

## 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 methods 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:

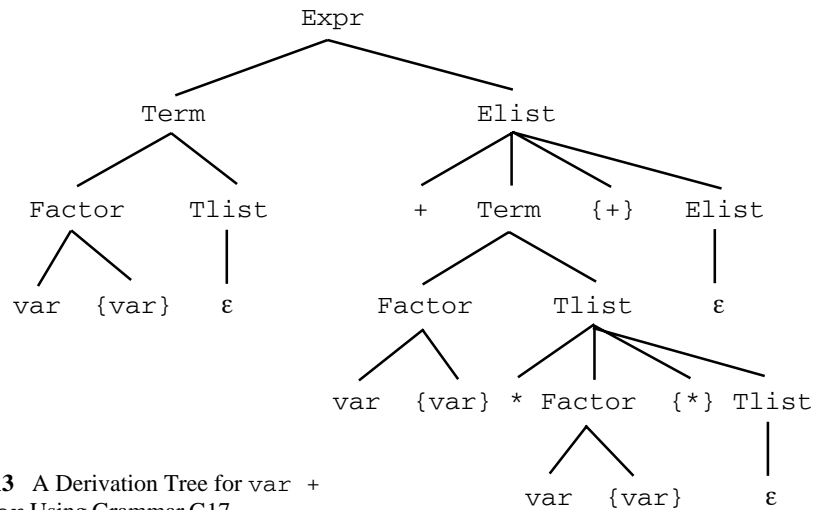
1.  $\text{Expr} \rightarrow \text{Term Elist}$
2.  $\text{Elist} \rightarrow + \text{Term } \{+\} \text{ Elist}$
3.  $\text{Elist} \rightarrow \epsilon$
4.  $\text{Term} \rightarrow \text{Factor Tlist}$
5.  $\text{Tlist} \rightarrow * \text{Factor } \{*\} \text{ Tlist}$
6.  $\text{Tlist} \rightarrow \epsilon$
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 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 method, 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 method  $S$  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')
    {  getInp();                // apply rule 1
      System.out.println ("print");
      S();
    }
    // end rule 1
  else if (inp=='b')
    {  getInp();                // apply rule 2
      B();
    }
    // end rule 2
  else Reject ();
}

void B ()
{  if (inp=='↔') System.out.println ("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:

```

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

void Elist ()
{  if (inp=='+')
    {  getInp();                // apply rule 2
      Term ();
      System.out.println ('+');
      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=='*')
    {   getInp();           // apply rule 5
        Factor ();
        System.out.println ('*');
        Tlist ();
    }   // end rule 5
    else if (inp=='+' || inp=='-') || inp=='<') ;
        // apply rule 6
    else Reject ();
}

void Factor ()
{   if (inp=='(')
    {   getInp();           // apply rule 7
        Expr ();
        if (inp==')') getInp();
        else Reject ();
    }   // end rule 7
    else if (inp==var)
    {   getInp();           // apply rule 8
        System.out.println ("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 (Bb) 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.  $S \rightarrow A b B$
2.  $A \rightarrow \{w\} a c$
3.  $A \rightarrow b A \{x\}$
4.  $B \rightarrow \{y\}$

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 *Java 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.

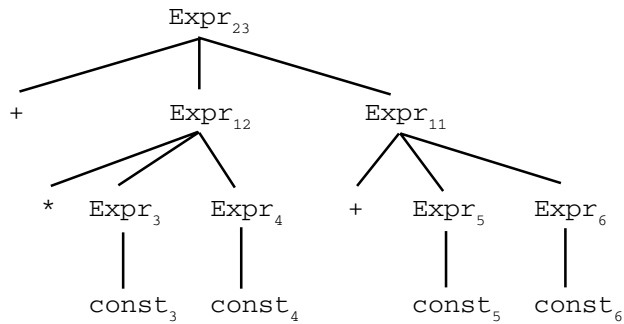
As an example, grammar G19 is an attributed (simple) grammar for prefix expressions involving addition and multiplication. The attributes, shown as subscripts, are intended to evaluate the arithmetic expression. The attribute on the terminal `const` is the value of the constant as supplied by the lexical 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 `int`, `float`, etc. This grammar is used to specify type declarations,



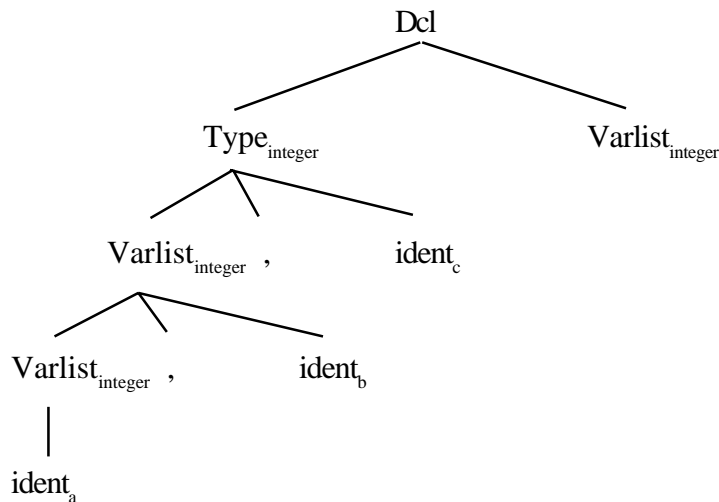
**Figure 4.15** An Attributed Derivation Tree for the Prefix Expression  $+ * 3 4 + 5 6$

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.  $Dcl \rightarrow type_t Varlist_v$   $v \leftarrow t$
2.  $Varlist_v \rightarrow Varlist_w , ident_x$   $w \leftarrow v$
3.  $Varlist_v \rightarrow ident_x$

An attributed derivation tree for the input string `int 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



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

down to a lower level (rules 2 and 3). It is important to remember that the 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 parameters or instance variables in the methods defining nonterminals. For example, if  $S_{a,b}$  is a nonterminal with two attributes, then the method  $S$  will have two parameters,  $a$  and  $b$ . Synthesized attributes are used to return information to the calling method and, hence, must be implemented with objects (i.e. with reference types). If the attribute is to store a whole number, we would be tempted to use the Java wrapper class `Integer` for synthesized attributes. Unfortunately, the `Integer` class is not mutable, i.e. `Integer` objects cannot be changed. Thus we will build our own wrapper class, called `MutableInt`, to store a whole number whose value can be changed. This class is shown in below:

```
// Wrapper class for ints which lets you change the value.
// This class is needed to implement attributed grammars
// with recursive descent

class MutableInt extends Object
{ int value;           // store a single int

MutableInt (int i)    // Initializing constructor
{ value = i; }

MutableInt ()        // Default constructor
{ value = 0;        // default value is 0
}

int get()            // Accessor
{ return value; }

void set (int i)     // Mutator
{ value = i; }

public String toString() // For printing
{ return (new Integer (value)).toString(); }
}
```

Since inherited attributes pass information to the called method, they may be passed by value or by using primitive types. Action symbol attributes will be implemented as instance 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:

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

The recursive descent method for  $S$  would be:

```
void S (MutableInt p)
{
    if (token.get_class()=='a')
        { token.getToken();
          A(r);
          B(s);
          // this must come after calls to A(r), B(s)
          p.set(r.get() + s.get());
        }
}
```

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

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

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

#### Sample Problem 4.6

Show a recursive descent parser for the attributed grammar  $G_{19}$ . Assume that the `Token` class has inspector methods, `get_class()` and `get_val()`, which return the class and value parts of a lexical token, respectively. The method `getToken()` reads in a new token.

**Solution:**

```

class RecDescent
{
    final int Num=0;           // token classes
    final int Op=1;
    final int End=2;
    Token token;

    void Eval ()
    { MutableInt p = new MutableInt(0);
      token = new Token();
      token.getToken();       // Read a token from stdin
      Expr (p);

                               // show final result
      if (token.get_class()==End) System.out.println (p);
      else reject();
    }

    void Expr (MutableInt p)
    { MutableInt q = new MutableInt(0), r = new
      MutableInt(0);         // Attributes q,r
      if (token.get_class()==Op) // Operator?
        if (token.get_value()== (int)'+') // apply rule 1
          { token.getToken(); // read next token
            Expr(q);
            Expr(r);
            p.set (q.get() + r.get());
          }
          // end rule 1
        else // should be a *, apply rule 2
          { token.getToken(); // read next token
            Expr(q);
            Expr(r);
            p.set (q.get() * r.get());
          }
          // end rule 2
        else if (token.get_class()==Num) // Is it a number?
          { p.set (token.get_value()); // apply rule 3
            token.getToken(); // read next token
          }
          // end rule 3
        else reject();
    }
}

```

## 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, the Token class has methods `get_class()` and `get_value()` which return the class and value parts of a lexical token, and the Token class has a `getToken()` method which reads a token from the standard input file.
3. Show an *attributed derivation tree* for each input string using the following attributed grammar:

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

(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.  $\text{Factor}_p \rightarrow \text{ident}_p$

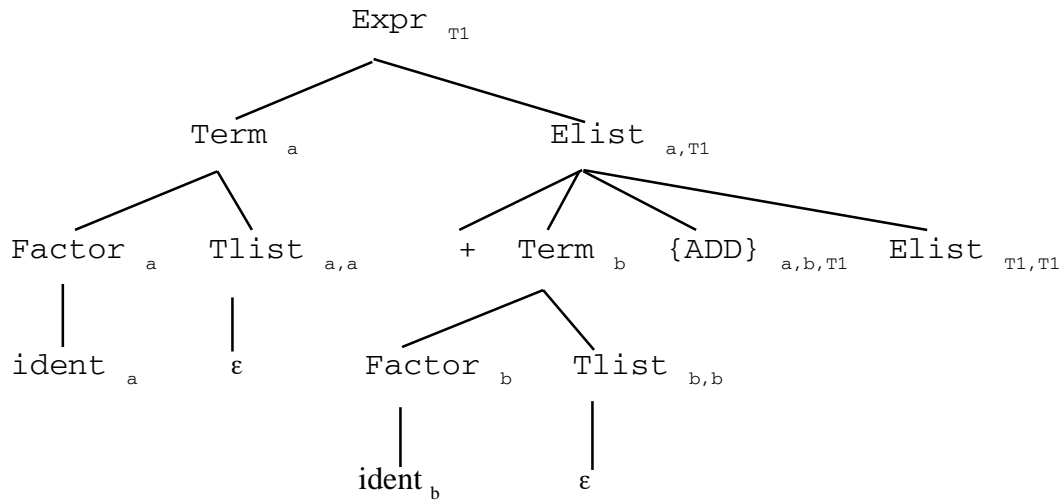
The intent of the action symbol  $\{\text{ADD}\}_{p,r,s}$  is to put out an ADD atom with operands  $p$  and  $r$  and result  $s$ . In many rules, several symbols share an attribute; this means that the attribute is to have the same value on those symbols. For example, in rule 1 the attribute of Term 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 method 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.

**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  $T1$ .

**4.7.1 Translating Expressions with Recursive Descent**

In the recursive descent translator which follows, synthesized attributes of nonterminals are implemented as references, and inherited attributes are implemented as primitives. The `alloc()` and `atom()` methods are not shown here. `alloc` simply allocates space for a temporary result, and `atom` 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 the `Token` class has methods `get_class()` and `getToken()`. Also, we use our wrapper class, `MutableInt`, for synthesized attributes.

```

class Expressions
{
Token token;

static int next = 0;    // for allocation of temporary
                        // storage

public static void main (String[] args)
{
    Expressions e = new Expressions();
    e.eval ();
}

void eval ()
{
    MutableInt p = new MutableInt(0);
    token = new Token();

    token.getToken();

    Expr (p);           // look for an expression
    if (token.get_class()!=Token.End) reject();
    else accept();
}

void Expr (MutableInt p)
    {
        MutableInt q = new MutableInt(0);
        if (token.get_class()==Token.Lpar ||
            token.get_class()==Token.Ident
            || token.get_class()==Token.Num)

            {
                Term (q);           // apply rule 1
                Elist (q.get(),p);
            }
            else reject();
    }

void Elist (int p, MutableInt q)

```

```

    { int s;
      MutableInt r = new MutableInt();
      if (token.get_class()==Token.Plus)
          { token.getToken(); // apply rule 2
            Term (r);
            s = alloc();
            atom ("ADD", p, r, s); // put out atom
            Elist (s,q);
          } // end rule 2
      else if (token.get_class()==Token.End ||
              token.get_class()==Token.Rpar)
          q.set(p); // rule 3
      else reject();
    }
void Term (MutableInt p)
    { MutableInt q = new MutableInt();
      if (token.get_class()==Token.Lpar
          || token.get_class()==Token.Ident
          || token.get_class()==Token.Num)
          { Factor (q); // apply rule 4
            Tlist (q.get(),p);
          } // end rule 4
      else reject();
    }

void Tlist (int p, MutableInt q)
    { int inp, s;
      MutableInt r = new MutableInt();
      if (token.get_class()==Token.Mult)
          { token.getToken(); // apply rule 5
            inp = token.get_class();
            Factor (r);
            s = alloc();
            atom ("MULT", p, r, s);
            Tlist (s,q);
          } // end rule 5
      else if (token.get_class()==Token.Plus
              || token.get_class()==Token.Rpar

```

```

        || token.get_class()==Token.End)
            q.set (p);          // rule 6
    else reject();
}

void Factor (MutableInt p)
{   if (token.get_class()==Token.Lpar)
    {   token.getToken(); // apply rule 7
        Expr (p);
        if (token.get_class()==Token.Rpar)
            token.getToken();
        else reject();
    } // end rule 7
    else if (token.get_class()==Token.Ident
        || token.get_class()==Token.Num)
    {   p.set(alloc()); // apply rule 8
        token.getToken();
    } // end rule 8
    else reject();
}

```

### Exercises 4.7

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

(a) $a + b * c$	(b) $(a + b) * c$
(c) (a)	(d) $a * b * c$
- In the recursive descent translator of Section 4.7.1, refer to the method `Tlist`. In it, there is the following statement:

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

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

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 Decaf 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 (boolean expressions) and assignments. Boolean expressions are expressions such as  $x > y$ , or  $y - 2 = 33$ . For those students familiar with C and C++, the comparison operators return ints (0=false, 1=true), but Java makes a distinction: the comparison operators return boolean results (true or false). If you have ever spent hours debugging a C or C++ program which contained `if (x=3) . . .` when you really intended `if (x==3) . . .`, then you understand the reason for this change. The Java compiler will catch this error for you.

Assignment is also slightly different in Java. In C/C++ assignment is an operator which produces a result in addition to the side effect of assigning a value to a variable. A statement may be formed by following any expression with a semicolon. This is not the case in Java. The expression statement must be an assignment or a method call. Since there are no methods in Decaf, we're left with an assignment.

At this point we need to introduce three new atoms indicated in Figure 4.17: LBL (label), JMP (jump, or unconditional branch), and TST (test, or conditional branch). A LBL atom is merely a placemaker in the stream of atoms that is put out. It represents a target location for the destination of a branch, such as JMP or TST. Ultimately, a LBL atom will represent the run-time memory address of an instruction in the object program. A LBL atom will always have one attribute which is a unique name for the label. A JMP atom is an unconditional branch; it must always be accompanied by a label name - the destination for the branch. A JMP atom will have one attribute - the name of the label which is the destination. A TST atom is used for a conditional branch. It will have four

<u>Atom</u>	<u>Attributes</u>	<u>Purpose</u>
LBL	label name	Mark a spot to be used as a branch destination
JMP	label name	Unconditional branch to the label specified
TST	Expr <sub>1</sub> Expr <sub>2</sub> comparison code label name	Compare Expr <sub>1</sub> and Expr <sub>2</sub> using the comparison code. Branch to the label if the result is true.

**Figure 4.17** Atoms used for Transfer of Control

attributes: two attributes will identify the locations of two expressions being compared; another attribute will be a comparison code (1 is ==, 2 is <, ...); the fourth attribute will be the name of the label which is the destination for the branch. The intent is for the branch to occur at run time only if the result of comparing the two expressions with the given comparison operator is true.

We will now explain the translation of boolean expressions. It turns out that every time we need to use a boolean expression in a statement, we really want to put out a TST atom that branches when the comparison is false. In the following two examples, we show an `if` statement and a `while` statement. In each case the source input is at the left, and the atoms to be put out are indented to the right. Rather than showing all the attributes at this point, their values are indicated with comments:

```

if
(x==3)
    [TST]           // Branch to the Label only if
Stmt             //   x==3 is false
    [Label]

////////////////////////////////////

while
    [Label1]
(x>2)
    [TST]           // Branch to Label2 only if
    Stmt           //   x>2 is false
    [JMP]           // Unconditional branch to Label1
    [Label2]

```

Recall our six comparison codes; to get the logical complement of any comparison, we simply subtract the code from 7 as shown below:

<u>Comparison</u>	<u>Code</u>	<u>Logical Complement</u>	<u>Code for complement</u>
=	1	!=	6
<	2	>=	5
>	3	<=	4
<=	4	>	3
>=	5	<	2
!=	6	=	1

Thus, to process a boolean expression all we need to do is put out a TST atom which allocates a new label name and branches to that label when the comparison is *false*. (The label atom itself will be handled later, in section 4.9).

Thus the attributed grammar rule for a boolean expression will be:

$$\text{BoolExpr}_{\text{Lbl}} \rightarrow \text{Expr}_p \text{ compare}_c \text{ Expr}_q \{ \text{TST} \}_{p,q,,7-c,\text{Lbl}}$$

The TST atom represents a conditional branch in the object program.  $\{\text{TST}\}a,b,,c,x$  will compare the values stored at  $a$  and  $b$ , using the comparison whose code is  $c$ , and branch to a label designated  $x$  if the comparison is true. In the grammar rule above the attribute of  $\text{BoolExpr}$ ,  $\text{Lbl}$ , is synthesized and represents the target label for the TST atom to branch in the object program. The attribute of the token  $\text{compare}$ ,  $c$ , is an integer from 1-6 representing the comparison code. The use of comparison code  $7-c$  inverts the sense of the comparison as desired.

Next we handle assignment; an assignment is an operator which returns a result that can be used as part of a larger expression. For example:

```
x = (y = 2) + (z = 3);           // y is 2, z is 3, x is 5
```

This means that we will need to put out a MOV atom to implement the assignment, in addition to giving it a synthesized attribute to be moved up the tree. The left operand of the assignment must be an identifier, or what is often called an *lvalue*. Also, note that unlike the arithmetic operators, this operator associates to the right, which permits multiple assignments such as:

```
x = y = z = 0;                 // x, y, and, z are now all 0.
```

We could use a translation grammar such as the following:

$$\begin{aligned} \text{Expr}_p &\rightarrow \text{AssignExpr}_p \\ \text{AssignExpr}_p &\rightarrow \text{ident}_p = \text{Expr}_q \{ \text{MOV} \}_{q,,p} \end{aligned}$$

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

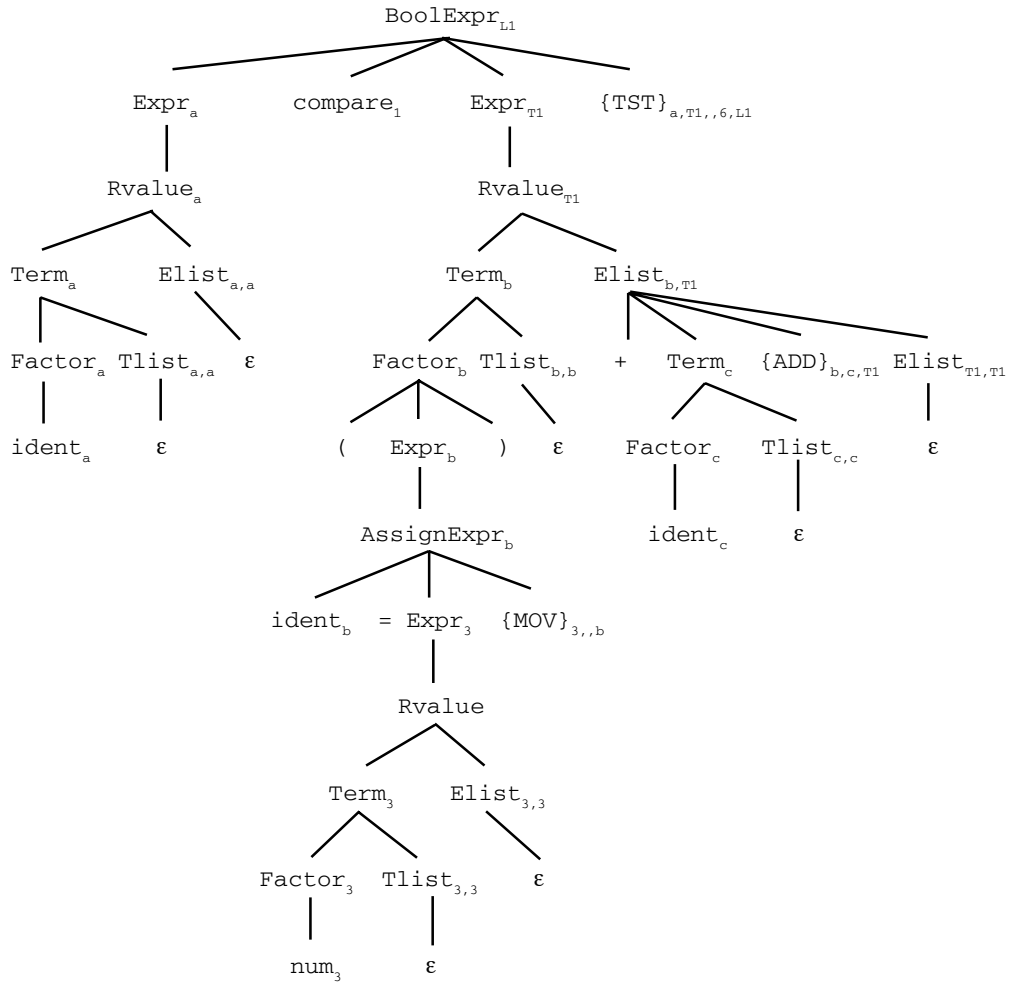
```
(MOV, 3, , c)
(ADD, b, c, T1)
(MOV, T1, , a)
```

An attributed translation grammar for Decaf expressions involving addition, subtraction, multiplication, division, comparisons, and assignment is shown below:

1.  $\text{BoolExpr}_{\text{L1}} \rightarrow \text{Expr}_p \text{ compare}_c \text{ Expr}_q \{ \text{TST} \}_{p,q,,7-c,\text{L1}}$   
L1  $\leftarrow$  newLabel()
2.  $\text{Expr}_p \rightarrow \text{AssignExpr}_p$
3.  $\text{Expr}_p \rightarrow \text{Rvalue}_p$
4.  $\text{AssignExpr}_p \rightarrow \text{ident}_p = \text{Expr}_q \{ \text{MOV} \}_{q,,p}$
5.  $\text{Rvalue}_p \rightarrow \text{Term}_q \text{ Elist}_{q,p}$
6.  $\text{Elist}_{p,q} \rightarrow + \text{Term}_r \{ \text{ADD} \}_{p,r,s} \text{ Elist}_{s,q}$  s  $\leftarrow$  alloc()

**Sample Problem 4.8**

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

**Solution:**

7.	$\text{Elist}_{p,q} \rightarrow$	$- \text{Term}_r \{ \text{SUB} \}_{p,r,s} \text{Elist}_{s,q}$	$s \leftarrow \text{alloc}()$
8.	$\text{Elist}_{p,q} \rightarrow$	$\epsilon$	$q \leftarrow p$
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.	$\text{Factor}_p \rightarrow$	$\text{ident}_p$	

### Exercises 4.8

1. Show an *attributed derivation tree* using the grammar for Decaf expressions given in this section for each of the following expressions or boolean expressions (in part (a) start with Expr; in parts (b,c,d,e) start with BoolExpr):

- |     |                      |     |                  |
|-----|----------------------|-----|------------------|
| (a) | $a = b = c$          | (b) | $a == b + c$     |
| (c) | $(a=3) <= (b=2)$     | (d) | $a == - (c = 3)$ |
| (e) | $a * (b=3) + c != 9$ |     |                  |

2. Show the recursive descent parser for the nonterminals BoolExpr, Rvalue, and Elist given in the grammar for Decaf expressions. Hint: the selection sets for the first eight grammar rules are:

- |         |   |                       |
|---------|---|-----------------------|
| Sel (1) | = | {ident, num, (, +, -} |
| Sel (2) | = | {ident}               |
| Sel (3) | = | {ident, num, (, +, -} |
| Sel (4) | = | {ident}               |
| Sel (5) | = | {ident, num, (, +, -} |
| Sel (6) | = | {+}                   |
| Sel (7) | = | {-}                   |
| Sel (8) | = | {), ←}                |

## 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:

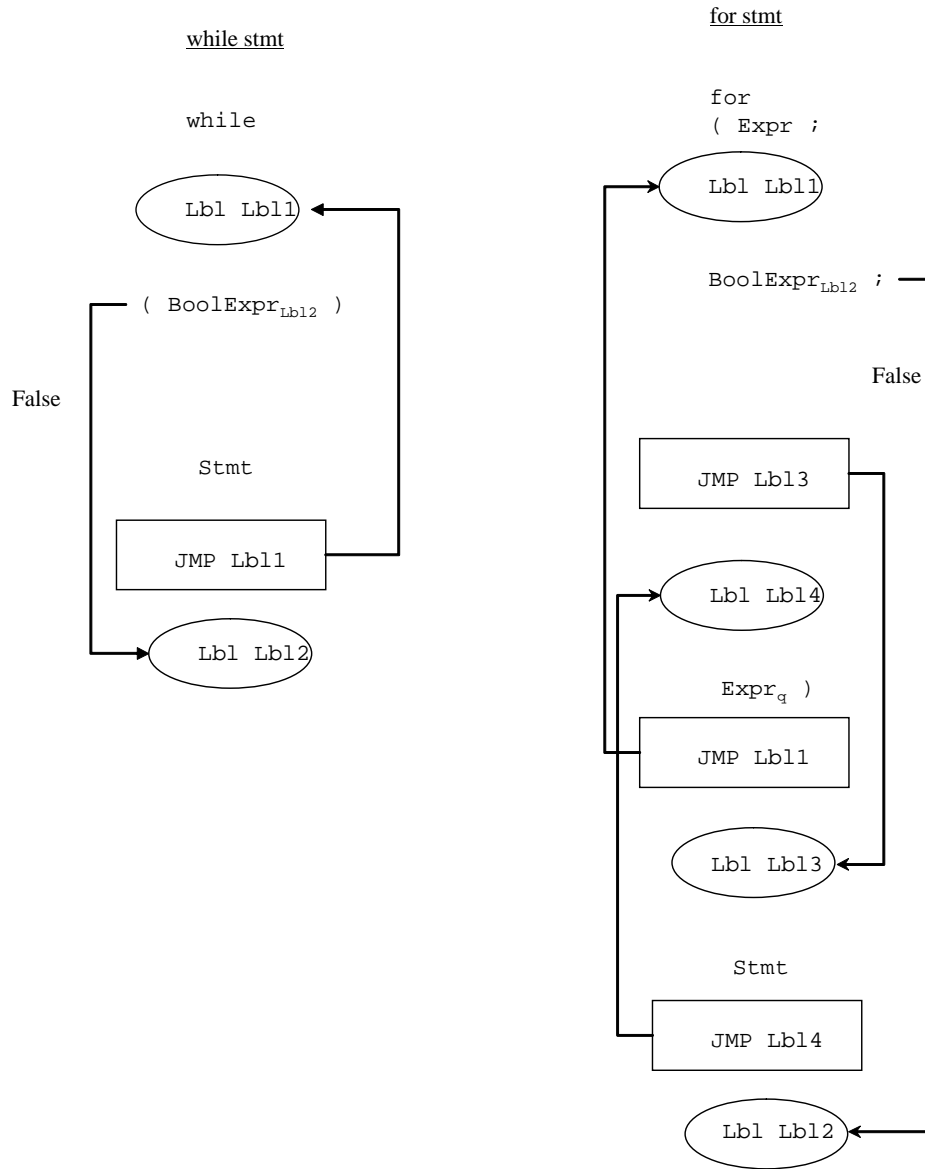
<code>=</code>	is 1	<code>&lt;=</code>	is 4
<code>&lt;</code>	is 2	<code>&gt;=</code>	is 5
<code>&gt;</code>	is 3	<code>!=</code>	is 6

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.

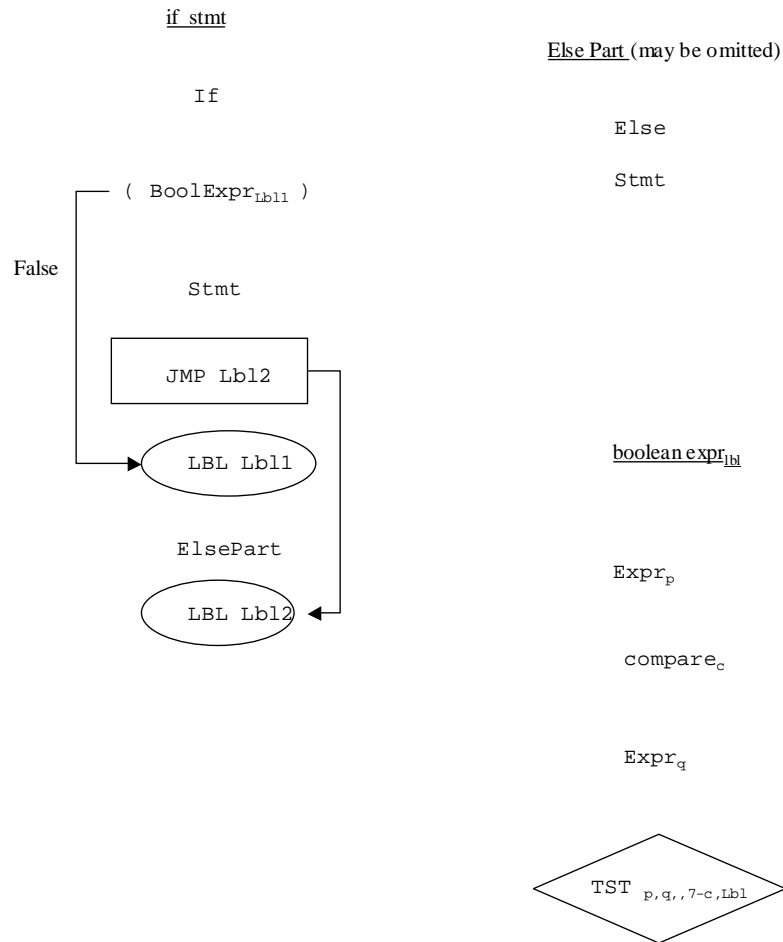
In addition, there is one more atom which is needed for assignment statements; it is a `Move` atom, which simply assigns its source operand to its target operand:

<code>MOV</code>	<code>Src</code>	-	<code>Trg</code>	-	-	<code>Trg = Src</code>
------------------	------------------	---	------------------	---	---	------------------------

Figure 4.18 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 4.18, the atoms are all enclosed in a boundary: `ADD` and `JMP` atoms are



**Figure 4.18 (a)** Control Structures for `while` and `for` Statements



**Figure 4.18 (b)** Control Structures for if Statements and Boolean Expressions.

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

The control structures in Figure 4.18 correspond to the following statement definitions:

1. Stmt  $\rightarrow$  while (BoolExpr) Stmt
2. Stmt  $\rightarrow$  for (Expr; BoolExpr; Expr) Stmt
3. Stmt  $\rightarrow$  if (BoolExpr) Stmt ElsePart

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

For the most part, Figure 4.18 is self explanatory. Figure 4.18 (a) shows the `while` and `for` structures; Figure 4.18(b) shows the `if` structure. In Figure 4.18(b) we also show that a boolean expression always puts out a `TST` atom which branches if the comparison is *false*. The boolean expression has an attribute which is the target label for the branch. Note that for `if` statements, we must jump around the statement which is not to be executed. 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 preceding unmatched `if`. When our parser for the nonterminal `ElsePart` encounters an `else`, it is never wrong to apply rule 4 because the closest preceding 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 Java statement:

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

#### Solution:

```
(LBL, L1)
(TST, x, 0, , 4, L2)           // Branch to L2 if x<=0
```

Atoms for `Stmt`

```
(JMP, L1)
(LBL, L2)
```

```

for (E1; boolean; E2)           E1 ;
    Stmt                       while (boolean)
                                { stmt
                                E2;
                                }

```

This means that after the atoms for the `stmt` are put out, we must put out a jump to the atoms corresponding to expression `E2`. In addition, there must be a jump after the atoms of expression `E2` back to the beginning of the loop, `boolean`. The LL(2) grammar for Decaf shown in the next section makes direct use of figure 4.18 for the control structures.

### Exercises 4.9

1. Show the *sequence of atoms* which would be put out according to Figure 4.18 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 4.18. Assume `if` statements always have an `else` part and that there is a method, `newlab`, which allocates a new statement label.
  1. `WhileStmt → while (BoolExpr) Stmt`
  2. `ForStmt → for (AssignExpr; BoolExpr;
 AssignExpr) Stmt`
  3. `IfStmt → if (BoolExpr) Stmt else Stmt`
3. Show a *recursive descent translator* for your solutions to Problem 2. Show methods for `WhileStmt`, `ForStmt`, and `IfStmt`.

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

```
for (int i=0; i<100; i = i+1)
    {   System.out.println (i);
        i = 100;
    }
```

### 4.10 Case Study: A Top Down Parser for Decaf

In this section we show how the concepts of this chapter can be applied to a compiler for a subset of Java, i.e. the Decaf language. We will develop a top down parser for Decaf and implement it using the technique known as recursive descent. The program is written in Java and is available at <http://www.rowan.edu/~bergmann/books>. 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. The SableCC grammar file, `decaf.grammar`, is used solely for lexical analysis at this point.

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 4.19. 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 Java, like C and C++, was designed to be amenable to top-down parsing (in the tradition of Pascal); the developer of C++ recommends recursive descent parsers (see Stroustrup 1994). To see this, note that most of the constructs 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 `identifier`. In other words, the part of the Decaf grammar defining statements is simple (in the technical sense). Note that the C language permits a statement to be formed by appending a semicolon to any expression; Java, however, requires a simple statement to be either an assignment or a method call.

In Figure 4.19 there are two method 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 4.19 are taken from Figure 4.18 and described in Section 4.9. As mentioned in that section, the convoluted logic flow in the `for` statement results 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 4.19. The only departure is in the description of iterative structures such as `IdentList` and `ArgList`. 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:

Program	→	class identifier { public static void main (String [ ] identifier) CompoundStmt }
Declaration	→	Type IdentList ;
Type	→	int   float
IdentList	→	identifier , IdentList identifier
Stmt	→	AssignStmt   ForStmt   WhileStmt   IfStmt   CompoundStmt   Declaration   ;
AssignStmt	→	AssignExpr <sub>p</sub> ;
ForStmt	→	for (OptAssignExpr <sub>r</sub> ; {LBL} <sub>Lb11</sub> OptBoolExpr <sub>Lb14</sub> ; {JMP} <sub>Lb12</sub> {LBL} <sub>Lb13</sub> OptAssignExpr <sub>r</sub> ) {JMP} <sub>Lb11</sub> {LBL} <sub>Lb12</sub> Stmt {JMP} <sub>Lb13</sub> {LBL} <sub>Lb14</sub> Lb11←newlab() Lb12←newlab() Lb13←newlab()
WhileStmt	→	while {LBL} <sub>Lb11</sub> ( BoolExpr <sub>Lb12</sub> ) Stmt {JMP} <sub>Lb11</sub> {LBL} <sub>Lb12</sub> Lb11←newlab()
IfStmt	→	if ( BoolExpr <sub>Lb11</sub> ) Stmt {JMP} <sub>Lb12</sub> {LBL} <sub>Lb11</sub> ElsePart {LBL} <sub>Lb12</sub> Lb12←newlab()
ElsePart	→	else Stmt   ε
CompoundStmt	→	{ StmtList }
StmtList	→	StmtList Stmt   ε
OptAssignExpr	→	AssignExpr <sub>p</sub>   ε
OptBoolExpr <sub>Lb11</sub>	→	BoolExpr <sub>Lb11</sub>   ε
BoolExpr <sub>Lb11</sub>	→	Expr <sub>p</sub> compare <sub>c</sub> Expr <sub>q</sub> {TST} <sub>p,q,,7-c,Lb11</sub> Lb11 ← newlab()
Expr <sub>p</sub>	→	AssignExpr <sub>p</sub>   Rvalue <sub>q</sub> Elist <sub>q,p</sub>
AssignExpr <sub>p</sub>	→	identifier <sub>p</sub> = Expr <sub>q</sub> {MOV} <sub>q,,p</sub>
Rvalue <sub>p</sub>	→	Term <sub>q</sub> Elist <sub>p,q</sub>

Figure 4.19 An Attributed Translation Grammar for Decaf (continued on next page)

Elist <sub>p,q</sub>	→	+ Term <sub>q</sub> {ADD} <sub>p,r,s</sub> Elist <sub>s,q</sub>	s ← alloc()
		- Term <sub>r</sub> {SUB} <sub>p,r,s</sub> Elist <sub>s,q</sub>	s ← alloc()
		ε	q ← p
Term <sub>p</sub>	→	Factor <sub>q</sub> Tlist <sub>q,p</sub>	
Tlist <sub>p,q</sub>	→	* Factor <sub>r</sub> {MUL} <sub>p,r,s</sub> Tlist <sub>s,q</sub>	s ← alloc()
		/ Factor <sub>r</sub> {DIV} <sub>p,r,s</sub> Tlist <sub>s,q</sub>	s ← alloc()
		ε	q ← p
Factor <sub>p</sub>	→	( Expr <sub>p</sub> )	
Factor <sub>p</sub>	→	+ Factor <sub>p</sub>	
Factor <sub>p</sub>	→	- Factor <sub>q</sub> {Neg} <sub>q,,p</sub>	p ← alloc()
Factor <sub>p</sub>	→	num <sub>p</sub>	
Factor <sub>p</sub>	→	identifier <sub>p</sub>	

Figure 4.19 (Continued)

```

if (token.get_class() != IDENTIFIER) error();
while (token.get_class() == IDENTIFIER)
{
    token.getToken();
    if (token.get_class() == ',')
        token.getToken();
}

```

We use this methodology also for the methods `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 (ignoring unary operations):

```

void Factor (MutableInt p);
{
    if (token.get_class() == '(')
    {
        token.getToken();
        Expr (p);
        if (token.get_class() == ')') token.getToken();
        else error();
    }
    else if (inp == IDENTIFIER)
    {
        // store the value part of the identifier
        p.set (token.get_value());
        token.getToken();
    }
}

```

```

else if (inp == NUMBER)
  {   p.set (token.get_value());
      token.getToken();
  }
else //    check for unary operators +, - ...

```

### Exercises 4.10

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

```

if (a==3) { a = 4;
           for (i = 2; i<5; i=0 ) i = i + 1;
           }
else while (a>5) i = i * 8;

```

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

```

ForStmt    →    for ( OptExprp; {LBL}Lb11 OptBoolExprLb13;
                  {JMP}Lb12 {LBL}Lb14 OptExprr ) {JMP}Lb11
                  {LBL}Lb12 Stmt {JMP}Lb14 {LBL}Lb13
                  Lb11←newlab() Lb12←newlab()
                  Lb13←newlab() Lb14←newlab()

```

3. The Java language has a `switch` statement.
  - (a) Include a definition of the `switch` statement in the *attributed translation grammar* for Decaf.
  - (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*, `decaf.java` and `parse.java`.
4. Using the grammar of Figure 4.19, show an attributed derivation tree for the statement given in problem 1, above.
5. Implement a `do-while` statement in `decaf`, following the guidelines in problem 3.

## 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 techniques for top down parsing: (1) *pushdown machines* and (2) *recursive descent*. These techniques 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 techniques 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 – Decaf. This parser is written as a recursive descent parser in Java, and makes use of SableCC for the lexical scanner. In the interest of keeping costs down, it is not shown in the appendix; however, it is available along with the other software files at <http://www.rowan.edu/~bergmann/books>.