22 Bottom-up Parsing

Introduction

There are essentially two ways to parse: top-down and bottom-up. In top-down parsing, we start at the top of the parse tree with the start symbol and work our way down to the terminals at the bottom of the parse tree. In bottom-up parsing, we start at the bottom of the parse tree with the terminal symbols and work our way up to the start symbol.

Here are some of the distinguishing features of top-down and bottom-up parsers:

- Bottom-up parsing works with a larger class of context-free grammars than top-down parsing. For example, bottom-up parsers work for both left recursive and right recursive grammars. Top-down parsers cannot handle left recursive grammars.

- Top-down parsers can easily be written by hand. The grammar along with the predict sets is essentially a flowchart for the parser code. Bottom-up parsers have a complex structure that make them difficult to implement by hand, except for very simple grammars.

- A top-down parser applies productions in the grammar, in effect replacing the left side with the right side. For example, suppose a grammar has the production

  ```
  S → AbC
  ```

  A top-down parser replaces `S` with `AbC`. But a bottom-up parser replaces `AbC` with S. That is, a bottom-up parser applies the productions in the grammar in the *reverse* direction.

- A top-down parser determines the leftmost derivation of the input string. A bottom-up parser determines the rightmost derivation of the input string.

Principles of Bottom-up Parsing

Let's perform a bottom-up parse of `aaa` using the following grammar, which we call the g1 grammar:

```
1) S → Sa          g1 grammar
2) S → a
```

The bottom-up parse of the input string `aaa` is in Fig. 22.1 In this example, the parse constructs the parse tree.

1)

```
a     a     a
^
```

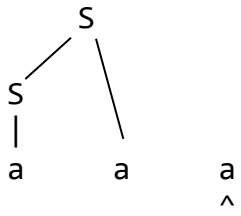
Reduce by production 2 and advance in the input string.

2)

```
S
|
a     a     a
      ^
```

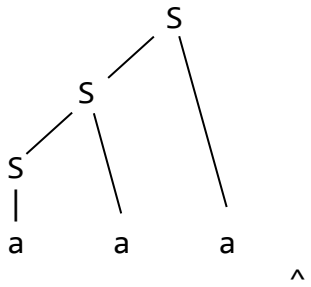
Reduce by production 1 and advance in the input.

3)



Reduce by production 1 and advance in the input.

4)



Accept (because only nonterminal remaining is S, and the current input is the end-of-input marker)

Figure 22.1

In snapshot (1), we have the given input string aaa. The caret symbol (^) marks the current input. The initial a in the input string comes from production 2 (S → a). So we replace a (the right side of the production 2) with S (the left side of the production). We show this replacement in snapshot (2) by drawing a line from a to S above it. We call this replacement operation in which we replace an occurrence of the right side of a production with its left side a *reduce operation*.

In snapshot (2), we now have Sa (S from the preceding reduce operation and the current input a). Sa comes from production 1. So we now reduce by production 1: We replace Sa with S to get snapshot (3).

In snapshot (3), we again have Sa (the S from the preceding reduce operation and the current input a). So we again reduce by production 1 to get snapshot (4). At this point, the parse has completed successfully, which indicates the input string is in the language defined by the grammar. A parse is successful if the constructed parse tree generates the entire input string, and its top node is the start symbol of the grammar and is the only node that has not be replaced by a reduce operation. When the parser successfully completes a parse, we say the parser *accepts* the input string. Otherwise, we say the parser *rejects* the input string.

Bottom-up Parsing Using a Stack

If we use a stack, we can perform a bottom-up parse without building a parse tree. In this approach we *shift* the characters in the input string one by one onto the stack. Whenever the right side of a production matches the top symbol(s) on the stack and that production produced those symbols in a rightmost derivation of the input string, we *reduce* by that production—that is, we replace the top symbols on the stack that match the production's right side with the production's left side. For example, suppose our grammar is the g1 grammar from the preceding section:

```
1) S → Sa          g1 grammar
2) S → a
```

Then whenever `a` (but not `Sa`) is on top of the stack, we reduce by production 2 (i.e., we replace the `a` on the stack with `S`). Whenever `Sa` is on top of the stack, we reduce by production 1 (i.e., we replace `Sa` on the stack with `S`). Let's do a bottom-up parse for this grammar for the input string `aaa` (this parse is the stack version of the parse in Fig. 22.1). We start with an empty stack (line 1). The symbols `$` and `#` are the bottom-of-stack and end-of-input markers, respectively.

```
Stack        Operation          Input
 1)   $                         aaa#
 2)             shift           ^
 3)   $a                        aa#
 4)             reduce(2)       ^
 5)   $S                        aa#
 6)             shift           ^
 7)   $Sa                       a#
 8)             reduce(1)       ^
 9)   $S                        a#
10)             shift           ^
11)   $Sa                       #
12)             reduce(1)       ^
13)   $S                        # ◄——— Accept configuration so accept
14)             accept          ^
```

Figure 22.2

Line 1 is the initial configuration: the stack is empty, and the current input is the first character in the input string. We obviously cannot reduce (because the stack is empty and the grammar does not have any lambda productions), so we shift the current input onto the stack to get line 3. We now have the right side of production 2 on top of the stack, so we reduce by production 2 to get line 5. `S` by itself is not the right side of a production, so we shift to get line 7, which gives `Sa` on top of the stack. `Sa` is the right side of production 1. So we reduce by production 1 to get line 9. We again do another shift and another reduce, at which point we can do neither a shift or reduce so the parse terminates. The final configuration has only `S` on the stack and the current input is the end-of-input marker. The parser started with the input string and ended with the start symbol. That is, the parser performed a derivation of the input string in reverse order. Thus, because a derivation of the input string exists, the input string is in the language defined by the grammar.

In Fig. 22.3, we show the strings produced if we concatenate the stack symbols with the remaining

input for each stack-input configuration in Fig. 22.2. For example, on line 9 in Fig. 22.2 we concatenate the S on the stack with the remaining input a in the input to get Sa.

	stack	remaining input	concatenation
line 1:	$	aaa#	= aaa
line 3:	$a	aa#	= aaa
line 5:	$S	aa#	= Saa
line 7:	$Sa	a#	= Saa
line 9:	$S	a#	= Sa
line 11:	$Sa	#	= Sa
line 13:	$S	#	= S

Figure 22.3

We get the following sequence of strings (with repeats omitted):

aaa, Saa, Sa, S

Compare it with the sequence of strings produced by the rightmost derivation of aaa:

S ⇒ Sa ⇒ Saa ⇒ aaa

The sequences are the same except that the former is in *reverse* order. This example illustrates that bottom-up parsing performs a rightmost derivation of the input string in reverse order. Thus, a parse that ends in the accept configuration (only the start symbol on the stack and the end-of-input marker as the current input) indicates that the input string is in the language defined by the grammar. Incidentally, we call the sequence of strings in a derivation *sentential forms*. In the derivation above, S, Sa, Saa, and aaa are the sentential forms in the derivation of aaa from S.

Important observation: In a successful bottom-up parse (i.e., a parse that ends in the accept configuration), concatenating the symbols on the stack with the remaining input for each configuration during the parse produces a string that is a sentential form in a rightmost derivation of the original input string. For example, on line 9 above, the concatenation produces the string Sa. Thus, Sa must be a sentential form in the derivation of aaa. Moreover, because Sa is a sentential form, then a must be in the FOLLOW set of S (because it obviously can follow S, as evidenced by the Sa sentential form).

Handles

The following example shows that the parser does not necessarily reduce when the right side of a production is on top of the stack. Consider the following g2 grammar:

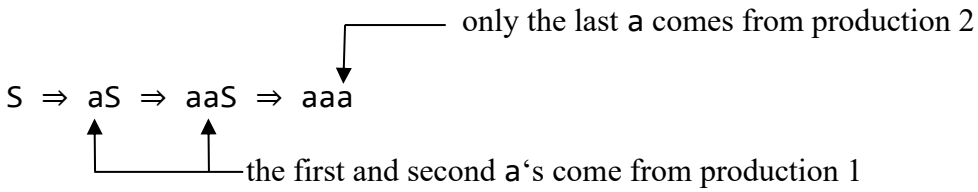
1) S → aS g2 grammar
2) S → a

The parse of the string aaa using this grammar is given in Fig. 22.4.

```
 1)    $                        aaa#
 2)          shift
 3)    $a                       aa#       (do not reduce here)
 4)          shift
 5)    #aa                      #
 6)          shift
 7)    #aaa                     #
 8)          reduce(2)
 9)    #aaS                     #
10)          reduce(1)
11)    #aS                      #
12)          reduce(1)
13)    #S                       #
14)          accept
```

Figure 22.4

We start by shifting the initial a onto the stack. a is the right side of production 2. But this a *does not come from production 2* as the rightmost derivation of aaa illustrates:



For this reason, we should *not* reduce by production 2. Here is the general rule on reducing: Reduce only if

1) The top symbol (or symbols) on the stack matches the right side of a production, and
2) these symbols on the stack in a rightmost derivation come from *that* production.

If these two conditions hold, we say that the symbols on top of the stack that match the right side of the production that generates them are a *handle*. A parser should reduce only when a handle is on top of the stack. The challenge in implementing bottom-up parsers is to determine when a handle is on top of the stack.

Parsing with Right Versus Left Recursive grammars

Top-down parsers can use right recursive grammars but not left recursive grammars. Bottom-up parsers, however, can use either right recursive or left recursive grammars. Earlier (Fig. 22.2), we parsed the string aaa using the g1 left-recursive grammar:

```
1) S → Sa    left-recursive (i.e., left side nonterminal leftmost on right side)
2) S → a
```

In Fig. 22.4, we parsed the same input string using the g2 right-recursive grammar:

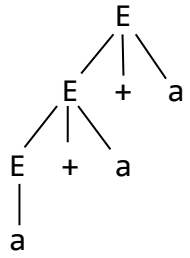
```
1) S → aS    right recursive (i.e., left side nonterminal rightmost of right side)
2) S → a
```

Comparing Fig. 22.2 and Fig. 22.4, we can see that for both grammars, the parser shifts three times and reduces three times. However, the order is different. With the right-recursive grammar, the three shifts occur first, then the three reduces. With the left recursive grammar, the parser alternates shifting and reducing. Which grammar is better for bottom-up parsing? The parsers for both grammars do the same about of work: one shift and one reduce for each letter in the input string. However, they differ with respect to the maximum stack size. In the right recursive case, the parser pushes the entire input string onto the stack before it reduces. The maximum stack size is equal to the size of the input string. Because there is no upper limit on the size of the input string, there is *no upper limit on the stack size* required to parse strings. In the left-recursive case, the parser alternates shifting with reducing. Thus, the stack size never exceeds two regardless of the length of the input string. Although the work both parsers do is the same, the left recursive parser is better because with it there is no possibility of a stack overflow.

Most arithmetic operators are *left associative* (i.e., operations of the same precedence are performed left to right). Thus, another advantage of left-recursive grammars is that they imply left associativity. Consider the two grammars in Fig. 22.5, one left recursive, the other right recursive. They generate the same language (an initial a followed by zero or more occurrences of "+a"). Observe in Fig. 22.5 that the parse tree from the left recursive grammar implies left associativity (because the left operand of the right + is the result of the operation specified by the left +; the parse from the right recursive grammar implies right associativity (because the right operand of the left + is the result of the operation specified by the right +).

Left Recursive

```
1) E → E + a     g3 grammar
2) E → a
```



Right Recursive

```
E → a + E     g4 grammar
E → a
```

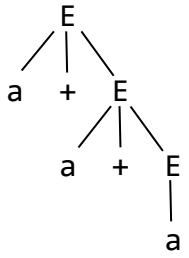


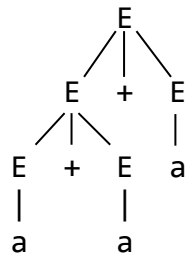
Figure 22.5

Bottom-up Parsing with Ambiguous Grammars

By definition, a grammar is *ambiguous* if there is at least one string with more than one parse tree. For example, in the following grammar, two parse trees exist for the string a+a+a (see Fig. 22.6):

```
1) E → E + E          g5 grammar
2) E → a
```

a)



b)

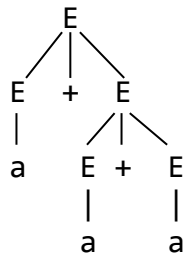


Figure 22.6

The parse tree in Fig. 22.6a implies left associativity. The parse tree in Fig. 22.6b implies right associativity (i.e., the operations are performed right to left).

Because there are two parse trees for `aaa`, there must be a choice at some point in the bottom-up parse of `aaa`. One alternative leads to the tree in Fig. 22.6a. The other alternative leads to the tree in Fig. 22.6b. Let's look at the first 11 lines of the parse of `aaa` in Fig. 22.7.

```
 1)   $                  a+a+a#
 2)          shift
 3)   $a                 +a+a#
 4)          reduce(3)
 5)   $E                 +a+a#
 6)          shift
 7)   $E+                a+a#
 8)          shift
 9)  $E+a                +a#
10)          reduce(3)
11)  $E+E                +a#
```

```
             reduce(1)          or                  shift
12a) $E                 +a#     |   12b) $E+E+                a#
             shift              |                  shift
13a) $E+                 a#     |   13b) $E+E+a                #
             shift              |                  reduce(3)
14a) $E+a                #      |   14b) $E+E+E                #
             reduce(3)          |                  reduce(1)
15a) $E+E                #      |   15b) $E+E                  #
             reduce(1)          |                  reduce(1)
16a) $E                 +z#     |   16b) $E                    #
```

Figure 22.7

Up to line 11, there is no choice in the operations that the bottom-up parser performs. However, on line 11, there is a choice between shift and reduce. We call this situation a *shift/reduce conflict*. If the parser reduces at this point, it in effect is parsing according to the parse tree in Fig. 22.6a. If, however, it shifts, it is parsing according to the parse tree in Fig. 22.6b. So should the parser shift or reduce? It should reduce because the parse tree in Fig. 22.6a implies the correct associativity for the addition operator (left

associativity). This example illustrates the following rule:

> If an operator is on the stack and a second operator is the current input, reducing gives higher precedence to the operator on the stack, and shifting gives higher precedence to the operator that is the current input.

The g3 grammar:

```
1) E → E + a
2) E → a
```

defines the same language as the g5 grammar. Unlike the g5 grammar, its SLR(1) parser does not have any shift/reduce conflicts. As this example illustrates, one way to avoid shift/reduce conflicts is simply to use a grammar that does not have any. Unfortunately, a simple grammar that does not produce any shift/reduce conflicts is not always available. The classic example of this is the `if` statement. Here is the standard grammar for an `if` statement:

```
1) <ifStatement> → 'if' '(' <expression> ')' <statement> <elsePart>
2) <elsePart>    → 'else' <statement>
3) <elsePart>    → λ
```

With this grammar, two parse trees exist for

```
if (a) if (b) c = 1; else d = 1;
```

In one parse tree, the `else` associates with the inner `if`. In the other, the `else` associates with the outer `if`. When `else` is the current input in a bottom-up parse of this input string, a shift-reduce conflict occurs. If the parser reduces at this point, it makes the inner `if` a simple `if` statement rather than an `if-else` statement. Thus, a reduce at this point has the effect of associating the `else` with the outer `if`. If, instead, the parser shifts, the `else` associates with the inner `if`. Typically, we want the `else` to associate with the inner `if`. Thus, the parser should resolve this shift/reduce conflict in favor of a shift.

Do-Not-Reduce Rule

Consider the parse of `aa` in Fig. 22.2 that uses the g2 grammar:

```
1) S → aS       FOLLOW(S) = {#}
2) S → a
```

The parse starts this way:

```
1)    $                        aa#
2)          shift
3)    $a                       a#
```

If the parser reduces at this point with the production `S → a`, the `a` on the stack is replaced by `S`, resulting in the following configuration:

```
4)   $S                          a#
```

As we previously observed, the concatenation of the symbols on the stack with the remaining input necessarily is one of the sentential forms in rightmost derivation of the input string. Performing this concatenation on the line 4 above, we get `Sa`. If `Sa` is a sentential form, then `a` must be in the FOLLOW set of `S`. But `a` is *not* in the FOLLOW set of `S` (the FOLLOW set of `S` is {#}). Thus, reducing on line 3 above by `S → a` *cannot* be a correct step in a bottom-up parse. Generalizing from this example, we get the following *do-not-reduce rule*:

> Do not reduce by a production if the current input is not in the FOLLOW set of left side of that production.

SLR(1) Parsing

Recall that an LL(1) grammar is the preferable type of context-free grammar for top-down parsing. The first "L" indicates the grammar is for inputs that are scanned left to right. The second "L" indicates the parse uncovers the leftmost derivation of the input string. The number "1" indicates the *lookahead* is 1—that is, the parser never needs to look at more than one input symbol—the current input—to determine the correct action to perform.

LR parsers is one category of bottom-up parsers. SLR(1) parsers are the simplest type of LR parsers. The "S" in "SLR(1)" stands for "simple". The "L" indicates the grammar is for input strings that are scanned left to right. The "R" indicates the parser uncovers the rightmost derivation of the input string. The number "1" indicates the lookahead is 1.

Here is the key idea that underlies LR parsing: Whenever the parser has to push a grammatical symbol onto the stack, it pushes instead a state number that represents not only that grammatical symbol but what is currently on the stack. With this approach, the parser uses a stack that initially contains the state `0`:

```
$0      represents an empty stack
```

The state `0` represents the empty stack. Suppose the parser has to shift `a` onto the stack. It instead pushes a state that not only represents the `a` but also indicates what is below the `a` on the stack. Let's say our parser uses state 1 to indicate an `a` is on top of stack that otherwise is empty. Then at this point, it would push 1 to get

```
$01          1  represents a on top of an empty stack
```

Suppose the parser now has to shift `b` onto the stack. It would push a state (assume it is state 2) onto the stack that represents the `b` on top of an `a`:

```
$012         2  represents b on top of a
```

Suppose the parser now has to shift `c` onto the stack. It would push a state (assume it is state 3) that

represents a c on top of a b on top of an a:

```
$0123          represents c on top of b
```

Suppose the parser now has to reduce by the production

```
A → bc
```

The parser would first pop the top two symbols (these symbols—23—represent bc, the right side of the reducing production) and then push A, the left side of the production. But instead of pushing A, it would push a state (assume it is state 4) that represents A on top of state 1—that is, an A on top of an a. The stack becomes

```
$014
```

With this approach, the state of top of the stack provides the parser with the information it needs about the contents of the stack. Thus, the parser never has to look below the top of the stack. The state on the top of the stack does not have to represent the entire stack. It need represent only those aspects of the stack that the parser needs to know to make its parsing decisions. When a state is on top of the stack, we say the parser is "in that state." For example, when the stack contains

```
$014
```

we say the parser is in state 4.

To determine the stack states needed for a given grammar, we construct a finite automaton (FA) based on the grammar. The states of this finite automaton are the states the parser pushes onto the stack during a parse. Let's construct the required finite automaton for the g3 grammar:

```
1) E → E + a              g3 grammar
2) E → a
```

First, we create a new start symbol, Q, and add the production Q → E. Let's number this new production with the number 0. Our grammar becomes

```
0) Q → E
1) E → E + a
2) E → a
```

This modification does not change the language defined by the grammar.

We will label the states of the finite automaton we construct with productions from our grammar. Each production will have an embedded period that indicates the current location of the parse in the input string. We call a production with an embedded period an *item*. For example, the item

```
E → E .+ a
```

indicates that the parse is between the E and "+ a". That is, the state number for E is on top of the stack and the parser is about to process the terminal symbols "+ a". Think of the period as marking the line between the stack contents and the remaining input.

We label the start state of the finite automaton with the item

```
Q → .E
```

The period before the E indicates that the parser is about to process the terminals in the input string that correspond to E (which should be the entire input string). If the parser is about to process the terminals for E, then by production 1 (E → E + a), the parser is also about to start to process the terminals corresponding to E + a. Thus, the item Q → .E implies a second item by virtue of production 1:

```
E → .E + a
```

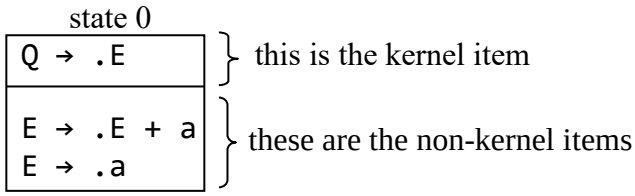
But E also can generate just an a by production 2 (E → a). Thus, a second possibility is that the parser is about to process the terminal symbol a, which means another item,

```
E → .a
```

is implied by

```
Q → .E
```

The items above together represent the possible starting configurations of the parser. So we label the starting state of our finite automaton with them. We get

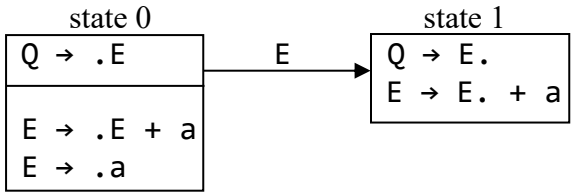


Our initial item, Q → .E, that gives rise to the other two items is called the *kernel item*.

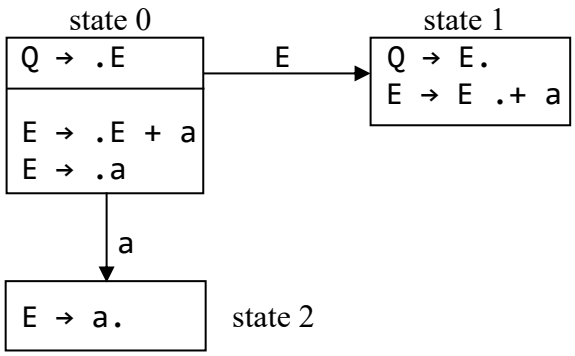
In state 0, the period abuts E on the left in the first item and second items, and an a in the third item. For each distinct symbol the period abuts on the left (E and a in this example), we create a new state. The kernel items of the new state are obtained from the items in state 0 by moving the period over one position. For example, to get state 1, we move the period over one position in Q → .E and in E → .E + a to get

```
Q → E.
E → E .+ a
```

We also draw an arrow from state 0 to state 1 labeled with E to indicate that E causes this state change. We get



In state 0, a period also abuts a on the left. So we create another new state in whose kernel item is obtained by moving the period over the a. We get



Here are the rules that govern the construction of the FA:

- If a period abuts on the left a nonterminal in a kernel item, then that kernel item gives rise to additional items in that state. For example, in state 0, a period abuts the nonterminal `E` on the left. Thus, we add all the `E` productions as items. In each added `E` production, we insert the period at the start of its right side. For example, the `E` production `E → E + a` becomes `E → .E + a`

- The number of outgoing arrows from each state is determined by the number of distinct symbols that are abutted on the left by a period. For example, in state 0, a period abuts `E` and `a`. Thus, state 0 has two outgoing arrows, one labeled with `E` and one labeled with `a`. State 2 does not have any symbols abutted on the left with a period. Thus, it has no outgoing arrows. State 1 has only one symbol (+) abutted on the left with a period. Thus, when the FA is completed, state 1 will have only one outgoing arrow.

- Let's refer to a state from which an arrow points as the "from state", and the state to which that arrow points as the "to state." The kernel items in the "to state" are derived from the items in the "from state" in which a period abuts on the left the label on the arrow. The kernel items in the new state are the items in the "from state" in which the period is abutting on the left the label on the arrow but with one modification: the period is moved to after the label on the arrow. This sounds quite complicated. But it is really straightforward. For example, an arrow labeled with `E` from state 0 points to state 1. In state 0, two items have an `E` abutted on the left with a period:

  ```
  Q → .E
  E → .E + a
  ```

  Thus, the kernel items in state 1 (the state which state 0 points to with an arrow labeled `E`) are these items but with the period moved across the `E`:

  ```
  Q → E.
  E → E .+ a
  ```

Here is the completed FA:

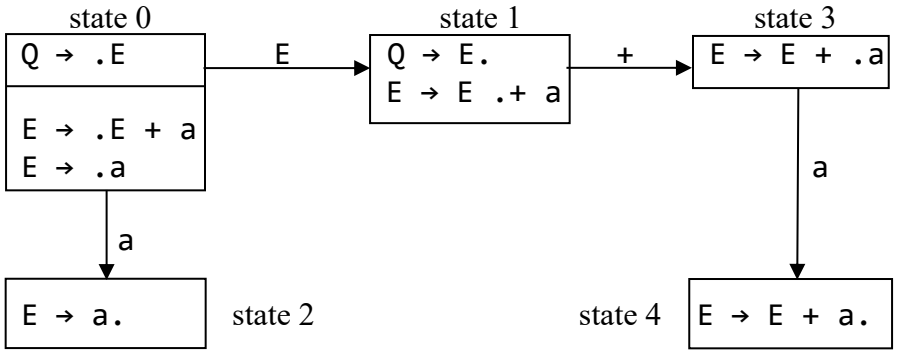


Figure 22.8

From this FA, let's determine the parsing table that specifies what the parser should do in various states. An item with the period rightmost indicates that a handle (the right side of the item) is on the stack. Thus, when in these states, the parser should reduce. For example, in state 4, the parser should perform an "`r1`" operation: that is, reduce by production 1 (`E → E + a`). By the do-no-reduce rule, the parser should reduce only when the current input is in the FOLLOW set of `E.` The FOLLOW set of `E` is {`+`, `#`}. Thus, when in state 4, the parser should reduce by production 1 but only if the current input is `+` or `#`.

State 1 calls for a reduce operation by production 0 (`Q → E.`) when the current input is in the FOLLOW set of Q (which is {`#`}). However, reduction by this production indicates the parse has successfully completed. Thus, the parser simply accepts and terminates.

For those states that have an outgoing arrow labeled with a terminal, the parser should change to the state indicated by the arrow if the current input matches the label on the arrow. For example, if the parser is in state 0 (i.e., 0 is on top of the stack) and the current input is `a`, the parser should go to state 2 (i.e., push 2 onto the stack). This operation is indicated by the arrow from state 0 to state 2 labeled with `a`.

In a parsing table, we use the letter "`s`" to represent a shift operation and the letter "`r`" to represent a reduce operation. For example, `s2` means to shift into *state* 2 (i.e., push 2 onto the stack). `r1` means to reduce by *production* 1. The left side of a parsing table specifies the shift and reduce actions. The right side specifies the state that is pushed when a reduce is performed. The table has an entry for each state that has an outgoing arrow and for each state in which an item has the period rightmost. The blank entries in a parsing table represent reject configurations. Fig. 22.9 shows the complete parsing table corresponding to the FA in Fig. 22.8:

	+	a	#	E
0		s2		1
1	s3		accept	
2	r2		r2	
3		s4		
4	r1		r1	

Figure 22.9

Let's now parse a+a using the parsing table in Fig. 22.9. We start in state 0:

```
$0              a+a#        (shift a onto stack)
```

The table tells us that in state 0 with a current input of a, the parser should perform the operation s2 (i.e, push st`ate 2 and advance in the input). We get

```
$02             +a#         (reduce by production 2)
```

The next operation is r2. So we pop the 2 (which represents the right side of production 2) and then push the state number that represents E on top of state 0. This state number (1) is given in the parsing table in the 0 row, E column. We get

```
$01             +a#         (shift + onto stack)
```

Next, the parser performs a s3 operation to get

```
$013             a#         (shift a onto stack)
```

followed by an s4 operation to get

```
$0134            #          (reduce by production 1)
```

In state 4 with # as the current input, the parser reduces with production 1. Because the right side of production 1 has three symbols, the top three symbols (4, 3, and 1) on the stack are popped. Then 1 (the state number that represents E on top of state 0) is pushed.

```
$01              #          (accept)
```

This is the accept configuration, so the parser accepts and terminates.

Shift/Reduce Conflicts

Let's now consider an SLR(1) parser for the g5 grammar:

```
1) E → E + E          g5 grammar
2) E → a
```

It defines the same language as the preceding example (a(+a)*), but it has a shift/reduce conflict. Here is its FA:

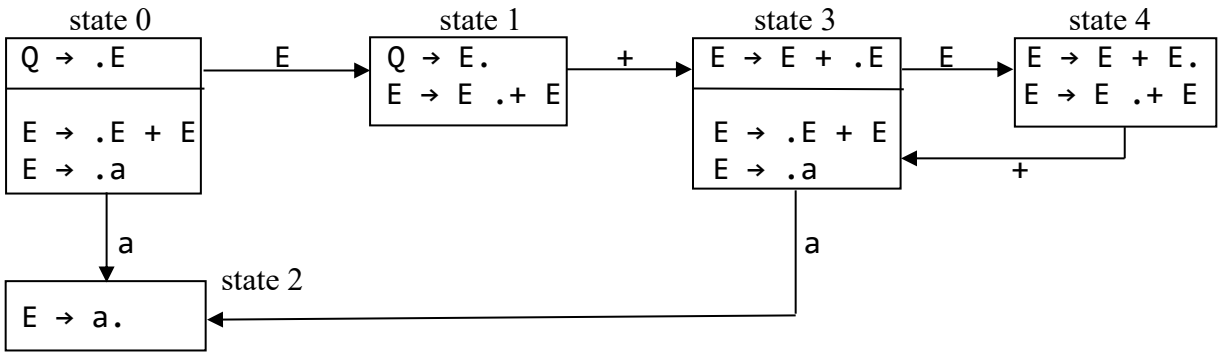


Figure 22.10

	+	a	#	E
0		s2		1
1	s3		accept	
2	r2		r2	
3		s2		4
4	s3/r1		r1	

Figure 22.11

Notice that state 4 calls for a shift if the current input is +. But it also calls for a reduce by production 1 if the current input is in the FOLLOW set of E (which is {+, #}). Thus, the parser has a shift/reduce conflict when it is in state 4 and the current input is +.

We previously learned that for the g5 grammar if the parser reduces whenever the shift/reduce conflict occurs, the parser in effect is giving the addition operator left associativity; if it shifts, the parser is giving the addition operator right associativity (see Fig. 22.6). The proper associativity for addition is left associativity. Thus, we program the parser for g5 as if the s3/r1 entry in Fig. 22.11 were just r1. But be careful when using this technique. Shift/reduce conflicts can occur for grammars that are not ambiguous. The presence of a shift/reduce conflict for a non-ambiguous grammar indicates that for some input strings in the language, a shift is needed, but for other strings in the language, a reduce is needed. Thus, if at a conflict, the parser always shifts or always reduces, there will be strings *in the language* that cannot be successfully parsed. For example, consider the following grammar:

```
1) S → aSa        g6 grammar
2) S → bSb
3) S → λ
```

Its SLR(1) parser has shift/reduce conflicts. If you force the parser to always shift or always reduce at a conflict, then the parse will fail for some strings in the language.

Reduce/Reduce Conflicts

If a parser has a choice between two reduce operations for some configuration, we say the parser has a *reduce/reduce conflict*. Consider the following grammar:

```
1) S → a          g7 grammar
2) S → AbA
3) A → a
```

The FA for SRL(1) parsing for this grammar has a state in which two reductions are called for. Here are states 0, 1, and 2:

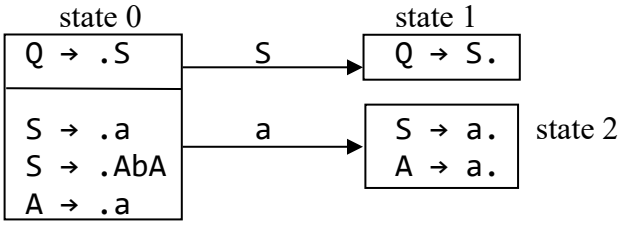


Figure 22.12

We can see from the FA that a reduce/reduce conflict occurs when the parser in state 2 and the input is #. In this configuration the parser has a choice between r1 and r3.

We can avoid the reduce/reduce conflict in the parsing table in Fig. 22.12 by using a grammar that does not produce any conflicts. Alternately we can use the original grammar with the more complex parsing table that we discuss in the next section.

If the parsing table for a grammar contains no shift/reduce or reduce/reduce conflicts, we say that the grammar is SLR(1). If at least one grammar exists for a language that is SLR(1), we say the language is SLR(1). g7 is not an SLR(1) grammar (because of the reduce/reduce conflict in its parsing table). However, the language it defines is SLR(1) because the following grammar is equivalent to g7 and its parsing table has no conflicts:

```
1) S → aA         grammar g8
2) A → ba
3) A → λ
```

LR(1) Parsing

The LR(1) parsing algorithm is the same as the SLR(1) parsing algorithm. The difference between an LR(1) parser and an SLR(1) parser is in the construction of FA and its corresponding parsing table. The items used in the FA for an LR(1) parser contains *two* components: a production with an embedded period and one or more lookaheads. For example, in the following LR(1) item,

```
A → B.CD, a/b
```

a and b are the lookaheads. The lookaheads in an LR(1) item indicate the possible inputs that will be current at that point later in the parse when the states representing the right side of the production are on

the stack, ready to be reduced. For example, the lookaheads in the preceding LR(1) item indicate that when the states corresponding to B, C, and D have been pushed onto the stack, the current input should be either a or b. Thus, at that point the parse should reduce but only if the current input is a or b.

The states in a LR(1) parser carry more information that the states in an SLR(1) parser. Because the top of the stack provides more information on the state of the parse, a LR(1) in general is better at determining the parser actions. It is oftentimes the case that a conflict that occurs in an SLR(1) parser does not occur in a LR(1) parser.

Let's construct the LR(1) FA and parsing table for the following grammar:

```
1) S → Sa          g9 grammar
2) S → b
```

First, we add a new start state Q and the production Q → S to the grammar to get

```
0) Q → S
1) S → Sa
2) S → b
```

The S in production 0 should generate the input string. Thus, after the parser has processed the string that this S generates, the current input should be the end-of-input marker #. Accordingly, the lookahead is the end-of-input marker #:

```
Q → .S, #
```

Because the period abuts the S on the left in this kernel item, it gives rise to the items

```
S → .Sa, #
S → .b, #
```

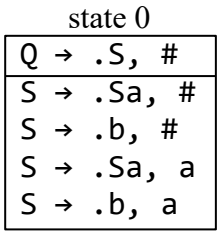
The S on the left side of these two items is the initial S. Thus, after the parser processes the input string that is generated by their right sides, the current input should be #. So the lookahead for both items is also #. The item

```
S → .Sa, #
```

gives rise to two more items because the period it abutting the S on the left. But observe that the S on the right side of this item is not the initial S. After the parser processes the string it generates, *the current input should be* a (because this S is followed immediately by a). Thus, the lookahead for the items with this S on the left side have the lookahead a:

```
S → .Sa, a
S → .b, a
```

Thus, the start state consists of all these items:



Continuing in this fashion, we can construct the FA (see Fig. 22.13) from which we can then construct the parsing table (see Fig. 22.14). But in place of the do-not-reduce rule, we use the lookaheads to determine for which inputs a reduce operation should be performed. For example, the items in state 2 indicate the parse should reduce by the production 2 but only if the current input is # or a.

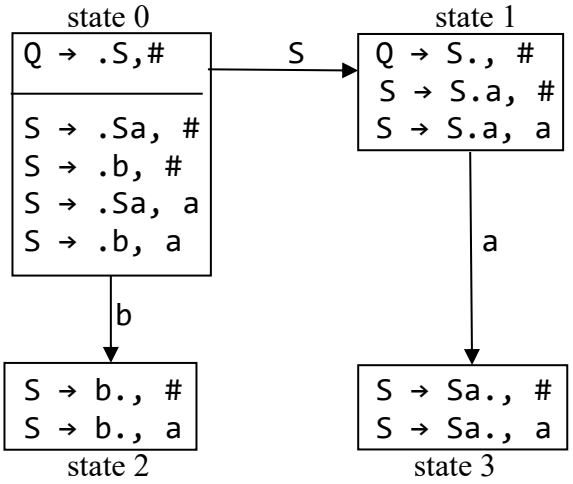
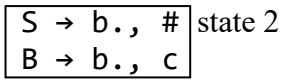


Figure 22.13

	a	b	#	S
0		s2		1
1	s3		accept	
2	r2		r2	
3	r1		r1	

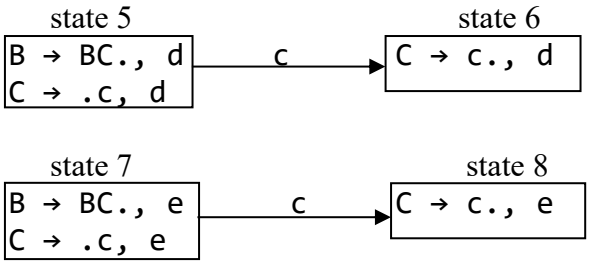
Figure 22.14

An LR(1) grammar is a grammar whose LR(1) parsing table contains no conflicts. LR(1) parsing is more powerful than SLR(1) parsing. That is, all SLR(1) grammars are also LR(1), but not all LR(1) grammars are SLR(1). In other words, if SLR(1) parsing works (i.e., no conflicts) then so will LR(1) parsing. But for some grammars LR(1) parsing works but not SLR(1) parsing. As an example of the latter, consider g7 whose partial parsing table is in Fig. 22.12. A reduce/reduce conflict occurs in state 2. But the LR(1) parsing table for g7 does not have any conflicts. Here is its state 2:

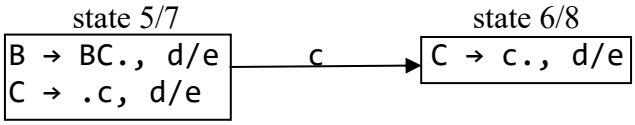


Notice that the items in state 2 have different lookaheads. When in this state, the parser reduces by `S → b` if the current input is `#`, or by `B → b` if the current input is `c`. The lookaheads have eliminated the reduce/reduce conflict that is present in the SLR(1) parser for the same grammar. Thus, this grammar is LR(1) but not SLR(1).

One disadvantage of LR(1) parsing is that the number of states can be excessive, resulting in a very large parsing table. However, generally many of the states for an LR(1) parser can be merged, reducing significantly the total number states. The resulting parser is called an LALR(1) parser to distinguish it from a regular LR(1) parser (the "LA" in "LALR" stands for "lookahead"). For example, suppose a LR(1) parser has the following states, among others:



Notice the first components of the items in state 5 match the first components of the items in state 7. We have a similar match between states 6 and 8. Thus, we can merge states 5 and 7, and states 6 and 8, combining the lookahead inputs. We get a parser that works the same as the LR(1) parser but has two fewer states:



yacc Input File

`yacc` is a LALR(1) *parser generator*. It inputs a grammar and outputs the programming code that implements a parser for that grammar. Most `yacc` versions generate C or C++ code. We will use the Berkeley version of `yacc` that generates C code and runs on Windows. As of this writing, it and its documentation are available at

http://gnuwin32.sourceforge.net/packages/byacc.htm

The letters in `yacc` stand for "yet another compiler compiler"—a likely indication that the original

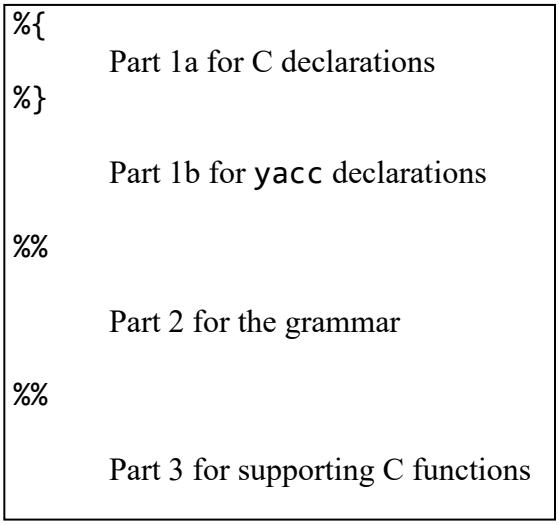
implementor went through many preliminary versions.

You can also use `bison`, a yacc-compatible (more or less) parser generator. Another parser generator, `ply`, is oriented to python. It is quite different from `yacc`. For example, it does not output code. The `ply` file that contains the specification of the grammar is itself the python program that does the parsing based on that grammar. For most applications, `yacc`/`bison` is more appropriate than `ply` since the former generates C code which compiles to efficient executable standalone code.

A parser generator outputs code that implements a parser. Similarly, a *lexer generator* outputs code that implements a *lexer* (i.e., tokenizer). Two popular (and compatible) lexer generators, `lex` and `flex`, will generate the C code for a lexer when provided with the specification of the tokens (in the form of regular expressions). Unlike LALR(1) parsers, lexers are easy to implement by hand. For this reason, in the examples that follow, the lexers are handwritten.

yacc Input and Output Files

A `yacc` input file consists of three parts:



`yacc` input file

Dividing each part from the next is line starting with `%%`. Part 1 consists of two subparts. Subpart 1a contains C code bracketed with "`%{`" and "`%}`". This code is carried over as is to the output file. Subpart 1b contains `yacc` declarations.

It is common practice to use the extension "`.y`" for `yacc` input files, although it is not required.. The default output file name is `y.tab.c`. The function `yyparse()` is the parser function created by `yacc` that is included in `y.tab.c`.

`yyparse()` expects two functions to be available that `yacc` does not generate: `yylex()` (the lexer) and `yyerror()` (the function `yyparse()` calls on an error). If we place `yylex()` and `yyerror()` in part 3 of the `yacc` input file along with a C `main` functioon, then the output file created by `yacc` will be a complete C program.

For each token, `yylex()` returns to `yyparse()` the number than indicates the category of the token. To do this, it uses the C `return` statement. For each single character token that has its own unique category (like '`+`'), `yylex()` returns its ASCII code as its category number. For a category like `DIGIT` that represents multiple tokens (0, 1, 2, …, or 9), `yylex()` returns an arbitrary number determined by

`yacc` that represents that category. `yacc` uses numbers greater than 256 for these types of tokens to avoid conflicts with ASCII codes (there are no ASCII codes greater than 256). For example, for a token that is a digit, `yacc` returns 257; for '`+`', `yacc` returns 43 decimal (its ASCII code).

To return the value (as opposed to its category) of a token, `yylex()` assigns the value to the variable `yylval`. Not all tokens have values. For example, '`+`' has no value, but the digit token '5' has the value 5. The declaration for `yylval` is in the code generated by `yacc`. Thus, a declaration for `yylval` should *not* appears in the `yacc` input file. However, to return other properties of a token (e.g., token's lexeme, line number, or column number), `yylex` should use global variables (such as `yytext`) that are declared in part 1a of the `yacc` input file.

To summarize:

- `yylex()` should return the category of a token using the C `return` statement.
- `yylex()` should assign the value of a token (if it has one) to `yylval`.
- To return other properties of a token to `yyparse()`, use global variables declared in part 1a of the `yacc` input file.

Part 2 of a yacc Input File

The `yacc` input file `g3.y` in Fig. 22.15 contains only part 2. It specifies the g3 grammar which we examined earlier.

g3.y

```
%%
E:    E '+' 'a'
    | 'a'
    ;
```

Figure 22.15

The colon (`:`) takes the place of the arrow that is typically used in grammars. The quoted items (`'a'`) are the terminal symbols. The vertical bar (`|`) is the OR operator we use to specify multiple right sides of productions with the same left side. The semicolon (`;`) ends the specification of the `E` productions. The remaining items are the nonterminals. This form of a content-free grammar is call *Backus-Naur Form* (BNF).

To process `g3.y` with `yacc`, enter on the command line

```
yacc -v g3.y
```

Because of the `-v` argument, `yacc` outputs not only the `y.tab.c` file but also the file `y.output`. The former is the C code for the parser; the latter is a representation of the FA constructed by `yacc` from which it determines the parsing table (see Fig. 22.16).

The file `y.output` starts with a listing of the g3 grammar to which a production has been added (production 0):

```
0  $accept : E $end        new start symbol is  $accept, end-of-input marker is  $end
1  E : E '+' 'a'
2    | 'a'
```

`$accept` is the new start symbol (we used `Q` in are previous examples). `$end` represents the end-of-input marker.

```
state 0
        $accept : . E $end  (0)
        'a'  shift 1
        .  error
        E  goto 2

state 1
        E : 'a' .  (2)
        .  reduce 2

state 2
        $accept : E . $end  (0)
        E : E . '+' 'a'  (1)
        $end  accept
        '+'  shift 3
        .  error

state 3
        E : E '+' . 'a'  (1)
        'a'  shift 4
        .  error

state 4
        E : E '+' 'a' .  (1
        .  reduce 1

4 terminals, 2 nonterminals
3 grammar rules, 5 states
```

`(0)` indicates the production number is 0

"`.`" is any other character

Figure 22.16

For each state, `y.output` shows only the kernel items. For example, in state 0 the kernel item is

```
$accept : . E $end
```

The items implied by this kernel item,

```
E : .E '+' 'a'
E : .'a'
```

are omitted.

In addition to the kernel item, for each state the actions to be taken when in that state are listed. For example, for state 2 the actions listed are

```
$end  accept      Accept if at end-of-input marker.
'+'  shift 3      Shift 3 onto the stack if the current input is '+' and advance in the input.
.  error          '.' represents any input other than previous choices.
```

The parser for g3 has no shift/reduce conflicts. But the equivalent g5 grammar does:

g5.y

```
%%
E:   E '+' E
     | 'a'
     ;
```

Figure 22.17

For state 4 corresponding to `g5.y`, the `y.output` file shows

```
4: shift/reduce conflict (shift 3, reduce 1) on '+'
state 4
        E : E . '+' E  (1)
        E : E '+' E .  (1)
        '+'  shift 3
        $end  reduce 1

State 4 contains 1 shift/reduce conflict.
```

A number within parentheses is the production number for that item. For example, "`(1)`" indicates the corresponding item is from production 1. The item

```
        E : E '+' E .
```

calls for a reduce operation. By the do-not-reduce rule, the reduce should be performed when the current input is '+' or #. But the other item calls for a shift if the current is '+'. Thus, when in state 4 and the current input is '+', both a shift and a reduce are called for. Note that the action listed for '+' is to shift. This is the default action the parser takes when a shift/reduce conflict occurs.

Simple Interpreter Created by yacc

Now that we understand the basic structure of a `yacc` input file, let's examine an example that implements a simple interpreter for the language defined by the following grammar in BNF form:

```
Q: E                    g10 grammar
   ;
E: E '+' E
   | E '*' E
   | DIGIT
   ;
```

Actions can be embedded in the productions in part 2 of a `yacc` input file. Actions are enclosed in braces. For example, here is the first production in the g10 grammar with an embedded action:

```
Q: E  {printf("%d\n", $1);} // display expr value
```

The embedded action in this example is a C `printf` statement. This action is executed (i.e., the `printf` is executed) when a reduce by this production is performed by the parser.

Actions do not have to be rightmost in a production. For example, a production of the following form is permitted:

```
A: B {action} C
```

For this example, the action is performed right after the reduce operation that pushes B onto the stack occurs and before the string produced by C is parsed. Grammars that have embedded actions are called *attribute grammars*.

The `yacc` input file for the g10 grammar is in `g10.y` (see Fig. 22.18). To convert the `g10.y` to an executable interpreter, enter on the command line

```
yacc g10.y
```

to create the C program `y.tab.c`. Next, compile `y.tab.c` by entering

```
gcc y.tab.c -o g10
```

Then execute the compiled program in `g10` by entering `g10` and the input string (an arithmetic expression) on the command line:

```
g10 2+3*5
```

The `g10` program will respond by computing and displaying the value of the expression entered:

```
17
```

Let's now examine in detail the `yacc` input file for the g10 grammar in Fig. 22.18.

`g10.y` calls several C library functions: `printf` (line 14), `strcpy` (24), and `isdigit` (30). The C compiler requires that for each of these functions the appropriate "`.h`" file be included in the program. Lines 2, 3, and 4 include these files. For a reference on these C library functions, see `cfunctions.txt` in the software package.

The input string is provided to the `g10` program via the command line via the `argv[1]` parameter in the `main` function. To tokenize the input string, the `yylex()` function obviously has to access the input string. But `argv[1]` is local to the `main` function. Thus, `yylex()` cannot directly access it. To make the input string available to `yylex()`, the `main` function copies the input string accessed with `argv[1]` to

the character array `input` (line 24). The array `input` is declared on line 5. Because its declaration is outside any function, it has global scope. Thus, `yylex()` can access it (lines 30, 32, and 35).

Whenever the C compiler compiles a function call, it checks that the arguments in the call are consistent with the parameters in the function definition. To do this, either the function definition or the function prototype must *precede* the function call. A function prototype is essentially the first line of the function definition (which lists the parameters). Line 6 contains the prototype for the `yyparse()` function. It allows the compiler to check if the arguments in the call of `yyparse()` on line 25 are consistent with parameters specified by its prototype.

The token declaration of line 9 indicates that `DIGIT` is a token—not a nonterminal. Without this declaration, `DIGIT` would be treated as a nonterminal on line 18.

Lines 10 and 11 indicate that '`+`' and '`*`' are left associative operators. Moreover, because the entry for '`*`' is listed *after* the entry for '`+`', these two lines indicate that '`*`' has higher precedence than '`+`'. Without these two lines, the grammar would have four shift/reduce conflicts. These conflicts occur during the parse when an operator is on the stack and the current input is also an operator. Here are the specific configurations in which a shift/reduce conflict would occur (*Note*: For clarity's sake, we are indicating what is on the stack with grammatical symbols rather than the state numbers that are actually there):

	stack	current input	action
1)	`$...E+E`	`+`	reduce
2)	`$...E+E`	`*`	shift
3)	`$...E*E`	`+`	reduce
4)	`$...E*E`	`*`	reduce

```
 1 %{
 2 #include <stdio.h>   // for printf()
 3 #include <string.h>  // for strcpy()
 4 #include <ctype.h>   // for isdigit
 5 char input[80];      // global array that holds the input string
 6 int yyparse();       // prototype
 7 %}
 8
 9 %token DIGIT   // indictes DIGIT is a token
10 %left '+'      // indicates + and * are left associative and
11 %left '*'      // * has higher precedence than +
12
13 %%
14 Q: E          {printf("%d\n", $1);} // display expr value
15    ;                                 // indicates end of Q productions
16 E: E '+' E    {$$ = $1 + $3;}       // assign left side the sum
17    | E '*' E  {$$ = $1 * $3;}       // assign left side the product
18    | DIGIT    {$$ = $1;}            // assign left side the right
19    ;                                 // indicates end of E productions
20
21 %%
22 int main(int argc, char *argv[])
23 {
24    strcpy(input, argv[1]);         // copy argv[1] to input array
25    return yyparse();               // call  yyparse()
26 }
```

```
27 int yylex()
28 {
29    static int i = 0;               // i retains value between calls
30    if (isdigit(input[i]))
31    {
32       yylval = input[i++] - '0';   // return in value via yylval
33       return DIGIT;                // return category
34    }
35    return input[i++];              // return ASCII code
36 }
37 void yyerror(char *p)
38 {
39    printf("%s\n", p);              // displays error message
40 }
```

Figure 22.18 `g10.y` (with line numbers added)

Recall when an operator in on the stack and a second operator is the current input, reducing gives precedence to the operator on the stack, and shifting gives precedence to the operator that is the current input (see Fig. 22.7 and the related discussion). Because lines 10 and 11 provide the associativity and precedence of the '+' and '*' operators, `yacc` can determine the correct action to take for the four configurations given above that otherwise would be shift/reduce conflicts. Try processing `g10.y` with `yacc` without lines 10 and 11. `yacc` will warn you of the shift/reduce conflicts. By default, the parser created by `yacc` will shift for all four configurations, thereby giving '+' and '*' equal precedence and right associativity. Thus, the expression 2*3+5 would evaluate to 16 (2*[3+5] = 16) rather than to the correct value 11 ([2*3]+5 = 11).

Line 25 calls `yyparse()`. On completion, `yyparse()` returns a return code to `main`. The `return` statement on line 25 returns this return code to startup code, which in turn returns it to the operating system.

Line 32 determines the integer value of a digit by subtracting the ASCII code for '0' from the ASCII code for the digit. For example, the ASCII code for '7' is 37 hex. The ASCII code for '0' is 30 hex. Thus, '7' – '0' = 37 - 30 = 7. The value computed on line 32 is assigned to the parser variable `yylval`. For a digit token, `yylex()` returns the digit's value via `yylval` and its category number via the C `return` statement. `yacc` assigns to the `DIGIT` token and arbitrary integer greater than 256 to represent its category (in this example, 257 decimal represents the DIGIT category). `yylex()` uses a `return` statement to return the category number (line 33).

When `yyparse()` pushes a state number that corresponds to token or a nonterminal, it pushes the current value in `yylval` on a separate stack. Thus, `yyparse()` handles two stacks during a parse: the *parsing stack* (for the state numbers) and the *value stack*. The value of each item on the parsing stack is in the corresponding slot in the value stack. For example, suppose the input string is 1+7 + 5. When the parser has processed the 1, +, and 7 and is about to reduce by the production

```
E: E '+' E
```

the two stacks look like this:

```
                E+E ◄——— grammatical symbols corresponding to 346
parsing stack: $0346
value stack:   $x1x7 ◄——— values of E (1), + (none), and E (7)
```

The 3 and 6 on the parsing stack are the states corresponding to the two `E`'s on the right side of the reducing production. The 4 is the state number corresponding to the '+'. The `x` indicates the corresponding parsing stack item has no value ('+' and state 0 have no values). In the grammar in part 2 of the `yacc` input file (see line 16 in Fig. 22.18), we access the two values in the value stack at this point in the parse with `$1` and `$3`. `$1` accesses the value of the first item on the right side of the production. `$3` accesses the third item on the right side of the reducing production. The reduce operation pops 346 off the parsing stack and then pushes an `E` (represented by the state number 3). The parsing stack becomes

```
parsing stack: $03
```

But the reducing operation also pops the corresponding three items from the value stack (7, x, 1). It then performs the operation specified in the action given in the reducing production (line 16 in Fig. 22.18):

```
E: E '+' E    {$$ = $1 + $3;}
```

The action appears within the braces to the right of the production. It is performed by the parser when it reduces by this production. It indicates that the values accessed by `$1` and `$3` (1 and 7) should be added. `$$` in the action represent the left side of the production. The assignment statement assigns 1+7 = 8 to the `E` on the left side of the production. More precisely, the 8 is the value pushed onto the value stack that corresponds to the `E` (represented by the state 3) pushed onto the parsing stack during the reduce operation. After the reduce operation, the parsing and value stacks look like this:

```
parsing stack: $03
value stack:   $x8
```

As the input is parsed, the values of the subcomponents of the expression are computed and in effect passed up the parse tree during the reduce operations. When the reduce operation for the first production

```
Q: E          {printf("%d\n", $1);}
```

is about to occur, the value associated with the `E` on the right of this production is the value of the entire expression. The action performed displays this value.

Let's now extend the interpreter example in Fig. 22.18 so that it supports the assignment and print statements and multicharacter integer and identifier tokens. In addition, the input string is obtained from a file.

Part 2 of the `yacc` file appears on lines 37 to 85 in Fig. 22.19. The variable `yytext` appears on lines 55 and 84. It is a global variable declared on line 12 that `yylex()` uses to provide the parser with the lexeme of the current token. For example, for each of the six tokens in the assignment statement

```
total = a + 123
```

`yylex()` returns the lexeme (i.e., the token itself) via `yytext` and category of the token via the C `return` statement:

token	lexeme returned via `yytext`	category returned via C `return` statement
`total`	"`total`"	ID (257)
`=`	"`=`"	ASCII code for "`=`"
`a`	"`a`"	ID (257)
`+`	"`+`"	ASCII code for "`+`"
`123`	"`123`"	UNSIGNED (258)
`newline`	"`\n`"	ASCII code for "`\n`"

For a single character token that is neither an unsigned integer nor an identifier, `yylex()` treats its ASCII code as its category number, and returns it to the parser via a C `return` statement. ID and UNSIGNED represent token categories (see lines 30 and 31 in Fig. 22.19). `yacc` defines them as the integers greater than 256 so they will not conflict with the categories that are ASCII codes (ASCII codes are all less than 257).

Values of identifiers are stored in a symbol table, which consists of two parallel arrays: a character pointer array `symtab` and an integer array `symval`. When an assignment statement assigns a value to an identifier, the pointer to the identifier is entered into `symtab`, and the value is entered into the corresponding slot in `symval`.

Here is the definition of the assignment production:

```
53 assignmentStatement:
54    ID
55    { ii = enter(yytext); }  // enter lexeme into symbol table
56    '='
57    expr {symval[ii] = $4; } // store its value in symbol table
58    '\n'
59 ;
```

On line 55, the `enter()` function enters the lexeme in `yytext` (which is the left side of the assignment statement) into a slot of the `symtab` array. It returns the index of the slot that is used, which is assigned to `ii`. On line 57, the value of the right side of the assignment statement (accessed by `$4`) is stored in `symval` at the index given by `ii`.

When an identifier appears as a factor in an expression, the call of the `getValue()` function on line 83 determines the index of its slot in `symtab` using the identifier's lexeme in `yytext`.

```
83    ID { $$ = getValue(yytext); }
```

It then accesses the corresponding slot in `symval` to gets the identifier's value, which it assigns to `$$`. The effect is to pass the value of the identifier up the parse tree.

```
 1 // ======================================== part 1a
 2 %{
 3 // include files
 4 #include <stdio.h>  // for I/O functions
 5 #include <string.h> // for strcmp(), strdup()
 6 #include <ctype.h>  // for isdigit(), isalpha()
 7 #include <stdlib.h> // for exit()
 8
 9 // global variables
10 char *symtab[100];      // symbol table
11 int symval[100];        // symbol table
```

```
12 char yytext[100];       // holds lexeme
13 int currentChar = -1;   // forces read of first line of source program
14 int currentColumn;
15 int currentLine;
16 int i, ii;
17 int nextIndex;   // index of next available slot in symbol table
18 FILE* inFile;    // file pointer for input file
19
20 // function prototypes
21 int yyparse();
22 int getNextChar();
23 int yylex();
24 int enter(char *);
25 int getValue(char *);
26 void yyerror(char *);
27 %}
28
29 // ========================================== part 1b
30 %token ID
31 %token UNSIGNED
32 %token PRINT
33 %left '+'  // indicates left associative
34 %left '*'  // indicates * higher precedence than +
35
36 %%  // ======================================== part 2
37 program:
38    statementList
39 ;
40 //------------------------------
41 statementList:
42    statementList statement
43  |
44    statement
45 ;
46 //------------------------------
47 statement:
48    assignmentStatement
49  |
50    printStatement
51 ;
52 //------------------------------
53 assignmentStatement:
54    ID
55    { ii = enter(yytext); }  // enter lexeme into symbol table
56    '='
57    expr {symval[ii] = $4; } // store its value in symbol table
58    '\n'
59 ;
60 //------------------------------
61 printStatement:
62    PRINT
63    '('
64    expr { printf("%d\n", $3); }
65    ')'
66    '\n'
67 ;
68 //------------------------------
69 expr:
70    expr '+' expr { $$ = $1 + $3; }
```

```
 71  |
 72    expr '*' expr { $$ = $1 * $3; }
 73  |
 74    factor { $$ = $1; }
 75 ;
 76 //------------------------------
 77 factor:
 78    '+' factor { $$ = $2; }
 79  |
 80    '-' factor { $$ = -$2; }
 81  |
 82    UNSIGNED { $$ = $1; }
 83  |
 84    ID { $$ = getValue(yytext); } // get value of lexeme in yytext
 85 ;
 86
 87 %%  // ==================================== part 3
 88 int main(int argc, char *argv[])
 89 {
 90   // open file whose name given on command line
 91   inFile = fopen(argv[1], "r");  // open for input
 92   return yyparse();
 93 }
 94 //-----------------------------------
 95 // enter or find ID in symbol table, return index
 96 int enter(char *p) // p points to ID
 97 {
 98    i = 0;
 99    while (i < nextIndex && strcmp(p, symtab[i]))
100       i++;
101    // i < nextIndex if ID in symbol table
102    if (i < nextIndex)
103       return i;
104    // add ID to symbol table
105    // strdup allocates memory for ID
106    // and copies ID there
107    symtab[nextIndex] = strdup(p);
108    return nextIndex++;
109 }
110 //-----------------------------------
111 // returns value of ID
112 int getValue(char *p)
113 {
114    i = 0;
115    // search for ID
116    while (i < nextIndex && strcmp(p, symtab[i]))
117       i++;
118    if (i < nextIndex)
119       return symval[i];   // found it at index i
120    // symbol not in symbol table so no value
121    printf("name '%s' on line %d column %d is not defined\n",
122            p, currentLine, currentColumn);
123    exit(1);  // terminate the program
124 }
125 //-----------------------------------
126 // returns next character on current line
127 int getNextChar()
128 {
129   // input retains values
```

```
130   // between calls of getNextChar()
131   // because of keyword static
132   static char input[100];
133
134   if (currentChar == '\n' || currentChar == -1)  // need next line?
135   {
136     if (fgets(input, sizeof(input), inFile))     // read line into input array
137     {
138       currentLine++;       // update
139       currentColumn = 0;   // reset for new line
140     }
141     else  // at EOF
142     {
143       return 0;   // 0 represents EOF
144     }
145   }
146   return input[currentColumn++]; // return next char from input array
147 }
148 //-----------------------------------
149 // lexer (i.e., tokenizer)
150 int yylex()
151 {
152   int category;
153
154   // skip space and tab
155   while (currentChar == ' ' || currentChar == '\t'
156          || currentChar == -1 )  // -1 forces read of first line of source code
157     currentChar = getNextChar();
158
159   // check for EOF
160   if (currentChar == 0)
161   {
162     strcpy(yytext, "<EOF>");     // give EOF a lexeme
163     category = 0;  // 0 represents EOF
164   }
165
166   else  // check for unsigned int
167   if (isdigit(currentChar))
168   {
169     i = 0;
170     do  // store lexeme in yytext
171     {
172       yytext[i++] = currentChar;
173       currentChar = getNextChar();
174     } while (isdigit(currentChar));
175     yytext[i] = '\0';
176     // convert ASCII string in yytest to int
177     sscanf(yytext, "%d", &yylval);
178     category = UNSIGNED;
179   }
180
181   else  // check for identifier
182   if (isalpha(currentChar) || currentChar == '_')
183   {
184     i = 0;
185     do  // store lexeme in yytext
186     {
187       yytext[i++] = currentChar;
188       currentChar = getNextChar();
```

```
189     } while (isalpha(currentChar) || currentChar == '_' || isdigit(currentChar));
190     yytext[i] = '\0';  // terminate string with null char
191     // check if keyword
192     if (!strcmp(yytext, "print"))
193       category = PRINT;
194     else  // not a keyword so category is ID
195       category = ID;
196   }
197
198   else  // do this if preceding cases do not apply
199   {
200     // use character itself as its category value
201     category = currentChar;
202     yytext[0] = currentChar;      // make character a string in yytext
203     yytext[1] = '\0';             // terminate string with null char
204     currentChar = getNextChar();  // always read one char beyond end of token
205   }
206
207   return category;
208 }
209 //-----------------------------------
210 void yyerror(char *p)
211 {
212    printf("%s %s '%s' %s %d %s %d\n", p, "on", yytext,
213           "on line", currentLine, "column", currentColumn);
214 }
```

Figure 22.19 `g11.y` (with line numbers added)

To determine the end of an identifier or unsigned integer token, `yylex()` has to read beyond the end of the token. For example, when processing an unsigned integer token. `yylex()` reads characters until it reads a character that is not a digit (see line 174 in Fig. 22.19). Thus, on the next call of `yylex()`, `currentChar` already has the first character that follows the unsigned integer token. To keep the logic of `yylex()` as simple as possible, for all tokens `yylex()` reads one character beyond the end of each token it processes (see lines 174, 189, and 204).

When the parser parses an unsigned integer, it in effect passes its value up the parse tree (see line 82). If the parent node generates the unsigned integer preceded by a unary minus, then the value is negated before it in turn is passed up the tree (see line 80). This process is repeated for each unary minus that precedes the unsigned integer. Thus, the value that the `expr` node ultimately receives is the value of the unsigned integer negated once for each unary minus that precedes it.

To create an executable program from `g11.y`, enter

```
yacc g11.y              (creates y.tab.c)
gcc y.tab.c -o g11      (compiles y.tab.c and creates g11.exe)
```

Let's now use our `g11` program to interpret `g11.in`:

g11.in

```
sum = ---30 + +50
print(sum)
```

To do this, enter

```
g11 g11.in
```

`g11` responds by computing and displaying the value of `sum`:

```
20
```

Problems

1. Construct the SLR(1) FA and the parsing table for g1. Show the parse of `aaa`.

2. Construct the SLR(1) FA and the parsing table for g2. Show the parse of `aaa`.

3. Construct the SLR(1) FA and the parsing table for g4. Show the parse of `a+a`.

4. Construct the SLR(1) FA and the parsing table for g6. Show the parse of `abba`.

5. Construct the SLR(1) FA and the parsing table for g7. Show the parse of `a` and `aba`.

6. Construct the SLR(1) FA and the parsing table for g8. Show the parse of `a` and `aba`.

7. Construct the SLR(1) FA and the parsing table for the following grammar:

   ```
   E → E + T
   E → T
   T → T * F
   T → F
   F → a
   F → ( E )
   ```

   Show the parse of `(a+a)*a`.

8. Implement *by hand* (i.e., do not use `yacc`) the SLR(1) parser for g3. Test your program with `2`, `2+3`, and `2+3+4`. Your parser should evaluate and display the value of the expression entered. Get the expression to process from the command line.

9. Construct the SLR(1) parser for

   ```
   S → a
   S → Sb
   S → Bcd
   B → b
   ```

   Are there any conflicts? Construct the LR(1) parser for the grammar. Are there any conflicts? Is this grammar SLR(1)? Is it LR(1)?

10. Show that the merging of states in the process of constructing a LALR(1) parser never creates a shift/reduce conflict.

11. Are LR(1) grammars ever ambiguous? Justify your answer.

12. Create `g12.y` by extending `g11.y` so that is supports division and strings. Test your implementation with `g12.in`.

13. Create `g13.y` by extending `g11.y` so that it supports comments. Test your implementation with `g12.in`.

14. What is accessed on the value stack by `$0`? What is accessed on the value stack by `$-1`?

15. The grammar in part 2 of a `yacc` file permits the passing of values up the parse tree. For example, with embedded actions like

```
{$$ = $1 + $3;}
```

    How are values passed down the parse tree? *Hint*: See problem 14.

16. Using `yacc`, implement the `i1` interpreter.

17. Using `yacc`, implement the `h1` interpreter.

18. Using `yacc`, implement the `c1` compiler.

19. By examining `y.output`, determine the grammar the `yacc`-generated parser uses for a production that has the following format:

```
A: B {action} C
```

20. For which strings in the language defined by the grammar will the SLR(1) parser for g6 fail if the parser aways shifts at a shift/reduce conflict? If the parser always reduces at a shift/reduce conflict?

21. What is actually in the value stack corresponding to a token like '`+`'. Is it 0? Is it whatever happened to be `yylval` when the '`+`' was pushed onto the stack?

22. Construct the SLR(1) FA and parsing table for the following grammar:

```
S → aS
S → λ
```

    Show the parse of `aaa`.

23. Construct the LR(1) FA and the LALR(1) FA for

```
S → AA
B → Ab
B → cC
C → Cd
C → e
```

Index