

4.5 Syntax-Directed Translation

Thus far we have explored top down parsing in a way that has been exclusively concerned with syntax. In other words, the programs we have developed can check only for syntax errors; they cannot produce output, and they do not deal at all with *semantics* – the intent or meaning of the source program. For this purpose, we now introduce **action symbols** which are intended to give us the capability of producing output and/or calling other functions while parsing an input string. A grammar containing action symbols is called a **translation grammar**. We will designate the action symbols in a grammar by enclosing them in curly braces $\{\}$. The meaning of the action symbol, by default, is to produce output – the action symbol itself.

To find the selection sets in a translation grammar, simply remove all the action symbols. This results in what is called the **underlying grammar**. Note that a rule of the form:

$$A \rightarrow \{\text{action}\}$$

is an epsilon rule in the underlying grammar. An example of a translation grammar to translate infix expressions involving addition and multiplication to postfix form is shown below.

G17:

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

The underlying grammar of grammar G17 is grammar G16 in Section 4.4. A derivation tree for the expression $\text{var} + \text{var} * \text{var}$ is shown in Figure 4.13 (see p. 134). Note that the action symbols are shown as well. Listing the leaves of the derivation tree, the derived string is shown below:

$\text{var } \{\text{var}\} + \text{var } \{\text{var}\} * \text{var } \{\text{var}\} \{*\} \{+\}$

in which input symbols and action symbols are interspersed. Separating out the action symbols gives the output defined by the translation grammar:

$\{\text{var}\} \{\text{var}\} \{\text{var}\} \{*\} \{+\}$

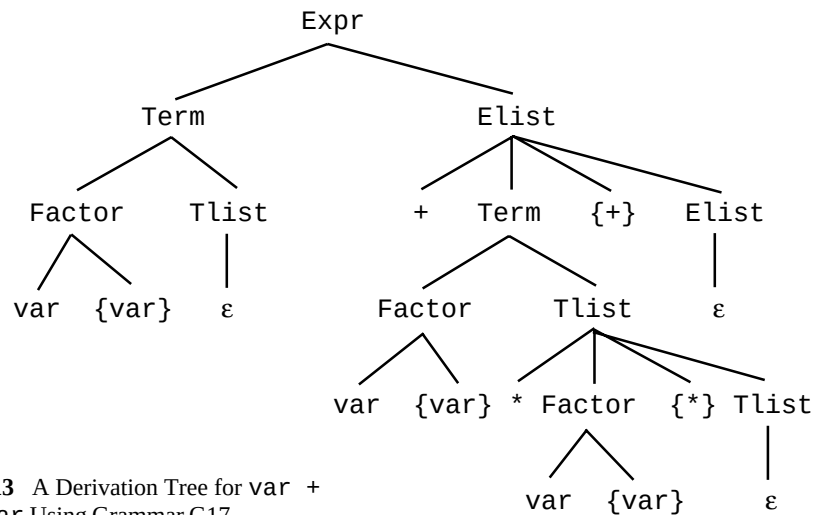


Figure 4.13 A Derivation Tree for `var + var * var` Using Grammar G17

4.5.1 Implementing Translation Grammars with Pushdown Translators

To implement a translation grammar with a pushdown machine, action symbols should be treated as stack symbols and are pushed onto the stack in exactly the same way as terminals and nonterminals occurring on the right side of a grammar rule. In addition, each action symbol $\{A\}$ representing output should label a row of the pushdown machine table. Every column of that row should contain the entry $\text{Out}(A)$, Pop , Retain . A pushdown machine to parse and translate infix expressions into postfix, according to the translation grammar G17, is shown in Figure 4.14 (see p. 135), in which all empty cells are assumed to be **Reject**.

4.5.2 Implementing Translation Grammars with Recursive Descent

To implement a translation grammar with a recursive descent translator, action symbols should be handled as they are encountered in the right side of a grammar rule. Normally this will mean simply writing out the action symbol, but it could mean to call a function, depending on the purpose of the action symbol. The student should be careful in grammars such as G18:

G18:

1. $S \rightarrow \{\text{print}\}aS$
2. $S \rightarrow bB$
3. $B \rightarrow \{\text{print}\}$

	var	+	*	(	)	↵
Expr	Rep(Elist Term) Retain			Rep(Elist Term) Retain		
Elist		Rep(Elist {+}Term+) Retain			Pop Retain	Pop Retain
Term	Rep(Tlist Factor) Retain			Rep(Tlist Factor) Retain		
Tlist		Pop Retain	Rep(Tlist {*}Factor*) Retain		Pop Retain	Pop Retain
Factor	Rep({var} var) Retain			Rep()Expr() Retain		
var	Pop Advance					
+		Pop Advance				
*			Pop Advance			
(				Pop Advance		
)					Pop Advance	
{var}	Pop Retain Out (var)	Pop Retain Out (var)	Pop Retain Out (var)	Pop Retain Out (var)	Pop Retain Out (var)	Pop Retain Out (var)
{+}	Pop Retain Out (+)	Pop Retain Out (+)	Pop Retain Out (+)	Pop Retain Out (+)	Pop Retain Out (+)	Pop Retain Out (+)
{*}	Pop Retain Out (*)	Pop Retain Out (*)	Pop Retain Out (*)	Pop Retain Out (*)	Pop Retain Out (*)	Pop Retain Out (*)
▽						Accept

Expr
▽

Initial
Stack

Figure 4.14 An Extended Pushdown Infix to Postfix Translator Constructed from Grammar G17

The function *S* for the recursive descent translator will print the action symbol *print* only if the input is an *a*. It is important always to check for the input symbols in the selection set before processing action symbols. Also, rule 3 is really an epsilon rule in the underlying grammar, since there are no terminals or nonterminals. Using the algorithm for selection sets, we find that:

```
Sel(1) = {a}
Sel(2) = {b}
Sel(3) = {↔}
```

The recursive descent translator for grammar G18 is shown below:

```
void S ()
{
    if (inp=='a')
    {
        cin >> inp;                // apply rule 1
        cout << "print";
        S();
    }                                // end rule 1
    else if (inp=='b')
    {
        cin >> inp;                // apply rule 2
        B();
    }                                // end rule 2
    else Reject ();
}

void B ()
{
    if (inp=='↔') cout << "print"; // apply rule 3
    else Reject ();
}
```

With these concepts in mind, we can now write a recursive descent translator to translate infix expressions to postfix according to grammar G17:

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

void Elist ()
{
    if (inp=='+')
    {
        cin >> inp;                // apply rule 2
        Term ();
        cout << '+';
        Elist ();
    }                            // end rule 2
    else if (inp=='↔' || inp==')') ; // apply rule 3
}
```

```

        else Reject ();
    }

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

void Tlist ()
{   if (inp=='*')
    {   cin >> inp;           // apply rule 5
        Factor ();
        cout << '*';
        Tlist ();
    }           // end rule 5
    else if (inp=='+' || inp=='-' || inp=='<') ;
           // apply rule 6
    else Reject ();
}

void Factor ()
{   if (inp=='(')
    {   cin >> inp;           // apply rule 7
        Expr ();
        if (inp==')') cin >> inp;
        else Reject ();
    }           // end rule 7
    else if (inp==var)
    {   cin >> inp;           // apply rule 8
        cout << "var";
    }           // end rule 8
    else Reject ();
}

```

Sample Problem 4.5

Show an extended pushdown translator for the translation grammar G18.

Solution:

	a	b	↵
S	Rep (Sa{print}) Retain	Rep (Bv) Retain	Reject
B	Reject	Reject	Rep ({print}) Retain
a	Pop Adv		
b	Reject	Pop Adv	
{print}	Pop Retain Out ({print})	Pop Retain Out ({print})	Pop Retain Out ({print})
▽	Reject	Reject	Accept

S
▽

Initial
Stack

Exercises 4.5

1. Consider the following translation grammar with starting nonterminal S, in which action symbols are put out:
1. S → A b B

2. A → {w} a c

3. A → b A {x}

4. B → {y}
- Show a *derivation tree* and the *output string* for the input **bacb**.
2. Show an *extended pushdown translator* for the translation grammar of Problem 1.
3. Show a *recursive descent translator* for the translation grammar of Problem 1.

4. Write the C++ statement which would appear in a recursive descent parser for each of the following translation grammar rules:

(a) $A \rightarrow \{w\} \ a \ \{x\} \ B \ C$

(b) $A \rightarrow a \ \{w\} \ \{x\} \ B \ C$

(c) $A \rightarrow a \ \{w\} \ B \ \{x\} \ C$

4.6 Attributed Grammars

It will soon become clear that many programming language constructs cannot be adequately described with a context-free grammar. For example, many languages stipulate that a loop control variable must not be altered within the scope of the loop. This is not possible to describe in a practical way with a context-free grammar. Also, it will be necessary to propagate information, such as a location for the temporary result of a subexpression, from one grammar rule to another. Therefore, we extend grammars further by introducing **attributed grammars**, in which each of the terminals and nonterminals may have zero or more attributes, normally designated by subscripts, associated with it. For example, an attribute on an `Expr` nonterminal could be a pointer to the stack location containing the result value of an evaluated expression. As another example, the attribute of an input symbol (a lexical token) could be the value part of the token.

In an attributed grammar there may be zero or more attribute computation rules associated with each grammar rule. These attribute computation rules show how the attributes are assigned values as the corresponding grammar rule is applied during the parse. One way for the student to understand attributed grammars is to build derivation trees for attributed grammars. This is done by first eliminating all attributes from the grammar and building a derivation tree. Then, attribute values are entered in the tree according to the attribute computation rules. Some attributes take their values from attributes of higher nodes in the tree, and some attributes take their values from attributes of lower nodes in the tree. For this reason, the process of filling in attribute values is not straightforward.

As an example, grammar G19 is an attributed (simple) grammar for prefix expressions involving addition and multiplication. The attributes, shown as subscripts, are intended to evaluate the arithmetic expression. The attribute on the terminal `const` is the value of the constant as supplied by the lexical phase. Note that this kind of expression evaluation is typical of what is done in an interpreter, but not in a compiler.

G19:

- | | | |
|----|---|----------------------|
| 1. | $\text{Expr}_p \rightarrow + \text{Expr}_q \text{Expr}_r$ | $p \leftarrow q + r$ |
| 2. | $\text{Expr}_p \rightarrow * \text{Expr}_q \text{Expr}_r$ | $p \leftarrow q * r$ |
| 3. | $\text{Expr}_p \rightarrow \text{const}_q$ | $p \leftarrow q$ |

An attributed derivation tree for the input `+ * 3 4 + 5 6` is shown in Figure 4.15 (see p. 141). The attribute on each subexpression is the value of that subexpression. This example was easy because all of the attribute values are taken from a lower node in the tree. These are called **synthesized attributes**.

A second example, involving attribute values taken from higher nodes in the tree is shown in Figure 4.16 (see p. 141). These are called **inherited attributes**. As an example of a grammar with inherited attributes, the following is a grammar for declarations of a numeric data type. In grammar G20, `type` is a terminal (token) whose value part may be a type such as `Integer`, `Real`, etc. This grammar is used to specify type

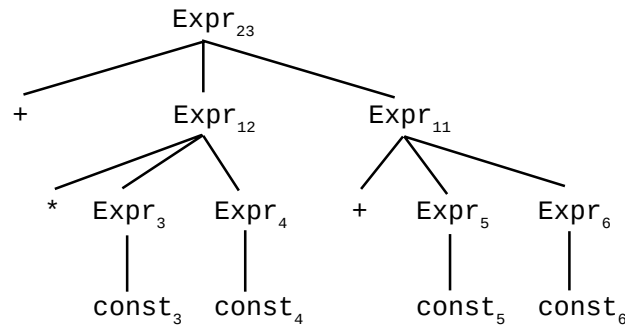


Figure 4.15 An Attributed Derivation Tree for the Prefix Expression + * 3 4 + 5 6

declarations, such as `Integer x, y, z`. We wish to store the type of each identifier with its symbol table entry. To do this, the type must be passed down the derivation tree to each of the variables being declared.

G20:

1. $\text{Dcl} \rightarrow \text{type}_t \text{Varlist}_v \quad v \leftarrow t$
2. $\text{Varlist}_v \rightarrow \text{Varlist}_w , \text{ident}_x \quad w \leftarrow v$
3. $\text{Varlist}_v \rightarrow \text{ident}_x$

An attributed derivation tree for the input string `Integer A, B, C` is shown, below, in Figure 4.16. Note that the values of the attributes move either horizontally on one level (rule 1) or down to a lower level (rules 2 and 3). It is important to remember that the

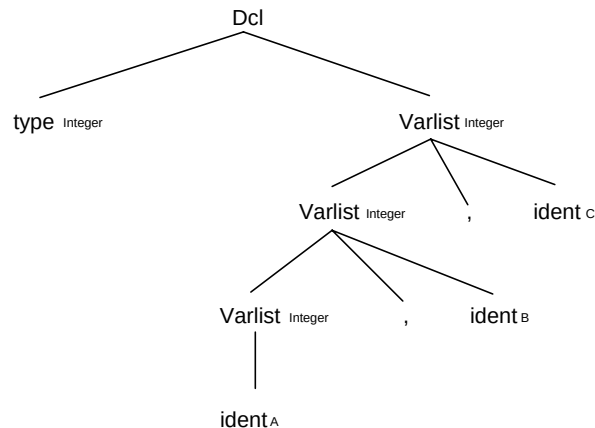


Figure 4.16 An Attributed Derivation Tree for Integer A,B,C Using Grammar G20

number and kind of attributes of a symbol must be consistent throughout the entire grammar. For example, if the symbol $A_{i,s}$ has two attributes, the first being inherited and the second synthesized, then this must be true everywhere the symbol A appears in the grammar.

4.6.1 Implementing Attributed Grammars with Recursive Descent

To implement an attributed grammar with recursive descent, the attributes will be implemented as arguments or local variables in the functions defining nonterminals. For example, if $S_{a,b}$ is a nonterminal with two attributes, then the function S will have two arguments, a and b . Synthesized attributes are used to return information to the calling function and, hence, must be implemented with **reference arguments** (call by reference, denoted by $\&$). Since inherited attributes pass information to the called function, they may be passed by value or by reference (no $\&$ is needed). Action symbol attributes will be implemented as local variables. Care must be taken that the attribute computation rules are included at the appropriate places. Also, care must be taken in designing the attributed grammar that the computation rules do not constitute a contradiction. For example:

$$S_p \rightarrow aA_rB_s \qquad p \leftarrow r + s$$

The recursive descent function for S would be:

```
void S (int & p)
{
    if (inp=='a')
    {
        cin >> inp;
        A(r);
        B(s);
        p = r + s;           // this must come after calls
                           // to A(r), B(s)
    }
}
```

In this example, the functions S , A , and B (A and B are not shown) all return values in their arguments (they are synthesized attributes, implemented with **reference arguments**), and there is no contradiction. However, assuming that the attribute of the B is synthesized, the following rule could not be implemented:

$$S \rightarrow aA_pB_q \qquad p \leftarrow q$$

In the function S , q will not have a value until function B has been called and terminated. Therefore, it will not be possible to assign a value to p before calling function A . This assumes, as always, that input is read from left to right.

Exercises 4.6

1. Consider the following attributed translation grammar with starting nonterminal S , in which action symbols are output:

$$\begin{array}{llll} 1. & S_p & \rightarrow & A_q \ b_r \ A_t \quad p \leftarrow r+t \\ 2. & A_p & \rightarrow & a_p \ \{w\}_p \ c \\ 3. & A_p & \rightarrow & b_q \ A_r \ \{x\}_p \quad p \leftarrow q+r \end{array}$$

Show an *attributed derivation tree* for the input string $a_1cb_2b_3a_4c$, and show the output symbols with attributes corresponding to this input.

2. Show a recursive descent translator for the grammar of Problem 1. Assume that all attributes are integers and that, as in sample problem 4.6, a token has member functions `getClass()` and `getVal()` which return the class and value parts of a lexical token.
3. Show an *attributed derivation tree* for each input string using the following attributed grammar:

$$\begin{array}{llll} 1. & S_p & \rightarrow & A_{q,r} \ B_t \quad \begin{array}{l} p \leftarrow q * t \\ r \leftarrow q + t \end{array} \\ 2. & A_{p,q} & \rightarrow & b_r \ A_{t,u} \ c \quad \begin{array}{l} u \leftarrow r \\ p \leftarrow r + t + u \end{array} \\ 3. & A_{p,q} & \rightarrow & \epsilon \quad p \leftarrow 0 \\ 4. & B_p & \rightarrow & a_p \end{array}$$

(a) a_2 (b) b_1ca_3 (c) $b_2b_3cca_4$

4. Is it possible to write a recursive descent parser for the attributed translation grammar of Problem 3?

4.7 An Attributed Translation Grammar for Expressions

In this section, we make use of the material presented thus far on top down parsing to implement a translator for infix expressions involving addition and multiplication. The output of the translator will be a stream of atoms, which could be easily translated to the appropriate instructions on a typical target machine. Each atom will consist of four parts: (1) an operation, **ADD** or **MULT**, (2) a left operand, (3) a right operand, and (4) a result. (Later, in Section 4.8, we will need to add two more fields to the atoms.) For example, if the input were $A + B * C + D$, the output could be the following three atoms:

MULT	B	C	Temp1
ADD	A	Temp1	Temp2
ADD	Temp2	D	Temp3

Note that our translator will have to find temporary storage locations (or use a stack) to store intermediate results at run time. It would indicate that the final result of the expression is in **Temp3**. In the attributed translation grammar G21, shown below, all nonterminal attributes are synthesized, with the exception of the first attribute on **Elist** and **Tlist**, which are inherited:

G21:

1. $\text{Expr}_p \rightarrow \text{Term}_q \text{Elist}_{q,p}$
2. $\text{Elist}_{p,q} \rightarrow + \text{Term}_r \{\text{ADD}\}_{p,r,s} \text{Elist}_{s,q} \quad s \leftarrow \text{Alloc}()$
3. $\text{Elist}_{p,q} \rightarrow \epsilon \quad q \leftarrow p$
4. $\text{Term}_p \rightarrow \text{Factor}_q \text{Tlist}_{q,p}$
5. $\text{Tlist}_{p,q} \rightarrow * \text{Factor}_r \{\text{MULT}\}_{p,r,s} \text{Tlist}_{s,q} \quad s \leftarrow \text{Alloc}()$
6. $\text{Tlist}_{p,q} \rightarrow \epsilon \quad q \leftarrow p$
7. $\text{Factor}_p \rightarrow (\text{Expr}_p)$
8. $\text{Factor}_p \rightarrow \text{ident}_p$

The intent of the action symbol $\{\text{ADD}\}_{p,r,s}$ is to put out an **ADD** atom with operands **p** and **r** and result **s**. In many rules, several symbols share an attribute; this means that the attribute is to have the same value on those symbols. For example, in rule 1 the attribute of **Term** is supposed to have the same value as the first attribute of **Elist**. Consequently, those attributes are given the same name. This also could have been done in rules 3 and 6, but we chose not to do so in order to clarify the recursive descent parser. For this reason, only four attribute computation rules were needed, two of which involved a call to **Alloc()**. **Alloc()** is a function which allocates space for a temporary result and returns a pointer to it (in our examples, we will call these temporary results **Temp1**, **Temp2**, **Temp3**, etc). The attribute of an **ident** token is the value part of that token, indicating the run-time location for the variable.

4.7.1 Translating Expressions with Recursive Descent

In the recursive descent translator which follows, synthesized attributes of nonterminals are implemented as reference arguments of the corresponding function, and inherited attributes are implemented ordinary arguments (pass by value). The `Alloc()` and `Out()` functions are not shown here. `Alloc` simply allocates space for a temporary numeric value, and `Out` simply puts out an atom, which could be a record consisting of four fields as described, above, in Section 4.7. Note that the underlying grammar of G21 is the grammar for expressions, G16, given in Section 4.4. The selection sets for this grammar are shown in that section. As in sample problem 4.6, we assume that a token has member functions `getClass()`, `getVal()`, and `getToken()`.

```
void Expr (int & p)
{   int q;
    if (inp=='(' || inp==Ident)
        {   Term (q);
            Elist (q,p);
        }
    else Reject();
}

void Elist (int p, int & q)
{   int r,s;
    if (inp=='+')
        {   token.getToken();
            inp = token.getClass();    // apply rule 2
            Term (r);
            s = Alloc();
            Out (ADD, p, r, s);
            Elist (s,q);
        }
    else if (inp=='<' || inp==')') q = p; // rule 3
    else Reject();
}

void Term (int & p)
{   int q;
    if (inp=='(' || inp==Ident)
        {   Factor (q);
            Tlist (q,p);
        }
    else Reject();
}
```

```

void Tlist (int p, int & q)
{   int r,s;
    if (inp=='*')
    {   token.getToken();
        inp = token.getClass();    // apply rule 5
        Factor (r);
        s = Alloc();
        Out (MULT, p, r, s);
        Tlist (s,q);
    }
    else if (inp=='+' || inp=='-' || inp=='<')
        q = p;                    // rule 6
    else Reject();
}

void Factor (int & p)
{   if (inp=='(')
    {   token.getToken();
        inp = token.getClass();    // apply rule 7
        Expr (p);
        if (inp==')') { token.getToken();
                        inp = token.getClass();
                      }
        else Reject();
    }
    else if (inp==Ident)
    {   p = token.getVal();          // apply rule 8
        token.getToken();
        inp = token.getClass();
    }
    else Reject();
}

```

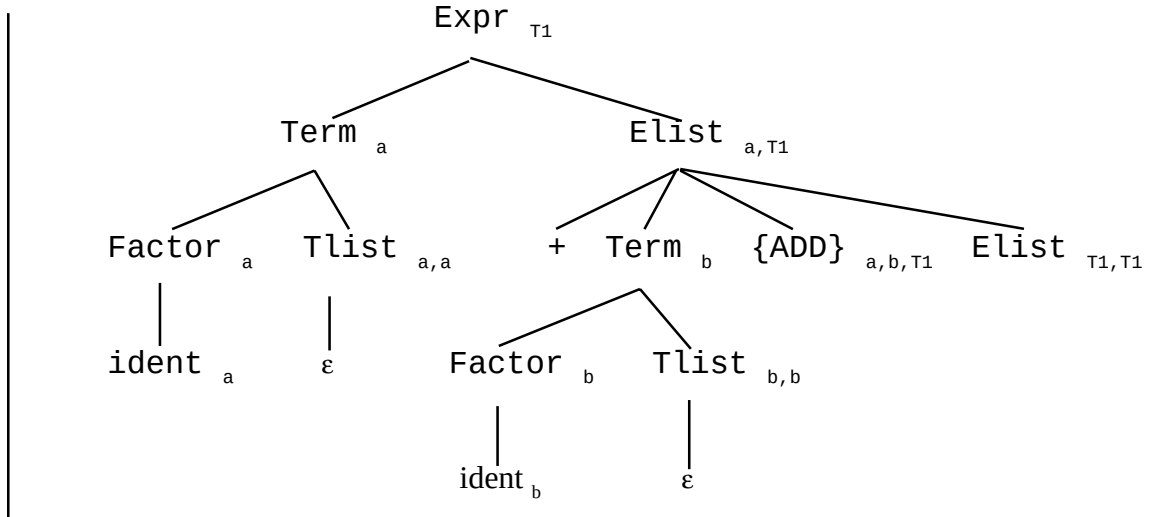
Note that the lexical analyser always leaves the value part of the last token read in a Token object, which can be accessed with the member function `getVal()`. In the grammar, this is the attribute of the terminal `ident`.

Sample Problem 4.7

Show an attributed derivation tree for the expression `a+b`, using grammar G21.

Solution:

The subscripts in this tree all represent pointers: `a` represents a pointer to the entry for the variable `a` in the symbol table, and `T1` represents a pointer to the temporary location or stack position for `T1`.



Exercises 4.7

1. Show an *attributed derivation tree* for each of the following expressions, using grammar G21. Assume that the `ALLOC` function returns a new temporary location each time it is called (`Temp1`, `Temp2`, `Temp3`, ...).

(a) $a + b * c$

(b) $(a + b) * c$

(c) (a)

(d) $a * b * c$

2. In the recursive descent translator of Section 4.7.1, refer to the function `Tlist`. In it, there is the following statement:

`Out (MULT, p, r, s)`

Explain how the three variables `p`, `r`, `s` obtain values before being put out.

3. Improve grammar G21 to include the operations of subtraction and division, as well as unary plus and minus. Assume that there are `SUB` and `DIV` atoms to handle subtraction and division. Also assume that there is a `NEG` atom with two attributes to handle unary minus; the first is the expression being negated and the second is the result.

4.8 MiniC Expressions

Thus far we have seen how to translate simple infix expressions involving only addition and multiplication into a stream of atoms. Obviously, we also need to include subtraction and division operators in our language, but this is straightforward, since subtraction has the same precedence as addition and division has the same precedence as multiplication. In this section, we extend the notion of expression to include comparisons and assignments. Comparisons are expressions such as $x > y$, or $y - 2 == 33$. The result of a comparison will be 0 if the comparison is false and 1 if the comparison is true. An assignment is an operator which produces a result in addition to the side effect of assigning a value to a variable.

First we will explain the translation of comparisons. All comparison operators have two operands, and the result will always be 0 (false) or 1 (true). One relatively easy way to implement comparisons is by moving a 1 into the result location and then testing to see whether the 1 should be replaced by a 0. For example, the comparison $x > y$ would result in the following atoms:

```
(MOV, 1, , T1)      // assume result is true
(TST, x, y, , 3, L1) // x > y? If so, skip the MOV.
(MOV, 0, , T1)      // result is false
(LBL, L1)
```

Expressing this as an attributed translation grammar, we have:

$$\text{Expr}_p \rightarrow \text{Expr}_q \text{ comp}_c \text{ Expr}_r \begin{cases} \{\text{MOV}\}_{1,,p} \\ \{\text{LBL}\}_{\text{Lbl1}} \end{cases} \begin{cases} \{\text{TST}\}_{q,r,,c,L1} \\ p \leftarrow \text{alloc}() \end{cases} \begin{cases} \{\text{MOV}\}_{0,,p} \\ \text{Lbl1} \leftarrow \text{newlab}() \end{cases}$$

in which the attribute of *Expr* is synthesized and the attribute of *comp* is an integer from 1-6 representing the comparison code. The attribute *p* represents the location of the result. The label atom (LBL) is used as a target for a branch or jump in the object program, and the test atom (TST) is a conditional branch to the label if the comparison of *q* with *r*, using comparison code *c* is true. The label and test atoms will be explained further in section 4.9. The function `alloc()` allocates memory for a temporary result, and the function `newlab()` returns a new label number.

This translation for comparisons presumes that there will always be memory locations which store the constants 0 and 1. Later in this section we show how to represent this translation with an LL(1) grammar.

Next we handle assignment; in the C language assignment can be done in an expression, whereas in languages such as Pascal a statement is used for assignments. Assignment should be thought of as an operator which returns a result. The result is the value of the expression being assigned. Assignment also has an important side effect, that of assigning a value to a variable, but syntactically it is merely another operator. For example, we could write:

$a = b + (c = 3);$

This means to assign 3 to c , add the value of c to b , and assign the result to a . The main problem we will have here is that the left operand of the assignment operator cannot be an arbitrary expression; it must be a variable, or what is often called an *lvalue*. We could use a translation grammar such as the following:

$$\text{Expr}_p \rightarrow \text{ident}_p = \text{Expr}_q \{ \text{MOV} \}_{q,,p}$$

in which the location of the result is the same as the location of the identifier receiving the value. The attribute of the Expr , again, is synthesized. The output for an expression such as $a = b + (c = 3)$ will be:

```
(MOV, 3,,c)
(ADD, b,c,T1)
(MOV, T1,,a)
```

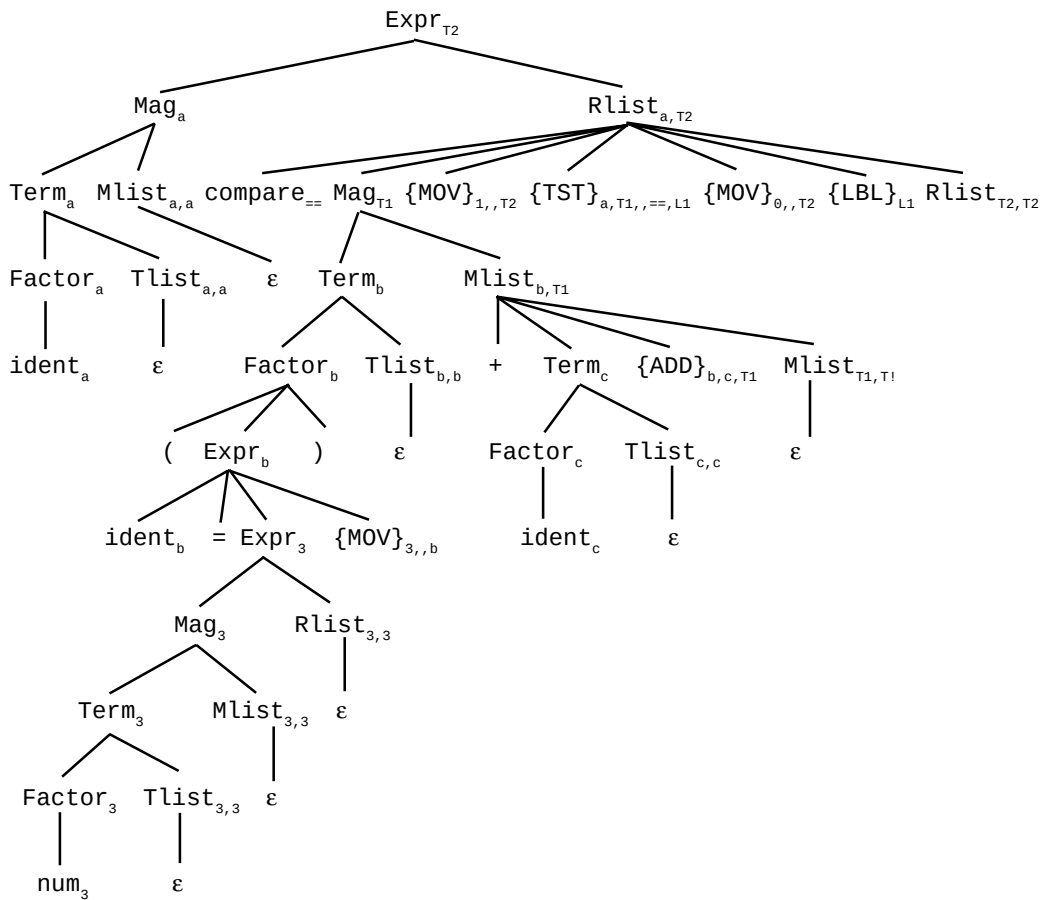
Unfortunately, this language of expressions with assignment operators is not LL(1), i.e. there is no LL(1) grammar for this language. However, we can still construct a top down parser if we permit our parser to look ahead one additional input symbol to determine whether or not an identifier is followed by an assignment operator. Thus our grammar is technically LL(2). The complete LL(2) attributed translation grammar for MiniC expressions involving addition, subtraction, multiplication, division, comparisons, and assignment is shown below:

1. $\text{Expr}_p \rightarrow \text{ident}_p = \text{Expr}_q \{ \text{MOV} \}_{q,,p}$
2. $\text{Expr}_p \rightarrow \text{Mag}_q \text{Rlist}_{q,p}$
3. $\text{Rlist}_{p,q} \rightarrow \text{compare}_c \text{Mag}_r \{ \text{MOV} \}_{1,,s} \{ \text{TST} \}_{p,r,,c,L} \{ \text{MOV} \}_{\emptyset,,s} \{ \text{LBL} \}_L \text{Rlist}_{s,q}$
 $s \leftarrow \text{alloc}()$
 $L \leftarrow \text{newlab}()$
4. $\text{Rlist}_{p,q} \rightarrow \epsilon$
 $q \leftarrow p$
5. $\text{Mag}_p \rightarrow \text{Term}_q \text{Mlist}_{q,p}$
6. $\text{Mlist}_{p,q} \rightarrow + \text{Term}_r \{ \text{ADD} \}_{p,r,s} \text{Mlist}_{s,q}$
 $s \leftarrow \text{alloc}()$
7. $\text{Mlist}_{p,q} \rightarrow - \text{Term}_r \{ \text{SUB} \}_{p,r,s} \text{Mlist}_{s,q}$
 $s \leftarrow \text{alloc}()$
8. $\text{Mlist}_{p,q} \rightarrow \epsilon$
 $q \leftarrow p$
9. $\text{Term}_p \rightarrow \text{Factor}_q \text{Tlist}_{q,p}$
10. $\text{Tlist}_{p,q} \rightarrow * \text{Factor}_r \{ \text{MUL} \}_{p,r,s} \text{Tlist}_{s,q}$
 $s \leftarrow \text{alloc}()$
11. $\text{Tlist}_{p,q} \rightarrow / \text{Factor}_r \{ \text{DIV} \}_{p,r,s} \text{Tlist}_{s,q}$
 $s \leftarrow \text{alloc}()$
12. $\text{Tlist}_{p,q} \rightarrow \epsilon$
 $q \leftarrow p$
13. $\text{Factor}_p \rightarrow (\text{Expr}_p)$
14. $\text{Factor}_p \rightarrow + \text{Factor}_p$
15. $\text{Factor}_p \rightarrow - \text{Factor}_q \{ \text{Neg} \}_{q,,p} p \leftarrow \text{alloc}()$
16. $\text{Factor}_p \rightarrow \text{num}_p$
17. $\text{Factor}_p \rightarrow \text{ident}_p$

Sample Problem 4.8

Using the grammar for MiniC expressions given in this section, show an attributed derivation tree for the expression $a == (b = 3) + c$

Solution:



Since the grammar for MiniC expressions is LL(2), we will need to implement a lookahead mechanism in our parser. This is easily done with a token object that stores two lexical tokens instead of one. This object will have four member functions: `void getToken()` will read in the next token from the input stream, `int getClass()` will return the token's class, `int getVal()` will return token's value part, and `void unget()` will place the last token read back on the input stream. This will be needed in the function for `Expr` when the parser needs to choose between rules 1 and 2. If the next input is an assign token (=), apply rule 1; otherwise, "unget" that input and apply rule 2.

Exercises 4.8

1. Show an *attributed derivation tree* using the grammar for MiniC expressions given in this section for each of the following expressions:

(a) <code>a = b = c</code>	(b) <code>a==b!=c</code>
(c) <code>(a=3)<=(b=2)</code>	(d) <code>a==-(c=3)</code>
(e) <code>a*(b=3)+c!=9</code>	

2. Show the *recursive descent parser* for the nonterminals `Expr`, `Mag`, and `Rlist` given in the grammar for MiniC expressions. Assume that `void getToken()`, `int getClass()`, `int getVal()`, and `void unget()` member functions are defined for a `Token` object. Hint: the selection sets for the first 5 grammar rules are:


```

sel(1) = {ident}
sel(2) = {ident, num, (, +, -}
sel(3) = {compare}
sel(4) = {), ←}
sel(5) = {ident, num, (, +, -}
      
```

3. Show a definition of the *Token class* with member functions `void getToken()`, `int getClass()`, `int getVal()`, and `void unget()`.

4.9 Translating Control Structures

In order to translate control structures, such as `for`, `while`, and `if` statements, we must first consider the set of primitive control operations available on the target machine. These are typically simple operations such as `Unconditional Jump` or `Goto`, `Compare`, and `Conditional Jump`. In order to implement these `Jump` operations, we need to establish a jump, or destination, address.

During the parse or syntax analysis phase there are, as yet, no machine addresses attached to the output. In addition, we must handle forward jumps when we don't know the destination of the jump. To solve this problem we introduce a special atom called a `Label`, which is used to mark the destination of a jump. During code generation, a machine address is associated with each `Label` atom. At this point, we need to add two additional fields to our atoms – one for comparison codes (1-6) and one for `jump` destinations.

We will use the following atoms to implement control structures:

<code>JMP</code>	-	-	-	-	<code>Lbl</code>	Unconditional jump to the specified label
<code>TST</code>	<code>E1</code>	<code>E2</code>	-	<code>Cmp</code>	<code>Lbl</code>	Conditional branch if comparison is true
<code>LBL</code>	-	-	-	-	<code>Lbl</code>	Label used as branch destination

The purpose of the `TST` atom is to compare the values of expressions `E1` and `E2` using the specified comparison operator, `Cmp`, and then branch to the label `Lbl` if the comparison is true. The comparison operator will be the value part of the comparison token (there are six, shown below). For example, `TST, A, C, , 4, L3` means jump to label `L3` if `A` is less than or equal to `C`. The six comparison operators and codes are:

<code>==</code>	is 1	<code><=</code>	is 4
<code><</code>	is 2	<code>>=</code>	is 5
<code>></code>	is 3	<code>!=</code>	is 6

The `LBL` atom is used as a tag or marker so that `JMP` and `TST` atoms can refer to a branch destination without knowing target machine addresses for these destinations.

In addition, there is one more atom which is needed for assignment statements; it is a `MOVE` atom, which simply assigns its source operand to its target operand:

`MOV Src - Trg - - Trg = Src`

Figure 4.17 shows the sequence in which atoms are put out for the common control structures (from top to bottom). It shows the input tokens, as well, so that you understand when the atoms are put out. The arrows indicate flow of control at **run time**; the arrows do **not** indicate the sequence in which the compiler puts out atoms. In Figure 4.17, the atoms are all enclosed in a boundary: `ADD` and `JMP` atoms are

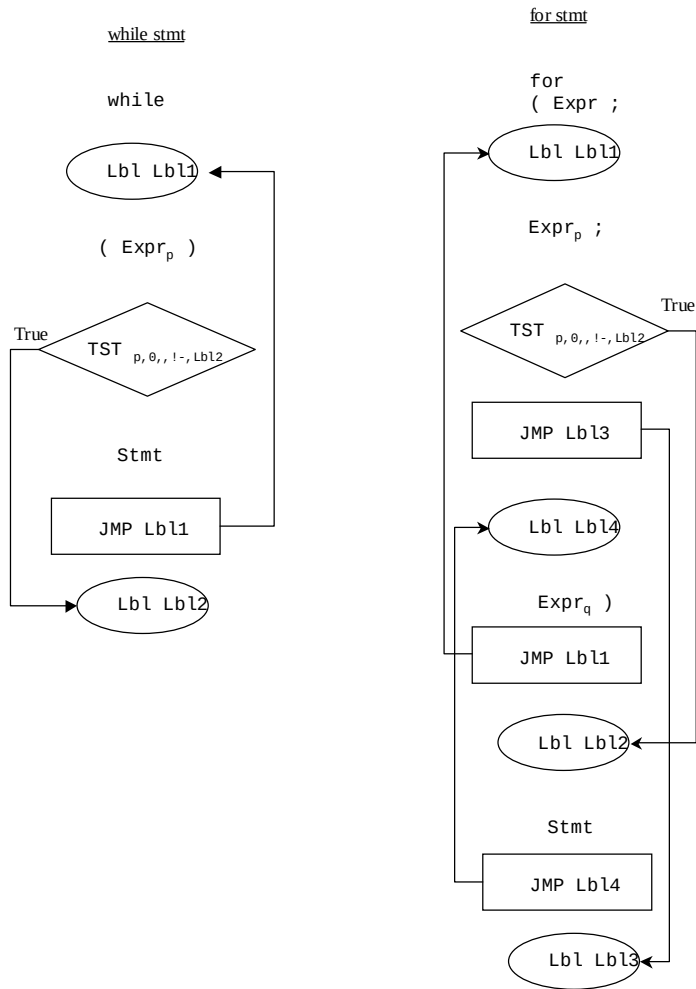


Figure 4.17 (a) Control Structures for while and for Statements

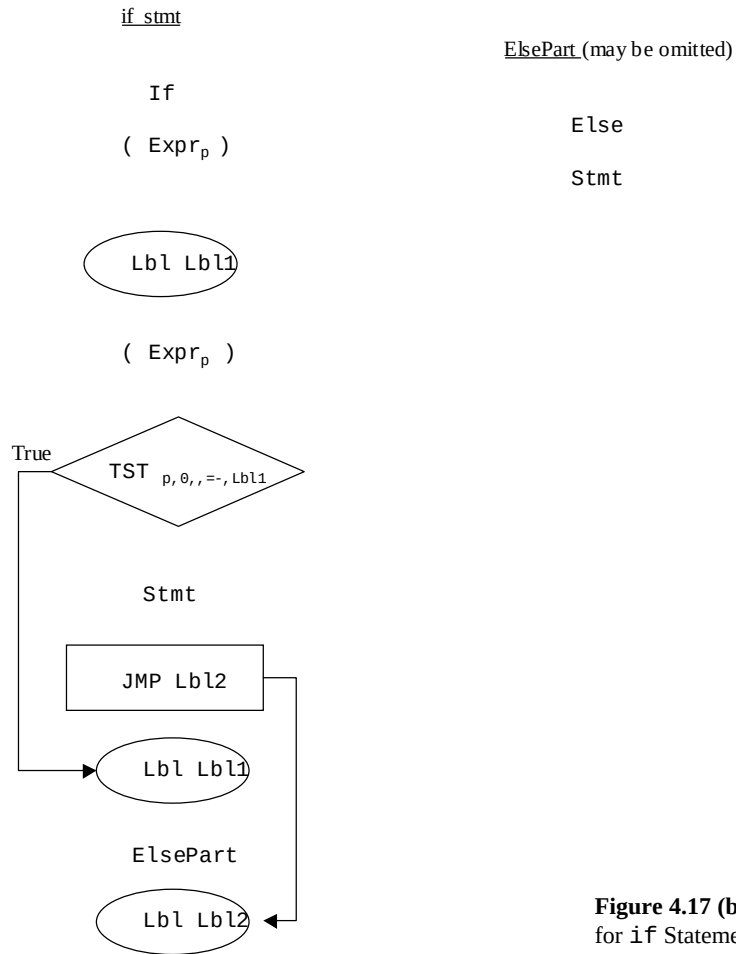


Figure 4.17 (b) Control Structures for if Statements

enclosed in rectangles, LBL atoms are enclosed in ovals, and TST atoms are enclosed in diamond shaped parallelograms.

The control structures in Figure 4.17 correspond to the following statement definitions (which are similar to the C++ control structures):

1. Stmt → while (Expr) Stmt
2. Stmt → for (Expr; Expr; Expr) Stmt
3. Stmt → if (Expr) Stmt ElsePart

- 4. `ElsePart` $\rightarrow$ `else Stmt`
- 5. `ElsePart` $\rightarrow \epsilon$

For the most part, Figure 4.17 is self explanatory. Note that for `if` statements, we must jump around the statement which is not to be executed. The value of the expression in parentheses, denoted by `p`, is compared against zero to determine whether the statement is to be executed. Recall that an integer value of zero represents a false condition, and a non-zero value represents a true condition. This provides for a relatively simple translation.

Unfortunately, the grammar shown above is not LL(1). This can be seen by finding the follow set of `ElsePart`, and noting that it contains the keyword `else`. Consequently, rules 4 and 5 do not have disjoint selection sets. However, it is still possible to write a recursive descent parser. This is due to the fact that all `elses` are matched with the closest preceeding unmatched `if`. When our parser for the nonterminal `ElsePart` encounters an `else`, it is never wrong to apply rule 4 because the closest preceeding unmatched `if` must be the one on top of the recursive call stack. Aho [1986] claims that there is no LL(1) grammar for this language. This is apparently one of those rare instances where theory fails, but our practical knowledge of how things work comes to the rescue.

The `for` statement requires some additional explanation. The following `for` statement and `while` statement are equivalent:

Sample Problem 4.9

Show the atom string which would be put out that corresponds to the following C statement:

```
while (x>0) Stmt
```

Solution:

```
(LBL, L1)
(MOV, 1, , T1)
(TST, x, 0, , 3, L2)           // x>0 ?
(MOV, 0, , T1)                // X>0 is false
(LBL, L2)
(TST, T1, 0, , 1, L3)         // T1 == false?
```

Atoms for `Stmt`

```
(JMP, L1)
(LBL, L3)
```

```

for (E1; E2; E3) stmt          E1 ;
                                while (E2)
                                { stmt
                                E3;
                                }

```

This means that after the atoms for the `stmt` are put out, we must put out a jump to the atoms corresponding to expression `E3`. In addition, there must be a jump after the atoms of expression `E3` back to the beginning of the loop, expression `E2`. The LL(2) grammar for MiniC shown in the next section makes direct use of figure 4.17 for the control structures.

Exercises 4.9

1. Show the *sequence of atoms* which would be put out according to Figure 4.17 for each of the following input strings:
 - (a) `if (a==b) while (x<y) Stmt`
 - (b) `for (i = 1; i<=100; i = i+1)`
`for (j = 1; j<=i; j = j+1) Stmt`
 - (c) `if (a==b) for (i=1; i<=20; i=i+1) Stmt1`
`else while (i>0) Stmt2`
 - (d) `if (a==b) if (b>0) Stmt1 else while (i>0) Stmt2`
2. Show an *attributed translation grammar rule* for each of the control structures given in Figure 4.17. Assume `if` statements always have an `else` part and that there is a function, `newlab`, which allocates a new statement label.
 1. `WhileStmt` $\rightarrow$ `while (Expr) Stmt`
 2. `ForStmt` $\rightarrow$ `for (Expr; Expr; Expr) Stmt`
 3. `IfStmt` $\rightarrow$ `if (Expr) Stmt else Stmt`
3. Show a *recursive descent translator* for your solutions to Problem 2. Show functions for `WhileStmt`, `ForStmt`, and `IfStmt`.

4. Does your C compiler permit a loop control variable to be altered inside the loop, as in the following example?

```
for (i=0; i!=100; i = i+1)
{   cout << i << endl;
    i = 200;
}
```

4.10 Case Study: A Top Down Parser for MiniC

In this section we show how the concepts of this chapter can be applied to a compiler for a subset of C, i.e. the MiniC language. We will develop a top down parser for MiniC and implement it using the method of recursive descent. The program is written in C++ and is available as described in Appendix B.1. Note that in the software package there are actually two parsers: one is top-down (for this chapter) and the other is bottom up (for Chapter 5). The bottom up parser is the one that is designed to work with the other phases to form a complete compiler. The top down parser is included only so that we can have a case study for this chapter.

In order to write the recursive descent parser, we will need to work with a grammar that is LL. This grammar is relatively easy to write and is shown in Figure 4.18. The most difficult part of this grammar is the part which defines arithmetic expressions. It is taken directly from Section 4.8.

The rest of the grammar is relatively easy because C was designed to be amenable to top-down parsing; the developer of C++ recommends recursive descent parsers (see Stroustrup 1994). To see this, note that most of the constructs in C begin with key words. For example, when expecting a statement, the parser need examine only one token to decide which kind of statement it is. The possibilities are simply `for`, `if`, `while`, `{`, or one of the symbols which begin an expression. In other words, MiniC is, for the most part, a simple language (in the technical sense).

In Figure 4.18 there are two function calls: `alloc()` and `newlab()`. The purpose of `alloc()` is to find space for a temporary result, and the purpose of `newlab()` is to generate new label numbers, which we designate `L1`, `L2`, `L3`,

The control structures in Figure 4.18 are taken from Figure 4.17 and described in Section 4.9. As mentioned in that section, the convoluted logic flow in the `for` statement result from the fact that the third expression needs to be evaluated after the loop body is executed, and before testing for the next iteration.

The recursive descent parser is taken directly from Figure 4.18. The only departure is in the description of iterative structures such as `IdentList` and `ArgtList`. Context-free grammars are useful in describing recursive constructs, but are not very useful in describing these iterative constructs. For example, the definition of `IdentList` is:

```
IdentList  →      identifier
              | identifier , IdentList
```

While this is a perfectly valid definition of `IdentList`, it leads to a less efficient parser (for compilers that don't translate tail recursion into loops). What we really want to do is use a loop to scan for a list of identifiers separated by commas. This can be done as follows:

```
if (inp!=IDENTIFIER_) error();
while (inp == IDENTIFIER_)
```

Function	→	Type identifier (ArgList) CompoundStmt
ArgList	→	Arg ArgList , Arg
Arg	→	Type identifier
Declaration	→	Type IdentList ;
Type	→	int float
IdentList	→	identifier , IdentList identifier
Stmt	→	ForStmt WhileStmt Expr _p ; IfStmt CompoundStmt Declaration ;
ForStmt	→	for (Expr _p ; {LBL} _{Lb11} Expr _q ; {TST} _{q,0,,!=,Lb12} {JMP} _{Lb13} {LBL} _{Lb14} Expr _r) {JMP} _{Lb11} {LBL} _{Lb12} Stmt {JMP} _{Lb14} {LBL} _{Lb13} Lb11←newlab() Lb12←newlab() Lb13←newlab() Lb14←newlab()
WhileStmt	→	while {LBL} _{Lb11} (Expr _p) {TST} _{p,0,,==,Lb12} Stmt {JMP} _{Lb11} {LBL} _{Lb12} Lb11←newlab() Lb12←newlab()
IfStmt	→	if (Expr _p) {TST} _{p,0,,==,Lb11} Stmt {JMP} _{Lb12} {LBL} _{Lb11} ElsePart {LBL} _{Lb12} Lb11←newlab() Lb12←newlab()
ElsePart	→	else Stmt ε
CompoundStmt	→	{ StmtList }
StmtList	→	StmtList Stmt ε
Expr _p	→	ident _p = Expr _q {MOV} _{q,,p} Mag _q Rlist _{q,p}
Rlist _{p,q}	→	compare _c Mag _r {MOV} _{1,,s} {TST} _{p,r,,c,Lb11} {MOV} _{0,,s} {LBL} _{Lb11} Rlist _{s,q} Lb11 ← newlab() s ← alloc()
Mag _p	→	Term _q Mlist _{q,p} q ← p
Mlist _{p,q}	→	+ Term _r {ADD} _{p,r,s} Mlist _{s,q} s ← alloc() - Term _r {SUB} _{p,r,s} Mlist _{s,q} s ← alloc() ε q ← p

Figure 4.18 An Attributed Translation Grammar for MiniC (continued on next page)

Term _p	→	Factor _q Tlist _{q,p}	
Tlist _{p,q}	→	* Factor _r {MUL} _{p,r,s} Tlist _{s,q}	s ← alloc()
		/ Factor _r {DIV} _{p,r,s} Tlist _{s,q}	s ← alloc()
		ε	q ← p
Factor _p	→	(Expr _p)	
Factor _p	→	+ Factor _p	
Factor _p	→	- Factor _q {Neg} _{q,,p}	p ← alloc()
Factor _p	→	num _p	
Factor _p	→	ident _p	

Figure 4.18 (Continued)

```

    { token.getToken();
      inp = token.getClass();
      if (inp == ',') { token.getToken();
                      inp = token.getClass();
                      }
    }
}

```

We use this method also for the functions ArgList() and StmtList().

Note that the fact that we have assigned the same attribute to certain symbols in the grammar, saves some effort in the parser. For example, the definition of Factor uses a subscript of p on the Factor as well as on the Expr, identifier, and number on the right side of the arrow. This simply means that the value of the Factor is the same as the item on the right side, and the parser is simply:

```

void Factor (int & p);
{
    if (inp == '(')
    {
        token.getTokein();
        inp = token.getClass();
        Expr (p);
        if (inp == ')') inp = token.getClass();
        else error();
    }
    else if (inp == IDENTIFIER)
    {
        p = token.getVal(); // value part of token
        inp.getToken();
        inp = token.getClass();
    }
    else if (inp == NUMBER)
    {
        p = token.getVal(); // value part of token
        inp.getToken();
    }
}

```

```

    inp = token.getClass();
    }
}

```

Exercises 4.10

1. Show the atoms put out as a result of the following MiniC statement:

```

if (a=3) { a == 4;
          for (i = 2; i<5; i ) i = i + 1;
        }
else while (a>5) i * 8;

```

2. Explain the purpose of each atom put out in our MiniC attributed translation grammar for the `for` statement:

```

ForStmt    →    for ( Exprp; {LBL}Lb11 Exprq;
                  {TST}q,0,,!=,Lb12 {JMP}Lb13 {LBL}Lb14 Exprr )
                  {JMP}Lb11 {LBL}Lb12 Stmt {JMP}Lb14 {LBL}Lb13
                  Lb11←newlab() Lb12←newlab()
                  Lb13←newlab() Lb14←newlab()

```

3. The C language has a `switch` statement.
 - (a) Include a definition of the `switch` statement in the *attributed translation grammar* for MiniC.
 - (b) Check your grammar by building an *attributed derivation tree* for a sample `switch` statement of your own design.
 - (c) Include code for the `switch` statement in the *recursive descent parser*, MiniC.C
4. Using the grammar of Figure 4.18, show an attributed derivation tree for the statement given in problem 1, above.

4.11 Chapter Summary

A *parsing algorithm* reads an input string one symbol at a time, and determines whether the string is a member of the language of a particular grammar. In doing so, the algorithm uses a *stack* as it applies rules of the grammar. A *top down parsing algorithm* will apply the grammar rules in a sequence which corresponds to a downward direction in the derivation tree.

Not all context-free grammars are suitable for top down parsing. In general, the algorithm needs to be able to decide which grammar rule to apply without looking ahead at additional input symbols. We present an algorithm for finding *selection sets*, which are sets of input symbols, one set for each rule, and are used to direct the parser. Since the process of finding selection sets is fairly complex, we first define *simple* and *quasi-simple* grammars in which the process is easier.

We present two methods for top down parsing: (1) *pushdown machines* and (2) *recursive descent*. These methods can be used whenever all rules defining the same nonterminal have *disjoint selection sets*. We then define *translation grammars*, which are capable of specifying output, and *attributed grammars*, in which it is possible for information to be passed from one grammar rule to another during the parse.

After learning how to apply these methods to context-free grammars, we turn our attention to *grammars for programming languages*. In particular, we devise an *attributed translation grammar* for *arithmetic expressions* which can be parsed top down. In addition, we look at an *attributed translation grammar* for some of the common *control structures*: *while*, *for*, and *if-else*.

Finally, we examine a *top down parser* for our case study language – MiniC. This parser is written as a recursive descent parser in C++. In the interest of keeping costs down, it is not shown in the appendix; however, it is available along with the other software files as described in Appendix B.1.