

## Chapter 4

---

---

# Top Down Parsing

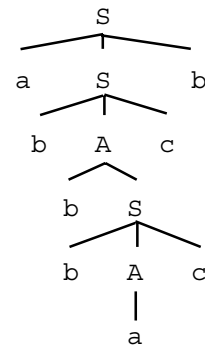
The parsing problem was defined in Section 3.2 as follows: given a grammar and an input string, determine whether the string is in the language of the grammar, and, if so, determine its structure. Parsing algorithms are usually classified as either *top down* or *bottom up*, which refers to the sequence in which a derivation tree is built or traversed; in this chapter we consider only top down algorithms.

In a top down parsing algorithm, grammar rules are applied in a sequence which corresponds to a general top down direction in the derivation tree. For example, consider the grammar G8:

G8:

1.  $S \rightarrow a S b$
2.  $S \rightarrow b A c$
3.  $A \rightarrow b S$
4.  $A \rightarrow a$

We show a derivation tree for the input string  $abbbaccb$  in Figure 4.1. A parsing algorithm will read one input symbol at a time and try to decide, using the grammar, whether the input string can be derived. A top down algorithm will begin with the starting nonterminal and try to decide which rule of the grammar should be applied. In the example of Figure 4.1, the algorithm is able to make this decision by examining a single input symbol and comparing it with the first symbol on the right side of the rules. Figure 4.2 shows the sequence of events, as input symbols are read, in which the numbers in circles indicate which grammar rules are being applied, and the underscored symbols are the ones



**Figure 4.1** A Derivation Tree for  $abbbaccb$

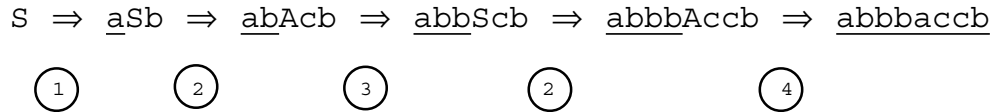


Figure 4.2 Sequence of Events in a Top Down Parse

which have been read by the parser. Careful study of Figures 4.1 and 4.2, above, reveals that this sequence of events corresponds to a top down construction of the derivation tree.

In this chapter, we describe some top down parsing algorithms and, in addition, we show how they can be used to generate output in the form of atoms or syntax trees. This is known as *syntax directed translation*. However, we need to begin by describing the subclass of context-free grammars which can be parsed top down. In order to do this we begin with some preliminary definitions from discrete mathematics.

### 4.0 Relations and Closure

Whether working with top down or bottom up parsing algorithms, we will always be looking for ways to automate the process of producing a parser from the grammar for the source language. This will require, in most cases, the use of mathematics involving sets and relations. A **relation** is a set of ordered pairs. Each pair may be listed in parentheses and separated by commas, as in the following example:

R1

- (a, b)
- (c, d)
- (b, a)
- (b, c)
- (c, c)

Note that (a, b) and (b, a) are not the same. Sometimes the name of a relation is used to list the elements of the relation:

- 4 < 9
- 5 < 22
- 2 < 3
- 3 < 0

If R is a relation, then the **reflexive transitive closure** of R is designated R\*; it is a relation made up of the same elements of R with the following properties:

1. All pairs of R are also in R\* .
2. If (a, b) and (b, c) are in R\* , then (a, c) is in R\* (Transitive).

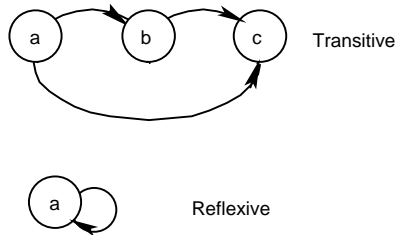


Figure 4.3 Reflexive and Transitive Elements of a Relation

3. If  $a$  is in one of the pairs of  $R$ , then  $(a, a)$  is in  $R^*$  (Reflexive).

A diagram using an arrow to represent the relation can be used to show what we mean by *transitive* and *reflexive* properties and is shown, above, in Figure 4.3. In rule 2 for transitivity note that we are searching the pairs of  $R^*$ , not  $R$ . This means that as additional pairs are added to the closure for transitivity, those new pairs must also be checked to see whether they generate new pairs.

#### Sample Problem 4.0

Show  $R1^*$  the reflexive transitive closure of  $R1$ .

#### Solution:

$R1^*$ :

$(a, b)$   
 $(c, d)$   
 $(b, a)$  (from  $R1$ )  
 $(b, c)$   
 $(c, c)$   
  
 $(a, c)$   
 $(b, d)$  (transitive)  
 $(a, d)$   
  
 $(a, a)$   
 $(b, b)$  (reflexive)  
 $(d, d)$

Note in Sample Problem 4.0 that we computed the transitive entries before the reflexive entries. The pairs can be listed in any order, but reflexive entries can never be used to derive new transitive pairs, consequently the reflexive pairs were listed last.

### Exercises 4.0

1. Show the *reflexive transitive closure* of each of the following relations:
 

(a)	(a, b)	(b)	(a, a)	(c)	(a, b)
	(a, d)		(a, b)		(c, d)
	(b, c)		(b, b)		(b, c)
					(d, a)
  
2. The mathematical relation “less than” is denoted by the symbol  $<$ . Some of the elements of this relation are:  $(4, 5)$   $(0, 16)$   $(-4, 1)$   $(1.001, 1.002)$ . What do we normally call the relation which is the reflexive transitive closure of “less than”?
  
3. Write a program in Java or C++ to read in from the keyboard, ordered pairs (of strings, with a maximum of eight characters per string) representing a relation, and print out the *reflexive transitive closure* of that relation in the form of ordered pairs. You may assume that there will be, at most, 100 ordered pairs in the given relation, involving, at most, 100 different symbols. (Hint: Form a *boolean matrix* which indicates whether each symbol is related to each symbol).

### 4.1 Simple Grammars

At this point, we wish to show how top down parsers can be constructed for a given grammar. We begin by restricting the form of context-free grammar rules so that it will be very easy to construct a parser for the grammar. These grammars will not be very useful, but will serve as an appropriate introduction to top down parsing.

A grammar is a *simple grammar* if every rule is of the form:

$$A \rightarrow a\alpha$$

(where A represents any nonterminal, a represents any terminal, and  $\alpha$  represents any string of terminals and nonterminals), and every pair of rules defining the same nonterminal begin with different terminals on the right side of the arrow. For example, the grammar G9 below is simple, but the grammar G10 is not simple because it contains an epsilon rule and the grammar G11 is not simple because two rules defining S begin with the same terminal.

G9:

$$\begin{aligned} S &\rightarrow aSb \\ S &\rightarrow b \end{aligned}$$

G10:

$$\begin{aligned} S &\rightarrow aSb \\ S &\rightarrow \epsilon \end{aligned}$$

G11:

$$\begin{aligned} S &\rightarrow aSb \\ S &\rightarrow a \end{aligned}$$

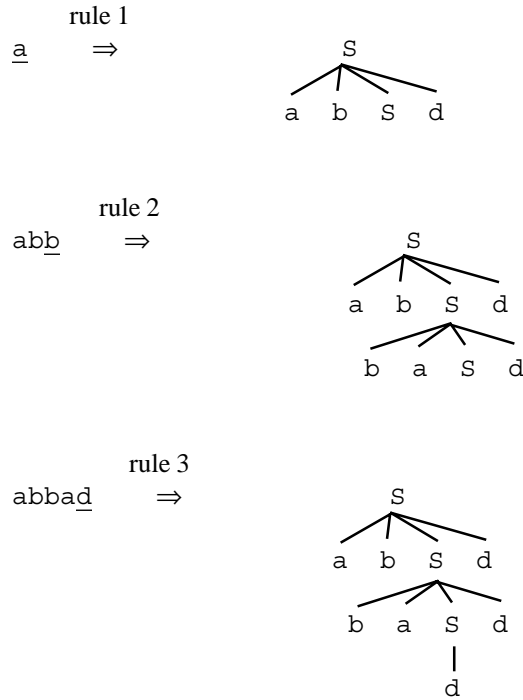
A language which can be specified by a simple grammar is said to be a *simple language*. Simple languages will not be very useful in compiler design, but they serve as a good way of introducing the class of languages which can be parsed top down. The parsing algorithm must decide which rule of the grammar is to be applied as a string is parsed. The set of input symbols (i.e. terminal symbols) which imply the application of a grammar rule is called the *selection set* of that rule. For simple grammars, the selection set of each rule always contains exactly one terminal symbol – the one beginning the right hand side. In grammar G9, the selection set of the first rule is {a} and the selection set of the second rule is {b}. In top down parsing in general, rules defining the same nonterminal must have *disjoint* (non-intersecting) selection sets, so that it is always possible to decide which grammar rule is to be applied.

For example, consider grammar G12 below:

G12:

$$\begin{aligned} 1. \quad S &\rightarrow a \underline{b} S d \\ 2. \quad S &\rightarrow \underline{b} a S d \\ 3. \quad S &\rightarrow d \end{aligned}$$

Figure 4.4 illustrates the construction of a derivation tree for the input string abbadd, using grammar G12. The parser must decide which of the three rules to apply as input symbols are read. In Figure 4.4 the underscored input symbol is the one which determines



**Figure 4.4** Using the Input Symbol to Guide the Parsing of the String *abbaddd*

which of the three rules is to be applied, and is thus used to guide the parser as it attempts to build a derivation tree. The input symbols which direct the parser to use a particular rule are the members of the selection set for that rule. In the case of simple grammars, there is exactly one symbol in the selection set for each rule, but for other context-free grammars, there could be several input symbols in the selection set.

### 4.1.1 Parsing Simple Languages with Pushdown Machines

In this section, we show that it is always possible to construct a one-state pushdown machine to parse the language of a simple grammar. Moreover, the construction of the machine follows directly from the grammar; i.e., it can be done mechanically. Consider the simple grammar *G*<sub>13</sub> below:

*G*<sub>13</sub>:

1.  $S \rightarrow aSB$
2.  $S \rightarrow b$
3.  $B \rightarrow a$
4.  $B \rightarrow bBa$

We now wish to construct an extended pushdown machine to parse input strings consisting of a's and b's, according to the rules of this grammar. The strategy we use is to begin with a stack containing a bottom marker ( $\nabla$ ) and the starting nonterminal, S. As the input string is being parsed, the stack will always correspond to the portion of the input string which has not been read. As each input symbol is read, the machine will attempt to apply one of the four rules in the grammar. If the top stack symbol is S, the machine will apply either rule 1 or 2 (since they define an S); whereas if the top stack symbol is B, the machine will apply either rule 3 or rule 4 (since they define a B). The current input symbol is used to determine which of the two rules to apply by comparing it with the selection sets (this is why we impose the restriction that rules defining the same nonterminal have disjoint selection sets).

Once we have decided which rule is to be entered in each cell of the pushdown machine table, it is applied by replacing the top stack symbol with the symbols on the right side of the rule in reverse order, and retaining the input pointer. This means that we will be pushing terminal symbols onto the stack in addition to nonterminals. When the top stack symbol is a terminal, all we need to do is ascertain that the current input symbol matches that stack symbol. If this is the case, simply pop the stack and advance the input pointer. Otherwise, the input string is to be rejected. When the end of the input string is encountered, the stack should be empty (except for  $\nabla$ ) in order for the string to be accepted. The extended pushdown machine for grammar G13 is shown in Figure 4.5, below. The sequence of stack configurations produced by this machine for the input aba is shown in Figure 4.6.

In general, given a simple grammar, we can always construct a one state extended pushdown machine which will parse any input string. The construction of the machine could be done automatically:

1. Build a table with each column labeled by a terminal symbol (and endmarker  $\leftarrow$ ) and each row labeled by a nonterminal or terminal symbol (and bottom marker  $\nabla$ ).

	a	b	$\leftarrow$
S	Rep (Bsa) Retain	Rep (b) Retain	Reject
B	Rep (a) Retain	Rep (aBb) Retain	Reject
a	pop Advance	Reject	Reject
b	Reject	pop Advance	Reject
$\nabla$	Reject	Reject	Accept

S
$\nabla$

Initial  
Stack

Figure 4.5 Pushdown Machine for Grammar G13

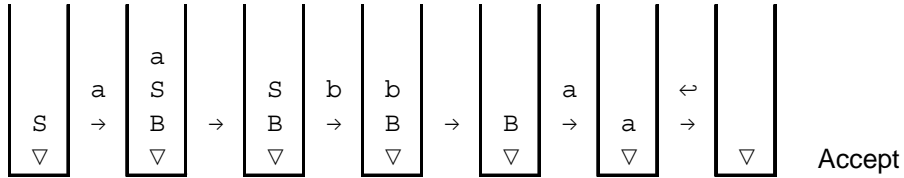


Figure 4.6 Sequence of Stacks for Machine of Figure 4.5 for Input aba

2. For each grammar rule of the form  $A \rightarrow a\alpha$ , fill in the cell in row A and column a with:  $REP(\alpha^r a)$ , retain, where  $\alpha^r$  represents  $\alpha$  reversed (here, a represents a terminal, and  $\alpha$  represents a string of terminals and nonterminals).
3. Fill in the cell in row a and column a with pop, advance, for each terminal symbol a.
4. Fill in the cell in row  $\nabla$  and column  $\leftrightarrow$  with Accept.
5. Fill in all other cells with Reject.
6. Initialize the stack with  $\nabla$  and the starting nonterminal..

This means that we could write a program which would accept, as input, any simple grammar and produce, as output, a pushdown machine which will accept any string in the language of that grammar and reject any string not in the language of that grammar. There is software which is able to do this for a grammar for a high level programming language – i.e., which is able to generate a parser for a compiler. Software which generates a compiler automatically from specifications of a programming language is called a **compiler-compiler**. We will study a popular compiler-compiler called *SableCC* in the section on bottom up parsing.

### 4.1.2 Recursive Descent Parsers for Simple Grammars

A second way of implementing a parser for simple grammars is to use a methodology known as **recursive descent**, in which the parser is written using a traditional programming language, such as Java or C++. A method is written for each nonterminal in the grammar. The purpose of this method is to scan a portion of the input string until an **example** of that nonterminal has been read. By an example of a nonterminal, we mean a string of terminals or input symbols which can be derived from that nonterminal. This is done by using the first terminal symbol in each rule to decide which rule to apply. The method then handles each succeeding symbol in the rule; it handles nonterminals by calling the corresponding methods, and it handles terminals by reading another input symbol. For example, a recursive descent parser for grammar G13 is shown below:

```

class RDP                                // Recursive Descent Parser
{
char inp;

public static void main (String[] args) throws IOException
{ InputStreamReader stdin = new InputStreamReader
                                (System.in);

    RDP rdp = new RDP();
    rdp.parse();
}

void parse ()
{ inp = getInp();
  S ();                                // Call start nonterminal
  if (inp=='\n') accept();             // end of string marker
  else reject();
}

void S ()
{ if (inp=='a')                        // apply rule 1
  { inp = getInp();
    S ();
    B ();
  }
  // end rule 1
  else if (inp=='b') inp = getInp();    // apply rule 2
  else reject();
}

void B ()
{ if (inp=='a') inp = getInp();         // rule 3
  else if (inp=='b')                    // apply rule 4
  { inp = getInp();
    B();
    if (inp=='a') inp = getInp();
    else reject();
  }
  // end rule 4
  else reject();
}

void accept()                            // Accept the input
{ System.out.println ("accept"); }

void reject()                             // Reject the input

```

```

{ System.out.println ("reject");
  System.exit(0);          // terminate parser
}

char getInp()
{ try
  { return (char) System.in.read(); }
  catch (IOException ioe)
  { System.out.println ("IO error " + ioe); }
  return '#';              // must return a char
}
}

```

Note that the main method (`parse`) reads the first input character before calling the method for nonterminal  $S$  (the starting nonterminal). Each method assumes that the initial input symbol in the example it is looking for has been read before that method has been called. It then uses the selection set to determine which of the grammar rules defining that nonterminal should be applied. The method  $S$  calls itself (because the nonterminal  $S$  is defined in terms of itself), hence the name *recursive descent*. When control is returned to the `parse` method, it must ensure that the entire input string has been read before accepting it. The methods `accept()` and `reject()`, which have been omitted from the above program, simply indicate whether or not the input string is in the language of the grammar. The method `getInp()` is used to obtain one character from the standard input file (keyboard). In subsequent examples, we omit the `main`, `accept`, `reject`, and `getInp` methods to focus attention on the concepts of recursive descent parsing. The student should perform careful hand simulations of this program for various input strings, to understand it fully.

#### Sample Problem 4.1

Show a one state pushdown machine and a recursive descent parser (show methods  $S()$  and  $A()$  only) for the following grammar:

1.  $S \rightarrow 0 S 1 A$
2.  $S \rightarrow 1 0 A$
3.  $A \rightarrow 0 S 0$
4.  $A \rightarrow 1$

**Solution:**

This grammar is simple because all rules begin with a terminal – rules 1 and 2 which define  $S$ , begin with different terminals, and rules 3 and 4 which define  $A$ , begin with different terminals. The pushdown machine is shown below:

	0	1	$\leftarrow$
$S$	Rep (A1S0) Retain	Rep (A01) Retain	Reject
$A$	Rep (0S0) Retain	Rep (1) Retain	Reject
0	pop Advance	Reject	Reject
1	Reject	pop Advance	Reject
$\nabla$	Reject	Reject	Accept

$S$
$\nabla$

Initial Stack

The recursive descent parser is shown below:

```

void S()
{
    if (inp=='0') // apply rule 1
    {
        inp = getInp();
        S ();
        if (inp=='1') getInp();
        else reject;
        A ();
    } // end rule 1
    else if (inp=='1') // apply rule 2
    {
        getInp();
        if (inp=='0') getInp();
        else reject();
        A ();
    } // end rule 2
    else reject();
}

void A ()
{
    if (inp=='0') // apply rule 3

```

```

    {   getInp();
        S ();
        if (inp=='0') getInp();
        else reject();
    }
else if (inp=='1') getInp(); // end rule 3
else reject(); // apply rule 4
}

```

### Exercises 4.1

1. Determine which of the following grammars are *simple*. For those which are simple, show an *extended one-state pushdown machine* to accept the language of that grammar.
  - (a)
    1.  $S \rightarrow a S b$
    2.  $S \rightarrow b$
  - (b)
    1.  $\text{Expr} \rightarrow \text{Expr} + \text{Term}$
    2.  $\text{Expr} \rightarrow \text{Term}$
    3.  $\text{Term} \rightarrow \text{var}$
    4.  $\text{Term} \rightarrow ( \text{Expr} )$
  - (c)
    1.  $S \rightarrow a A b B$
    2.  $A \rightarrow b A$
    3.  $A \rightarrow a$
    4.  $B \rightarrow b A$
  - (d)
    1.  $S \rightarrow a A b B$
    2.  $A \rightarrow b A$
    3.  $A \rightarrow b$
    4.  $B \rightarrow b A$
  - (e)
    1.  $S \rightarrow a A b B$
    2.  $A \rightarrow b A$
    3.  $A \rightarrow \varepsilon$
    4.  $B \rightarrow b A$

2. Show the *sequence of stacks* for the pushdown machine of Figure 4.5 for each of the following input strings:
  - (a)  $aba \leftarrow$
  - (b)  $abbaa \leftarrow$
  - (c)  $aababaa \leftarrow$
3. Show a *recursive descent parser* for each simple grammar of Problem 1, above.

## 4.2 Quasi-Simple Grammars

We now extend the class of grammars which can be parsed top down by permitting  $\epsilon$  rules in the grammar. A *quasi-simple grammar* is a grammar which obeys the restriction of simple grammars, but which may also contain rules of the form:

$$N \rightarrow \epsilon$$

(where  $N$  represents any nonterminal) as long as all rules defining the same nonterminal have disjoint selection sets.

For example, the following is a quasi-simple grammar:

G14:

1.  $S \rightarrow a A S$
2.  $S \rightarrow b$
3.  $A \rightarrow c A S$
4.  $A \rightarrow \epsilon$

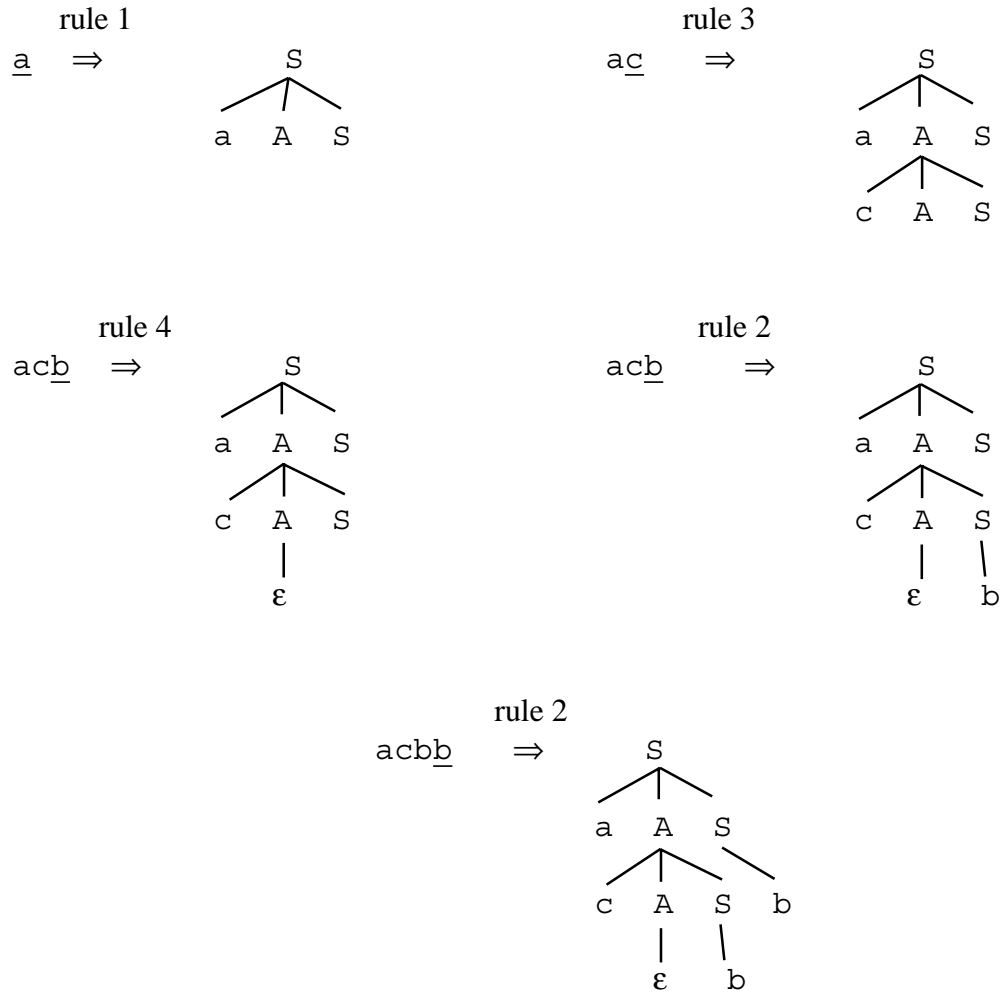
In order to do a top down parse for this grammar, we will again have to find the selection set for each rule. In order to find the selection set for  $\epsilon$  rules (such as rule 4) we first need to define some terms. The *follow set* of a nonterminal  $A$ , designated  $Fol(A)$ , is the set of all terminals (or endmarker  $\epsilon$ ) which can immediately follow an  $A$  in an intermediate form derived from  $S \leftarrow$ , where  $S$  is the starting nonterminal. For grammar G14, above, the follow set of  $S$  is  $\{a, b, \epsilon\}$  and the follow set of  $A$  is  $\{a, b\}$ , as shown by the following derivations:

$$\begin{aligned} S \leftarrow &\Rightarrow aAS \leftarrow \Rightarrow acASS \leftarrow \Rightarrow acASaAS \leftarrow \\ &\Rightarrow acASb \leftarrow \quad Fol(S) = \{a, b, \epsilon\} \end{aligned}$$

$$\begin{aligned} S \leftarrow &\Rightarrow aAS \leftarrow \Rightarrow aAaAS \leftarrow \\ &\Rightarrow aAb \leftarrow \quad Fol(A) = \{a, b\} \end{aligned}$$

For the present, we rely on the student's ingenuity to find all elements of the follow set. In a later section, we will present an algorithm for finding follow sets. The selection set for an  $\epsilon$  rule is simply the follow set of the nonterminal on the left side of the arrow. For example, in grammar G14, above, the selection set of rule 4 is  $Se1(4) = Fol(A) = \{a, b\}$ . We use the follow set because these are the terminals which could be the current input symbol when, for instance, an example of an  $A$  in recursive descent is being sought.

To understand selection sets for quasi-simple grammars, consider the case where the parser for grammar G14 is attempting to build a derivation tree for the input string  $acbb$ . Once again, it is the selection set of each rule that guides the parser to apply that rule, as illustrated in Figure 4.7. If the parser is trying to decide how to rewrite an  $A$ , it will



**Figure 4.7** Construction of a Parse Tree for  $acbb$  Using Selection Sets

choose rule 3 if the input symbol is a  $c$ , but it will choose rule 4 if the input symbol is either an  $a$  or a  $b$ .

#### 4.2.1 Pushdown Machines for Quasi-Simple Grammars

To build a pushdown machine for a quasi-simple grammar, we need to add only one step to the algorithm given in Section 4.1.1. We need to apply an  $\epsilon$  rule by simply popping the nonterminal off the stack and retaining the input pointer. We do this only when the input symbol is in the follow set of the nonterminal defined in the  $\epsilon$  rule. We would add the following step to the algorithm between steps 4 and 5:

4.5 For each  $\epsilon$  rule in the grammar, fill in cells of the row corresponding to the nonterminal on the left side of the arrow, but only in those columns corresponding to elements of the follow set of the nonterminal. Fill in these cells with Pop, Retain.

This will cause the machine to apply an  $\epsilon$  rule by popping the nonterminal off the stack without reading any input symbols.

For example, the pushdown machine for grammar G14 is shown in Figure 4.8, above. Note, in particular, that the entries in columns a and b for row A (Pop, Retain) correspond to the  $\epsilon$  rule (rule 4).

	a	b	c	$\leftarrow$
S	Rep (Saa) Retain	Rep (b) Retain	Reject	Reject
A	Pop Retain	Pop Retain	Rep (Sac) Retain	Reject
a	Pop Advance	Reject	Reject	Reject
b	Reject	Pop Advance	Reject	Reject
c	Reject	Reject	Pop Advance	
$\nabla$	Reject	Reject	Reject	Accept

S
$\nabla$

Initial Stack

Figure 4.8 A Pushdown Machine for Grammar G14

### 4.2.2 Recursive Descent for Quasi-Simple Grammars

Recursive descent parsers for quasi-simple grammars are similar to those for simple grammars. The only difference is that we need to check for all the input symbols in the selection set of an  $\epsilon$  rule. If any of these are the current input symbol, we simply return to the calling method without reading any input. By doing so, we are indicating that  $\epsilon$  is an example of the nonterminal for which we are trying to find an example. A recursive descent parser for grammar G14 is shown below:

```

char inp;
void parse ()
{   inp = getInp();
    S ();
    if (inp==' $\leftarrow$ ') accept();
    else reject();
}
    
```

```

void S ()
{   if (inp=='a')                               // apply rule 1
    {   inp = getInp();
        A();
        S();
    }
    else if (inp=='b') inp = getInp(); // apply rule 2
    else reject();
}

void A ()
{   if (inp=='c')                               // apply rule 3
    {   inp = getInp();
        A ();
        S ();
    }
    else if (inp=='a' || inp=='b') ; // apply rule 4
    else reject();
}

```

Note that rule 4 is applied in method `A ()` when the input symbol is a or b. Rule 4 is applied by returning to the calling method without reading any input characters. This is done by making use of the fact that Java permits null statements (at the comment `// apply rule 4`). It is not surprising that a null statement is used to process the null string.

### 4.2.3 A Final Remark on $\epsilon$ Rules

It is not strictly necessary to compute the selection set for  $\epsilon$  rules in quasi-simple grammars. In other words, there is no need to distinguish between `Reject` entries and `Pop`, `Retain` entries in the row of the pushdown machine for an  $\epsilon$  rule; they can all be marked `Pop`, `Retain`. If the machine does a `Pop`, `Retain` when it should `Reject` (i.e., it applies an  $\epsilon$  rule when it really has no rule to apply), the syntax error will always be detected subsequently in the parse. However, this is often undesirable in compilers, because it is generally a good idea to detect syntax errors as soon as possible so that a meaningful error message can be put out.

For example, in the pushdown machine of Figure 4.8, for the row labeled A, we have filled in `Pop`, `Retain` under the input symbols a and b, but `Reject` under the input symbol  $\epsilon$ ; the reason is that the selection set for the  $\epsilon$  rule is  $\{a, b\}$ . If we had not computed the selection set, we could have filled in all three of these cells with `Pop`, `Retain`, and the machine would have produced the same end result for any input.

**Sample Problem 4.2**

Find the selection sets for the following grammar. Is the grammar quasi-simple? If so, show a pushdown machine and a recursive descent parser (show methods  $S()$  and  $A()$  only) corresponding to this grammar.

1.  $S \rightarrow b A b$
2.  $S \rightarrow a$
3.  $A \rightarrow \epsilon$
4.  $A \rightarrow a S a$

**Solution:**

In order to find the selection set for rule 3, we need to find the follow set of the nonterminal  $A$ . This is the set of terminals (including  $\epsilon$ ) which could follow an  $A$  in a derivation from  $S\epsilon$ .

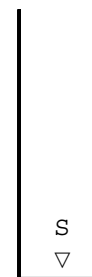
$$S\epsilon \Rightarrow bAb\epsilon$$

We cannot find any other terminals that can follow an  $A$  in a derivation from  $S\epsilon$ . Therefore,  $FOL(A) = \{b\}$ . The selection sets can now be listed:

$$\begin{aligned} Sel(1) &= \{b\} \\ Sel(2) &= \{a\} \\ Sel(3) &= FOL(A) = \{b\} \\ Sel(4) &= \{a\} \end{aligned}$$

The grammar is quasi-simple because the rules defining an  $S$  have disjoint selection sets and the rules defining an  $A$  have disjoint selection sets. The pushdown machine is shown below:

	a	b	$\epsilon$
S	Rep (a) Retain	Rep (bAb) Retain	Reject
A	Rep (aSa) Retain	Pop Retain	Reject
a	Pop Advance	Reject	Reject
b	Reject	Pop Advance	Reject
$\nabla$	Reject	Reject	Accept



Initial Stack

The recursive descent parser is shown below:

```

void S ()
{   if (inp=='b')                               // apply rule 1
    {   getInp();
        A ();
        if (inp=='b') getInp();
        else reject();
    }
    else if (inp=='a') getInp();                // apply rule 2
    else reject();
}

void A ()
{   if (inp=='b') ;                             // apply rule 3
    else if (inp=='a')                          // apply rule 4
    {   getInp();
        S ();
        if (inp=='a') getInp();
        else reject();
    }
    else reject();
}

```

Note that rule 3 is implemented with a null statement. This should not be surprising since rule 3 defines A as the null string.

### Exercises 4.2

1. Show the *sequence of stacks* for the pushdown machine of Figure 4.8 for each of the following input strings:
  - (a)  $ab\leftarrow$
  - (b)  $acbb\leftarrow$
  - (c)  $aab\leftarrow$
2. Show a *derivation tree* for each of the input strings in Problem 1, using grammar G14. Number the nodes of the tree to indicate the sequence in which they were applied by the pushdown machine.
3. Given the following grammar:

1.  $S \rightarrow a A b S$
2.  $S \rightarrow \epsilon$
3.  $A \rightarrow a S b$
4.  $A \rightarrow \epsilon$

- (a) Find the *follow set* for each nonterminal.
- (b) Show an *extended pushdown machine* for the language of this grammar.
- (c) Show a *recursive descent parser* for this grammar.

### 4.3 LL(1) Grammars

We now generalize the class of grammars that can be parsed top down by allowing rules of the form  $A \rightarrow \alpha$  where  $\alpha$  is any string of terminals and nonterminals. However, the grammar must be such that any two rules defining the same nonterminal must have disjoint selection sets. If it meets this condition, the grammar is said to be **LL(1)**, and we can construct a one-state pushdown machine parser or a recursive descent parser for it. The name LL(1) is derived from the fact that the parser finds a *left*-most derivation when scanning the input from *left* to right if it can look ahead no more than *one* input symbol. In this section we present an algorithm to find selection sets for an arbitrary context-free grammar.

The algorithm for finding selection sets of any context-free grammar consists of twelve steps and is shown below. Intuitively, the selection set for each rule in the grammar is the set of terminals which the parser can expect to encounter when applying that grammar rule. For example, in grammar G15, below, we would expect the terminal symbol *b* to be in the selection set for rule 1, since:

$$S \Rightarrow ABC \Rightarrow bABC$$

In this discussion, the phrase “any string” always includes the null string, unless otherwise stated. As each step is explained, we present the result of that step when applied to the example, grammar G15.

G15:

1.  $S \rightarrow ABC$
2.  $A \rightarrow bA$
3.  $A \rightarrow \epsilon$
4.  $B \rightarrow c$

Step 1. Find all nullable rules and nullable nonterminals:

Remove, temporarily, all rules containing a terminal. All  $\epsilon$  rules are **nullable rules**. The nonterminal defined in a nullable rule is a **nullable nonterminal**. In addition, all rules in the form

$$A \rightarrow B C D \dots$$

where  $B, C, D, \dots$  are all nullable non-terminals are nullable rules, and they define nullable nonterminals. In other words, a nonterminal is nullable if  $\epsilon$  can be derived from it, and a rule is nullable if  $\epsilon$  can be derived from its right side.

For grammar G15–  
 Nullable rules: rule 3

Nullable nonterminals: A

Step 2. Compute the relation “Begins Directly With” for each nonterminal:

$A \text{ BDW } X$  if there is a rule  $A \rightarrow \alpha X \beta$  such that  $\alpha$  is a nullable string (a string of nullable non-terminals). A represents a nonterminal and X represents a terminal or nonterminal.  $\beta$  represents any string of terminals and nonterminals.

For G15–

S BDW A (from rule 1)  
 S BDW B (also from rule 1, because A is nullable)  
 A BDW b (from rule 2)  
 B BDW c (from rule 4)

Step 3. Compute the relation “Begins With”:

$X \text{ BW } Y$  if there is a string beginning with Y that can be derived from X. BW is the reflexive transitive closure of BDW. In addition, BW should contain pairs of the form  $a \text{ BW } a$  for each terminal a in the grammar.

For G15–

S BW A  
 S BW B (from BDW)  
 A BW b  
 B BW c  
  
 S BW b (transitive)  
 S BW c  
  
 S BW S  
 A BW A  
 B BW B (reflexive)  
 b BW b  
 c BW c

Step 4. Compute the set of terminals "First(x)" for each symbol x in the grammar.

At this point, we can find the set of all terminals which can begin a sentential form when starting with a given symbol of the grammar.

$\text{First}(A)$  = set of all terminals b, such that  $A \text{ BW } b$  for each nonterminal A.  
 $\text{First}(t)$  = {t} for each terminal.

ForG15–

$\text{First}(S) = \{b, c\}$   
 $\text{First}(A) = \{b\}$   
 $\text{First}(B) = \{c\}$   
 $\text{First}(b) = \{b\}$   
 $\text{First}(c) = \{c\}$

Step 5. Compute "First" of right side of each rule:

We now compute the set of terminals which can begin a sentential form derivable from the right side of each rule.

$$\begin{aligned} \text{First}(XYZ\dots) &= \{\text{First}(X)\} \\ &\cup \{\text{First}(Y)\} \quad \text{if } X \text{ is nullable} \\ &\cup \{\text{First}(Z)\} \quad \text{if } Y \text{ is also nullable} \\ &\vdots \\ &\vdots \\ &\vdots \end{aligned}$$

In other words, find the union of the  $\text{First}(x)$  sets for each symbol on the right side of a rule, but stop when reaching a non-nullable symbol.

ForG15–

1.  $\text{First}(ABc) = \text{First}(A) \cup \text{First}(B) = \{b, c\}$  (because A is nullable)
2.  $\text{First}(bA) = \{b\}$
3.  $\text{First}(\epsilon) = \{\}$
4.  $\text{First}(c) = \{c\}$

If the grammar contains no nullable rules, you may skip to step 12 at this point.

Step 6. Compute the relation "Is Followed Directly By":

B FDB X if there is a rule of the form

$A \rightarrow \alpha B \beta X \gamma$

where  $\beta$  is a string of nullable nonterminals,  $\alpha, \gamma$  are strings of symbols, X is any symbol, and A and B are nonterminals.

ForG15–

A FDB B (from rule 1)  
B FDB c (from rule 1)

Note that if B were a nullable nonterminal we would also have  $A \text{ FDB } c$ .

Step 7. Compute the relation “Is Direct End Of”:

$X \text{ DEO } A$  if there is a rule of the form:

$$A \rightarrow \alpha X \beta$$

where  $\beta$  is a string of nullable nonterminals,  $\alpha$  is a string of symbols, and X is a single grammar symbol.

For G15 –

$c \text{ DEO } S$  (from rule 1)  
 $A \text{ DEO } A$  (from rule 2)  
 $b \text{ DEO } A$  (from rule 2, since A is nullable)  
 $c \text{ DEO } B$  (from rule 4)

Step 8. Compute the relation “Is End Of”:

$X \text{ EO } Y$  if there is a string derived from Y that ends with X. EO is the reflexive transitive closure of DEO. In addition, EO should contain pairs of the form  $N \text{ EO } N$  for each nullable nonterminal, N, in the grammar.

For G15 –

$c \text{ EO } S$   
 $A \text{ EO } A$  (from DEO)  
 $b \text{ EO } A$   
 $c \text{ EO } B$   
 (no transitive entries)  
 $c \text{ EO } c$   
 $S \text{ EO } S$  (reflexive)  
 $b \text{ EO } b$   
 $B \text{ EO } B$

Step 9. Compute the relation “Is Followed By”:

$W \text{ FB } Z$  if there is a string derived from  $S \Leftarrow$  in which W is immediately followed by Z.

If there are symbols X and Y such that

$W \text{ EO } X$   
 $X \text{ FDB } Y$   
 $Y \text{ BW } Z$

then  $W \text{ FB } Z$

For G15–

A EO A	A FDB B	B BW B	A FB B
		B BW c	A FB c
b EO A		B BW B	b FB B
		B BW c	b FB c
B EO B	B FDB c	c BW c	B FB c
c EO B		c BW c	c FB c

Step 10. Extend the FB relation to include endmarker:

$A \text{ FB } \leftrightarrow$  if  $A \text{ EO } S$  where A represents any nonterminal and S represents the starting nonterminal.

For G15–

$S \text{ FB } \leftrightarrow$  because  $S \text{ EO } S$

There are now seven pairs in the FB relation for grammar G15.

Step 11. Compute the Follow Set for each nullable nonterminal:

The follow set of any nonterminal A is the set of all terminals,  $t$ , for which  $A \text{ FB } t$ .

$$\text{Fol}(A) = \{t \mid A \text{ FB } t\}$$

To find selection sets, we need find follow sets for nullable nonterminals only.

For G15–

$$\text{Fol}(A) = \{c\} \text{ since } A \text{ is the only nullable nonterminal and } A \text{ FB } c.$$

Step 12. Compute the selection set for each rule:

i.  $A \rightarrow \alpha$

if rule i is not a nullable rule, then  $\text{Sel}(i) = \text{First}(\alpha)$

if rule i is a nullable rule, then  $\text{Sel}(i) = \text{First}(\alpha) \cup \text{Fol}(A)$

For G15–

$$\text{Sel}(1) = \text{First}(ABC) = \{b, c\}$$

$$\text{Sel}(2) = \text{First}(bA) = \{b\}$$

$$\text{Sel}(3) = \text{First}(\epsilon) \cup \text{Fol}(A) = \{\} \cup \{c\} = \{c\}$$

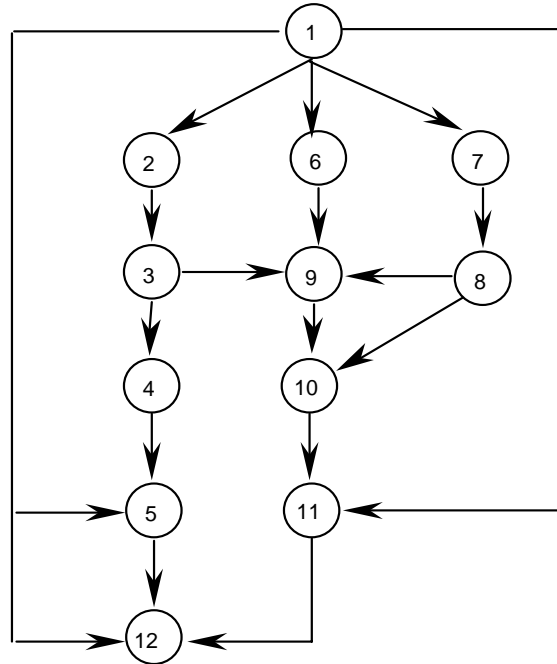
$$\text{Sel}(4) = \text{First}(c) = \{c\}$$

Notice that since we are looking for the follow set of a nullable nonterminal in step 12, we have actually done much more than necessary in step 9. In step 9 we need produce only those pairs of the form  $A \text{ FB } \tau$ , where  $A$  is a nullable nonterminal and  $\tau$  is a terminal.

The algorithm is summarized, below, in Figure 4.9. A context-free grammar is LL(1) if rules defining the same nonterminal always have disjoint selection sets. Grammar G15 is LL(1) because rules 2 and 3 (defining the nonterminal  $A$ ) have disjoint selection sets (the selection sets for those rules have no terminal symbols in common). Note that if there are no nullable rules in the grammar, we can get the selection sets directly from step 5 – i.e., we can skip steps 6-11. A graph showing the dependence of any step in this algorithm on the results of other steps is shown in Figure 4.10. For example, this graph shows that the results of steps 3, 6, and 8 are needed for step 9.

1. Find nullable rules and nullable nonterminals.
2. Find "Begins Directly With" relation (BDW).
3. Find "Begins With" relation (BW).
4. Find "First(x)" for each symbol, x.
5. Find "First(n)" for the right side of each rule, n.
6. Find "Followed Directly By" relation (FDB).
7. Find "Is Direct End Of" relation (DEO).
8. Find "Is End Of" relation (EO).
9. Find "Is Followed By" relation (FB).
10. Extend FB to include endmarker.
11. Find Follow Set,  $\text{Fol}(A)$ , for each nullable nonterminal, A.
12. Find Selection Set,  $\text{Sel}(n)$ , for each rule, n.

**Figure 4.9** Summary of Algorithm to Find Selection Sets of any Context-Free Grammar.



**Figure 4.10** Dependency Graph for the Steps in the Algorithm for Finding Selection Sets.

### 4.3.1 Pushdown Machines for LL(1) Grammars

Once the selection sets have been found, the construction of the pushdown machine is exactly as for quasi-simple grammars. For a rule in the grammar,  $A \rightarrow \alpha$ , fill in the cells in the row for nonterminal  $A$  and in the columns for the selection set of that rule with *Rep*, *Retain*, where  $\alpha^r$  represents the right side of the rule reversed. For  $\epsilon$  rules, fill in *Pop*, *Retain* in the columns for the selection set. For each terminal symbol, enter *Pop*, *Advance* in the cell in the row and column labeled with that terminal. The cell in the row labeled  $\nabla$  and the column labeled  $\Leftarrow$  should contain *Accept*. All other cells are *Reject*. The pushdown machine for grammar G15 is shown in Figure 4.11.

### 4.3.2 Recursive Descent for LL(1) Grammars

Once the selection sets have been found, the construction of the recursive descent parser is exactly as for quasi-simple grammars. When implementing each rule of the grammar, check for the input symbols in the selection set for that grammar. A recursive descent parser for grammar G15 is shown below:

```
void parse ()
```

	b	c	←
S	Rep (cBA) Retain	Rep (cBA) Retain	Reject
A	Rep (Ab) Retain	Pop Retain	Reject
B	Reject	Rep (c) Retain	Reject
b	Pop Advance	Reject	Reject
c	Reject	Pop Advance	Reject
▽	Reject	Reject	Accept

S
▽

Initial Stack

Figure 4.11 A Pushdown Machine for Grammar G15

```

{  getInp();
   S ();
   if (inp=='←') accept; else reject();
}

void S ()
{  if (inp=='b' || inp=='c')          // apply rule 1
   {  A ();
      B ();
      if (inp=='c') getInp();
      else reject();
   }
   else reject();
}

void A ()
{  if (inp=='b')                      // apply rule 2
   {  getInp();
      A ();
   }
   else if (inp=='c') ;               // apply rule 3
   else reject();
}

void B ()

```

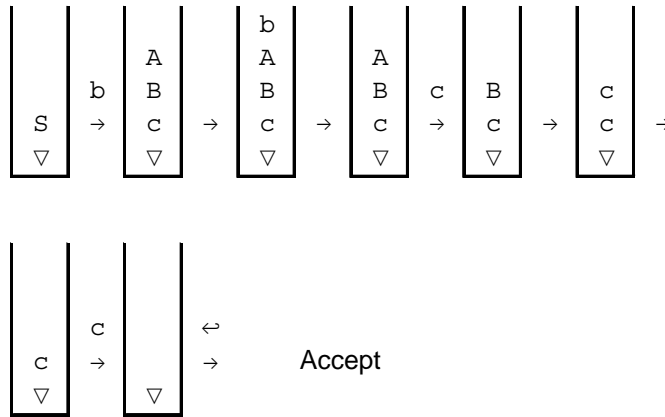
```
{  if (inp=='c') getInp();      // apply rule 4
   else reject();
}
```

Note that when processing rule 1, an input symbol is not read until a terminal is encountered in the grammar rule (after checking for a or b, an input symbol should not be read before calling procedure A).

**Sample Problem 4.3**

Show the sequence of stacks that occurs when the pushdown machine of Figure 4.11 parses the string  $bcc \Leftarrow$ .

**Solution:**



**Exercises 4.3**

- Given the following information, find the *Followed By* relation (FB) as described in step 9 of the algorithm for finding selection sets:

A EO A	A FDB D	D BW b
A EO B	B FDB a	b BW b
B EO B		a BW a

2. Find the *selection sets* of the following grammar and determine whether it is LL(1).
  1.  $S \rightarrow ABD$
  2.  $A \rightarrow aA$
  3.  $A \rightarrow \epsilon$
  4.  $B \rightarrow bB$
  5.  $B \rightarrow \epsilon$
  6.  $D \rightarrow dD$
  7.  $D \rightarrow \epsilon$
3. Show a *pushdown machine* for the grammar of Problem 2.
4. Show a *recursive descent parser* for the grammar of Problem 2.
5. Step 3 of the algorithm for finding selection sets is to find the “Begins With” relation by forming the reflexive transitive closure of the “Begins Directly With” relation. Then add “pairs of the form  $a \text{ BW } a$  for each terminal  $a$  in the grammar”; i.e., there could be terminals in the grammar which do not appear in the BDW relation. Find an example of a grammar in which the selection sets will not be found correctly if you don’t add these pairs to the BW relation (hint: see step 9).

#### 4.4 Parsing Arithmetic Expressions Top Down

Now that we understand how to determine whether a grammar can be parsed down, and how to construct a top down parser, we can begin to address the problem of building top down parsers for actual programming languages. One of the most heavily studied aspects of parsing programming languages deals with *arithmetic expressions*. Recall grammar G5 for arithmetic expressions involving only addition and multiplication, from Section 3.1. We wish to determine whether this grammar is LL(1).

G5:

1.  $\text{Expr} \rightarrow \text{Expr} + \text{Term}$
2.  $\text{Expr} \rightarrow \text{Term}$
3.  $\text{Term} \rightarrow \text{Term} * \text{Factor}$
4.  $\text{Term} \rightarrow \text{Factor}$
5.  $\text{Factor} \rightarrow ( \text{Expr} )$
6.  $\text{Factor} \rightarrow \text{var}$

In order to determine whether this grammar is LL(1), we must first find the selection set for each rule in the grammar. We do this by using the twelve step algorithm given in Section 4.3.

1. Nullable rules: none  
 Nullable nonterminals: none
  
2.
 

Expr	BDW	Expr
Expr	BDW	Term
Term	BDW	Term
Term	BDW	Factor
Factor	BDW	(
Factor	BDW	var
  
3.
 

Expr	BW	Expr
Expr	BW	Term
Term	BW	Term
Term	BW	Factor
Factor	BW	(
Factor	BW	var

Factor	BW	Factor
(	BW	(
var	BW	var

Expr	BW	Factor
------	----	--------

```

Expr      BW      (
Expr      BW      var
Term      BW      (
Term      BW      var
*         BW      *
+         BW      +
)         BW      )

```

4.    First(Expr)            = { (, var }  
       First(Term)           = { (, var }  
       First(Factor)        = { (, var }
5.    (1) First(Expr + Term)        = { (, var }  
       (2) First(Term)               = { (, var }  
       (3) First(Term \* Factor)      = { (, var }  
       (4) First(Factor)              = { (, var }  
       (5) First( ( Expr ) )         = { ( }  
       (6) First (var)                = { var }
12.   Sel(1) = { (, var }  
       Sel(2) = { (, var }  
       Sel(3) = { (, var }  
       Sel(4) = { (, var }  
       Sel(5) = { ( }  
       Sel(6) = { var }

Since there are no nullable rules in the grammar, we can obtain the selection sets directly from step 5. This grammar is not LL(1) because rules 1 and 2 define the same nonterminal, Expr, and their selection sets intersect. This is also true for rules 3 and 4.

Incidentally, the fact that grammar G5 is not suitable for top down parsing can be determined much more easily by inspection of the grammar. Rules 1 and 3 both have a property known as *left recursion* :

1. Expr → Expr + Term
3. Term → Term \* Factor

They are in the form:

$$A \rightarrow A\alpha$$

Note that any rule in this form cannot be parsed top down. To see this, consider the method for the nonterminal A in a recursive descent parser. The first thing it would do

would be to call itself, thus producing infinite recursion with no “escape hatch”. Any grammar with left recursion cannot be LL(1).

The left recursion can be eliminated by rewriting the grammar with an equivalent grammar that does not have left recursion. In general, the offending rule might be in the form:

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

in which we assume that  $\beta$  is a string of terminals and nonterminals that does not begin with an  $A$ . We can eliminate the left recursion by introducing a new nonterminal,  $R$ , and rewriting the rules as:

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

A more detailed and complete explanation of left recursion elimination can be found in Parsons [1992].

This methodology is used to rewrite the grammar for simple arithmetic expressions in which the new nonterminals introduced are *Elist* and *Tlist*. An equivalent grammar for arithmetic expressions involving only addition and multiplication, *G16*, is shown below. A derivation tree for the expression `var+var*var` is shown in Figure 4.12.

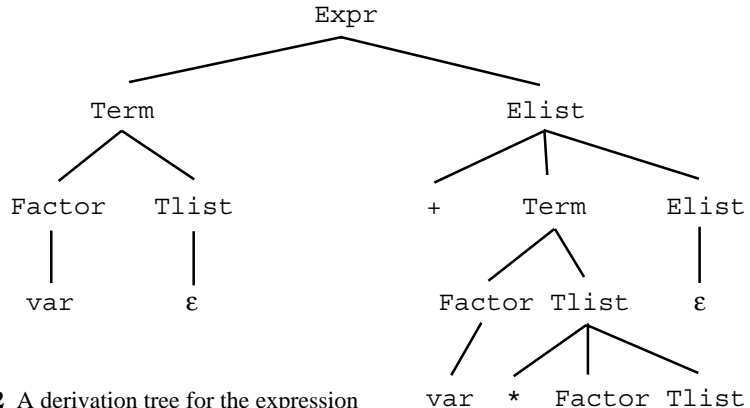
*G16*:

1. `Expr`  $\rightarrow$  `Term Elist`
2. `Elist`  $\rightarrow$   `+ Term Elist`
3. `Elist`  $\rightarrow$   $\epsilon$
4. `Term`  $\rightarrow$  `Factor Tlist`
5. `Tlist`  $\rightarrow$   `* Factor Tlist`
6. `Tlist`  $\rightarrow$   $\epsilon$
7. `Factor`  $\rightarrow$  `( Expr )`
8. `Factor`  $\rightarrow$  `var`

Note in grammar *G16* that an *Expr* is still the sum of one or more *Terms* and a *Term* is still the product of one or more *Factors*, but the left recursion has been eliminated from the grammar. We will see, later, that this grammar also defines the precedence of operators as desired. The student should construct several derivation trees using grammar *G16* in order to be convinced that it is not ambiguous.

We now wish to determine whether this grammar is LL(1), using the algorithm to find selection sets:

1. Nullable rules: 3, 6



**Figure 4.12** A derivation tree for the expression  $\text{var}+\text{var}*\text{var}$  using grammar G16

Nullable nonterminals: Elist, Tlist, var, ε

2.
 

Expr	BDW	Term	
Elist	BDW	+	
Term	BDW	Factor	
Tlist	BDW	*	
Factor	BDW	(	
Factor	BDW	var	
  
3.
 

Expr	BW	Term	
Elist	BW	+	
Term	BW	Factor	(fromBDW)
Tlist	BW	*	
Factor	BW	(	
Factor	BW	var	

Expr	BW	Factor	
Term	BW	(	
Term	BW	var	(transitive)
Expr	BW	(	
Expr	BW	var	

Expr	BW	Expr	
Term	BW	Term	
Factor	BW	Factor	
Elist	BW	Elist	
Tlist	BW	Tlist	(reflexive)
Factor	BW	Factor	
+	BW	+	

	*	BW	*	
	(	BW	(	
	var	BW	var	
	)	BW	)	
4.	First (Expr)	=	{(, var}	
	First (Elist)	=	{+}	
	First (Term)	=	{(, var}	
	First (Tlist)	=	{*}	
	First (Factor)	=	{(, var}	
5.	1. First(Term Elist)	=	{(, var}	
	2. First(+ Term Elist)	=	{+}	
	3. First( $\epsilon$ )	=	{}	
	4. First(Factor Tlist)	=	{(, var}	
	5. First(* Factor Tlist)	=	{*}	
	6. First( $\epsilon$ )	=	{}	
	7. First(( Expr ))	=	{(}	
	8. First(var)	=	{var}	
6.	Term	FDB	Elist	
	Factor	FDB	Tlist	
	Expr	FDB	)	
7.	Elist	DEO	Expr	
	Term	DEO	Expr	
	Elist	DEO	Elist	
	Term	DEO	Elist	
	Tlist	DEO	Term	
	Factor	DEO	Term	
	Tlist	DEO	Tlist	
	Factor	DEO	Tlist	
	)	DEO	Factor	
	var	DEO	Factor	
8.	Elist	EO	Expr	
	Term	EO	Expr	
	Elist	EO	Elist	
	Term	EO	Elist	
	Tlist	EO	Term	
	Factor	EO	Term	(fromDEO)
	Tlist	EO	Tlist	
	Factor	EO	Tlist	
	)	EO	Factor	



10. Elist FB  $\leftarrow$   
 Term FB  $\leftarrow$   
 Expr FB  $\leftarrow$   
 Tlist FB  $\leftarrow$   
 Factor FB  $\leftarrow$
11. Fol (Elist) =  $\{), \leftarrow\}$   
 Fol (Tlist) =  $\{+, ), \leftarrow\}$
12. Sel (1) = First (Term Elist) =  $\{ (, \text{var}\}$   
 Sel (2) = First (+ Term Elist) =  $\{+\}$   
 Sel (3) = Fol (Elist) =  $\{), \leftarrow\}$   
 Sel (4) = First (Factor Tlist) =  $\{ (, \text{var}\}$   
 Sel (5) = First (\* Factor Tlist) =  $\{*\}$   
 Sel (6) = Fol (Tlist) =  $\{+, ), \leftarrow\}$   
 Sel (7) = First ( ( Expr ) ) =  $\{( \}$   
 Sel (8) = First (var) =  $\{\text{var}\}$

Since all rules defining the same nonterminal (rules 2 and 3, rules 5 and 6, rules 7 and 8) have disjoint selection sets, the grammar G16 is LL(1).

In step 9 we could have listed several more entries in the FB relation. For example, we could have listed pairs such as var FB + and Tlist FB Elist. These were not necessary, however; this is clear if one looks ahead to step 11, where we construct the follow sets for nullable nonterminals. This means we need to use only those pairs from step 9 which have a nullable nonterminal on the left and a terminal on the right. Thus, we

#### Sample Problem 4.4

Show a pushdown machine and a recursive descent translator for arithmetic expressions involving addition and multiplication using grammar G16.

#### Solution:

To build the pushdown machine we make use of the selection sets shown above. These tell us which columns of the machine are to be filled in for each row. For example, since the selection set for rule 4 is  $\{ (, \text{var}\}$ , we fill the cells in the row labeled Term and columns labeled ( and var with information from rule 4: Rep (Tlist Factor). The solution is shown on the next page.

We make use of the selection sets again in the recursive descent processor. In each procedure, the input symbols in the selection set tell us which rule of the grammar to apply. Assume that a var is represented by the integer 256.

	+	*	(	)	var	↵
Expr	Reject	Reject	Rep(Elist Term) Retain	Reject	Rep(Elist Term) Retain	Reject
Elist	Rep(Elist Term +) Retain	Reject	Reject	Pop Retain	Reject	Pop Retain
Term	Reject	Reject	Rep(Tlist Factor) Retain	Reject	Rep(Tlist Factor) Retain	Reject
Tlist	Pop Retain	Rep(Tlist Factor *) Retain	Reject	Pop Retain	Reject	Pop Retain
Factor	Reject	Reject	Rep( )Expr( ) Retain	Reject	Rep(var) Retain	Reject
+	Pop Advance	Reject	Reject	Reject	Reject	Reject
*	Reject	Pop Advance	Reject	Reject	Reject	Reject
(	Reject	Reject	Pop Advance	Reject	Reject	Reject
)	Reject	Reject	Reject	Pop Advance	Reject	Reject
var	Reject	Reject	Reject	Reject	Pop Advance	Reject
↵	Reject	Reject	Reject	Reject	Reject	Accept

Expr  
↵

Initial Stack

```

int inp; const int var = 256;
void Expr ()
{
    if (inp=='(' || inp==var) // apply rule 1
    {
        Term ();
        Elist ();
    } // end rule 1
    else reject ();
}
    
```

```
void Elist ()
{   if (inp=='+')           // apply rule 2
    {   getInp();
        Term ();
        Elist ();
    }
    else if (inp=='|' || inp=='←') ; // apply rule 3
    else reject ();
}

void Term ()
{   if (inp=='(' || inp==var) // apply rule 4
    {   Factor ();
        Tlist ();
    }
    else reject();           // end rule 4
}

void Tlist ()
{   if (inp=='*')           // apply rule 5
    {   getInp();
        Factor ();
        Tlist ();
    }
    else if (inp=='+' || inp=='|' || inp=='←') // end rule 5
        ; // apply rule 6
    else reject();
}

void Factor ()
{   if (inp=='(')           // apply rule 7
    {   getInp();
        Expr ();
        if (inp=='|') getInp();
        else reject();
    }
    else if (inp==var) getInp(); // end rule 7
    else reject();           // apply rule 8
}
```

will not need `var FB +` because the left member is not a nullable nonterminal, and we will not need `Tlist FB Elist` because the right member is not a terminal.

### Exercises 4.4

1. Show *derivation trees* for each of the following input strings, using grammar G16.
  - (a) `var + var`
  - (b) `var + var * var`
  - (c) `(var + var) * var`
  - (d) `((var))`
  - (e) `var * var * var`
  
2. We have shown that grammar G16 for simple arithmetic expressions is LL(1), but grammar G5 is not LL(1). What are the advantages, if any, of grammar G5 over grammar G16?
  
3. Suppose we permitted our parser to “peek ahead”  $n$  characters in the input stream to determine which rule to apply. Would we then be able to use grammar G5 to parse simple arithmetic expressions top down? In other words, is grammar G5  $LL(n)$ ?
  
4. Find two *null statements* in the recursive descent parser of the sample problem in this section. Which methods are they in and which grammar rules do they represent?
  
5. Construct part of a *recursive descent parser* for the following portion of a programming language:
  1. `Stmt`  $\rightarrow$  `if (Expr) Stmt`
  2. `Stmt`  $\rightarrow$  `while (Expr) Stmt`
  3. `Stmt`  $\rightarrow$  `{ StmtList }`
  4. `Stmt`  $\rightarrow$  `Expr ;`

Write the procedure for the nonterminal `Stmt`. Assume the selection set for rule 4 is `{(, identifier, number}`.
  
6. Show an  $LL(1)$  grammar for the language of regular expressions over the alphabet `{0, 1}`, and show a *recursive descent parser* corresponding to the grammar.

7. Show how to *eliminate the left recursion* from each of the grammars shown below:
- (a)
1.  $A \rightarrow A b c$
  2.  $A \rightarrow a b$
- (b)
1.  $\text{ParmList} \rightarrow \text{ParmList} , \text{Parm}$
  2.  $\text{ParmList} \rightarrow \text{Parm}$
8. A parameter list is a list of 0 or more parameters separated by commas; a parameter list neither begins nor ends with a comma. Show an LL(1) *grammar* for a parameter list. Assume that parameter has already been defined.