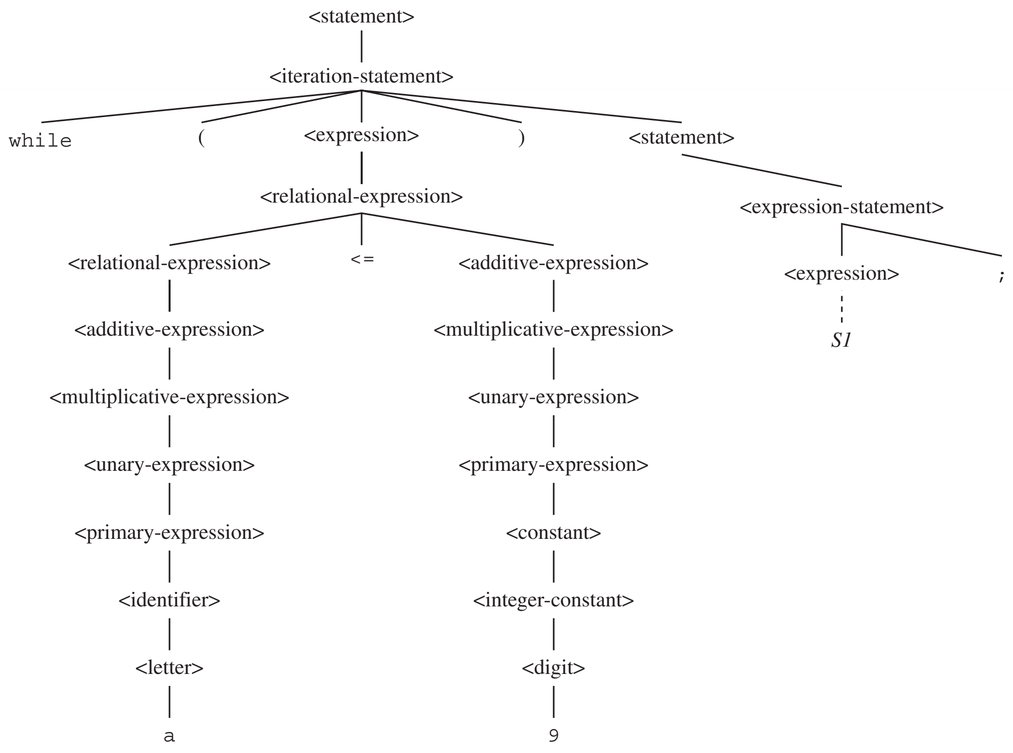


Figure 7.10 The syntax tree for the parse of `while(a<=9)` *S1* `;` in Figure 7.9.

ated, 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.

A context-sensitive grammar for C is not practical.

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

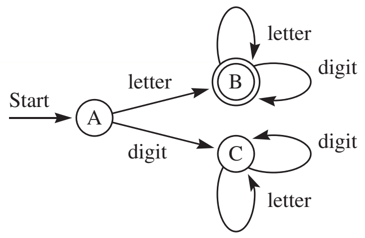


Figure 7.11 An FSM to parse an identifier.

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*.

An FSM to Recognize 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.

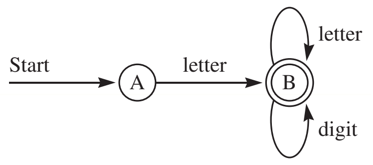


Figure 7.13 The FSM of Figure 7.11 without the failure state.

Example 7.5 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. ■

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

Figure 7.12 The state transition table for the FSM in Figure 7.11

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.

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.

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

Figure 7.14 The state transition table for the FSM in Figure 7.13

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.

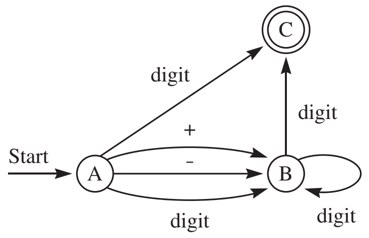


Figure 7.15 A nondeterministic FSM to parse a signed integer.

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

Example 7.6 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

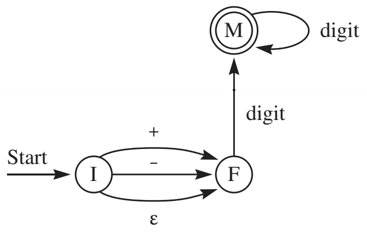


Figure 7.17 An FSM with an empty transition to parse a signed integer.

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

Figure 7.16 The state transition table for the FSM in Figure 7.15

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 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.

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

Figure 7.18 The state transition table for the FSM in Figure 7.17

Example 7.7 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.

The algorithm to remove an empty transition

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

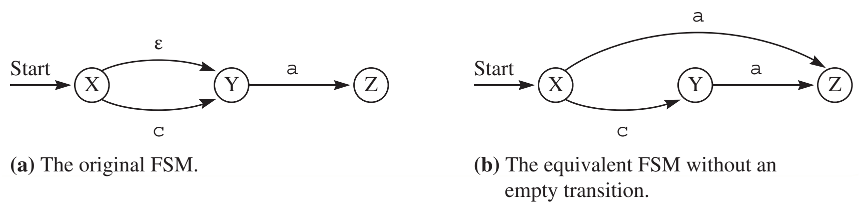


Figure 7.19 Removing an empty transition

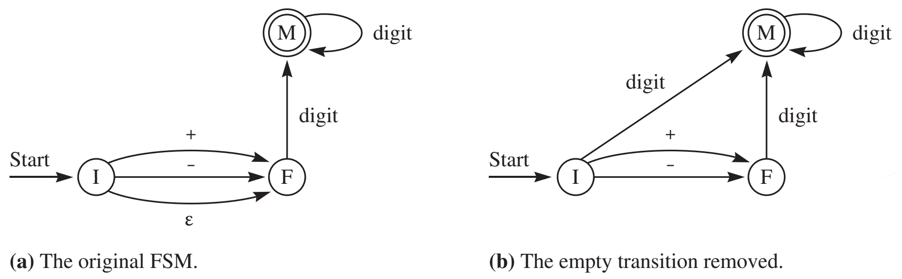


Figure 7.20 Removing the empty transition from the FSM of Figure 7.17

the algorithm to eliminate 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:

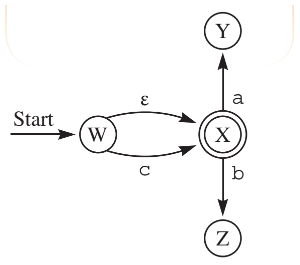
$\varepsilon a = a$

Example 7.8 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 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. ■

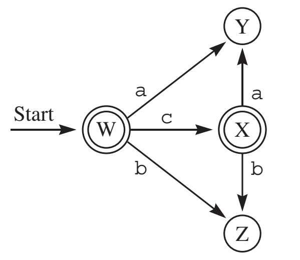
Example 7.9 Figure 7.20 shows this transformation on the FSM of Figure 7.17. The empty transition from I to F is replaced by the transition from I to M on digit, because there is a transition from F to M on digit. ■

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 7.10 To eliminate the empty transition from W to X in Figure 7.21, 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. ■



(a) The original FSM.



(b) The equivalent FSM without an empty transition.

Figure 7.21 Removing an empty transition.

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 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.

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

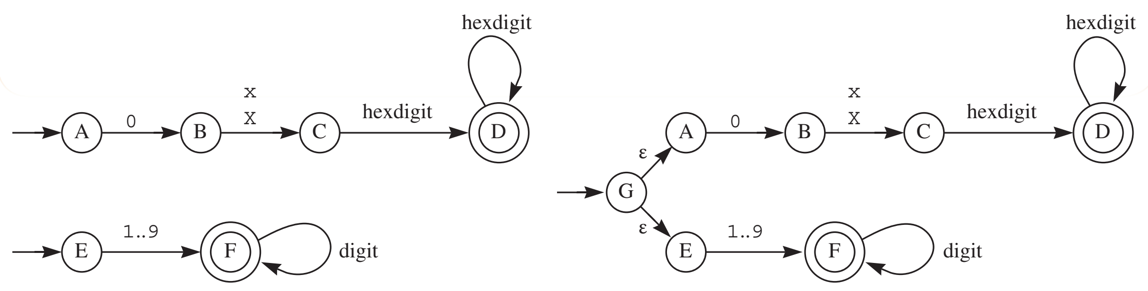
A token is a string of terminal characters that has meaning as a group. The characters usually correspond to some nonterminal in a language's grammar. For example, consider the Pep/10 assembly language statement

```
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.

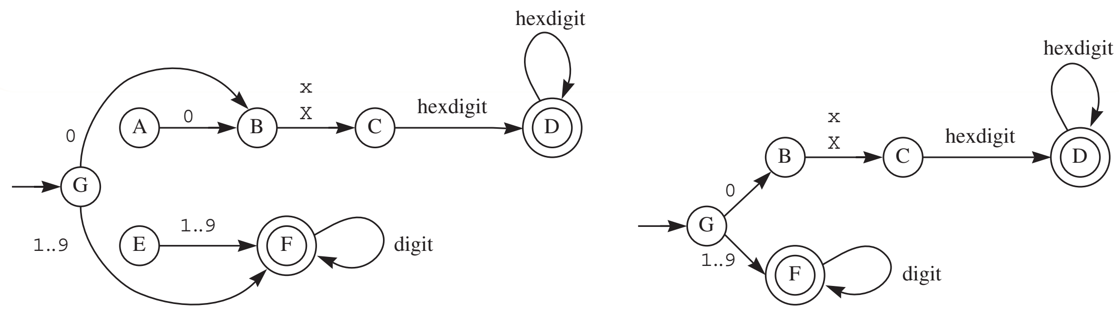
A common use of an FSM in a translator is to detect the tokens in the source string. Consider the assembler's job when confronted with this source line. Suppose the assembler has already determined that `mask:` is a symbol definition and `.WORD` is a dot command. It knows that either a decimal or hexadeci-



(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.

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



(a) Removing the empty transitions.

(b) Removing the inaccessible states.

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

mal constant can follow the dot command, so it must be programmed to accept either. It needs an FSM that recognizes both.

Figure 7.22 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`.

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 shows removal of the empty transitions for the FSM of Figure 7.22(b). After their removal, states A and E are 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 monadic 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.

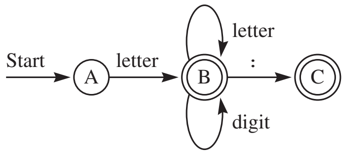


Figure 7.24 An FSM to parse a Pep/10 assembly language identifier or symbol definition.

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

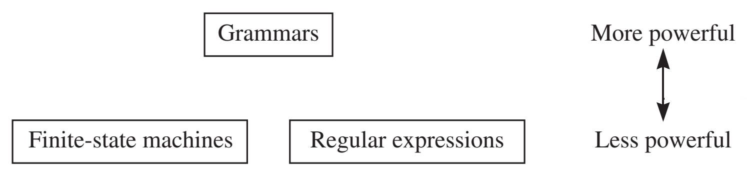


Figure 7.25 The power of grammars, FSMs, and regular expressions.

an identifier. The rules for forming a valid identifier are that the first 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.

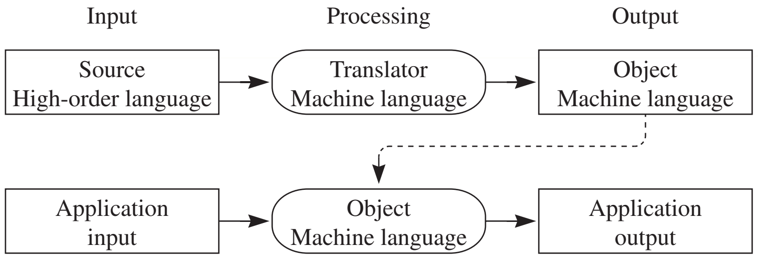
Figure 7.5 is a grammar for an expression. The language of expressions is so complex that it is mathematically impossible to specify an FSM that can parse an expression. The problem with FSMs for expressions is that you can have unlimited nested parentheses. Once the FSM scans a left parenthesis, it 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.

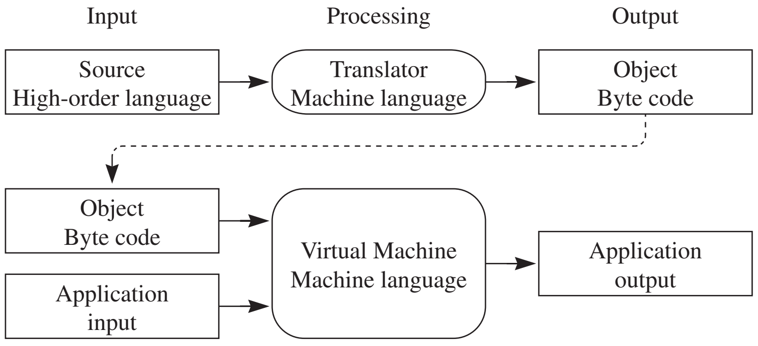
Although a description of regular expressions is beyond the scope of this text, how powerful are they? It turns out that for every regular expression there is an equivalent FSM, and for every FSM there is an equivalent regular expression. Consequently, FSMs and regular expressions are equal in power 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 Constructing a Lexical Analyzer

The remainder of this chapter shows how a language translator converts a Pep/10 assembly language source program into a machine language object program. It uses the Python language rather than C to illustrate the translation techniques. The syntax of Python is similar to that of C, and it has the advantage of a "batteries included" standard library. The programs in this chapter receive their input as a string of terminal characters via command line interface (CLI) arguments in a terminal and send the results of the translation to the terminal's standard output. The CLI programming details are not shown but are available with the software for this text.



(**a**) Compilation.



(**b**) Interpretation.

Figure 7.26 The difference between compilation and interpretation.

Python itself is an interpreted language, whose standard implementation is based on a virtual machine. **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 Python and Pep/10, both of which are based on virtual machines. In the first 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.

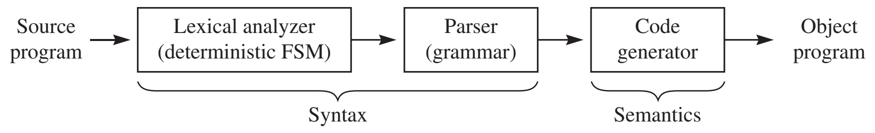


Figure 7.27 Steps in the compilation process.

The language maintainers must have a compiler for every platform. With an interpreted language like Python, 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.

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 Lexical Analyzer

The program in Figure 7.28 implements the FSM of Figure 7.11 with the table-lookup technique. `Table.transitions`, a nested dictionary, 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.

The second python fragment of Figure 7.28 illustrates how the terminal application creates an instance of the `Table` class, and invokes newly created instance's `parse()` method. The input comes either directly via text on a command line argument or from a file. The terminal application reads the specified text and passes it to `main()`. The input and output shown in Figure 7.28 are a reflection of the arguments and standard output of the terminal application .

The `for` loop processes the terminal characters one at a time by iterating over the contents of `text`. It looks up the next state in the FSM table from the current state and the current character kind.

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.

Python `fsm/table.py`

```python
class Table:
    class States(enum.IntEnum):
        A = 0
        B = 1
        C = 2

    class Kind(enum.IntEnum):
        Letter = 0
        Digit = 1

    transitions: Dict[States, Dict[Kind, States]] = {
        States.A: {Kind.Letter: States.B, Kind.Digit: States.C},
        States.B: {Kind.Letter: States.B, Kind.Digit: States.B},
        States.C: {Kind.Letter: States.C, Kind.Digit: States.C},
    }

    def parse(self, text: str):
        state = Table.States.A
        for ch in text:
            kind = Table.Kind.Letter if ch.isalpha() else Table.Kind.Digit
            state = Table.transitions[state][kind]
        return state == Table.States.B
```

Python

```python
from cs6th_ch7.fsm import Table

def main(text: str):
  fsm = Table()
  valid = fsm.parse(text)
  print(f"{text} is {'' if valid else 'not'} a valid identifier")
```

Input
```
h3ll0
```
Output
```
h3ll0 is a valid
identifier
```
(a) First run.

Input
```
3cab
```
Output
```
3cab is not a valid
identifier
```
(b) Second run.

Input
```
cab#3
```
Output
```
cab#3 is a valid
identifier
```
(c) Third run.

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

Python `fsm/direct.py`

```python
class Direct:
    class States(enum.Enum):
        I = 0
        F = 1
        M = 2
        STOP = 3

    def parse(self, text: str):
        text = text + "\n"
        state = Direct.States.I
        valid, magnitude, sign = True, 0, +1

        while state != Direct.States.STOP and valid:
            ch, text = text[0], text[1:] if len(text) > 1 else ""
            match state:
                case Direct.States.I:
                    if ch == "+":
                        sign, state = 1, Direct.States.F
                    elif ch == "-":
                        sign, state = -1, Direct.States.F
                    elif ch.isdigit():
                        magnitude, state = int(ch), Direct.States.M
                    else:
                        valid = False

                case Direct.States.F:
                    if ch.isdigit():
                        magnitude, state = int(ch), Direct.States.M
                    else:
                        valid = False

                case Direct.States.M:
                    if ch.isdigit():
                        magnitude = 10 * magnitude + int(ch)
                    elif ch == "\n":
                        state = Direct.States.STOP
                    else:
                        valid = False

        return valid, sign * magnitude if valid else None
```

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

A Direct-Code Lexical Analyzer

The program in Figure 7.29 uses the direct-code technique to parse a signed integer. It is an implementation of the FSM of Figure 7.20(b). The `Direct` class has one method named `parse()` that takes a string named `text` as input. If `text` is a valid signed integer, `parse()` returns a couple whose first element is `True` and second element is the integer value that `text` represents.

Otherwise, it returns a couple whose first element is `False` and second element is `None`.

The algorithm appends a newline character to `text`, which acts as a sentinel. Regardless of how many or few characters the user enters there will always be at least one character to process, even if the user provides an empty value to `text`.

The method has a local variable named `state`, whose possible values are the enumerated constants of `Direct.States`—`I`, `F`, or `M`—corresponding to the states of the FSM in Figure 7.20. An additional state named `STOP` is for terminating the loop. The method initializes `valid` to `True` and `state` to the start state `I`.

Each execution of the body of the `while` loop simulates one transition in the FSM. First, local variable `ch` gets the first character from `text`, and `text` is replaced by the remaining string, decreasing its length by one. Then, a single `match` statement determines the current state, and a single `if/elif/else` statement nested within each case processes the next character `ch`. 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. The test for the `while` loop

```
while state != Direct.States.STOP and valid:
```

corresponds to these stopping conditions.

`parse()` executes correctly even when called with an empty string. It initializes `state` to `I`, enters the `while` loop, and immediately sets `ch` to the newline character, which was appended to the empty input. Then `valid` gets `False`, and the loop terminates.

In addition to determining whether the string is valid, `parse()` converts the string of characters to the correct 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 `magnitude` to its proper value in state `I` or `F`. Its value is accumulated while in state `M` each time a succeeding digit is detected. `magnitude` is multiplied by `sign` when the function returns with a valid number.

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.

Integrating semantic actions with syntactic actions

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.

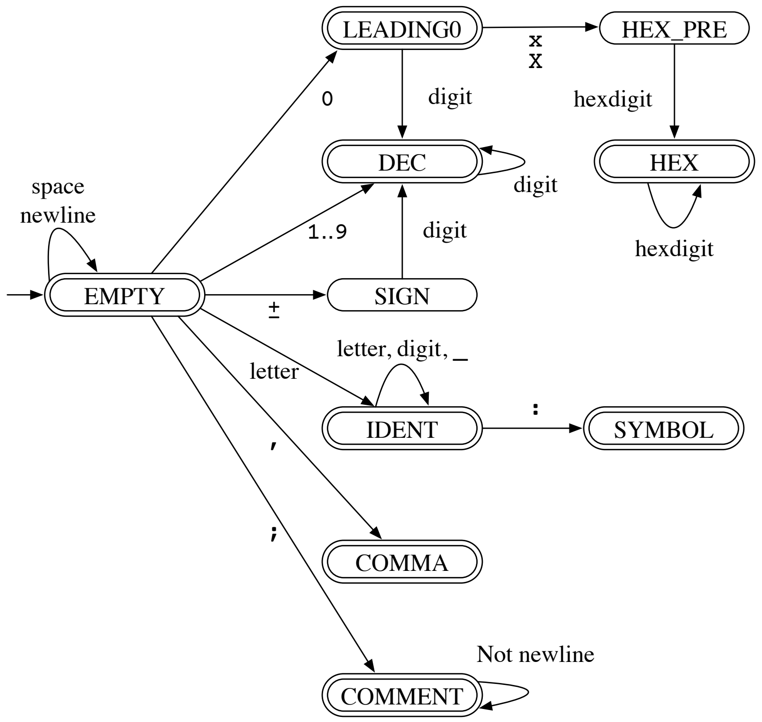


Figure 7.30 The lexical analyzer for a subset of Pep/10 assembly language.

A Multiple-Token Lexical Analyzer

The output of a lexer is a token, which is the input to the parser. The parser considers a token to be an element of its terminal alphabet. If the parser of a Pep/10 assembler is analyzing the string

```
main: LDWA
```

it knows that the next token could be an identifier such as `area`, a decimal constant such as `57`, or a hexadecimal constant such as `0x002B`. Because the parser does not know which token to expect, it calls a lexer that can recognize any token, as in Figure 7.30.

The start state, labeled EMPTY, is also a final state with a transition to itself on the space and newline characters. It allows for leading spaces before any token. Furthermore, if the only characters left to scan are trailing spaces at the end of a line, the lexer returns the empty token. So, the lexer returns the empty token at the end of every line, even empty lines.

Each final state of Figure 7.30 yields a different kind of token, listed in Figure 7.31. Except for the two final states that both produce decimal integer tokens, all other final states have unique token types. While in a final state,

Final state	Python class name	Description	Sample values
EMPTY	`Empty`	The terminal `\n`	
COMMENT	`Comment`	Comments beginning with `;`	`;local variable`
COMMA	`Comma`	The terminal `,`	`,`
DEC, LEADING0	`Decimal`	Decimal integers	`0, 257, -15`
HEX	`Hexadecimal`	Hexadecimal integers	`0x0, 0xFEED`
IDENT	`Identifier`	Identifiers	`main, a1_`
SYMBOL	`Symbol`	Symbol declarations	`main:, a1_:`

Figure 7.31 The token types of Figure 7.30.

Input
```
Here is A47 48B
    C-49 ALongIdentifier +50 D16-51
```

Output
```
Identifier(value='Here')
Identifier(value='is')
Identifier(value='A47')
Decimal(value=48)
Identifier(value='B')
Empty()
Identifier(value='C')
Decimal(value=-49)
Identifier(value='ALongIdentifier')
Decimal(value=50)
Identifier(value='D16')
Decimal(value=-51)
Empty()
```

(a) First run.

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

Output
```
Identifier(value='Here')
Identifier(value='is')
Identifier(value='A47')
Invalid()
Decimal(value=48)
Identifier(value='B')
Empty()
Identifier(value='C')
Decimal(value=49)
Empty()
```

(b) Second run.

Figure 7.32 The input/output of a program that recognizes Figure 7.30.

if the machine scans a terminal character and the current state has no transitions on that character, processing stops and the lexer returns the current token. The unused character must be put back into the input buffer, so that it may be scanned as the first terminal character of the next token.

Figure 7.32 shows the input and output of two runs from a program that implements the multiple-token recognizer of Figure 7.30. The first run has an input of two lines, the first line with five nonempty tokens and the second line with six nonempty tokens.

In first run of Figure 7.32a, the machine begins in the start state and scans the first terminal, `H`, and the state transitions from EMPTY to IDENT. 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. The machines puts the space terminal, which could not use in this state, back into the input buffer for use as the first terminal for the next token. It then declares that an identifier has been scanned.

The machine begins again in the start state. It uses the left over space to make a transition to EMPTY. A few more spaces produce a few more transitions to EMPTY, 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 DEC state. The `8` makes the transition to the same state. Now the machine scans the `B`. There is no transition from state DEC on the terminal `B`. Because the machine is in the final state for integers, it concludes that an integer has been scanned. The machines 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. The machine 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 or newlines in the input because the lexical analyzer inserts a trailing newline automatically.

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 principles:

- If a transition exists from the current state on the next terminal, that transition must be taken. This greedy strategy produces the longest match possible. It is also referred to as *maximal munch*.
- 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.

Maximal munch

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

Figure 7.33 shows the Python definitions of the token classes used by the Pep/10 assembler. Token types with a value member, like `Decimal`, need additional information between their lexical category to reconstruct the meaning of the token. For example, both `425` and `717` scan as `Decimals` tokens.

Python pep10/tokens.py

```python
@dataclass
class Identifier:
    value: str


@dataclass
class Symbol:
    value: str


@dataclass
class Comment:
    value: str
```

```python
@dataclass
class Empty: ...


@dataclass
class Invalid: ...


@dataclass
class Comma: ...
```

```python
@dataclass
class Decimal:
    value: int


@dataclass
class Hexadecimal:
    value: int
```

Figure 7.33 Pep/10 token classes of Figure 7.31

Without the value associated with that token, they cannot be distinguished. Other types without a value member, like `Empty`, only need to record their lexical category. It is not possible to distinguish one newline from another, so `Empty` does not require a value.

Figure 7.34 is a direct-code implementation of the FSM of Figure 7.30. The class's constructor (`__init__()`) receives a text buffer that has been loaded with the input string. Method `__next__()` returns an instance of one of the token classes from Figure 7.33. When combined with `__iter__()`, the `Lexer` class becomes *iterable*—like a `list` or `str`.

The states of the FSM in Figure 7.30 are contained in the enumerated constant `Lexer.States`. As in Figure 7.29, a `STOP` state has been added to terminate loop execution.

Each call to `__next__()` returns the next single token in the input stream. The body of `__next__()` begin by initializing the state variable to the start state. It also initializes several bookkeeping variables used to compute the values associated with tokens and records the position in the input stream prior to entering the loop. As in Figure 7.29, the `while` loop of Figure 7.34 simulates the transitions in the FSM. The loop continues until the machine reaches the stop state, an invalid token has been recognized, or the input stream has been exhausted.

To enable input to be backed up, the FSM uses a stream class from Python's standard library called `io.StringIO`. Calling `tell()` on a `io.StringIO` returns a marker to the stream's current position which can be used to return to the that position in the future with `seek()`. The FSM uses `read()` to advance sequential through the buffer. `read(1)` either returns the next single character in the stream and and increments the stream's position or returns an empty string because the stream has been exhausted. If the FSM `read()` function returns a character and decides to put that character back into the stream,

Python `pep10/lexer.py`

```python
class Lexer(TokenProducer[Token]):
    class States(Enum):
        START, COMMENT, IDENT, LEADING0, HEX_PRE = range(0, 5)
        HEX, SIGN, DEC, STOP = range(5, 9)

    def __init__(self, buffer: io.StringIO) -> None:
        self.buffer: io.StringIO = buffer

    def __iter__(self) -> "Lexer":
        return self

    def __next__(self) -> Token:
        state: Lexer.States = Lexer.States.START
        as_str_list: List[str] = []
        as_int: int = 0
        sign: Literal[-1, 1] = 1
        token: Token = tokens.Empty()
        initial_pos = self.buffer.tell()

        while state != Lexer.States.STOP and (
            type(token) is not tokens.Invalid
        ):
            prev_pos = self.buffer.tell()
            ch: str = self.buffer.read(1)
            if len(ch) == 0:
                if initial_pos == prev_pos:
                    raise StopIteration()
                ch = "\n"

            ...

        return token
```

Figure 7.34 A Pep/10 Lexer

it calls `seek()` with the marker previously returned by `tell()`. For this reason, every `read()` is preceded by a `tell()`.

After reading the next character, the FSM handles the case where it reached the end of the input stream. If some characters have been read since first entering the loop (e.g., `initial_pos != prev_pos`), then the machine should attempt to complete that token. It simulates a trailing newline in the input stream by setting `ch = '\n'` before entering the `match`. Otherwise, the machine needs to prevent callers from iterating further by raising a `StopIteration` exception. This is a requirement of Python's iteration protocol.

Figure 7.35 shows the start of a large `match` statement with one case for each state and nested ifs for each transition. Each of the `if` statements corresponds to on of the transitions of Figure 7.30. Each branch updates the state variable on an outbound transition.

The bookkeeping variables introduced in Figure 7.34 are updated using semantic actions. Textual sequences tokens (e.g., Identifiers, Comments) accu-

Python `pep10/lexer.py` (*Continued*)

```
            match state:
                case Lexer.States.START:
                    if ch == "\n":
                        state = Lexer.States.STOP
                    elif ch == ",":
                        state = Lexer.States.STOP
                        token = tokens.Comma()
                    elif ch.isspace():
                        pass
                    elif ch == ";":
                        state = Lexer.States.COMMENT
                    elif ch.isalpha():
                        as_str_list.append(ch)
                        state = Lexer.States.IDENT
                    elif ch == "0":
                        state = Lexer.States.LEADING0
                    elif ch.isdecimal():
                        state = Lexer.States.DEC
                        as_int = ord(ch) - ord("0")
                    elif ch == "+" or ch == "-":
                        state = Lexer.States.SIGN
                        sign = -1 if ch == "-" else 1
                    else:
                        token = tokens.Invalid()
```

Figure 7.35 A direct-code implementation of the START state of Figure 7.30

mulate characters in `as_str_list`, while numeric token accumulate integers values in `sign` and `as_int`.

Notice that the COMMA state has been collapsed into START. COMMA has no outbound transitions. Suppose a COMMA state was added. START would update the state variable to COMMA, and the next iteration of the loop would begin. The machine would read a character that would unconditionally be put back into the input stream, producing spurious calls to `tell()`, `read()`, and `seek()`. Instead, the machine transitions directly to the STOP state and recognizes the correct token type for COMMA. A similar transformation occurs for the SYMBOL state inside IDENT, since SYMBOL also has no outgoing transition.

As shown in Figure 7.36, 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. In the case the IDENT state, this corresponds to terminals that are not alphanumeric, nor the terminals **:** and _. The machine puts the character back into the input stream using `seek()`.

Python pep10/lexer.py (*Continued*)

```python
                case Lexer.States.IDENT:
                    if ch.isalnum() or ch == "_":
                        as_str_list.append(ch)
                    elif ch == ":":
                        state = Lexer.States.STOP
                        token = tokens.Symbol("".join(as_str_list))
                    else:
                        self.buffer.seek(prev_pos, os.SEEK_SET)
                        state = Lexer.States.STOP
                        as_str = "".join(as_str_list)
                        token = tokens.Identifier(as_str)
```

Figure 7.36 A direct-code implementation of the IDENT and SYMBOL states of Figure 7.30

7.4 Constructing a Parser

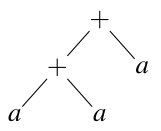
There are two strategies for constructing a parser: top-down and bottom-up. Top-down parsers start with a nonterminal and recursively expand the nonterminal using the productions of the grammar until a derivation matching the input is found. If the parser has the choice of multiple rules at a given step, it cannot declare the program ill-formed until it fully explores all branches. Exploring the different branches may involve significant backtracking.

Bottom-up parsers consume input tokens until a production rule has been matched, at which point a nonterminal has been recognized. The process of recognizing nonterminals continues until input has been exhausted or a starting nonterminal has been reached. While bottom-up parser handle broader classes of grammars than top-down parsers, this additional power comes at the cost of implementation complexity. Bottom-up parsers tend to require machine assistance to design, whereas top-down parsers are easy to implement by hand. This text focuses on top-down parsers for their ease of implementation.

Top-Down Parsing

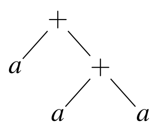
Top-down parsers employ a method known as ***L**eft-to-right, **L**eftmost derivation*, otherwise known as *LL*. Left-to-right means that input is scanned (e.g., tokens are consumed) from left-to-right, and leftmost derivation means an LL parser can only apply production rules to the leftmost nonterminal. An alternative parsing strategy employed by some bottom-up parsers is ***L**eft-to-right, **R**ightmost derivation* or *LR*. LR parsing still consumes tokens from left-to-right, but applies production rules to the rightmost nonterminal. Figure 7.37 shows how how the derivations and syntax trees for the same input and grammar change based on the choice of LL or LR parsing. Trees generated by LL parsers tend to "lean" left, while those generated by LR parsers "lean" right.

E ⇒	**E + T**	Rule 1
⇒	**E + T** + T	Rule 1
⇒	**T** + T + T	Rule 3
⇒	**F** + T + T	Rule 4
⇒	**a** + T + T	Rule 6
⇒	a + **F** + T	Rule 4
⇒	a + **a** + T	Rule 6
⇒	a + a + **F**	Rule 4
⇒	a + a + **a**	Rule 6



(a) Derivation and parse tree using left-to-right, leftmost derivation (LL)

E ⇒	**E + T**	Rule 1
⇒	E + **F**	Rule 4
⇒	E + **a**	Rule 6
⇒	**E + T** + a	Rule 1
⇒	E + **F** + a	Rule 4
⇒	E + **a** + a	Rule 6
⇒	**T** + a + a	Rule 3
⇒	**F** + a + a	Rule 4
⇒	**a** + a + a	Rule 6



(b) Derivation and parse tree using left-to-right, rightmost derivation (LR)

Figure 7.37 Parsing the input `a + a + a` using leftmost and rightmost derivations methods for the grammar of Figure 7.5

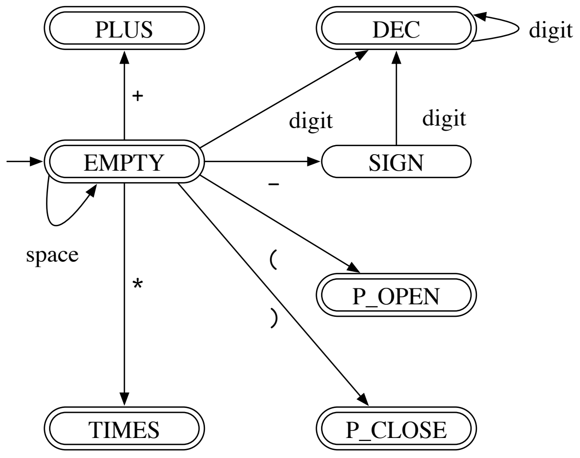
Example 7.11 LL parsers can be affected by infinite recursion through specific patterns of self-referencing rules. In the following derivation using the grammar of Figure 7.5, the LL parser applies the production rule in numeric order using.

E ⇒	**E + T**	Rule 1
⇒	**E + T** + T	Rule 1
⇒	**E + T** + T + T	Rule 1
$\Rightarrow^*$	**E + T** + T + ⋯ + T + T	

The parser continues to expand the leftmost nonterminal—E—indefinitely using Rule 1, producing an infinitely long sentence of nonterminals. Therefore, the grammar of Figure 7.5 cannot be implemented with an LL parser. ■

Grammars containing production rules of the form A ⇒ A⋯ are known as *left-recursive*, while those containing rules of the form A ⇒ ⋯A are *right-recursive*. The techniques for converting a left-recursive grammar to right-recursive grammar can mitigate issues with infinite recursion for LL parsers. However, the details of conversion are beyond the scope of this text.

Figure 7.38 extends the grammar of Figure 7.5 to use tokens as terminals rather than individual characters. The FSM for the lexical analyser is shown in Figure 7.38(a). Figure 7.38b exchanges left recursion for right recursion by reordering the nonterminals of Figure 7.5 to enable LL parsing. The terminal `a` is replaced by a token for signed decimals. The multiple rules for the nonterminals E and T have been combined using by specifying the operator and



(a) A FSM of a multi-token lexical analyzer.

N = { <E>, <T>, <F> }
T = { PLUS, TIMES, P_OPEN, P_CLOSE, DEC }
P = the productions

1. <E> → <T> [PLUS <E>]
2. <T> → <F> [TIMES <T>]
3. <F> → P_OPEN <E> P_CLOSE
4. <F> → DEC

S = <E>

(b) An expanded expression grammar.

Figure 7.38 An expanded expression grammar using a multiple-token lexical analyzer as input to a parser.

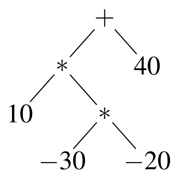
argument as *optional*, using square bracket grammar notation. The rule

<E> → <T> [PLUS <E>]

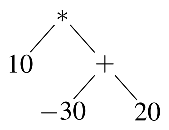
means

<E> produces <T> followed by zero or one occurrence of PLUS <E>.

Example 7.12 Figure 7.39 shows the parse trees corresponding to sample inputs. Despite the fact that Figure 7.38(b) is a right-recursive grammar, the syntax tree of Figure 7.39(a) leans left due to the precedence of operations. Addition has a lower precedence than multiplication, so the chained sequence of multiplications is recognized before the addition of `40`. The subtree for multiplication leans right as expected due to the right-recursion of the grammar. ■



(a) An abstract syntax tree for `10 * -30 * 20 + 40`



(b) An abstract syntax tree for `10 * (-30 + 20)`

Figure 7.39 Abstract syntax trees for runs of Figure 7.38.

Recursive Descent Parsers

The parsers in the remainder of this chapter use the same technique to view the next token without consuming it—often called a *lookahead* token—and conditionally consume the leading token based on its type. Instead of duplicating the token buffering code in each program, a reusable class is stored in a separate file and imported in each program. Figure 7.40 shows the implementation of the token buffer.

The `TokenBuffer`'s constructor takes a lexical analyzer as an argument. It creates an instance variable named `_buffer`, which will contain unconsumed tokens. `TokenBuffer` provides a method `peek()`, which returns the first token in the buffer without removing it, providing a lookeahead. If

Python utils/buffer.py

```python
class ParserBuffer:
    def __init__(self, producer: TokenProducer):
        self._producer = producer
        self._buffer: List = []

    def peek(self):
        if len(self._buffer) == 0:
            try:
                self._buffer.append(next(self._producer))
            except StopIteration:
                return None
        return self._buffer[0]

    def may_match(self, expected_type):
        if (token := self.peek()) and type(token) is expected_type:
            return self._buffer.pop(0)
        return None

    def must_match(self, expected_type):
        if ret := self.may_match(expected_type):
            return ret
        raise SyntaxError()
```

Figure 7.40 Buffering tokens from a lexical analyzer

the buffer is empty, a token will be extracted from the lexical analyzer before being enqueued into the buffer and returned.

The methods `may_match()` and `must_match()` both take a token type as input. Both methods are used to conditionally consume tokens from the buffer. If the first token in the buffer is an instance of the provided type, the token is popped from the buffer and returned. Both methods delegate to `peek()` to ensure that the buffer contains at least one token, In the case of a type mismatch, `may_match()` returns `None`, while `must_match()` raises a `SyntaxError`.

A direct strategy to implement a top-down parsing is called *recursive descent*. Recursive descent parsers contain one function per nonterminal in the grammar. The parsing function for each nonterminal covers all of the production rules which derive from that nonterminal. Each function either consumes terminals, calls other nonterminal parsing functions, or a combination of both. A nonterminal may have a production rule which derives to itself—like rule 1 of Figure 7.38. This leads to recursive function calls between the nonterminal parsing functions. This method generates an implicit syntax tree of return addresses and program counters using the program's stack.

Figure 7.41 implements a recursive descent parser for the grammar of Figure 7.38. Its token classes and multiple-token lexical analyzer mirror Figures 7.33 and 7.34, respectively, and their implementations are provided with the companion source distribution. The parser converts the arithmetic expressions

Python `expr/parser.py`

```python
class ExpressionParser:
    def __init__(self, buffer: io.StringIO):
        self._lexer = Lexer(buffer)
        self._buffer = ParserBuffer(self._lexer)

    # 3. <F> -> ( <E> )
    # 4. <F> -> DEC
    def F(self) -> list[Token]:
        if self._buffer.may_match(tokens.ParenOpen):
            e = self.E()
            self._buffer.must_match(tokens.ParenClose)
            return e
        return [self._buffer.must_match(tokens.Decimal)]

    # 2. <T> -> <F> [* <T>]
    def T(self) -> list[Token]:
        f = self.F()
        if times := self._buffer.may_match(tokens.Times):
            t = self.T()
            return [*f, *t, times]
        return f

    # 1. <E> -> <T> [+ <E>]
    def E(self) -> list[Token]:
        t = self.T()
        if plus := self._buffer.may_match(tokens.Plus):
            e = self.E()
            return [*t, *e, plus]
        return t
```

Figure 7.41 Implementing a parser for the grammar of Figure 7.38

from infix to postfix notation using semantic actions.

Besides the constructor, which takes a lexical analyzer as an argument, `ExpressionParser` has three methods. These methods `F()`, `T()` and `E()` correspond to the three nonterminals of Figure 7.38. Each method returns the list of matched tokens.

Method `F()` first checks if the next token if the buffer is an open parenthesis using `may_match()`. If there is a match, the parser infers that the only rule that could apply is Rule 3 of Figure 7.38 and continue by parsing an E. After parsing the E, the parser expects to see a close parenthesis to complete the rule, and will raise a syntax error via `must_match()` if this is not the case. Parenthesis have no usage in postfix notation, so the return value is the list of tokens of E. If the first token was not an open parenthesis, the parser infers only Rule 4 is applicable, and expects a decimal token, raising a syntax error if this is not the case.

The one production rule implemented by method `T()` always starts by matching an F. The parser then check for the existence of a plus token. When the next token is a plus, the parser assumes that the optional part of Rule 2 is present. The last line of the `if`

```
return [*f, *t, times]
```

unpacks the contents of the token list `f`, concatenated with the unpacked contents of the token list `F`, and concatenated with the times token. If a plus token was not present, the optional part of the rule will be skipped on this invocation. The logic of `E()` matches the structure of `T()` in that it matches a nonterminal before probing for an optional operator and that operators second argument.

Example 7.13 Figure 7.42 shows the parser of Figure 7.41 and it associated recursive call tree. The initial call to `E()` (1) immediately calls `T()` (2) which immediately calls `F()` (3). The next token is `10`, and is matched by `F()` (4..5) and propagated back to the starting `E()` (6..8).

The next token is now `*`, which is matched by the initial `E()` (9), causing the parser to recurse into another call to `E()` (10). The next token is now `(`, and nonterminals are called until it is parsed (11..13), recursing into `E()` again (14). The parser handles the nested expression `-30 + 20` (15..30) while waiting for a close parenthesis (31). The subexpression is propagated with a chain of returns to the original `E()` (32..36), where it is combine with the left hand side of the multiplication before being returned (37). ■

While the grammar of Figure 7.38 has been designed to avoid backtracking, it still attempts many failing matches, which is inefficient. Moreover, the asymptotic performance of recursive descent parsers is poor due when many alternatives branches need to be evaluated, which occurs when production rules for a nonterminal can generate a common prefix. More advanced top-down parsing implementation techniques like *predictive parsing* exist to overcome the limitations of recursive descent to improve performance.

Intermediate Representations and Symbol Tables

As a translator lowers the level of abstraction from the source program towards the object program, it must either produce the object program or a proof that the source program contains one or more syntax errors. While the translator is processing a source program, it discovers facts that are useful for producing the object program, or for producing appropriate error messages. A fact may relate to a single line—for example, whether the line is a monadic instruction, a dyadic instruction, or a pseudo-op. Or, the fact may relate to the program as a whole—for example, whether the program contains two definitions for the same symbol, or whether it uses a symbol that is not defined.

An *intermediate representation* (IR) is the collection of data structures that a translator uses to record these facts. The particular IR of a translator depends on the source language and the object language. Many data structures can be used to store an IR: lists, trees, graphs, to name a few. Pep/10 assembly language is line-oriented rather than block-oriented like C. Therefore, using a line based—or *linear*—intermediate representation fits best.

Input
```
10 * (-30 + 20)
```
Output
```
10 -30 20 + *
```
Call Tree
```
Function                                    Next Token Matched Input

E()                                            10
|-T()                                          10
| |-F()                                        10
| | |-may_match(OpenParen) -> None             10
| | |-must_match(Decimal)  -> 10               *          10
| | |-return [10]                              *          10
| |-may_match(Plus) -> None                    *          10
| |-return [10]                                *          10
|-may_match(Times) -> Times                    (          10 *
|-E()                                          (          10 *
| |-T()                                        (          10 *
| | |-F()                                      (          10 *
| | | |-may_match(OpenParen) -> OpenParen      -30        10 * (
| | | |-E()                                    -30        10 * (
| | | | | T()                                  -30        10 * (
| | | | | | F()                                -30        10 * (
| | | | | | |-may_match(OpenParen) -> None     -30        10 * (
| | | | | | |-must_match(Decimal)  -> -30      +          10 * (-30
| | | | | | |-return [-30]                     +          10 * (-30
| | | | | |-may_match(Plus) -> Plus            20         10 * (-30 +
| | | | | |-T()                                20         10 * (-30 +
| | | | | | |-F()                              20         10 * (-30 +
| | | | | | | |-may_match(OpenParen) -> None   20         10 * (-30 +
| | | | | | | |-must_match(Decimal)  -> 20     )          10 * (-30 + 20
| | | | | | | |-return [20]                    )          10 * (-30 + 20
| | | | | | |-may_match(Plus) -> None          )          10 * (-30 + 20
| | | | | | |-return [20]                      )          10 * (-30 + 20
| | | | | |-return [-30, 20, +]                )          10 * (-30 + 20
| | | | |-may_match(Times) -> None             )          10 * (-30 + 20
| | | | |-return [-30, 20, +]                  )          10 * (-30 + 20
| | | |-must_match(CloseParen) -> CloseParen              10 * (-30 + 20)
| | | |-return [-30, 20, +]                               10 * (-30 + 20)
| | |-may_match(Plus) -> None                             10 * (-30 + 20)
| | |-return [-30, 20, +]                                 10 * (-30 + 20)
| |-may_match(Times) -> None                              10 * (-30 + 20)
| |-return [-30, 20, +]                                   10 * (-30 + 20)
|-return [10, -30, 20, +, *]                              10 * (-30 + 20)
```

Figure 7.42 An input/output pair and call tree of Figure 7.41.

A translator for Pep/10 assembly language records two kinds of facts:

- The declaration and use of symbols
- The structured contents of source lines

A symbol table is a data structure that allows a parser to record symbol names and usage details. Figure 7.43 shows Python code that implements a symbol table for Pep/10 assembly language. The symbol table is a dictionary mapping strings to `SymbolEntry` s. The methods `__contains__()` and `__getitem__()` allow a `SymbolTable` instance to be used like the Python builtin class `dict`.

Each symbol entry records the following:

- A string `name`, which holds the name of the symbol
- An integer `definition_count`, which holds the number of times the symbol is defined
- An integer `value`, which holds an address assigned by the code generator

Method `is_undefined()` returns True if the definition count is zero. Method `is_multiply_defined()` returns True if the definition count is greater than one.

When the parser encounters a SYMBOL token, the parser calls `define()` on its symbol table with the name of the symbol. `define()` ensures the symbol is present in the symbol table by calling `reference()` before incrementing the definition count. Identifiers, on the other hand, are only inserted into the symbol table via `reference()` by the parser when used as arguments. Calls to `reference()` do not increase the definition count.

Symbol definitions insert a name into the symbol table even if that name is not used as an argument. The program of Figure 7.44 defines a symbol `if` but does not use it as an argument. An entry for `if` appears in the symbol table of Figure 7.44.

Unlike C, symbols can be referenced on a line before their definition in Pep/10 assembly language. Figure 7.44 defines `main` on the fourth line, but uses it on the first line. To enable forward branches, the parser must employ use-before-definition logic.

There are two semantic errors that involve symbols—using a symbol without defining it and defining a symbol multiple times. The parser defers checking for these errors to the code generator.

The intermediate representation is the output from the parser and the input to the code generator. Figure 7.45 defines the IR for the translation of Pep/10 assembly language. It uses a `Protocol`, which defines a contract based on method signatures rather than class inheritance. Any class that has the required methods automatically implements the protocol.

Here is the `Protocol` hierarchy.

Python pep10/symbol.py

```python
class SymbolEntry:
    def __init__(self, name: str):
        self.name: str = name
        self.definition_count: int = 0
        self.value: int | None = None

    def is_undefined(self):
        return self.definition_count == 0

    def is_multiply_defined(self):
        return self.definition_count > 1

class SymbolTable:
    def __init__(self) -> None:
        self._table: Dict[str, SymbolEntry] = {}

    def reference(self, name: str) -> SymbolEntry:
        if name not in self._table:
            self._table[name] = SymbolEntry(name)
        return self._table[name]

    def define(self, name: str) -> SymbolEntry:
        (sym := self.reference(name)).definition_count += 1
        return sym

    def __contains__(self, name: str) -> bool:
        return name in self._table

    def __getitem__(self, name: str):
        return self._table[name]
```

Figure 7.43 A symbol entry

- `IRLine`
  Implements `source()`
  Source line example:
  `;******* main()`
- `GeneratesObjectCode`
  Additionally implements `object_code()`, `__len__()`, and has a `memory_address` field
  Source line example:
  `for2:    CPWA    numPts,d    ;j <= numPts`

Section 7.5 describes the implementation of the methods in the IR.

Class `IRLine` is the base type specification for an intermediate representation line. The only requirement placed on every IR line is that it implements a `source()` method that formats the IR line in the canonical source code format of Pep/10. Examples include comment-only lines, error-notification lines, and the `.EQUATE` pseudo-op.

Source program

```
         BR      main
number:  .EQUATE 0            ;local variable #2d
;
main:    SUBSP   2,i          ;push #number
         @DECI   number,s     ;scanf("%d", &number)
if:      LDWA    number,s     ;if (number < 0)
         BRGE    endIf
         LDWA    number,s     ;number = -number
         NEGA
         STWA    number,s
endIf:   @DECO   number,s     ;printf("%d", number)
         ADDSP   2,i          ;pop #number
         RET
```

Symbol table

Symbol name	Symbol value	Definition count
`main`	0003	1
`number`	0000	1
`if`	000C	1
`endif`	0019	1

Figure 7.44 The symbol table for the source program of Figure 6.17.

Class `GeneratesObjectCode` is for those statements that generate object code. These statements implement the method of `IRLine` and additional methods `object_code()`, which generates the binary object code, and `__len__()`, which returns the number of bytes generated. It has a `memory_address` field, which is used for generating the listing. Examples include the dyadic instructions and the pseudo-ops `.WORD`, `.BYTE`, and `.BLOCK`.

Figure 7.45 shows the Pep/10 intermediate representation `Protocol` hierarchy, and the specification of four IR lines. An `EmptyLine` represents a line that contains only spaces and a newline. A `CommentLine` represents a line whose only content is some leading spaces followed by a comment. An `ErrorLine` formats similarly to a `CommentLine`, but emits a description of the error in its comment. Each of these three classes implements `IRLine`, and none generate object code.

Class `DyadicLine` represents a line containing a single dyadic instruction. It implements `GeneratesObjectCode` and generates object code. A dyadic instruction requires a mnemonic, an argument, and an addressing mode, optionally accompanied by a symbol declaration or comment. The symbol dec-

Python `pep10/ir.py`

```python
@runtime_checkable
class IRLine(Protocol):
    def source(self) -> str: ...

@runtime_checkable
class GeneratesObjectCode(IRLine, Protocol):
    memory_address: int | None

    def object_code(self) -> bytearray: ...
    def __len__(self) -> int: ...

@dataclass
class ErrorLine:
    comment: str | None = None

@dataclass
class EmptyLine:

class CommentLine:
    comment: str

@dataclass
class DyadicLine:
    mnemonic: str
    operand_spec: OperandType
    addressing_mode: AddressingMode
    symbol_decl: SymbolEntry | None = None
    comment: str | None = None
    memory_address: int | None = None
```

Figure 7.45 The Pep/10 intermediate representation `Protocol` hierarchy, and the specification of four IR lines.

N = { <argument>, <instruction>, <line>, <statement> }
T = { HEX, DEC, COMMA, EMPTY, IDENT, SYMBOL, COMMENT }
P = the productions
1. <argument> → HEX | DEC | IDENT
2. <instruction> → IDENT <argument> COMMA IDENT
3. <line> → <instruction> [COMMENT]
4. <statement> → [COMMENT | ([SYMBOL] <line>)] EMPTY

S = <statement>

Figure 7.46 A grammar for a subset of Pep/10 assembly language

laration uses the symbol table and `SymbolEntry` introduced in Figure 7.43.

A Recursive Descent Parser for Pep/10

Figure 7.46 is a grammar for a subset of Pep/10 assembly language. As in Figure 7.38, the terminals are tokens rather than characters. This grammar subset includes only dyadic instructions, comment-only lines, and empty lines. It excludes pseudo-ops and monadic instructions, which are problems for the student at the end of this chapter.

Parsing begins with <statement>, and can match the following constructs to yield IR instances.

COMMENT EMPTY
A line containing only a comment.

[SYMBOL] <line> EMPTY
A line containing a dyadic instruction.

EMPTY
A line containing only spaces and a newline.

Nonterminal <line> associates a comment with an <instruction>. Nonterminal <instruction> matches the various portions of a dyadic instruction. Nonterminal <argument> matches the various possibilities of the operand specifier. Upon completing a line by matching EMPTY and returning an IR object, the parser begins parsing the next line.

Figure 7.46 does not have any recursive rules, which simplifies the call trees of a recursive descent parser. Figure 7.42, the call tree of a recursive descent parser for a grammar with recursive rules, shows several instances of backtracking. With one token of lookahead, the next production rule for this grammar can always be selected correctly without backtracking.

Figure 7.47 shows the constructor for `Parser`, which takes a text buffer and symbol table as an argument. An instance of the multi-token lexical analyzer of Figure 7.34 is constructed from the buffer, and a token buffer of Figure 7.40 is constructed from the lexical analyzer. The symbol table parameter is assigned to the instance variable named `symbol_table` for future use.

Methods `__iter__()` and `__next__()` implement Python's iteration protocol. Method `__next__()` returns the intermediate representation for the next line of code by calling the method `statement()`. If no tokens remain in the buffer, `StopIteration` is raised to prevent further attempts at iteration. The remaining lines of `__next__()` convert the `SyntaxErrors` raised by `must_match()` on the token buffer to an `ErrorLine` IR instance.

Figure 7.48 shows the implementation of <statement> via recursive descent and how semantic actions return an intermediate representation line. If a symbol declaration is matched, the parser's symbol table increments the definition count of the symbol. Additional checks ensure that a symbol is always followed by an instruction. If not, statement raises a `SyntaxError`, which is converted into an error IR line by `__next__()`.

Python pep10/parser.py

```python
class Parser:
    def __init__(
        self, buffer: io.StringIO, symbol_table: SymbolTable | None = None
    ):
        self.lexer = Lexer(buffer)
        self._buffer = ParserBuffer(self.lexer)
        self.symbol_table = symbol_table if symbol_table else SymbolTable()

    def __iter__(self):
        return self

    def __next__(self) -> IRLine:
        if self._buffer.peek() is None:
            raise StopIteration()
        try:
            return self.statement()
        except SyntaxError as s:
            self._buffer.skip_to_next_line({tokens.Empty})
            return ErrorLine(comment=s.msg if s.msg else None)
        except KeyError:
            self._buffer.skip_to_next_line({tokens.Empty})
            return ErrorLine()

def parse(text: str, symbol_table: SymbolTable | None = None) -> List[IRLine]:
    # Ensure input is terminated with a single \n.
    parser = Parser(io.StringIO(text.rstrip() + "\n"), symbol_table)
    return [item for item in parser]
```

Figure 7.47 Pep/10 parser initialization code.

When the parser does not match a symbol, it proceeds to match <line>. <statement>s that fail all these checks raise a `SyntaxError` due to not matching production rules of Figure 7.46. Lastly, the parser verifies that the current line is terminated with a newline (that is, the EMPTY token) before returning the IR line.

Figure 7.48 also shows the parser code for nonterminal <line>. Method `line()` always attempts to match an <instruction>. The method returns `None` rather than raise a syntax error to avoid hiding <statement>'s syntax error about symbol declarations. If a COMMENT is matched, it is attached to the IR.

Figure 7.49 implements the nonterminal <instruction>. The semantic actions of `instruction()` are more complicated than the grammar would suggest. While <instruction> accepts any IDENT, the parser must limit itself to actual mnemonics of the Pep/10 instruction set. After parsing an <argument>, the parser must ensure that the argument is representable as a 16-bit integer. The FSM of Figure 7.30 places no limit on the values of decimal or hexadecimal constants, but dyadic Pep/10 instructions only have a 16-bit operand specifier.

The parser recognizes an addressing mode in two stages

Python pep10/parser.py

```python
    # statement ::= [COMMENT  | ([SYMBOL] line)] EMPTY
    def statement(self) -> IRLine:
        return_ir: IRLine | None = None
        if self._buffer.may_match(tokens.Empty):
            return EmptyLine()
        elif comment_token := self._buffer.may_match(tokens.Comment):
            return_ir = CommentLine(comment_token.value)
        elif symbol_token := self._buffer.may_match(tokens.Symbol):
            symbol_entry = self.symbol_table.define(symbol_token.value)
            if not (return_ir := self.line(symbol_entry)):
                message = "Symbol declaration must be followed by instruction"
                raise SyntaxError(message)
        elif not (return_ir := self.line(None)):
            raise SyntaxError("Failed to parse line")

        self._buffer.must_match(tokens.Empty)
        return return_ir
```

```python
    # line ::= instruction [COMMENT]
    def line(self, symbol_entry: SymbolEntry | None) -> DyadicLine | None:
        return_ir = self.instruction(symbol_entry)
        if not return_ir:
            return None
        elif comment := self._buffer.may_match(tokens.Comment):
            return_ir.comment = comment.value
        return return_ir
```

Figure 7.48 Pep/10 parser code for <statement> and <line>.

- Scan a COMMA token
- Scan an IDENT token

If either step fails to recognize the desired token type, then the current line must not be a valid <instruction>, and a syntax error is raised via `must_match()`. Just as the parser limits the acceptable IDENTs for mnemonics, the parser must limit the acceptable IDENTs for addressing modes. A syntax error is raised if the addressing mode is not allowed for the mnemonic or if the IDENT does not represent an addressing mode. A `DyadicLine` IR instance is constructed from the parsed components.

Figure 7.49 also shows the parser code for nonterminal <argument>. One by one, the parser matches each of the token types. The classes `Hexadecimal`, `Decimal`, and `Identifier` are intermediate representations used to format operand specifiers as 16-bit integer values. Matching the `Hexadecimal` and `Decimal` tokens wraps the token's value in the correct intermediate representation class.

Matching an `Identifier` requires extra handling compared to the other token types. Because the parser is processing the current identifier as an argument, it inserts a reference to that symbol into the symbol table. Inserting the

Python pep10/parser.py

```python
    # instruction ::= IDENT argument COMMA IDENT
    def instruction(
        self, symbol_entry: SymbolEntry | None
    ) -> DyadicLine | None:
        if not (mn_token := self._buffer.may_match(tokens.Identifier)):
            return None
        mn_str = mn_token.value.upper()
        if mn_str not in INSTRUCTION_TYPES:
            raise SyntaxError(f"Unrecognized mnemonic: {mn_str}")
        elif not (arg_ir := self.argument()):
            raise SyntaxError(f"Missing argument")

        try:
            arg_int = int(arg_ir)
            arg_int.to_bytes(2, signed=arg_int < 0)
        except OverflowError:
            raise SyntaxError("Number too large")

        self._buffer.must_match(tokens.Comma)
        addr_str = self._buffer.must_match(tokens.Identifier).value.upper()
        try:
            # Check if addressing mode is valid for this mnemonic
            addr_mode = cast(AddressingMode, AddressingMode[addr_str])
            mn_type = INSTRUCTION_TYPES[mn_str]
            if not mn_type.allows_addressing_mode(addr_mode):
                err = f"Invalid addressing mode {addr_str} for {mn_str}"
                raise SyntaxError(err)
        except KeyError:
            raise SyntaxError(f"Unknown addressing mode: {addr_str}")
        return DyadicLine(mn_str, arg_ir, addr_mode, symbol_decl=symbol_entry)
```

```python
    # argument ::= HEX | DEC | IDENT
    def argument(self) -> OperandType | None:
        if hex_token := self._buffer.may_match(tokens.Hexadecimal):
            return operands.Hexadecimal(hex_token.value)
        elif dec_token := self._buffer.may_match(tokens.Decimal):
            return operands.Decimal(dec_token.value)
        elif ident_token := self._buffer.may_match(tokens.Identifier):
            symbol_entry = self.symbol_table.reference(ident_token.value)
            return operands.Identifier(symbol_entry)
        return None
```

Figure 7.49 Pep/10 parser code for <instruction> and <argument>.

references enables the code generator to detect undefined symbols or multiply defined symbols during code generation.

Example 7.14 Figure 7.50 shows a program that will be used as an input to the parser and the corresponding intermediate representation. Executing the assembled source code on the Pep/10 virtual machine would output `hi` to `charOut` before shutting down. It uses only tokens from Figure 7.30, eschew-

Pep/10 Source Code

```
start:   BR      main
; SUBX,i is ASCII 'h'
h:       SUBX    0xFEED,i

main:    LDBA    h,d
         STBA    charOut,d
         ADDA    1,i          ;Increment accumulator to convert h to i
       StBa  charOut  ,  D
         STBA    pwrOff,d
```

Intermediate Representation

```
DyadicLine('start:BR main,i')
CommentLine(' SUBX,i is ASCII \'h\'')
DyadicLine('h:SUBX 0xfeed,i')
EmptyLine()
DyadicLine('main:LDBA h,d')
DyadicLine('STBA charOut,d')
DyadicLine('ADDA 1,i',comment='Increment accumulator to convert h to i')
DyadicLine('STBA charOut,d')
DyadicLine('STBA pwrOff,d')
```

Figure 7.50 A Pep/10 source code program and its intermediate representation

ing dot commands for data or character constants.

A program creates an instance of the parser is created with the input text. The program over the parser instance. On each iteration, the parser calls `statement()` which returns an IR instance. Those IR lines are accumulated in a list, which is formatted and written to the program's standard output. ■

A language translator should detect as many errors as possible on each run and should not abort on the first error. When the lexical analyzer detects an error on a line it returns an error token. When the parser detects a syntax error—whether because of an invalid token from the lexical analyzer or a grammatical error—it returns an intermediate representation `ErrorLine` and discards tokens from the lexical analyzer until an EMPTY token is scanned.

Figure 7.51 shows that the parser is capable of recognizing a large number of errors on a single run. It provides sufficient information to the programmer to modify their program to avoid the mistakes on future assembly attempts. However, this error recovery mechanism is incapable of detecting multiple errors on a single line.

Example 7.15 The source code line

Pep/10 Source Code

```
start: ;BR   main,i
       LDWY  0x10000,i
       ADDA  0x10000,i
       SUBSP 15,y
       STWA  0,i
```

Intermediate Representation

```
ErrorLine(';ERROR: Symbol declaration must be followed by instruction')
ErrorLine(';ERROR: Unrecognized mnemonic: LDWY')
ErrorLine(';ERROR: Number too large')
ErrorLine(';ERROR: Unknown addressing mode: Y')
ErrorLine(';ERROR: Invalid addressing mode I for STWA')
```

Figure 7.51 A Pep/10 source code program demonstrating error recovery in the parser

```
 LDWY 0x10000,y
```

in Figure 7.51 contains three errors—an invalid mnemonic, an argument incapable of being represented with 16-bits, and an unknown addressing mode. The lexical analyzer does not detect any of these three syntax errors. Once the parser detects the first error—an invalid mnemonic—it discards the remaining tokens on the line, missing the second two errors. ■

7.5 Constructing a Code Generator

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need