1 $P \rightarrow L$

2 $S \rightarrow$ id := id
3 $S \rightarrow$ while id do S
4 $S \rightarrow$ begin L end
5 $S \rightarrow$ if id then S
6 $S \rightarrow$ if id then S else S

7 $L \rightarrow S$
8 $L \rightarrow L\ ;\ S$

GRAMMAR 3.30.

well understood. Most shift-reduce conflicts, and probably all reduce-reduce conflicts, should not be resolved by fiddling with the parsing table. They are symptoms of an ill-specified grammar, and they should be resolved by eliminating ambiguities.

3.4 USING PARSER GENERATORS

The task of constructing LR(1) or LALR(1) parsing tables is simple enough to be automated. And it is so tedious to do by hand that LR parsing for realistic grammars is rarely done except using parser-generator tools. Yacc ("Yet another compiler-compiler") is a classic and widely used parser generator; *Bison* and *occs* are more recent implementations.

A Yacc specification is divided into three sections, separated by `%%` marks:

parser declarations
`%%`
grammar rules
`%%`
programs

The *parser declarations* include a list of the terminal symbols, nonterminals, and so on. The *programs* are ordinary C code usable from the semantic actions embedded in the earlier sections.

The *grammar rules* are productions of the form

```
exp :    exp PLUS exp    { semantic action }
```

where `exp` is a nonterminal producing a right-hand side of `exp+exp`, and `PLUS` is a terminal symbol (token). The *semantic action* is written in ordinary C and will be executed whenever the parser reduces using this rule.

```
%{
int yylex(void);
void yyerror(char *s) { EM_error(EM_tokPos, "%s", s); }
%}
%token ID | WHILE | BEGIN | END | DO | IF | THEN | ELSE | SEMI | ASSIGN
%start prog
%%

prog: stmlist

stm : ID ASSIGN ID
    | WHILE ID DO stm
    | BEGIN stmlist END
    | IF ID THEN stm
    | IF ID THEN stm ELSE stm

stmlist : stm
        | stmlist SEMI stm
```

GRAMMAR 3.31. Yacc version of Grammar 3.30. Semantic actions are omitted and will be discussed in Chapter 4.

Consider Grammar 3.30. It can be encoded in Yacc as shown in Grammar 3.31. The Yacc manual gives a complete explanation of the directives in a grammar specification; in this grammar, the terminal symbols are `ID`, `WHILE`, etc.; the nonterminals are `prog`, `stm`, `stmlist`; and the grammar's start symbol is `prog`.

CONFLICTS

Yacc reports shift-reduce and reduce-reduce conflicts. A shift-reduce conflict is a choice between shifting and reducing; a reduce-reduce conflict is a choice of reducing by two different rules. By default, Yacc resolves shift-reduce conflicts by shifting, and reduce-reduce conflicts by using the rule that appears earlier in the grammar.

Yacc will report that this Grammar 3.30 has a shift-reduce conflict. Any conflict is cause for concern, because it may indicate that the parse will not be as the grammar-designer expected. The conflict can be examined by reading the verbose description file that Yacc produces. Figure 3.32 shows this file.

A brief examination of state 17 reveals that the conflict is caused by the familiar dangling `else`. Since Yacc's default resolution of shift-reduce conflicts is to shift, and shifting gives the desired result of binding an `else` to the nearest `then`, this conflict is not harmful.

```
state 0:
  prog : . stmlist

  ID       shift 6
  WHILE    shift 5
  BEGIN    shift 4
  IF       shift 3
  prog     goto 21
  stm      goto 2
  stmlist  goto 1
  .        error

state 1:
  prog : stmlist .
  stmlist : stmlist . SEMI stm

  SEMI     shift 7
  .        reduce by rule 0

state 2:
  stmlist : stm .

  .        reduce by rule 6

state 3:
  stm : IF . ID THEN stm
  stm : IF . ID THEN stm ELSE stm

  ID       shift 8
  .        error

state 4:
  stm : BEGIN . stmlist END

  ID       shift 6
  WHILE    shift 5
  BEGIN    shift 4
  IF       shift 3
  stm      goto 2
  stmlist  goto 9
  .        error

state 5:
  stm : WHILE . ID DO stm

  ID       shift 10
  .        error

state 6:
  stm : ID . ASSIGN ID

  ASSIGNshift 11
  .        error

state 7:
  stmlist : stmlist SEMI . stm

  ID       shift 6
  WHILE    shift 5
  BEGIN    shift 4
  IF       shift 3
  stm      goto 12
  .        error

state 8:
  stm : IF ID . THEN stm
  stm : IF ID . THEN stm ELSE stm

  THEN     shift 13
  .        error

state 9:
  stm : BEGIN stmlist . END
  stmlist : stmlist . SEMI stm

  END      shift 14
  SEMI     shift 7
  .        error

state 10:
  stm : WHILE ID . DO stm

  DO       shift 15
  .        error

state 11:
  stm : ID ASSIGN . ID

  ID       shift 16
  .        error

state 12:
  stmlist : stmlist SEMI stm .
  .        reduce by rule 7

state 13:
  stm : IF ID THEN . stm
  stm : IF ID THEN . stm ELSE stm

  ID       shift 6
  WHILE    shift 5
  BEGIN    shift 4
  IF       shift 3
  stm      goto 17
  .        error

state 14:
  stm : BEGIN stmlist END .

  .        reduce by rule 3

state 15:
  stm : WHILE ID DO . stm

  ID       shift 6
  WHILE    shift 5
  BEGIN    shift 4
  IF       shift 3
  stm      goto 18
  .        error

state 16:
  stm : ID ASSIGN ID .

  .        reduce by rule 1

state 17: shift/reduce conflict
          (shift ELSE, reduce 4)
  stm : IF ID THEN stm .
  stm : IF ID THEN stm . ELSE stm

  ELSE     shift 19
  .        reduce by rule 4

state 18:
  stm : WHILE ID DO stm .

  .        reduce by rule 2

state 19:
  stm : IF ID THEN stm ELSE . stm

  ID       shift 6
  WHILE    shift 5
  BEGIN    shift 4
  IF       shift 3
  stm      goto 20
  .        error

state 20:
  stm : IF ID THEN stm ELSE stm .

  .        reduce by rule 5

state 21:

  EOF      accept

  .        error
```

FIGURE 3.32. LR states for Grammar 3.30.

	id	num	+	-	*	/	(	)	$	E
1	s2	s3					s4			g7
2			r1	r1	r1	r1		r1	r1	
3			r2	r2	r2	r2		r2	r2	
4	s2	s3					s4			g5
5								s6		
6			r7	r7	r7	r7		r7	r7	
7			s8	s10	s12	s14			a	
8	s2	s3					s4			g9
9			s8,r5	s10,r5	s12,r5	s14,r5		r5	r5	
10	s2	s3					s4			g11
11			s8,r6	s10,r6	s12,r6	s14,r6		r6	r6	
12	s2	s3					s4			g13
13			s8,r3	s10,r3	s12,r3	s14,r3		r3	r3	
14	s2	s3					s4			g15
15			s8,r4	s10,r4	s12,r4	s14,r4		r4	r4	

TABLE 3.33. LR parsing table for Grammar 3.5.

Shift-reduce conflicts are acceptable in a grammar if they correspond to well understood cases, as in this example. But most shift-reduce conflicts, and all reduce-reduce conflicts, are serious problems and should be eliminated by rewriting the grammar.

PRECEDENCE DIRECTIVES

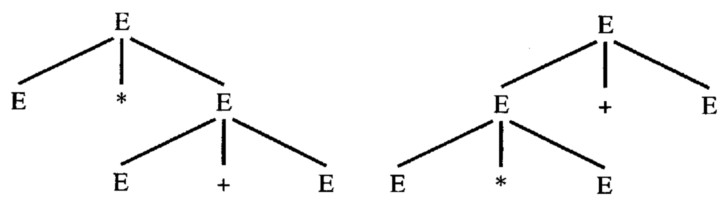
No ambiguous grammar is LR(k) for any k; the LR(k) parsing table of an ambiguous grammar will always have conflicts. However, ambiguous grammars can still be useful if we can find ways to resolve the conflicts.

For example, Grammar 3.5 is highly ambiguous. In using this grammar to describe a programming language, we intend it to be parsed so that $*$ and $/$ bind more tightly than $+$ and $-$, and that each operator associates to the left. We can express this by rewriting the unambiguous Grammar 3.8.

But we can avoid introducing the T and F symbols and their associated "trivial" reductions $E \rightarrow T$ and $T \rightarrow F$. Instead, let us start by building the LR(1) parsing table for Grammar 3.5, as shown in Table 3.33. We find many conflicts. For example, in state 13 with lookahead $+$ we find a conflict between *shift into state 8* and *reduce by rule 3*. Two of the items in state 13 are:

$E \rightarrow E * E\ .$	$+$
$E \rightarrow E\ .\ + E$	*(any)*

In this state the top of the stack is $\cdots E * E$. Shifting will lead to a stack $\cdots E * E+$ and eventually $\cdots E * E + E$ with a reduction of $E + E$ to E. Reducing now will lead to the stack $\cdots E$ and then the $+$ will be shifted. The parse trees obtained by shifting and reducing are:



Shift Reduce

If we wish $*$ to bind tighter than $+$, we should reduce instead of shift. So we fill the (13, $+$) entry in the table with r3 and discard the s8 action.

Conversely, in state 9 on lookahead $*$, we should shift instead of reduce, so we resolve the conflict by filling the (9, $*$) entry with s12.

The case for state 9, lookahead $+$ is

$E \rightarrow E + E\ .$	$+$
$E \rightarrow E\ .\ + E$	*(any)*

Shifting will make the operator right-associative; reducing will make it left-associative. Since we want left associativity, we fill (9, $+$) with r5.

Consider the expression $a - b - c$. In most programming languages, this associates to the left, as if written $(a - b) - c$. But suppose we believe that this expression is inherently confusing, and we want to force the programmer to put in explicit parentheses, either $(a - b) - c$ or $a - (b - c)$. Then we say that the minus operator is *nonassociative*, and we would fill the (11, $-$) entry with an error entry.

The result of all these decisions is a parsing table with all conflicts resolved (Table 3.34).

Yacc has *precedence directives* to indicate the resolution of this class of shift-reduce conflicts. A series of declarations such as

```
%nonassoc EQ NEQ
%left PLUS MINUS
%left TIMES DIV
%right EXP
```

	+	-	*	/	
			⋮		
9	r5	r5	s12	s14	
11 …			s12	s14	…
13	r3	r3	r3	r3	
15	r4	r4			
			⋮		

TABLE 3.34. Conflicts of Table 3.33 resolved.

indicates that + and - are left-associative and bind equally tightly; that * and / are left-associative and bind more tightly than +; that ^ is right-associative and binds most tightly; and that = and ≠ are nonassociative, and bind more weakly than +.

In examining a shift-reduce conflict such as

$E \rightarrow E * E\ .$	$+$
$E \rightarrow E\ .\ + E$	*(any)*

there is the choice of shifting a *token* and reducing by a *rule*. Should the rule or the token be given higher priority? The precedence declarations (`%left`, etc.) give priorities to the tokens; the priority of a rule is given by the last token occurring on the right-hand side of that rule. Thus the choice here is between a rule with priority * and a token with priority +; the rule has higher priority, so the conflict is resolved in favor of reducing.

When the rule and token have equal priority, then a `%left` precedence favors reducing, `%right` favors shifting, and `%nonassoc` yields an error action.

Instead of using the default "rule has precedence of its last token," we can assign a specific precedence to a rule using the `%prec` directive. This is commonly used to solve the "unary minus" problem. In most programming languages a unary minus binds tighter than any binary operator, so $-6 * 8$ is parsed as $(-6) * 8$, not $-(6 * 8)$. Grammar 3.35 shows an example.

The token UMINUS is never returned by the lexer; it is merely a placeholder in the chain of precedence (`%left`) declarations. The directive `%prec UMINUS` gives the rule `exp: MINUS exp` the highest precedence, so reducing by this rule takes precedence over shifting any operator, even a minus sign.

```
%{ declarations of yylex and yyerror %}
%token INT | PLUS | MINUS | TIMES | UMINUS
%start exp

%left PLUS MINUS
%left TIMES
%left UMINUS
%%

exp :   INT
    |   exp PLUS exp
    |   exp MINUS exp
    |   exp TIMES exp
    |   MINUS exp     %prec UMINUS
```

GRAMMAR 3.35.

Precedence rules are helpful in resolving conflicts, but they should not be abused. If you have trouble explaining the effect of a clever use of precedence rules, perhaps instead you should rewrite the grammar to be unambiguous.

SYNTAX VERSUS SEMANTICS

Consider a programming language with *arithmetic expressions* such as $x + y$ and *boolean expressons* such as $x + y = z$ or $a\&(b = c)$. Arithmetic operators bind tighter than the boolean operators; there are arithmetic variables and boolean variables; and a boolean expression cannot be added to an arithmetic expression. Grammar 3.36 gives a syntax for this language.

The grammar has a reduce-reduce conflict, as shown in Figure 3.37. How should we rewrite the grammar to eliminate this conflict?

Here the problem is that when the parser sees an identifier such as a, it has no way of knowing whether this is an arithmetic variable or a boolean variable – syntactically they look identical. The solution is to defer this analysis until the "semantic" phase of the compiler; it's not a problem that can be handled naturally with context-free grammars. A more appropriate grammar is:

$S \rightarrow \text{id} := E$
$E \rightarrow \text{id}$
$E \rightarrow E \,\&\, E$
$E \rightarrow E \;=\; E$
$E \rightarrow E \;+\; E$

Now the expression $a + 5\&b$ is syntactically legal, and a later phase of the compiler will have to reject it and print a semantic error message.

```
%{ declarations of yylex and yyerror %}
%token ID | ASSIGN | PLUS | MINUS | AND | EQUAL
%start stm
%left OR
%left AND
%left PLUS
%%

stm : ID ASSIGN ae
    | ID ASSIGN be

be  : be OR be
    | be AND be
    | ae EQUAL ae
    | ID

ae  : ae PLUS ae
    | ID
```

GRAMMAR 3.36.

3.5 ERROR RECOVERY

LR(k) parsing tables contain shift, reduce, accept, and error actions. On page 59 I claimed that when an LR parser encounters an error action it stops parsing and reports failure. This behavior would be unkind to the programmer, who would like to have *all* the errors in her program reported, not just the first error.

RECOVERY USING THE ERROR SYMBOL

Local error recovery mechanisms work by adjusting the parse stack and the input *at the point where the error was detected* in a way that will allow parsing to resume. One local recovery mechanism – found in many versions of the Yacc parser generator – uses a special *error* symbol to control the recovery process. Wherever the special *error* symbol appears in a grammar rule, a sequence of erroneous input tokens can be matched.

For example, in a Yacc grammar for Tiger, we might have productions such as

```
state 0:
  stm : . ID ASSIGN ae
  stm : . ID ASSIGN be

  ID      shift 1
  stm     goto 14
  .       error

state 1:
  stm : ID . ASSIGN ae
  stm : ID . ASSIGN be

  ASSIGN  shift 2
  .       error

state 2:
  stm : ID ASSIGN . ae
  stm : ID ASSIGN . be

  ID      shift 5
  be      goto 4
  ae      goto 3
  .       error

state 3:
  stm : ID ASSIGN ae .
  be : ae . EQUAL ae
  ae : ae . PLUS ae

  PLUS    shift 7
  EQUAL   shift 6
  .       reduce by rule 0

state 4:
  stm : ID ASSIGN be .
  be : be . AND be

  AND     shift 8
  .       reduce by rule 1

state 5: reduce/reduce conflict
         between rule 6 and
         rule 4 on EOF
  be : ID .
  ae : ID .

  PLUS    reduce by rule 6
  AND     reduce by rule 4
  EQUAL   reduce by rule 6
  EOF     reduce by rule 4
  .       error

state 6:
  be : ae EQUAL . ae

  ID      shift 10
  ae      goto 9
  .       error

state 7:
  ae : ae PLUS . ae

  ID      shift 10
  ae      goto 11
  .       error

state 8:
  be : be AND . be

  ID      shift 5
  be      goto 13
  ae      goto 12
  .       error

state 9:
  be : ae EQUAL ae .
  ae : ae . PLUS ae

  PLUS    shift 7
  .       reduce by rule 3

state 10:
  ae : ID .

  .       reduce by rule 6

state 11:
  ae : ae . PLUS ae
  ae : ae PLUS ae .

  .       reduce by rule 5

state 12:
  be : ae . EQUAL ae
  ae : ae . PLUS ae

  PLUS    shift 7
  EQUAL   shift 6
  .       error

state 13:
  be : be . AND be
  be : be AND be .

  .       reduce by rule 2

state 14:

  EOF     accept
  .       error
```

FIGURE 3.37. LR states for Grammar 3.36.

$$
\begin{aligned}
exp &\rightarrow ID \\
exp &\rightarrow exp\ +\ exp \\
exp &\rightarrow (\ exps\ ) \\
exps &\rightarrow exp \\
exps &\rightarrow exps\ ;\ exp
\end{aligned}
$$

Informally, we can specify that if a syntax error is encountered in the middle of an expression, the parser should skip to the next semicolon or right parenthesis (these are called *syncronizing tokens*) and resume parsing. We do this by adding error-recovery productions such as

$$exp \rightarrow (\ error \ )$$
$$exps \rightarrow error \ ; \ exp$$

What does the parser-generator do with the *error* symbol? In parser generation, *error* is considered a terminal symbol, and shift actions are entered in the parsing table for it as if it were an ordinary token.

When the LR parser reaches an error state, it takes the following actions:

1. Pop the stack (if necessary) until a state is reached in which the action for the *error* token is *shift*.
2. Shift the *error* token.
3. Discard input symbols (if necessary) until a state is reached that has a non-error action on the current lookahead token.
4. Resume normal parsing.

In the two *error* productions illustrated above, we have taken care to follow the *error* symbol with an appropriate synchronizing token – in this case, right parenthesis or semicolon. Thus, the "non-error action" taken in step 3 will always *shift*. If instead we used the production $exp \rightarrow error$, the "non-error action" would be *reduce*, and (in an SLR or LALR parser) it is possible that the original (erroneous) lookahead symbol would cause another error after the reduce action, without having advanced the input. Therefore, grammar rules that contain *error* not followed by a token should be used only when there is no good alternative.

Caution. One can attach *semantic actions* to Yacc grammar rules; whenever a rule is reduced, its semantic action is executed. Chapter 4 explains the use of semantic actions. Popping states from the stack can lead to seemingly "impossible" semantic actions, especially if the actions contain side effects. Consider this grammar fragment:

```
statements:  statements exp SEMICOLON
          |  statements error SEMICOLON
          |  /* empty */

exp : increment exp decrement
    | ID

increment:  LPAREN     {nest=nest+1;}
decrement:  RPAREN     {nest=nest-1;}
```

"Obviously" it is true that whenever a semicolon is reached, the value of `nest` is zero, because it is incremented and decremented in a balanced way according to the grammar of expressions. But if a syntax error is found after some left parentheses have been parsed, then states will be popped from the stack without "completing" them, leading to a nonzero value of `nest`. The best solution to this problem is to have side-effect-free semantic actions that build abstract syntax trees, as described in Chapter 4.

GLOBAL ERROR REPAIR

What if the best way to recover from the error is to insert or delete tokens from the input stream at a point *before* where the error was detected? Consider the following Tiger program:

```
let   type a := intArray [ 10 ] of 0   in ...
```

A local technique will discover a syntax error with `:=` as lookahead symbol. Error recovery based on *error* productions would likely delete the phrase from `type` to `0`, resynchronizing on the `in` token. Some local repair techniques can insert tokens as well as delete them; but even a local repair that replaces the `:=` by `=` is not very good, and will encounter another syntax error at the `[` token. Really, the programmer's mistake here is in using `type` instead of `var`, but the error is detected two tokens too late.

Global error repair finds the smallest set of insertions and deletions that would turn the source string into a syntactically correct string, *even if the insertions and deletions are not at a point where an LL or LR parser would first report an error.* In this case, global error repair would do a single-token substitution, replacing `type` by `var`.

Burke-Fisher error repair. I will describe a limited but useful form of global error repair, which tries every possible single-token insertion, deletion, or replacement at every point that occurs no earlier than K tokens before the point where the parser reported the error. Thus, with $K = 15$, if the parsing engine gets stuck at the 100th token of the input, then it will try every possible repair between the 85th and 100th token.

The correction that allows the parser to parse furthest past the original reported error is taken as the best error repair. Thus, if a single-token substitution of `var` for `type` at the 98th token allows the parsing engine to proceed past the 104th token without getting stuck, this repair is a successful one.

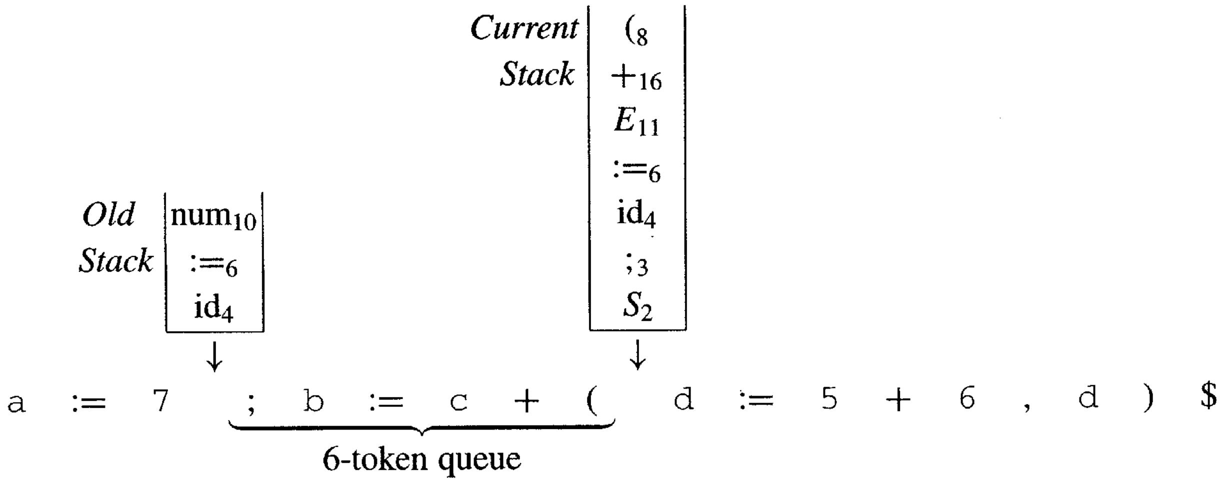


FIGURE 3.38. Burke-Fisher parsing, with an error-repair queue. Figure 3.18 shows the complete parse of this string according to Table 3.19.

Generally, if a repair carries the parser $R = 4$ tokens beyond where it originally got stuck, this is "good enough."

The advantage of this technique is that the LL(k) or LR(k) (or LALR, etc.) grammar is not modified at all (no *error* productions), nor are the parsing tables modified. Only the parsing engine, which interprets the parsing tables, is modified.

The parsing engine must be able to back up K tokens and reparse. To do this, it needs to remember what the parse stack looked like K tokens ago. Therefore, the algorithm maintains *two* parse stacks: the *current* stack and the *old* stack. A queue of K tokens is kept; as each new token is shifted, it is pushed on the *current* stack and also put onto the tail of the queue; simultaneously, the head of the queue is removed and shifted onto the *old* stack. With each *shift* onto the old or current stack, the appropriate reduce actions are also performed. Figure 3.38 illustrates the two stacks and queue.

Now suppose a syntax error is detected at the *current* token. For each possible insertion, deletion, or substitution of a token at any position of the queue, the Burke-Fisher error repairer makes that change to within (a copy of) the queue, then attempts to reparse from the *old* stack. The success of a modification is in how many tokens past the *current* token can be parsed; generally, if three or four new tokens can be parsed, this is considered a completely successful repair.

In a language with N kinds of tokens, there are $K + K \cdot N + K \cdot N$ possible deletions, insertions, and substitutions within the K-token window. Trying this many repairs is not very costly, especially considering that it happens only when a syntax error is discovered, not during ordinary parsing.

Semantic actions. Shift and reduce actions are tried repeatly and discarded during the search for the best error repair. Parser generators usually perform programmer-specified semantic actions along with each reduce action, but the programmer does not expect that these actions will be performed repeatedly and discarded – they may have side effects. Therefore, a Burke-Fisher parser does not execute any of the semantic actions as reductions are performed on the *current* stack, but waits until the same reductions are performed (permanently) on the *old* stack.

This means that the lexical analyzer may be up to $K + R$ tokens ahead of the point to which semantic actions have been performed. If semantic actions affect lexical analysis – as they do in C, compiling the `typedef` feature – this can be a problem with the Burke-Fisher approach. For languages with a pure context-free grammar approach to syntax, the delay of semantic actions poses no problem.

Semantic values for insertions. In repairing an error by insertion, the parser needs to provide a semantic value for each token it inserts, so that semantic actions can be performed as if the token had come from the lexical analyzer. For punctuation tokens no value is necessary, but when tokens such as numbers or identifiers must be inserted, where can the value come from? The ML-Yacc parser generator, which uses Burke-Fischer error correction, has a `%value` directive, allowing the programmer to specify what value should be used when inserting each kind of token:

```
%value ID  ("bogus")
%value INT (1)
%value STRING  ("")
```

Programmer-specified substitutions. Some common kinds of errors cannot be repaired by the insertion or deletion of a single token, and sometimes a particular single-token insertion or substitution is very commonly required and should be tried first. Therefore, in an ML-Yacc grammar specification the programmer can use the `%change` directive to suggest error corrections to be tried first, before the default "delete or insert each possible token" repairs.

```
%change     EQ -> ASSIGN  |  ASSIGN -> EQ
        |  SEMICOLON ELSE -> ELSE   |    -> IN INT END
```

Here the programmer is suggesting that users often write "`; else`" where they mean "`else`" and so on.

The insertion of `in 0 end` is a particularly important kind of correction, known as a *scope closer*. Programs commonly have extra left parentheses or right parentheses, or extra left or right brackets, and so on. In Tiger, another kind of nesting construct is `let` ··· `in` ··· `end`. If the programmer forgets to close a scope that was opened by left parenthesis, then the automatic single-token insertion heuristic can close this scope where necessary. But to close a `let` scope requires the insertion of three tokens, which will not be done automatically unless the compiler-writer has suggested "change *nothing* to `in 0 end`" as illustrated in the `%change` command above.

PROGRAM PARSING

Use Yacc to implement a parser for the Tiger language. Appendix A describes, among other things, the syntax of Tiger.

You should turn in the file `tiger.grm` and a `README`.

Supporting files available in `$TIGER/chap3` include:

`makefile` The "makefile."
`errormsg.[ch]` The Error Message structure, useful for producing error messages with file names and line numbers.
`lex.yy.c` The lexical analyzer. I haven't provided the source file `tiger.lex`, but I've provided the output of Lex that you can use if your lexer isn't working.
`parsetest.c` A driver to run your parser on an input file.
`tiger.grm` The skeleton of a file you must fill in.

You won't need `tokens.h` anymore; instead, the header file for tokens is `y.tab.h`, which is produced automatically by Yacc from the token specification of your grammar.

Your grammar should have as few shift-reduce conflicts as possible, and no reduce-reduce conflicts. Furthermore, your accompanying documentation should list each shift-reduce conflict (if any) and explain why it is not harmful.

My grammar has a shift-reduce conflict that's related to the confusion between

variable [expression]
type-id [expression] **of** expression

In fact, I had to add a seemingly redundant grammar rule to handle this confusion. Is there a way to do this without a shift-reduce conflict?

Use the precedence directives (`%left`, `%nonassoc`, `%right`) *when it is straightforward to do so.*

Do not attach any semantic actions to your grammar rules for this exercise.

Optional: Add *error* productions to your grammar and demonstrate that your parser can sometimes recover from syntax errors.

FURTHER READING

Conway [1963] describes a predictive (recursive-descent) parser, with a notion of FIRST sets and left-factoring. LL(k) parsing theory was formalized by Lewis and Stearns [1968].

LR(k) parsing was developed by Knuth [1965]; the SLR and LALR techniques by DeRemer [1971]; LALR(1) parsing was popularized by the development and distribution of Yacc [Johnson 1975] (which was not the first parser-generator, or "compiler-compiler," as can be seen from the title of the cited paper).

Figure 3.29 summarizes many theorems on subset relations between grammar classes. Heilbrunner [1981] shows proofs of several of these theorems, including LL(k) $\subset$ LR(k) and LL(1) $\not\subset$ LALR(1) (see Exercise 3.14). Backhouse [1979] is a good introduction to theoretical aspects of LL and LR parsing.

Aho et al. [1975] showed how deterministic LL or LR parsing engines can handle ambiguous grammars, with ambiguities resolved by precedence directives (as described in Section 3.4).

Burke and Fisher [1987] invented the error-repair tactic that keeps a K-token queue and two parse stacks.

EXERCISES

3.1 Translate each of these regular expressions into a context-free grammar.

a. $((xy^*x)|(yx^*y))?$

b. $((0|1)^+ \text{"."} (0|1)^*)|((0|1)^* \text{"."} (0|1)^+)$

***3.2** Write a grammar for English sentences using the words

```
time, arrow, banana, flies, like, a, an, the, fruit
```