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.

It is often possible to use ambiguous grammars by resolving shift-reduce conflicts in favor of shifting or reducing, as appropriate. But it is best to use this technique sparingly, and only in cases (such as the *dangling-else* described here, and operator-precedence to be described on page 74) that are 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. *CUP*[1] ("Construction of Useful Parsers") is one such tool, modeled on the classic *Yacc* ("Yet another compiler-compiler") parser generator.

A CUP specification has a *preamble*, which declares lists of terminal symbols, nonterminals, and so on, followed by *grammar rules*. The preamble also specifies how the parser is to be attached to a lexical analyzer and other such details.

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

[1] CUP, written by Scott Hudson of the Georgia Institute of Technology, is available from this book's Web site.


```
terminal ID, WHILE, BEGIN, END, DO, IF, THEN, ELSE, SEMI, ASSIGN;

non terminal  prog, stm, stmlist;

start with   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. CUP version of Grammar 3.30. Semantic actions are omitted and will be discussed in Chapter 4.

Java and will be executed whenever the parser reduces using this rule.

Consider Grammar 3.30. It can be encoded in CUP as shown in Grammar 3.31. The CUP 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

CUP 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, CUP resolves shift-reduce conflicts by shifting, and reduce-reduce conflicts by using the rule that appears earlier in the grammar.

CUP 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 CUP 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 CUP'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
 ASSIGN  shift 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

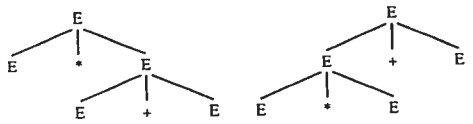
No ambiguous grammar is LR(k) for any k; the LR(k) parsing table of an ambiguous grammar will always have conflicts. However, ambiguous grammars can still be useful if we can find ways to resolve the conflicts.

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:

$E \to E * E .$	$+$
$E \to E . + E$	*(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

$E \to E + E .$	$+$
$E \to E . + E$	*(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).

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

```
precedence nonassoc EQ, NEQ;
precedence left PLUS, MINUS;
precedence left TIMES, DIV;
precedence 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

$E \to E * E .$	$+$
$E \to E . + E$	*(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 (`precedence 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's just a placeholder in the chain of `precedence` 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.

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.

```
terminal    INT, PLUS, MINUS, TIMES, UMINUS;
non terminal    exp;
precedence left PLUS, MINUS;
precedence left TIMES;
precedence left UMINUS;

start with   exp;

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

GRAMMAR 3.35.

```
terminal ID, ASSIGN, PLUS, MINUS, AND, EQUAL;
non terminal  stm, be, ae;
precedence left OR;
precedence left AND;
precedence left PLUS;
start with stm;

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.

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.


```
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.

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.

3.5 ERROR RECOVERY

LR(k) parsing tables contain shift, reduce, accept, and error actions. On page 60 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

exp → *ID*
exp → *exp* + *exp*
exp → (*exps*)
exps → *exp*
exps → *exps* ; *exp*

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 → (*error*)
exps → *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* → *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; :)
```

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 CUP 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/ErrorMsg.java` The `ErrorMsg` class, useful for producing error messages with file names and line numbers.

`Parse/Yylex.class` The lexical analyzer. I haven't provided the source file, but I've compiled it so you will be able to use the .class file if your lexer isn't working.

`Parsetest.java` A driver to run your parser on an input file.

`java_cup/runtime/*` Support files for CUP parser.

`java_cup/Main.java` The CUP parser generator.

`Grm.cup` The skeleton of a Tiger parser specification you must fill in.

The `sym.java` file will be automatically produced by CUP 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. $((0|1)^+\cdot\cdot(0|1)^*)|((0|1)^*\cdot\cdot(0|1)^+)$

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

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

and the semicolon. Be sure to include all the senses (noun, verb, etc.) of each word. Then show that this grammar is ambiguous by exhibiting more than one parse tree for "time flies like an arrow; fruit flies like a banana."

3.3 Write an unambigous grammar for each of the following languages. **Hint:** One way of verifying that a grammar is unambiguous is to run it through Yacc and get no conflicts.

a. Palindromes over the alphabet $\{a, b\}$ (strings that are the same backward and forward).

b. Strings that match the regular expression $a*b*$ and have more a's than b's.

c. Balanced parentheses and square brackets. Example: `([][(){([])}])`

*d. Balanced parentheses and brackets, where a closing bracket also closes any outstanding open parentheses (up to the previous open bracket). Example: `[([](){([])}]`. **Hint:** First, make the language of balanced parentheses and brackets, where extra open parentheses are allowed; then make sure this nonterminal must appear within brackets.

e. All subsets and permutations (without repetition) of the keywords `public final static synchronized transient`. (Then comment on how best to handle this situation in a real compiler.)

f. Statement blocks in Pascal or ML where the semicolons *separate* the statements:

```
( statement ; ( statement ; statement ) ; statement )
```

g. Statement blocks in C where the semicolons *terminate* the statements:

```
{ expression; { expression; expression; } expression; }
```

3.4 Write a grammar that accepts the same language as Grammar 3.1, but that is suitable for LL(1) parsing. That is, eliminate the ambiguity, eliminate the left recursion, and (if necessary) left-factor.

show the Yacc-style precedence directives that resolve the conflicts this way.

0 $S \rightarrow E\ \$$

1 $E \rightarrow$ while E do E
2 $E \rightarrow$ id $:= E$
3 $E \rightarrow E + E$
4 $E \rightarrow$ id

***3.16** Explain how to resolve the conflicts in this grammar, using precedence directives, or grammar transformations, or both. Use CUP as a tool in your investigations, if you like.

1 $E \rightarrow$ id
2 $E \rightarrow E\ B\ E$
3 $B \rightarrow +$
4 $B \rightarrow -$
5 $B \rightarrow \times$
6 $B \rightarrow /$

***3.17** Prove that Grammar 3.8 cannot generate parse trees of the form shown in Figure 3.9. **Hint:** What nonterminals could possibly be where the $?X$ is shown? What does that tell us about what could be where the $?Y$ is shown?

4 Abstract Syntax

ab-stract: disassociated from any specific instance

Webster's Dictionary

A compiler must do more than recognize whether a sentence belongs to the language of a grammar – it must do something useful with that sentence. The *semantic actions* of a parser can do useful things with the phrases that are parsed.

In a recursive-descent parser, semantic action code is interspersed with the control flow of the parsing actions. In a parser specified in CUP, semantic actions are fragments of Java program code attached to grammar productions.

4.1 SEMANTIC ACTIONS

Each terminal and nonterminal may be associated with its own type of semantic value. For example, in a simple calculator using Grammar 3.35, the type associated with `exp` and `INT` might be `int`; the other tokens would not need to carry a value. The type associated with a token must, of course, match the type that the lexer returns with that token.

For a rule $A \rightarrow B\ C\ D$, the semantic action must return a value whose type is the one associated with the nonterminal A. But it can build this value from the values associated with the matched terminals and nonterminals B, C, D.

RECURSIVE DESCENT

In a recursive-descent parser, the semantic actions are the values returned by parsing functions, or the side effects of those functions, or both. For each terminal and nonterminal symbol, we associate a *type* (from the implementation

```
class Token {int kind; Object val;
             Token(int k, Object v) {kind=k; val=v;}
            }
final int EOF=0, ID=1, NUM=2, PLUS=3, MINUS=4, ···

int lookup(String id) { ··· }

int F_follow[] = { PLUS, TIMES, RPAREN, EOF };

int F() {switch (tok.kind) {
         case ID:   int i=lookup((String)(tok.val)); advance(); return i;
         case NUM:  int i=((Integer)(tok.val)).intValue();
                    advance(); return i;
         case LPAREN:  eat(LPAREN);
                       int i = E();
                       eatOrSkipTo(RPAREN, F_follow);
                       return i;
         case EOF:
         default:   print("expected ID, NUM, or left-paren");
                    skipto(F_follow); return 0;
        }}

int T_follow[] = { PLUS, RPAREN, EOF };

int T() {switch (tok.kind) {
         case ID:
         case NUM:
         case LPAREN: return Tprime(F());
         default: print("expected ID, NUM, or left-paren");
                  skipto(T_follow);
                  return 0;
        }}

int Tprime(int a) {switch (tok.kind) {
         case TIMES: eat(TIMES); return Tprime(a*F());
         case PLUS:
         case RPAREN:
         case EOF:  return a;
         default:  ···
        }}

void eatOrSkipTo(int expected, int[] stop) {
   if (tok.kind==expected)
       eat(expected);
   else {print(···); skipto(stop);}
}
```

PROGRAM 4.1. Recursive-descent interpreter for part of Grammar 3.15.


```
terminal PLUS, MINUS, TIMES, UMINUS;
terminal Integer  INT;
non terminal Integer  exp;
start with   exp;

precedence left PLUS, MINUS;
precedence left TIMES;
precedence left UMINUS;

exp  ::=  INT:i
                {: RESULT = i; :}
     |    exp:e1 PLUS exp:e2
                {: RESULT = new Integer(e1.intValue()+e2.intValue()); :}
     |    exp:e1 MINUS exp:e2
                {: RESULT = new Integer(e1.intValue()-e2.intValue()); :}
     |    exp:e1 TIMES exp:e2
                {: RESULT = new Integer(e1.intValue()*e2.intValue()); :}
     |    MINUS exp:e %prec UMINUS
                {: RESULT = new Integer(-e.intValue()); :};
```

PROGRAM 4.2. CUP version of Grammar 3.35.

language of the compiler) of *semantic values* representing phrases derived from that symbol.

Program 4.1 is a recursive-descent interpreter for part of Grammar 3.15. The tokens `ID` and `NUM` must now carry values of type `string` and `int`, respectively. We will assume there is a lookup table mapping identifiers to integers. The type associated with E, T, F, etc. is `int`, and the semantic actions are easy to implement.

The semantic action for an artificial symbol such as T' (introduced in the elimination of left recursion) is a bit tricky. Had the production been $T \rightarrow T * F$ then the semantic action would have been

```
int a = T(); eat(TIMES); int b=F(); return a*b;
```

With the rearrangement of the grammar, the production $T' \rightarrow *FT'$ is missing the left operand of the $*$. One solution is for T to pass the left operand as an argument to T', as shown in Program 4.1.

CUP-GENERATED PARSERS

A parser specification for CUP consists of a set of grammar rules, each annotated with a semantic action that is a Java statement. Whenever the generated

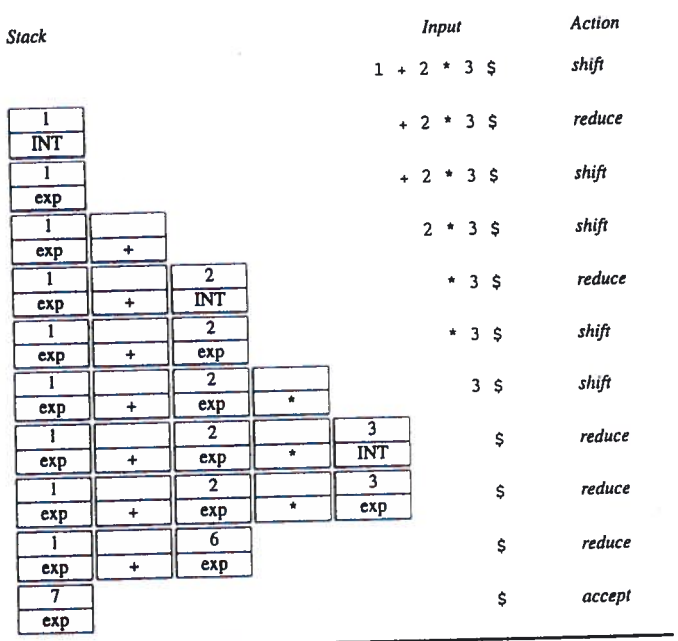


FIGURE 4.3. Parsing with a semantic stack.

parser reduces by a rule, it will execute the corresponding semantic action fragment.

Program 4.2 shows how this works for Grammar 3.35. Every `INT` terminal and every `exp` nonterminal carries an Integer value. To access this value, give the terminal or nonterminal a "name" in the grammar rule (such as `i` or `e1` in Program 4.2), and access this name as a variable in the semantic action. The variable `RESULT` is the "name" of the left-hand-side nonterminal of the grammar rule.

In a more realistic example, there might be several nonterminals each carrying a different type.

A CUP-generated parser implements semantic values by keeping a stack of them parallel to the state stack. Where each symbol would be on a simple parsing stack, now there is a semantic value. When the parser performs a reduction, it must execute a Java-language semantic action; it satisfies each

reference to a right-hand-side semantic value by a reference to one of the top k elements of the stack (for a rule with k right-hand-side symbols). When the parser pops the top k elements from the symbol stack and pushes a nonterminal symbol, it also pops k from the semantic value stack and pushes the value obtained by executing the Java semantic action code.

Figure 4.3 shows an LR parse of a string using Program 4.2. The stack holds states and semantic values (in this case, the semantic values are all integers). When a rule such as $E \rightarrow E + E$ is reduced (with a semantic action such as `exp1+exp2`), the top three elements of the semantic stack are `exp1`, empty (a place-holder for the trivial semantic value carried by +), and `exp2`, respectively.

A MINI-INTERPRETER IN SEMANTIC ACTIONS

To illustrate the power of semantic actions, let us write an interpreter for the language whose abstract syntax is given in Program 1.5. In Chapter 1 we interpreted the abstract syntax trees; with Yacc we can interpret the real thing, the concrete syntax of the language.

Program 4.4 is a CUP grammar with semantic actions that build expression-objects of classes `PlusExp`, `MinusExp`, `IdExp` (and so on) and statement-objects of classes `CompoundStm`, `AssignStm`, `PrintStm` (and so on).

Figure 4.5 implements the `Table` class. A `Table` has just one method, `lookup`, that maps an identifier to a number. There are two implementations of `Table`:

`EmptyTable` whose `lookup` method always throws an error, and
`Update` which makes a table just like `base`, except that the identifier `id` maps to `val`.

Chapter 5 discusses more efficient versions of such tables.

Program 4.6 shows the implementation of the `Exp` class. The method `eval` is a function from `Table` to integer; roughly, "you give me a table to look up identifiers, and I'll give you back an integer." Thus, `eval` for a simple identifier `IdExp(x)` is, "give me a table, and I'll look up `x` and give you what I find." The `eval` method for `NumExp(5)` is even simpler: "give me a table, and I'll ignore it and give you the integer 5." Finally, `eval` for $e_1 + e_2$ is, "give me a table t, and first I'll apply e_1 to t, then do $e_2(t)$, then add the resulting integers."

Program 4.7 shows the `Stm` class. The method `eval` is a function from `Table` to `Table`; roughly, "you tell me what the state of the world looked like

```
terminal Integer INT;
terminal String ID;
terminal PLUS, MINUS, TIMES, DIV, ASSIGN, PRINT,
         LPAREN, RPAREN, COMMA, SEMICOLON;
non terminal Exp exp;
non terminal Stm stm;
non terminal ExpList exps;
non terminal Table prog;

precedence right SEMICOLON;
precedence left PLUS, MINUS;
precedence left TIMES, DIV;

start with prog;

prog ::= stm:s                          {:RESULT=s.eval(new EmptyTable());:};

stm  ::= stm:a SEMICOLON stm:b          {:RESULT=new CompoundStm(a,b);:};
stm  ::= ID:i ASSIGN exp:e              {:RESULT=new AssignStm(i,e);:};
stm  ::= PRINT LPAREN exps:e RPAREN {:RESULT=new PrintStm(e);:};

exps ::= exp:e                          {:RESULT=new ExpList(e,null);:};
exps ::= exp:e COMMA exps:es            {:RESULT=new ExpList(e,es);:};

exp  ::= INT:i                          {:RESULT=new NumExp(i.intValue());:};
exp  ::= ID:id                          {:RESULT=new IdExp(id);:};
exp  ::= exp:a PLUS exp:b               {:RESULT=new PlusExp(a,b);:};
exp  ::= exp:a MINUS exp:b              {:RESULT=new MinusExp(a,b);:};
exp  ::= exp:a TIMES exp:b              {:RESULT=new TimesExp(a,b);:};
exp  ::= exp:a DIV exp:b                {:RESULT=new DivExp(a,b);:};
exp  ::= stm:s COMMA exp:e              {:RESULT=new EseqExp(s,e);:};
exp  ::= LPAREN exp:e RPAREN            {:RESULT=e;:};
```

PROGRAM 4.4. An interpreter for straight-line programs.

before this statement executed, I'll show you the 'after' state." The statement b:=6, applied to a table t, returns a new table t' that is just like t except that t'.lookup(b) $= 6$. And to implement the compound statement $s_1; s_2$ applied to a table t, we first get a table t' by s_1.eval(t), then calculate s_2.eval(t').

This interpreter contains a major error: an assignment statement inside an expression has no permanent effect (see Exercise 4.2).

AN IMPERATIVE INTERPRETER IN SEMANTIC ACTIONS

Program 4.2 and Program 4.4 show how semantic values for nonterminals can be calculated from the semantic values of the right-hand side of the pro-


```
abstract class Table {abstract int lookup(String id);}
class EmptyTable extends Table {
    int lookup(String id) {throw new Error("Empty Table");}
}
class Update extends Table {
    private Table base; String id; int val;
    Update(Table b, String i, int v) {base=b; id=i.intern(); val=v;}
    int lookup(String i) {
       if (i.intern()==id) return val;
       else return base.lookup(i);
    }
}
```

PROGRAM 4.5. Table class for Program 4.4.

```
abstract class Exp {abstract int eval(Table env);}

class Num extends Exp {private int i;
        Num(int ii) {i=ii;}
        int eval(Table env) {return i;}
}
class Id extends Exp {private String id;
        Id(String i) {id=i;}
        int eval(Table env) {return env.lookup(id);}
}
class Plus extends Exp {private Exp a,b;
        Plus(Exp aa, Exp bb) {a=aa; b=bb;}
        int eval(Table env) {return aa.eval(env) + bb.eval(env);}
}
class Minus extends Exp {private Exp a,b;
        Minus(Exp aa, Exp bb) {a=aa; b=bb;}
        int eval(Table env) {return aa.eval(env) - bb.eval(env);}
}
class Times extends Exp {private Exp a,b;
        Times(Exp aa, Exp bb) {a=aa; b=bb;}
        int eval(Table env) {return aa.eval(env) * bb.eval(env);}
}
class Div extends Exp {private Exp a,b;
        Div(Exp aa, Exp bb) {a=aa; b=bb;}
        int eval(Table env) {return aa.eval(env) / bb.eval(env);}
}
class Eseq extends Exp {private Stm stm, Exp exp;
        Eseq(Stm s, Exp e) {stm=s; exp=e;}
        int eval(Table env) {return exp.eval(stm.eval(env));}
}
```

PROGRAM 4.6. Exp class for Program 4.4.

```
abstract class Stm {abstract Table eval(Table env);}

class CompoundStm extends Stm {private Stm stm1, stm2;
        CompoundStm(Stm s1, Stm s2) {stm1=s1; stm2=s2;}
        Table eval(Table env) {return stm2.eval(stm1.eval(env));}
}
class AssignStm extends Stm {private String id; Exp exp;
        AssignStm(String i, Exp e) {id=i; exp=e;}
        Table eval(Table env) {return new Update(env,id,exp.eval(env));}
}
class PrintStm extends Stm {private ExpList exps;
        PrintStm(ExpList e) {exps=e;}
        Table eval(Table env) {exps.eval(env); return env;}
}

class ExpList {private Exp head, ExpList tail;
        ExpList(Exp h, ExpList t) {head=h; tail=t;}
        void eval(Table env) {System.out.print(head.eval(env));
                              if (tail!=null) tail.eval(env);}
}
```

PROGRAM 4.7. Stm and ExpList classes for Program 4.4.

ductions. The semantic actions in these examples do not have *side effects* that change any global state, so the order of evaluation of the right-hand-side symbols does not matter.

However, an LR parser does perform reductions, and associated semantic actions, in a deterministic and predictable order: a bottom-up, left-to-right traversal of the parse tree. In other words, the (virtual) parse tree is traversed in *postorder*. Thus, one can write *imperative* semantic actions with global side effects, and be able to predict the order of their occurrence.

Program 4.8 shows an imperative version of the interpreter.

4.2 ABSTRACT PARSE TREES

It is possible to write an entire compiler that fits within the semantic action phrases of a CUP parser. However, such a compiler is difficult to read and maintain. And this approach constrains the compiler to analyze the program in exactly the order it is parsed.

To improve modularity, it is better to separate issues of syntax (parsing) from issues of semantics (type-checking and translation to machine code).


```
action code {: java.util.Dictionary dict = new java.util.Hashtable();
               int get(String id) {
                     return ((Integer)dict.get(id.intern())).intValue();
               }
               void put(String id, int v) {
                       dict.put(id.intern(), new Integer(v));
               }
             :}

terminal Integer INT;
terminal String ID;
terminal PLUS, MINUS, TIMES, DIV, ASSIGN, PRINT,
         LPAREN, RPAREN, COMMA, SEMICOLON;
non terminal Integer exp;
non terminal stm, exps, prog;

precedence right SEMICOLON;
precedence left PLUS, MINUS;
precedence left TIMES, DIV;

start with prog;

prog ::= stm:s                          ;

stm  ::= stm:a SEMICOLON stm:b          ;
stm  ::= ID:id ASSIGN exp:e             {: put(id,e); :};
stm  ::= PRINT LPAREN exps:e RPAREN {: System.out.println(); :};

exps ::= exp:e                          {: System.out.print(e); :};
exps ::= exps COMMA exp:e               {: System.out.print(e); :};

exp  ::= INT:i                  {:RESULT=i.intValue();:};
exp  ::= ID:id                  {:RESULT=get(id);:};
exp  ::= exp:a PLUS exp:b       {:RESULT=new Integer(a.intValue() +
                                                     b.intValue());:};
exp  ::= exp:a MINUS exp:b      {:RESULT=new Integer(a.intValue() -
                                                     b.intValue());:};
exp  ::= exp:a TIMES exp:b      {:RESULT=new Integer(a.intValue() *
                                                     b.intValue());:};
exp  ::= exp:a DIV exp:b        {:RESULT=new Integer(a.intValue() /
                                                     b.intValue());:};
exp  ::= stm:s COMMA exp:e      {:RESULT=e;:};
exp  ::= LPAREN exp:e RPAREN {:RESULT=e;:};
```

PROGRAM 4.8. An interpreter in imperative style.

$S \to S\ ;\ S$	$L \to$
$S \to \texttt{id} := E$	$L \to L\ E$
$S \to \texttt{print}\ L$	
$E \to \texttt{id}$	$B \to +$
$E \to \texttt{num}$	$B \to -$
$E \to E\ B\ E$	$B \to \times$
$E \to S\ ,\ E$	$B \to /$

GRAMMAR 4.9. Abstract syntax of straight-line programs.

One way to do this is for the parser to produce a *parse tree* – a data structure that later phases of the compiler can traverse. Technically, a parse tree has exactly one leaf for each token of the input and one internal node for each grammar rule reduced during the parse.

Such a parse tree, which we will call a *concrete parse tree* representing the *concrete syntax* of the source language, is inconvenient to use directly. Many of the punctuation tokens are redundant and convey no information – they are useful in the input string, but once the parse tree is built, the structure of the tree conveys the structuring information more conveniently.

Furthermore, the structure of the parse tree depends too much on the grammar! The grammar transformations shown in Chapter 3 – factoring, elimination of left recursion, elimination of ambiguity – involve the introduction of extra nonterminal symbols and extra grammar productions for technical purposes. These details should be confined to the parsing phase and should not clutter the semantic analysis.

An *abstract syntax* makes a clean interface between the parser and the later phases of a compiler (or, in fact, for the later phases of other kinds of program-analysis tools such as dependency analyzers). The abstract syntax tree conveys the phrase structure of the source program, with all parsing issues resolved but without any semantic interpretation.

Many early compilers did not use an abstract syntax data structure because early computers did not have enough memory to represent an entire compilation unit's syntax tree. Modern computers rarely have this problem. And many modern programming languages (ML, Modula-3, Java) allow forward reference to identifiers defined later in the same module; using an abstract syntax tree makes compilation easier for these languages. It may be that Pascal and C require clumsy *forward* declarations because their designers wanted

to avoid an extra compiler pass on the machines of the 1970s.

Grammar 4.9 shows the abstract syntax of a straight-line-program language. This grammar is completely impractical for parsing: the grammar is quite ambiguous, since precedence of the operators is not specified, and many of the punctuation keywords are missing.

However, Grammar 4.9 is not meant for parsing. The parser uses the *concrete syntax* (Program 4.4) to build a parse tree for the *abstract syntax*. The semantic analysis phase takes this *abstract syntax tree*; it is not bothered by the ambiguity of the grammar, since it already has the parse tree!

The compiler will need to represent and manipulate abstract syntax trees as data structures. In Java, these data structures are organized according to the principles outlined in Section 1.3: an abstract class for each nonterminal, a subclass for each production, and so on. In fact, the classes of Programs 4.7 and 4.6 are abstract syntax classes for Grammar 4.9. An alternate arrangement, with all the different binary operators grouped into an `OpExp` class, is shown in Program 1.5.

IS IT OBJECT-ORIENTED?

The abstract syntax classes of Program 1.5 have no methods. The constructors build a syntax-tree data structure, which can then be examined by using `instanceof` and fetching public class variables that represent subtrees. This is a *syntax separate from interpretations* style of programming.

In contrast, the classes of Programs 4.6 and 4.7 have no public instance variables. Any use of these objects must be through their `eval` method. This is an *object-oriented* style of programming.

The choice of style affects the modularity of the compiler. In a situation such as this, we have several *kinds* of objects: compound statements, assignment statements, print statements, and so on. And we also may have several different *interpretations* of these objects: type-check, translate to Pentium code, translate to Sparc code, optimize, interpret, and so on.

Each *interpretation* must be applied to each *kind*; if we add a new kind we must implement each interpretation for it, and if we add a new interpretation we must implement it for each kind. Figure 4.10 illustrates the orthogonality of kinds and interpretations – for compilers, and for graphic user interfaces, where the *kinds* are different widgets and gadgets, and the *interpretations* are move, hide, and redisplay commands.

If the *syntax separate from interpretations* style is used, then it is easy and modular to add a new *interpretation:* one new function is written, with clauses

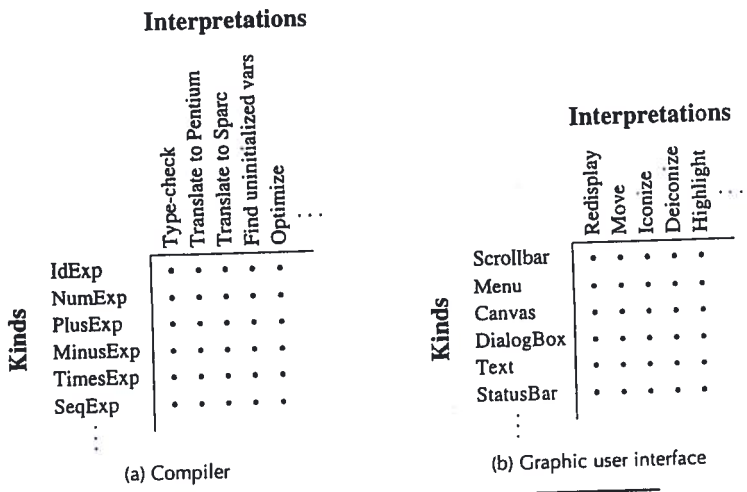


FIGURE 4.10. Orthogonal directions of modularity.

for the different kinds all grouped logically together. On the other hand, it will not be modular to add a new *kind*, since a new clause must be added to every interpretation function.

With the *object-oriented* style, each interpretation is just a *method* in all the classes. It is easy and modular to add a new *kind:* all the interpretations of that kind are grouped together as methods of the new class. But it is not modular to add a new *interpretation:* a new method must be added to every class.

For graphic user interfaces, each application will want to make its own kinds of widgets; it is impossible to predetermine one set of widgets for everyone to use. On the other hand, the set of common operations (interpretations) is fixed: the window manager demands that each widget support only a certain interface. Thus, the *object-oriented* style works well, and the *syntax separate from interpretations* style would not be as modular.

For programming languages, on the other hand, it works very well to fix a syntax and then provide many interpretations of that syntax. If we have a compiler where one interpretation is *translate to Pentium* and we wish to port that compiler to the Sparc, then not only must we add operations for generating Sparc code but we might also want to remove (in this configuration)

the Pentium code generation functions. This would be very inconvenient in the object-oriented style, requiring each class file to be edited. In the *syntax separate from interpretations* style, such a change is modular: we remove a Pentium-related module and add a Sparc module.

Thus, in this book we will generally use a non-object-oriented style for abstract syntax, and for other similar intermediate representations.

The CUP (or recursive-descent) parser, parsing the *concrete syntax*, constructs the *abstract* syntax tree. In fact, whether the syntax/interpretations style or the object-oriented style is used, the semantic actions of the parser look the same, as shown in Program 4.4.

POSITIONS

In a one-pass compiler, lexical analysis, parsing, and semantic analysis (type-checking) are all done simultaneously. If there is a type error that must be reported to the user, the *current* position of the lexical analyzer is a reasonable approximation of the source position of the error. In such a compiler, the lexical analyzer keeps a "current position" global variable, and the error-message routine just prints the value of that variable with each message.

A compiler that uses abstract-syntax-tree data structures need not do all the parsing and semantic analysis in one pass. This makes life easier in many ways, but slightly complicates the production of semantic error messages. The lexer reaches the end of file before semantic analysis even begins; so if a semantic error is detected in traversing the abstract syntax tree, the *current* position of the lexer (at end of file) will not be useful in generating a line-number for the error message. Thus, the source-file position of each node of the abstract syntax tree must be remembered, in case that node turns out to contain a semantic error.

To remember positions accurately, the abstract-syntax data structures must be sprinkled with `pos` fields. These indicate the position, within the original source file, of the characters from which these abstract syntax structures were derived. Then the type-checker can produce useful error messages.

The lexer must pass the source-file positions of the beginning and end of each token to the parser. We can augment the (abstract) classes `Exp` and `Stm` with a `position` field; then each constructor for all the subclasses of `Exp` and `Stm` must take a `pos` argument to initialize this field. The positions of leaf nodes of the syntax tree can be obtained from the tokens returned by the lexical analyzer; internal-node positions can be derived from the positions of their subtrees. This is tedious but straightforward.

ABSTRACT SYNTAX FOR Tiger

Figure 4.11 shows classes for the abstract syntax of Tiger. The meaning of each constructor in the abstract syntax should be clear after a careful study of Appendix A, but there are a few points that merit explanation.

In Java, definition of `Exp` and its subclass `StringExp` would actually be written as

```
/* Absyn/Absyn.java */
package Absyn;
abstract public class Absyn {public int pos;}

/* Absyn/Exp.java */
package Absyn;
abstract public class Exp extends Absyn {}

/* Absyn/StringExp.java */
package Absyn;
public class StringExp extends Exp {
   public String value;
   public StringExp(int p, String v) {pos=p; value=v;}
}
```

The Tiger program

```
(a := 5; a+1)
```

translates into abstract syntax as

```
SeqExp(1,ExpList(
          AssignExp(4,SimpleVar(2,symbol("a")),
                      IntExp(7,5)),
        ExpList(
          OpExp(11,VarExp(10,SimpleVar(10,symbol("a"))),
          PLUS,
          IntExp(12,1)),
        null)))
```

This is a *sequence expression* containing two expressions separated by a semicolon: an *assignment expression* and an *operator expression*. Within these are a *variable expression* and two *integer constant expressions*.

The positions (1, 4, 2, 7, 11, 10, 10, 12) sprinkled throughout are source-code character count. The position I have chosen to associate with an `AssignExp` is that of the `:=` operator, for an `OpExp` that of the `+` operator, and so on.


```
package Absyn;

abstract class Var
SimpleVar(int pos, Symbol name)
FieldVar(int pos, Var var, Symbol field)
SubscriptVar(int pos, Var var, Exp index)

abstract class Exp
VarExp(int pos, Var var)
NilExp(int pos)
IntExp(int pos, int value)
StringExp(int pos, String value)
CallExp(int pos, Symbol func, ExpList args)
OpExp(int pos, Exp left, int oper, Exp right)
RecordExp(int pos, Symbol typ, FieldExpList fields)
SeqExp(int pos, ExpList list)
AssignExp(int pos, Var var, Exp exp)
IfExp(int pos, Exp test, Exp thenclause)  // elseclause is null
IfExp(int pos, Exp test, Exp thenclause, Exp elseclause)
WhileExp(int pos, Exp test, Exp body)
ForExp(int pos, VarDec var, Exp hi, Exp body)
BreakExp(int pos)
LetExp(int pos, DecList decs, Exp body)
ArrayExp(int pos, Symbol typ, Exp size, Exp init)

abstract class Dec
FunctionDec(int pos, Symbol name, FieldList params, NameTy result,
            Exp body, FunctionDec next)   // result may be null
VarDec(int pos, Symbol name, NameTy typ, Exp init)  // typ may be null
TypeDec(int pos, Symbol name, Ty ty, TypeDec next)

abstract class Ty
NameTy(int pos, Symbol name)
RecordTy(int pos, FieldList fields)
ArrayTy(int pos, Symbol typ)

 miscellaneous classes
DecList(Dec head, DecList tail)
ExpList(Exp head, ExpList tail)
FieldExpList(int pos, Symbol name, Exp init, FieldExpList tail)
FieldList(int pos, Symbol name, Symbol typ, FieldList tail)

 constants for oper field of OpExp
final static int OpExp.PLUS, OpExp.MINUS, OpExp.MUL, OpExp.DIV,
   OpExp.EQ, OpExp.NE, OpExp.LT, OpExp.LE, OpExp.GT, OpExp.GE;
```

FIGURE 4.11. Abstract syntax for the Tiger language. Only the constructors are shown; the object field variables correspond exactly to the names of the constructor arguments.

These decisions are a matter of taste; they represent my guesses about how they will look when included in semantic error messages.

Now consider

```
let var a := 5
    function f() : int = g(a)
    function g(i: int) = f()
 in f()
end
```

The Tiger language treats *adjacent* function declarations as (possibly) mutually recursive. The `FunctionDec` constructor of the abstract syntax takes a *list* of function declarations, not just a single function. The intent is that this list is a maximal consecutive sequence of function declarations. Thus, functions declared by the same `FunctionDec` can be mutually recursive. Therefore, this program translates into the abstract syntax,

```
new LetExp(
  new DecList(new VarDec(symbol("a"),
                  null,new IntExp(5)),
  new DecList(new FunctionDec(symbol("f"),null,symbol("int"),
                  new CallExp(symbol("g"), ···),
                new FunctionDec(symbol("g"),
                  new FieldList(symbol("i"),symbol("int"),null),
                  null,
                  new CallExp(symbol("f"), ···),
                null)),
   null)),
   new CallExp(symbol("f"), null))
```

where the positions are omitted for clarity.

The `TypeDec` constructor also takes a list of type declarations, for the same reason; consider the declarations

```
type tree = {key: int, children: treelist}
type treelist = {head: tree, tail: treelist}
```

which translate to *one* `DecList` element containing a two-`TypeDec` sequence:

```
new DecList(
 new TypeDec((symbol("tree"),
   new RecordTy(
     new FieldList(symbol("key"),symbol("int"),
```



```
     new FieldList(symbol("children"),symbol("treelist"),
     null))),
 new TypeDec(symbol("treelist"),
   new RecordTy(
     new FieldList(symbol("head"),symbol("tree"),
     new FieldList(symbol("tail"),symbol("treelist"),
     null))),
   null))),
 null)
```

There is no abstract syntax for "`&`" and "`|`" expressions; instead, e_1`&`e_2 is translated as `if` e_1 `then` e_2 `else 0`, and e_1`|`e_2 is translated as though it had been written `if` e_1 `then 1 else` e_2.

Similarly, unary negation ($-i$) should be represented as subtraction ($0 - i$) in the abstract syntax.[1] Also, where the body of a `LetExp` has multiple statements, we must use a `SeqExp`. An empty statement is represented by `SeqExp(null)`.

By using these representations for `&`, `|`, and unary negation, we keep the abstract syntax data type smaller and make fewer cases for the semantic analysis phase to process. On the other hand, it becomes harder for the type-checker to give meaningful error messages that relate to the source code.

The lexer returns `ID` tokens with `string` values. The abstract syntax requires identifiers to have `symbol` values. Function `Symbol.Symbol.symbol` (package `Symbol`, class `Symbol`, method `symbol`) converts strings to symbols, and the method `toString()` converts back. The representation of symbols is discussed in Chapter 5.

The semantic analysis phase of the compiler will need to keep track of which local variables are used from within nested functions. The `escape` component of a `VarDec` or `FieldList` is used to keep track of this. This `escape` field is not mentioned in the class constructor parameters, but is always initialized to `true`, which is a conservative approximation. The class `FieldList` is used for both formal parameters and record fields; `escape` has meaning for formal parameters, but for record fields it can be ignored.

Having the `escape` fields in the abstract syntax is a "hack," since escaping is a global, nonsyntactic property. But leaving `escape` out of the `Absyn` would require another data structure for describing escapes.

[1] This might not be adequate in an industrial-strength compiler. The most negative two's complement integer of a given size cannot be represented as $0 - i$ for any i of the same size. In floating point numbers, $0 - x$ is not the same as $-x$ if $x = 0$. We will neglect these issues in the Tiger compiler.

PROGRAM ABSTRACT SYNTAX

Add semantic actions to your parser to produce abstract syntax for the Tiger language.

You should turn in the file `Grm.cup`.

For modularity, all the abstract syntax classes reside in the package `Absyn`. This means that all uses of the constructors must be prefixed by "`Absyn.`" which is tedious but straightforward.

Supporting files available in `$TIGER/chap4` include:

`Absyn/*` The abstract syntax classes for Tiger.
`Absyn/Print.java` A pretty-printer for abstract syntax trees, so you can see your results.
`ErrorMsg/*` As before.
`Parse/Yylex.class` Use this only if your own lexical analyzer still isn't working.
`Symbol/*` A module to turn strings into `symbols`.
`makefile` As usual.
`Parse/Parse.java` A driver to run your parser on an input file.
`Grm.cup` The skeleton of a grammar specification.

FURTHER READING

Many compilers mix recursive-descent parsing code with semantic-action code, as shown in Program 4.1; Gries [1971] and Fraser and Hanson [1995] are ancient and modern examples. Machine-generated parsers with semantic actions (in special-purpose "semantic-action mini-languages") attached to the grammar productions were tried out in 1960s [Feldman and Gries 1968]; Yacc [Johnson 1975] was one of the first to permit semantic action fragments to be written in a conventional, general-purpose programming language.

The notion of *abstract syntax* is due to McCarthy [1963], who designed the abstract syntax for Lisp [McCarthy et al. 1962]. The abstract syntax was intended to be used writing programs until designers could get around to creating a concrete syntax with human-readable punctuation (instead of Lots of Irritating Silly Parentheses), but programmers soon got used to programming directly in abstract syntax.

The search for a theory of programming-language semantics, and a notation for expressing semantics in a compiler-compiler, led to ideas such as

denotational semantics [Stoy 1977]. The semantic interpreter shown in Programs 4.4–4.7 is inspired by ideas from denotational semantics, as is the idea of separating concrete syntax from semantics using the abstract syntax as a clean interface.

EXERCISES

4.1 Write a package of Java classes to express the abstract syntax of regular expressions.

4.2 When Program 4.4 interprets straight-line programs, statements embedded inside expressions have no permanent effect – any assignments made within those statements "disappear" at the closing parenthesis. Thus, the program

```
a := 6; a := (a := a+1, a+4) + a; print(a)
```

prints 17 instead of 18. To fix this problem, change the type of semantic values for exp so that it can produce a value *and* a new table; that is:

```
class ValAndTable {int val; Table table;
                   ValAndTable(···){···}}
abstract class Exp {ValAndTable eval(Table env);}
```

Then change the semantic actions of Program 4.4 accordingly.

4.3 Program 4.4 is not a purely functional program; the semantic action for PRINT contains a Java print statement, which is a side effect. You can make the interpreter purely functional by having each statement return, not only a table, but also a list of values that would have been printed. Thus,

```
class IntList {int head; IntList tail;
               IntList(···) {···} }
class TableAndIntList {Table table;  IntList list;
                   TableAndIntList(···){···} }
abstract class Stm {TableAndIntList eval(Table env);}
```

Adjust the `Exp` class accordingly, and rewrite Program 4.4.

4.4 Combine the ideas of Exercises 4.2 and 4.3 to write a purely functional version of the interpreter that handles printing and statements-within-expressions correctly.

4.5 Implement Program 4.4 as a recursive-descent parser, with the semantic actions embedded in the parsing functions.

4 Abstract Syntax

ab-stract: disassociated from any specific instance

Webster's Dictionary

A compiler must do more than recognize whether a sentence belongs to the language of a grammar – it must do something useful with that sentence. The *semantic actions* of a parser can do useful things with the phrases that are parsed.

In a recursive-descent parser, semantic action code is interspersed with the control flow of the parsing actions. In a parser specified in JavaCC, semantic actions are fragments of Java program code attached to grammar productions. SableCC, on the other hand, automatically generates syntax trees as it parses.

4.1 SEMANTIC ACTIONS

Each terminal and nonterminal may be associated with its own type of semantic value. For example, in a simple calculator using Grammar 3.37, the type associated with exp and INT might be int; the other tokens would not need to carry a value. The type associated with a token must, of course, match the type that the lexer returns with that token.

For a rule $A \rightarrow B\ C\ D$, the semantic action must return a value whose type is the one associated with the nonterminal A. But it can build this value from the values associated with the matched terminals and nonterminals B, C, D.

RECURSIVE DESCENT

In a recursive-descent parser, the semantic actions are the values returned by parsing functions, or the side effects of those functions, or both. For each ter-


```
class Token {int kind; Object val;
             Token(int k, Object v) {kind=k; val=v;}
            }
final int EOF=0, ID=1, NUM=2, PLUS=3, MINUS=4, ···

int lookup(String id) { ··· }

int F_follow[] = { PLUS, TIMES, RPAREN, EOF };

int F() {switch (tok.kind) {
         case ID:   int i=lookup((String)(tok.val)); advance(); return i;
         case NUM:  int i=((Integer)(tok.val)).intValue();
                    advance(); return i;
         case LPAREN:  eat(LPAREN);
                       int i = E();
                       eatOrSkipTo(RPAREN, F_follow);
                       return i;
         case EOF:
         default:   print("expected ID, NUM, or left-paren");
                    skipto(F_follow); return 0;
        }}

int T_follow[] = { PLUS, RPAREN, EOF };

int T() {switch (tok.kind) {
         case ID:
         case NUM:
         case LPAREN: return Tprime(F());
         default: print("expected ID, NUM, or left-paren");
                  skipto(T_follow);
                  return 0;
        }}

int Tprime(int a)  {switch (tok.kind) {
        case TIMES: eat(TIMES); return Tprime(a*F());
        case PLUS:
        case RPAREN:
        case EOF:  return a;
        default:  ···
       }}

void eatOrSkipTo(int expected, int[] stop) {
   if (tok.kind==expected)
        eat(expected);
   else {print(···); skipto(stop);}
}
```

PROGRAM 4.1. Recursive-descent interpreter for part of Grammar 3.15.

```
void Start() :
{ int i; }
{ i=Exp() <EOF>  { System.out.println(i); }
}
int Exp() :
{ int a,i; }
{ a=Term()
  ( "+" i=Term() { a=a+i; }
  | "-" i=Term() { a=a-i; }
  )*
  { return a; }
}
int Term() :
{ int a,i; }
{ a=Factor()
  ( "*" i=Factor() { a=a*i; }
  | "/" i=Factor() { a=a/i; }
  )*
  { return a; }
}
int Factor() :
{ Token t; int i; }
{ t=<IDENTIFIER>       { return lookup(t.image); }
| t=<INTEGER_LITERAL> { return Integer.parseInt(t.image); }
| "(" i=Exp() ")"      { return i; }
}
```

PROGRAM 4.2. JavaCC version of a variant of Grammar 3.15.

minal and nonterminal symbol, we associate a *type* (from the implementation language of the compiler) of *semantic values* representing phrases derived from that symbol.

Program 4.1 is a recursive-descent interpreter for part of Grammar 3.15. The tokens `ID` and `NUM` must now carry values of type `string` and `int`, respectively. We will assume there is a lookup table mapping identifiers to integers. The type associated with E, T, F, etc., is `int`, and the semantic actions are easy to implement.

The semantic action for an artificial symbol such as T' (introduced in the elimination of left recursion) is a bit tricky. Had the production been $T \rightarrow T * F$, then the semantic action would have been

```
int a = T(); eat(TIMES); int b=F(); return a*b;
```


With the rearrangement of the grammar, the production $T' \rightarrow *FT'$ is missing the left operand of the $*$. One solution is for T to pass the left operand as an argument to T', as shown in Program 4.1.

AUTOMATICALLY GENERATED PARSERS

A parser specification for JavaCC consists of a set of grammar rules, each annotated with a semantic action that is a Java statement. Whenever the generated parser reduces by a rule, it will execute the corresponding semantic action fragment.

Program 4.2 shows how this works for a variant of Grammar 3.15. Every `INTEGER_CONSTANT` terminal and every nonterminal (except `Start`) carries a value. To access this value, give the terminal or nonterminal a name in the grammar rule (such as `i` in Program 4.2), and access this name as a variable in the semantic action.

SableCC, unlike JavaCC, has no way to attach action code to productions. However, SableCC automatically generates syntax tree classes, and a parser generated by SableCC will build syntax trees using those classes. For JavaCC, there are several companion tools, including JJTree and JTB (the Java Tree Builder), which, like SableCC, generate syntax tree classes and insert action code into the grammar for building syntax trees.

4.2 ABSTRACT PARSE TREES

It is possible to write an entire compiler that fits within the semantic action phrases of a JavaCC or SableCC parser. However, such a compiler is difficult to read and maintain, and this approach constrains the compiler to analyze the program in exactly the order it is parsed.

To improve modularity, it is better to separate issues of syntax (parsing) from issues of semantics (type-checking and translation to machine code). One way to do this is for the parser to produce a *parse tree* – a data structure that later phases of the compiler can traverse. Technically, a parse tree has exactly one leaf for each token of the input and one internal node for each grammar rule reduced during the parse.

Such a parse tree, which we will call a *concrete parse tree*, representing the *concrete syntax* of the source language, may be inconvenient to use directly. Many of the punctuation tokens are redundant and convey no information – they are useful in the input string, but once the parse tree is built, the structure

$E \to E + E$
$E \to E - E$
$E \to E * E$
$E \to E / E$
$E \to$ id
$E \to$ num

GRAMMAR 4.3. Abstract syntax of expressions.

of the tree conveys the structuring information more conveniently.

Furthermore, the structure of the parse tree may depend too much on the grammar! The grammar transformations shown in Chapter 3 – factoring, elimination of left recursion, elimination of ambiguity – involve the introduction of extra nonterminal symbols and extra grammar productions for technical purposes. These details should be confined to the parsing phase and should not clutter the semantic analysis.

An *abstract syntax* makes a clean interface between the parser and the later phases of a compiler (or, in fact, for the later phases of other kinds of program-analysis tools such as dependency analyzers). The abstract syntax tree conveys the phrase structure of the source program, with all parsing issues resolved but without any semantic interpretation.

Many early compilers did not use an abstract syntax data structure because early computers did not have enough memory to represent an entire compilation unit's syntax tree. Modern computers rarely have this problem. And many modern programming languages (ML, Modula-3, Java) allow forward reference to identifiers defined later in the same module; using an abstract syntax tree makes compilation easier for these languages. It may be that Pascal and C require clumsy *forward* declarations because their designers wanted to avoid an extra compiler pass on the machines of the 1970s.

Grammar 4.3 shows an abstract syntax of the expression language is Grammar 3.15. This grammar is completely impractical for parsing: The grammar is quite ambiguous, since precedence of the operators is not specified.

However, Grammar 4.3 is not meant for parsing. The parser uses the *concrete syntax* to build a parse tree for the *abstract syntax*. The semantic analysis phase takes this *abstract syntax tree*; it is not bothered by the ambiguity of the grammar, since it already has the parse tree!

The compiler will need to represent and manipulate abstract syntax trees as


```
Exp Start() :
  { Exp e; }
  { e=Exp() { return e; }
  }
Exp Exp() :
  { Exp e1,e2; }
  { e1=Term()
      ( "+" e2=Term() { e1=new  PlusExp(e1,e2); }
      | "-" e2=Term() { e1=new MinusExp(e1,e2); }
      )*
    { return e1; }
  }
Exp Term() :
  { Exp e1,e2; }
  { e1=Factor()
      ( "*" e2=Factor() { e1=new  TimesExp(e1,e2); }
      | "/" e2=Factor() { e1=new DivideExp(e1,e2); }
      )*
    { return e1; }
  }
Exp Factor() :
  { Token t; Exp e; }
  { ( t=<IDENTIFIER>       { return new Identifier(t.image); } |
      t=<INTEGER_LITERAL> { return new IntegerLiteral(t.image); } |
      "(" e=Exp() ")"      { return e; } )
  }
```

PROGRAM 4.4. Building syntax trees for expressions.

data structures. In Java, these data structures are organized according to the principles outlined in Section 1.3: an abstract class for each nonterminal, a subclass for each production, and so on. In fact, the classes of Program 4.5 are abstract syntax classes for Grammar 4.3. An alternate arrangement, with all the different binary operators grouped into an `OpExp` class, is also possible.

Let us write an interpreter for the expression language in Grammar 3.15 by first building syntax trees and then interpreting those trees. Program 4.4 is a JavaCC grammar with semantic actions that produce syntax trees. Each class of syntax-tree nodes contains an `eval` function; when called, such a function will return the value of the represented expression.

POSITIONS

In a one-pass compiler, lexical analysis, parsing, and semantic analysis (type-checking) are all done simultaneously. If there is a type error that must be reported to the user, the *current* position of the lexical analyzer is a reason-

```
public abstract class Exp {
   public abstract int eval();
}
public class PlusExp extends Exp {
   private Exp e1,e2;
   public PlusExp(Exp a1, Exp a2) { e1=a1; e2=a2; }
   public int eval() {
       return e1.eval()+e2.eval();
   }
}
public class MinusExp extends Exp {
   private Exp e1,e2;
   public MinusExp(Exp a1, Exp a2) { e1=a1; e2=a2; }
   public int eval() {
       return e1.eval()-e2.eval();
   }
}
public class TimesExp extends Exp {
   private Exp e1,e2;
   public TimesExp(Exp a1, Exp a2) { e1=a1; e2=a2; }
   public int eval() {
       return e1.eval()*e2.eval();
   }
}
public class DivideExp extends Exp {
   private Exp e1,e2;
   public DivideExp(Exp a1, Exp a2) { e1=a1; e2=a2; }
   public int eval() {
       return e1.eval()/e2.eval();
   }
}
public class Identifier extends Exp {
   private String f0;
   public Identifier(String n0) { f0 = n0; }
   public int eval() {
       return lookup(f0);
   }
}
public class IntegerLiteral extends Exp {
   private String f0;
   public IntegerLiteral(String n0) { f0 = n0; }
   public int eval() {
       return Integer.parseInt(f0);
   }
}
```

PROGRAM 4.5. Exp class for Program 4.4.

able approximation of the source position of the error. In such a compiler, the lexical analyzer keeps a "current position" global variable, and the error-message routine just prints the value of that variable with each message.

A compiler that uses abstract-syntax-tree data structures need not do all the parsing and semantic analysis in one pass. This makes life easier in many ways, but slightly complicates the production of semantic error messages. The lexer reaches the end of file before semantic analysis even begins; so if a semantic error is detected in traversing the abstract syntax tree, the *current* position of the lexer (at end of file) will not be useful in generating a line number for the error message. Thus, the source-file position of each node of the abstract syntax tree must be remembered, in case that node turns out to contain a semantic error.

To remember positions accurately, the abstract-syntax data structures must be sprinkled with `pos` fields. These indicate the position, within the original source file, of the characters from which these abstract-syntax structures were derived. Then the type-checker can produce useful error messages. (The syntax constructors we will show in Figure 4.9 do not have `pos` fields; any compiler that uses these exactly as given will have a hard time producing accurately located error messages.)

The lexer must pass the source-file positions of the beginning and end of each token to the parser. We can augment the types `Exp`, etc. with a `position` field; then each constructor must take a `pos` argument to initialize this field. The positions of leaf nodes of the syntax tree can be obtained from the tokens returned by the lexical analyzer; internal-node positions can be derived from the positions of their subtrees. This is tedious but straightforward.

4.3 VISITORS

Each abstract syntax class of Program 4.5 has a constructor for building syntax trees, and an `eval` method for returning the value of the represented expression. This is an *object-oriented* style of programming. Let us consider an alternative.

Suppose the code for evaluating expressions is written *separately* from the abstract syntax classes. We might do that by examining the syntax-tree data structure by using `instanceof` and by fetching public class variables that represent subtrees. This is a *syntax separate from interpretations* style of programming.

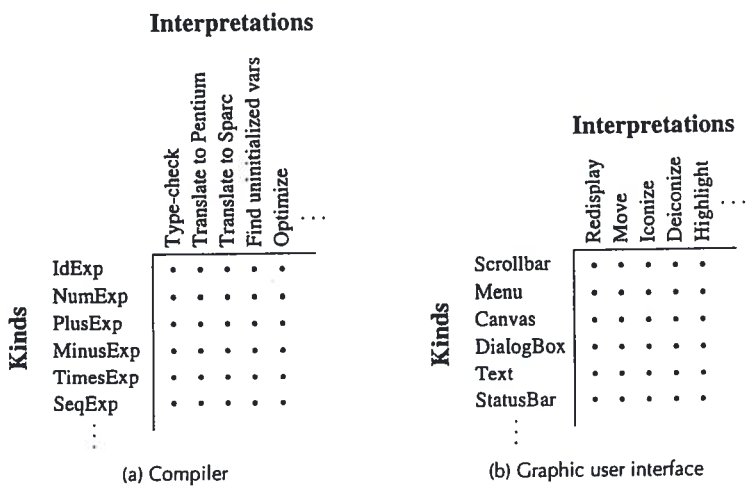


FIGURE 4.6. Orthogonal directions of modularity.

The choice of style affects the modularity of the compiler. In a situation such as this, we have several *kinds* of objects: compound statements, assignment statements, print statements, and so on. And we also may have several different *interpretations* of these objects: type-check, translate to Pentium code, translate to Sparc code, optimize, interpret, and so on.

Each *interpretation* must be applied to each *kind*; if we add a new kind, we must implement each interpretation for it; and if we add a new interpretation, we must implement it for each kind. Figure 4.6 illustrates the orthogonality of kinds and interpretations – for compilers, and for graphic user interfaces, where the *kinds* are different widgets and gadgets, and the *interpretations* are move, hide, and redisplay commands.

If the *syntax separate from interpretations* style is used, then it is easy and modular to add a new *interpretation:* One new function is written, with clauses for the different kinds all grouped logically together. On the other hand, it will not be modular to add a new *kind*, since a new clause must be added to every interpretation function.

With the *object-oriented* style, each interpretation is just a *method* in all the classes. It is easy and modular to add a new *kind:* All the interpretations of that kind are grouped together as methods of the new class. But it is not


```
public abstract class Exp {
   public abstract int accept(Visitor v);
}
public class PlusExp extends Exp {
   public Exp e1,e2;
   public PlusExp(Exp a1, Exp a2) { e1=a1; e2=a2; }
   public int accept(Visitor v) {
       return v.visit(this);
   }
}
public class MinusExp extends Exp {
   public Exp e1,e2;
   public MinusExp(Exp a1, Exp a2) { e1=a1; e2=a2; }
   public int accept(Visitor v) {
       return v.visit(this);
   }
}
public class TimesExp extends Exp {
   public Exp e1,e2;
   public TimesExp(Exp a1, Exp a2) { e1=a1; e2=a2; }
   public int accept(Visitor v) {
       return v.visit(this);
   }
}
public class DivideExp extends Exp {
   public Exp e1,e2;
   public DivideExp(Exp a1, Exp a2) { e1=a1; e2=a2; }
   public int accept(Visitor v) {
       return v.visit(this);
   }
}
public class Identifier extends Exp {
   public String f0;
   public Identifier(String n0) { f0 = n0; }
   public int accept(Visitor v) {
       return v.visit(this);
   }
}
public class IntegerLiteral extends Exp {
   public String f0;
   public IntegerLiteral(String n0) { f0 = n0; }
   public int accept() {
       return v.visit(this);
   }
}
```

PROGRAM 4.7. Syntax classes with accept methods.

```
public interface Visitor {
   public int visit(PlusExp n);
   public int visit(MinusExp n);
   public int visit(TimesExp n);
   public int visit(DivideExp n);
   public int visit(Identifier n);
   public int visit(IntegerLiteral n);
}

public class Interpreter implements Visitor {
   public int visit(PlusExp n) {
       return n.e1.accept(this)+n.e2.accept(this);
   }
   public int visit(MinusExp n) {
       return n.e1.accept(this)-n.e2.accept(this);
   }
   public int visit(TimesExp n) {
       return n.e1.accept(this)*n.e2.accept(this);
   }
   public int visit(DivideExp n) {
       return n.e1.accept(this)/n.e2.accept(this);
   }
   public int visit(Identifier n) {
       return lookup(n.f0);
   }
   public int visit(IntegerLiteral n) {
       return Integer.parseInt(n.f0);
   }
}
```

PROGRAM 4.8. An interpreter visitor.

modular to add a new *interpretation:* A new method must be added to every class.

For graphic user interfaces, each application will want to make its own kinds of widgets; it is impossible to predetermine one set of widgets for everyone to use. On the other hand, the set of common operations (interpretations) is fixed: The window manager demands that each widget support only a certain interface. Thus, the *object-oriented* style works well, and the *syntax separate from interpretations* style would not be as modular.

For programming languages, on the other hand, it works very well to fix a syntax and then provide many interpretations of that syntax. If we have a compiler where one interpretation is *translate to Pentium* and we wish to port that compiler to the Sparc, then not only must we add operations for generating Sparc code but we might also want to remove (in this configuration) the Pentium code-generation functions. This would be very inconvenient in the object-oriented style, requiring each class to be edited. In the *syntax separate from interpretations* style, such a change is modular: We remove a Pentium-related module and add a Sparc module.

We prefer a syntax-separate-from-interpretations style. Fortunately, we can use this style without employing `instanceof` expressions for accessing syntax trees. Instead, we can use a technique known as the Visitor pattern. A visitor implements an interpretation; it is an object which contains a `visit` method for each syntax-tree class. Each syntax-tree class should contain an `accept` method. An `accept` method serves as a hook for all interpretations. It is called by a visitor and it has just one task: It passes control back to an appropriate method of the visitor. Thus, control goes back and forth between a visitor and the syntax-tree classes.

Intuitively, the visitor calls the `accept` method of a node and asks "what is your class?" The `accept` method answers by calling the corresponding `visit` method of the visitor. Code for the running example, using visitors, is given in Programs 4.7 and 4.8. Every visitor implements the interface `Visitor`. Notice that each `accept` method takes a visitor as an argument, and that each `visit` method takes a syntax-tree-node object as an argument.

In Programs 4.7 and 4.8, the `visit` and `accept` methods all return `int`. Suppose we want instead to return `String`. In that case, we can add an appropriate `accept` method to each syntax tree class, and we can write a new visitor class in which all `visit` methods return `String`.

The main difference between the object-oriented style and the syntax-separate-from-interpretations style is that, for example, the interpreter code in Program 4.5 is in the `eval` methods while in Program 4.8 it is in the `Interpreter` visitor.

In summary, with the Visitor pattern we can add a new interpretation without editing and recompiling existing classes, provided that each of the appropriate classes has an `accept` method. The following table summarizes some advantages of the Visitor pattern:

	Frequent type casts?	Frequent recompilation?
Instanceof and type casts	Yes	No
Dedicated methods	No	Yes
The Visitor pattern	No	No

ABSTRACT SYNTAX FOR MiniJava

Figure 4.9 shows classes for the abstract syntax of MiniJava. The meaning of each constructor in the abstract syntax should be clear after a careful study of Appendix A, but there are a few points that merit explanation.

Only the constructors are shown in Figure 4.9; the object field variables correspond exactly to the names of the constructor arguments. Each of the six list classes is implemented in the same way, for example:

```
public class ExpList {
   private Vector list;
   public ExpList() {
      list = new Vector();
   }
   public void addElement(Exp n) {
      list.addElement(n);
   }
   public Exp elementAt(int i)  {
      return (Exp)list.elementAt(i);
   }
   public int size() {
      return list.size();
   }
}
```

Each of the nonlist classes has an accept method for use with the visitor pattern. The interface `Visitor` is shown in Program 4.10.

We can construct a syntax tree by using nested `new` expressions. For example, we can build a syntax tree for the MiniJava statement:

```
x = y.m(1,4+5);
```

using the following Java code:

```
ExpList el = new ExpList();
el.addElement(new IntegerLiteral(1));
el.addElement(new Plus(new IntegerLiteral(4),
                       new IntegerLiteral(5)));
Statement s = new Assign(new Identifier("x"),
                         new Call(new IdentifierExp("y"),
                                  new Identifier("m"),
                                  el));
```

SableCC enables automatic generation of code for syntax tree classes, code for building syntax trees, and code for template visitors. For JavaCC, a companion tool called the Java Tree Builder (JTB) enables the generation of sim-


```
package syntaxtree;

Program(MainClass m, ClassDeclList cl)
MainClass(Identifier i1, Identifier i2, Statement s)

abstract class ClassDecl
ClassDeclSimple(Identifier i, VarDeclList vl, MethodDeclList ml)
ClassDeclExtends(Identifier i, Identifier j,
                 VarDeclList vl, MethodDeclList ml)  see Ch.14

VarDecl(Type t, Identifier i)
MethodDecl(Type t, Identifier i, FormalList fl, VarDeclList vl,
           StatementList sl, Exp e)
Formal(Type t, Identifier i)

abstract class Type
IntArrayType() BooleanType() IntegerType() IdentifierType(String s)

abstract class Statement
Block(StatementList sl)
If(Exp e, Statement s1, Statement s2)
While(Exp e, Statement s)
Print(Exp e)
Assign(Identifier i, Exp e)
ArrayAssign(Identifier i, Exp e1, Exp e2)

abstract class Exp
And(Exp e1, Exp e2)
LessThan(Exp e1, Exp e2)
Plus(Exp e1, Exp e2) Minus(Exp e1, Exp e2) Times(Exp e1, Exp e2)
ArrayLookup(Exp e1, Exp e2)
ArrayLength(Exp e)
Call(Exp e, Identifier i, ExpList el)
IntegerLiteral(int i)
True()
False()
IdentifierExp(String s)
This()
NewArray(Exp e)
NewObject(Identifier i)
Not(Exp e)

Identifier(String s)

 list classes
ClassDeclList() ExpList() FormalList() MethodDeclList() StatementList() VarDeclList()
```

FIGURE 4.9. Abstract syntax for the MiniJava language.

```
public interface Visitor {
  public void visit(Program n);
  public void visit(MainClass n);
  public void visit(ClassDeclSimple n);
  public void visit(ClassDeclExtends n);
  public void visit(VarDecl n);
  public void visit(MethodDecl n);
  public void visit(Formal n);
  public void visit(IntArrayType n);
  public void visit(BooleanType n);
  public void visit(IntegerType n);
  public void visit(IdentifierType n);
  public void visit(Block n);
  public void visit(If n);
  public void visit(While n);
  public void visit(Print n);
  public void visit(Assign n);
  public void visit(ArrayAssign n);
  public void visit(And n);
  public void visit(LessThan n);
  public void visit(Plus n);
  public void visit(Minus n);
  public void visit(Times n);
  public void visit(ArrayLookup n);
  public void visit(ArrayLength n);
  public void visit(Call n);
  public void visit(IntegerLiteral n);
  public void visit(True n);
  public void visit(False n);
  public void visit(IdentifierExp n);
  public void visit(This n);
  public void visit(NewArray n);
  public void visit(NewObject n);
  public void visit(Not n);
  public void visit(Identifier n);
}
```

PROGRAM 4.10. MiniJava visitor

ilar code. The advantage of using such tools is that once the grammar is written, one can go straight on to writing visitors that operate on syntax trees. The disadvantage is that the syntax trees supported by the generated code may be less abstract than one could desire.

PROGRAM ABSTRACT SYNTAX

Add semantic actions to your parser to produce abstract syntax for the MiniJava language. Syntax-tree classes are available in `$MINIJAVA/chap4`, together with a `PrettyPrintVisitor`. If you use JavaCC, you can use JTB to generate the needed code automatically. Similarly, with SableCC, the needed code can be generated automatically.

FURTHER READING

Many compilers mix recursive-descent parsing code with semantic-action code, as shown in Program 4.1; Gries [1971] and Fraser and Hanson [1995] are ancient and modern examples. Machine-generated parsers with semantic actions (in special-purpose "semantic-action mini-languages") attached to the grammar productions were tried out in 1960s [Feldman and Gries 1968]; Yacc [Johnson 1975] was one of the first to permit semantic action fragments to be written in a conventional, general-purpose programming language.

The notion of *abstract syntax* is due to McCarthy [1963], who designed the abstract syntax for Lisp [McCarthy et al. 1962]. The abstract syntax was intended to be used for writing programs until designers could get around to creating a concrete syntax with human-readable punctuation (instead of Lots of Irritating Silly Parentheses), but programmers soon got used to programming directly in abstract syntax.

The search for a theory of programming-language semantics, and a notation for expressing semantics in a compiler-compiler, led to ideas such as *denotational semantics* [Stoy 1977]. The semantic interpreter shown in Programs 4.4 and 4.5 is inspired by ideas from denotational semantics, as is the idea of separating concrete syntax from semantics using the abstract syntax as a clean interface.

EXERCISES

4.1 Write a package of Java classes to express the abstract syntax of regular expressions.

4.2 Extend Grammar 3.15 such that a program is a sequence of either assignment statements or print statements. Each assignment statement assigns an expression

to an implicitly-declared variable; each print statement prints the value of an expression. Extend the interpreter in Program 4.1 to handle the new language.

4.3 Write a JavaCC version of the grammar from Exercise 4.2. Insert Java code for interpreting programs, in the style of Program 4.2.

4.4 Modify the JavaCC grammar from Exercise 4.3 to contain Java code for building syntax trees, in the style of Program 4.4. Write two interpreters for the language: one in object-oriented style and one that uses visitors.

4.5 In `$MINIJAVA/chap4/handcrafted/visitor`, there is a file with a visitor `PrettyPrintVisitor.java` for pretty printing syntax trees. Improve the pretty printing of nested `if` and `while` statements.

4.6 The visitor pattern in Program 4.7 has `accept` methods that return `int`. If one wanted to write some visitors that return integers, others that return class *A*, and yet others that return class *B*, one could modify all the classes in Program 4.7 to add two more `accept` methods, but this would not be very modular. Another way is to make the visitor return `Object` and cast each result, but this loses the benefit of compile-time type-checking. But there is a third way.

Modify Program 4.7 so that all the `accept` methods return `void`, and write two extensions of the `Visitor` class: one that computes an `int` for each `Exp`, and the other that computes a `float` for each `Exp`. Since the `accept` method will return `void`, the visitor object must have an instance variable into which each `accept` method can place its result. Explain why, if one then wanted to write a visitor that computed an object of class *C* for each `Exp`, no more modification of the `Exp` subclasses would be necessary.

5 Semantic Analysis

se-man-tic: of or relating to meaning in language

Webster's Dictionary

The *semantic analysis* phase of a compiler connects variable definitions to their uses, checks that each expression has a correct type, and translates the abstract syntax into a simpler representation suitable for generating machine code.

5.1 SYMBOL TABLES

This phase is characterized by the maintenance of *symbol tables* (also called *environments*) mapping identifiers to their types and locations. As the declarations of types, variables, and functions are processed, these identifiers are bound to "meanings" in the symbol tables. When *uses* (nondefining occurrences) of identifiers are found, they are looked up in the symbol tables.

Each local variable in a program has a *scope* in which it is visible. For example, in a MiniJava method `m`, all formal parameters and local variables declared in `m` are visible only until the end of `m`. As the semantic analysis reaches the end of each scope, the identifier bindings local to that scope are discarded.

An environment is a set of *bindings* denoted by the $\mapsto$ arrow. For example, we could say that the environment σ_0 contains the bindings {`g` $\mapsto$ `string`, `a` $\mapsto$ `int`}, meaning that the identifier `a` is an integer variable and `g` is a string variable.

Consider a simple example in the Java language: