11 SYNTAX-DIRECTED TRANSLATION

In this chapter we build on the ideas developed in the last two, and continue towards our goal of developing translators for computer languages, by discussing how syntax analysis can form the basis for driving a translator, or similar programs that process input strings that can be described by a grammar. Our discussion will be limited to methods that fit in with the top-down approach studied so far, and we shall make the further simplifying assumption that the sentences to be analysed are essentially syntactically correct.

11.1 Embedding semantic actions into syntax rules

The primary goal of the types of parser studied in the last chapter - or, indeed, of any parser - is the recognition or rejection of input strings that claim to be valid sentences of the language under consideration. However, it does not take much imagination to see that once a parser has been constructed it might be enhanced to perform specific actions whenever various syntactic constructs have been recognized.

As usual, a simple example will help to crystallize the concept. We turn again to the grammars that can describe simple algebraic expressions, and in this case to a variant that can handle parenthesized expressions in addition to the usual four operators:

```
   Expression =  Term { "+" Term | "-" Term } .
   Term       =  Factor { "*" Factor | "/" Factor } .
   Factor     =  identifier | number | "(" Expression ")" .
```

It is easily verified that this grammar is LL(1). A simple recursive descent parser is readily constructed, with the aim of accepting a valid input expression, or aborting with an appropriate message if the input expression is malformed.

```
  void Expression(void);  // function prototype

  void Factor(void)
  // Factor = identifier | number | "(" Expression ")" .
  { switch (SYM.sym)
    { case identifier:
      case number:
        getsym(); break;
      case lparen:
        getsym(); Expression();
        accept(rparen, " Error - ')' expected"); break;
      default:
        printf("Unexpected symbol\n"); exit(1);
    }
  }

  void Term(void)
  // Term = Factor { "*" Factor | "/" Factor } .
  { Factor();
    while (SYM.sym == times || SYM.sym == slash)
    { getsym(); Factor(); }
  }

  void Expression(void)
  // Expression = Term { "+" Term | "-" Term } .
  { Term();
    while (SYM.sym == plus || SYM.sym == minus)
    { getsym(); Term(); }
  }
```

Note that in this and subsequent examples we have assumed the existence of a lower level scanner that recognizes fundamental terminal symbols, and constructs a globally accessible variable `SYM` that has a structure declared on the lines of

```
  enum symtypes {
    unknown, eofsym, identifier, number, plus, minus, times, slash,
    lparen, rparen, equals
  };

  struct symbols {
    symtypes sym;      // class
    char name;         // lexeme
    int num;           // value
  };

  symbols SYM; // Source token
```

The parser proper requires that an initial call to `getsym()` be made before calling `Expression()` for the first time.

We have also assumed the existence of a severe error handler, similar to that used in the last chapter:

```
  void accept(symtypes expectedterminal, char *errormessage)
  { if (SYM.sym != expectedterminal) { puts(errormessage); exit(1); }
    getsym();
  }
```

Now consider the problem of reading a valid string in this language, and translating it into a string that has the same meaning, but which is expressed in *postfix* (that is, "reverse Polish") notation. Here the operators follow the pair-wise operands, and there is no need for parentheses. For example, the infix expression

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

is to be translated into its postfix equivalent

$$a\ b + c\ d - *$$

This is a well-known problem, admitting of a fairly straightforward solution. As we read the input string from left to right we immediately copy all the operands to the output stream as soon as they are recognized, but we delay copying the operators until we can do so in an order that relates to the familiar precedence rules for the operations. With a little thought the reader should see that the grammar and the parser given above capture the spirit of these precedence rules. Given this insight, it is not difficult to see that the augmented routine below not only parses input strings; the execution of the carefully positioned output statements effectively produces the required postfix translation.

```
  void Factor(void)
  // Factor = identifier | number | "(" Expression ")" .
  { switch (SYM.sym)
    { case identifier:
      case number:
        printf(" %c ", SYM.name); getsym(); break;
      case lparen:
        getsym(); Expression();
        accept(rparen, " Error - ')' expected"); break;
      default:
        printf("Unexpected symbol\n"); exit(1);
    }
  }

  void Term(void)
  // Term = Factor { "*" Factor | "/" Factor } .
  { Factor();
    while (SYM.sym == times || SYM.sym == slash)
    { switch (SYM.sym)
      { case times: getsym(); Factor(); printf(" * "); break;
        case slash: getsym(); Factor(); printf(" / "); break;
      }
    }
  }

  void Expression(void)
  // Expression = Term { "+" Term | "-" Term } .
  { Term();
    while (SYM.sym == plus || SYM.sym == minus)
    { switch (SYM.sym)
      { case plus:  getsym(); Term(); printf(" + "); break;
        case minus: getsym(); Term(); printf(" - "); break;
      }
    }
  }
```

In a very real sense we have moved from a parser to a compiler in one easy move! What we have illustrated is a simple example of a syntax-directed program; one in which the underlying algorithm is readily developed from an understanding of an underlying syntactic structure. Compilers are obvious candidates for this sort of development, although the technique is more generally applicable, as hopefully will become clear.

The reader might wonder whether this idea could somehow be reflected back to the formal grammar from which the parser was developed. Various schemes have been proposed for doing this. Many of these use the idea of adding **semantic actions** into context-free BNF or EBNF production schemes.

Unfortunately there is no clear winner among the notations proposed for this purpose. Most, however, incorporate the actions by writing statements in some implementation language (for example, Modula-2 or C++) between suitably chosen meta-brackets that are not already bespoke in that language. For example, Coco/R uses EBNF for expressing the productions and brackets the actions with "`(.`" and "`.)`", as in the example below.

```
   Expression
   =  Term
      {   "+" Term            (. Write('+'); .)
        | "-" Term            (. Write('-'); .)
      } .

   Term
   =  Factor
      {   "*" Factor          (. Write('*'); .)
        | "/" Factor          (. Write('/'); .)
      } .

   Factor
   =   ( identifier | number ) (. Write(SYM.name); .)
     | "("  Expression  ")" .
```

The **yacc** parser generator on UNIX systems uses unextended BNF for the productions and uses braces "`{`" and "`}`" around actions expressed in C.

Exercises

11.1 Extend the grammar and the parsers so as to handle an expression language in which one may have an optional leading + or **-** sign (as exemplified by + *a* * (- *b* + *c*)).

11.2 Attribute grammars

A little reflection will show that, although an algebraic expression clearly has a semantic meaning (in the sense of its "value"), this was not brought out when developing the last example. While the idea of incorporating actions into the context-free productions of a grammar gives a powerful tool for documenting and developing syntax- directed programs, what we have seen so far is still inadequate for handling the many situations where some deeper semantic meaning is required.

We have seen how a context-free grammar can be used to describe many features of programming languages. Such grammars effectively define a derivation or parse tree for each syntactically correct program in the language, and we have seen that with care we can construct the grammar so that a parse tree in some way reflects the meaning of the program as well.

As an example, consider the usual old chestnut language, albeit expressed with a slightly different (non-LL(1)) grammar

```
  Goal        =  Expression .
  Expression  =  Term  |  Expression  "+"  Term  |  Expression  "-"  Term.
  Term        =  Factor  |  Term  "*"  Factor  |  Term  "/"  Factor .
  Factor      =  identifier  |  number  |  "("  Expression  ")" .
```

and consider the phrase structure tree for *x* + *y* * *z*, shown in Figure 11.1.

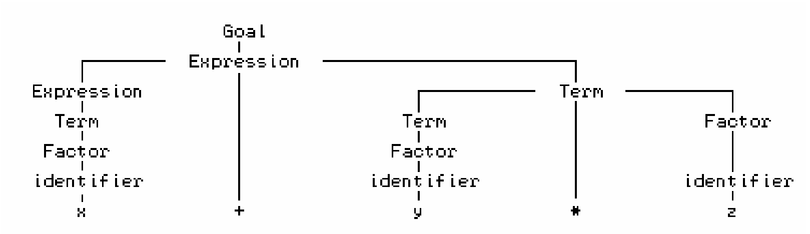

Figure 11.1 Phrase structure tree for x + y * z

Suppose *x*, *y* and *z* had associated numerical values of 3, 4 and 5, respectively. We can think of these as **semantic attributes** of the leaf nodes *x*, *y* and *z*. Similarly we can think of the nodes ’+’ and ’*’ as having attributes of "add" and "multiply". Evaluation of the whole expression can be regarded as a process where these various attributes are passed "up" the tree from the terminal nodes and are semantically transformed and combined at higher nodes to produce a final result or attribute at the root - the value (23) of the *Goal* symbol. This is illustrated in Figure 11.2.

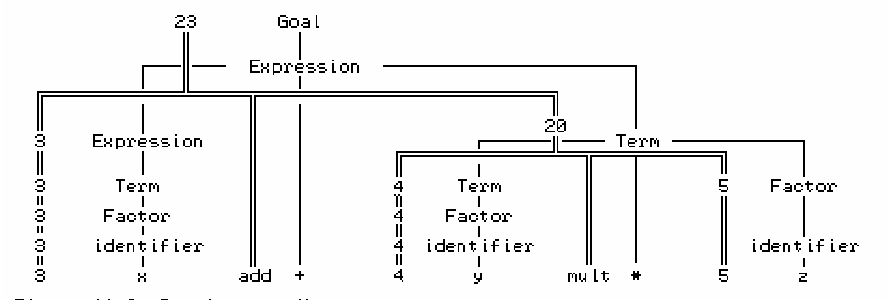

Figure 11.2 Passing attributes up a parse tree

In principle, and indeed in practice, parsing algorithms can be written whose embedded actions explicitly construct such trees as the input sentences are parsed, and also *decorate* or *annotate* the nodes with the semantic attributes. Associated tree-walking algorithms can then later be invoked to process this semantic information in a variety of ways, possibly making several passes over the tree before the evaluation is complete. This approach lends itself well to the construction of optimizing compilers, where repeatedly walking the tree can be used to prune or graft nodes in a way that a simpler compiler cannot hope to do.

The parser constructed in the last section for recognizing this language did not, of course, construct an explicit parse tree. The grammar we have now employed seems to map immediately to parse trees in which the usual associativity and precedence of the operators is correctly reflected. It is left recursive, and thus unsuitable as the basis on which to construct a recursive descent parser. However, as we saw in section 10.6, it is possible to construct other forms of parser to handle grammars that employ left recursion. For the moment we shall not pursue the interesting problem of whether or how a recursive descent parser could be modified to generate an explicit tree. We shall content ourselves with the observation that the execution of such a parser effectively walks an implicit structure, whose nodes correspond to the various calls made to the sub-parsers as the parse proceeds.

Notwithstanding any apparent practical difficulties, our notions of formal grammars may be extended to try to capture the essence of the attributes associated with the nodes, by extending the notation still further. In one scheme, attribute rules are associated with the context-free productions in much the same way as we have already seen for actions, giving rise to what is known as an **attribute grammar**. As usual, an example will help to clarify:

```
Goal
=   Expression            (. Goal.Value := Expr.Value .) .
Expression
=   Term                  (. Expr.Value := Term.Value .)
  | Expression  "+"  Term (. Expr.Value := Expr.Value + Term.Value .)
  | Expression  "-"  Term (. Expr.Value := Expr.Value - Term.Value .) .
Term
=   Factor                (. Term.Value := Fact.Value .)
  | Term  "*"  Factor     (. Term.Value := Term.Value * Fact.Value .)
  | Term  "/"  Factor     (. Term.Value := Term.Value / Fact.Value .) .
Factor
=   identifier            (. Fact.Value := identifier.Value .)
  | number                (. Fact.Value := number.Value .)
  | "("  Expression  ")"  (. Fact.Value := Expr.Value .) .
```

Here we have employed the familiar "dot" notation that many imperative languages use in designating the elements of record structures. Were we to employ a parsing algorithm that constructed an explicit tree, this notation would immediately be consistent with the declarations of the tree nodes used for these structures.

It is important to note that the semantic rules for a given production specify the relationships between attributes of other symbols in the *same* production, and are essentially "local".

It is not necessary to have a left recursive grammar to be able to provide attribute information. We could write an iterative LL(1) grammar in much the same way:

```
Goal
=  Expression             (. Goal.Value := Expr.Value .) .
Expression
=  Term                   (. Expr.Value := Term.Value .)
   {   "+"  Term          (. Expr.Value := Expr.Value + Term.Value .)
     | "-"  Term          (. Expr.Value := Expr.Value - Term.Value .)
   } .
Term
=  Factor                 (. Term.Value := Fact.Value .)
   {   "*"  Factor        (. Term.Value := Term.Value * Fact.Value .)
     | "/"  Factor        (. Term.Value := Term.Value / Fact.Value .)
   } .
Factor
=   identifier            (. Fact.Value := identifier.Value .)
  | number                (. Fact.Value := number.Value .)
```

```
    | "("  Expression  ")"  (. Fact.Value := Expr.Value .) .
```

Our notation does yet lend itself immediately to the specification and construction of those parsers that do *not* construct explicit structures of decorated nodes. However, it is not difficult to develop a suitable extension. We have already seen that the construction of parsers can be based on the idea that expansion of each non-terminal is handled by an associated routine. These routines can be parameterized, and the parameters can transmit the attributes to where they are needed. Using this idea we might express our expression grammar as follows (where we have introduced yet more meta-brackets, this time denoted by "<" and ">"):

```
Goal < Value >
=  Expression  < Value > .
Expression < Value >
=  Term < Value >
   {   "+"  Term < TermValue >     (. Value := Value + TermValue .)
     | "-"  Term < TermValue >     (. Value := Value - TermValue .)
   } .
Term < Value >
=  Factor < Value >
   {   "*"  Factor < FactorValue > (. Value := Value * FactorValue .)
     | "/"  Factor < FactorValue > (. Value := Value / FactorValue .)
   } .
Factor < Value >
=    identifier < Value >
  |  number  < Value >
  |  "("  Expression  < Value > ")" .
```

11.3 Synthesized and inherited attributes

A little contemplation of the parse tree in our earlier example, and of the attributes as given here, should convince the reader that (in this example at least) we have a situation in which the attributes of any particular node depend only on the attributes of nodes in the subtrees of the node in question. In a sense, information is always passed "up" the tree, or "out" of the corresponding routines. The parameters must be passed "by reference", and the grammar above maps into code of the form shown below (where we shall, for the moment, ignore the issue of how one attributes an identifier with an associated numeric value).

```
void Factor(int &value)
// Factor = identifier | number | "(" Expression ")" .
{ switch (SYM.sym)
  { case identifier:
    case number:
      value = SYM.num; getsym(); break;
    case lparen:
      getsym(); Expression(value);
      accept(rparen, " Error - ')' expected"); break;
    default:
      printf("Unexpected symbol\n"); exit(1);
  }
}

void Term(int &value)
// Term = Factor { "*" Factor | "/" Factor } .
{ int factorvalue;
  Factor(value);
  while (SYM.sym == times || SYM.sym == slash)
  { switch (SYM.sym)
    { case times:
        getsym(); Factor(factorvalue); value *= factorvalue; break;
      case slash:
        getsym(); Factor(factorvalue); value /= factorvalue; break;
    }
  }
}

void Expression(int &value)
// Expression = Term { "+" Term | "-" Term } .
{ int termvalue;
```

```
    Term(value);
    while (SYM.sym == plus || SYM.sym == minus)
    { switch (SYM.sym)
      { case plus:
          getsym(); Term(termvalue); value += termvalue; break;
        case minus:
          getsym(); Term(termvalue); value -= termvalue; break;
      }
    }
  }
```

Attributes that travel in this way are known as synthesized attributes. In general, given a context-free production rule of the form

$$A = \alpha\, B\, \gamma$$

then an associated semantic rule of the form

$$A.\text{attribute}_i = f\ (\alpha.\text{attribute}_j,\ B.\text{attribute}_k,\ \gamma.\text{attribute}_l\ )$$

is said to specify a **synthesized attribute** of *A*.

Attributes do not always travel up a tree. As a rather grander example, consider the very small CLANG program:

```
PROGRAM Silly;
  CONST
    Bonus = 4;
  VAR
    Pay;
  BEGIN
    WRITE(Pay + Bonus)
  END.
```

which has the phrase structure tree shown in Figure 11.3.

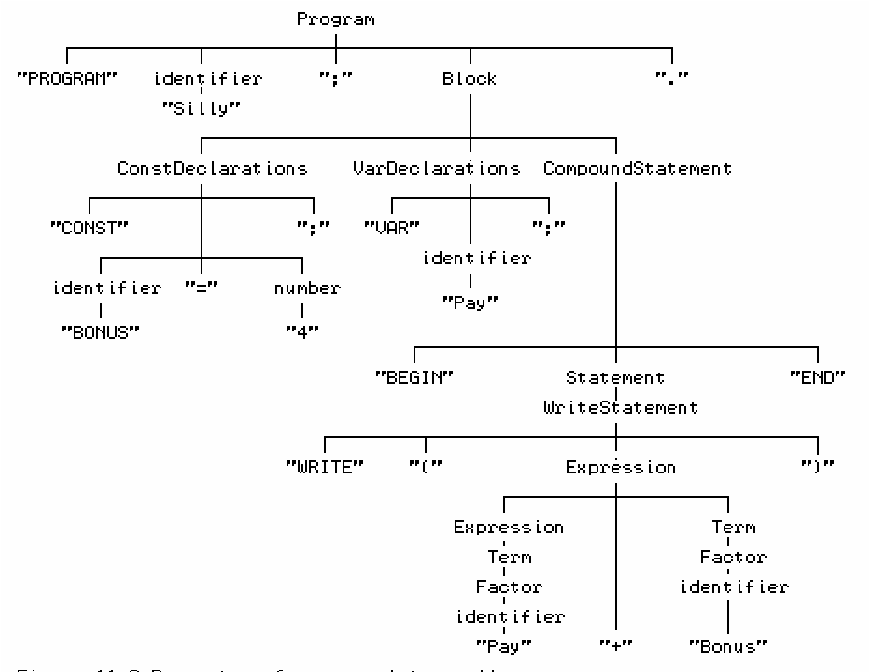

Figure 11.3 Parse tree for a complete small program

In this case we can think of the Boolean *IsConstant* and *IsVariable* attributes of the nodes `CONST` and `VAR` as being passed up the tree (*synthesized*), and then later passed back down and *inherited* by other nodes like `Bonus` and `Pay` (see Figure 11.4). In a sense, the context in which the identifiers were declared is being remembered - the system is providing a way of handling context-sensitive features of an otherwise context-free language.

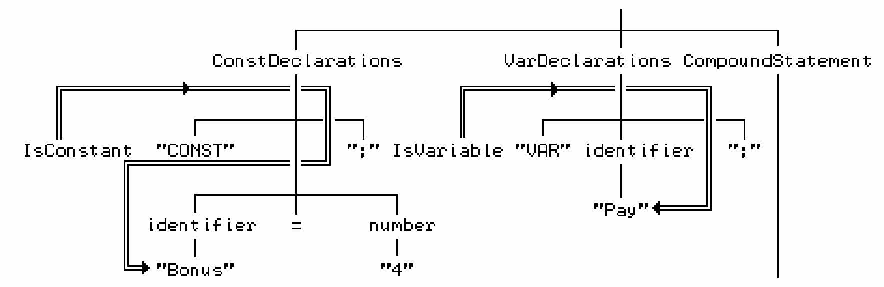

Figure 11.4 Attributes passed up and down a parse tree

Of course, this idea must be taken much further. Attributes like this form part of what is usually termed an **environment**. Compilation or parsing of programs in a language like Pascal or Modula-2 generally begins in a "standard" environment, into which pervasive identifiers like `TRUE, FALSE, ORD, CHR` and so on are already incorporated. This environment is inherited by *Program* and then by *Block* and then by *ConstDeclarations*, which augments it and passes it back up, to be inherited in its augmented form by *VarDeclarations* which augments it further and passes it back, so that it may then be passed down to the *CompoundStatement*. We may try to depict this as shown in Figure 11.5.

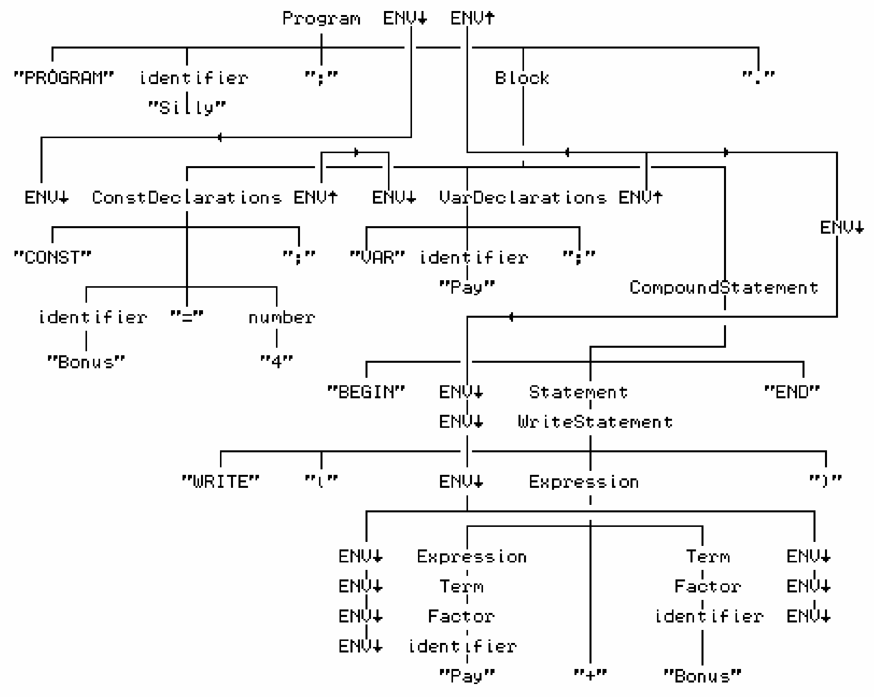

Figure 11.5 Modification of the parsing environment

More generally, given a context-free production rule of the form

$$A = \alpha\, B\, \gamma$$

an associated semantic rule of the form

$$B.\text{attribute}_i = f\ (\alpha.\text{attribute}_j, A.\text{attribute}_k, \gamma.\text{attribute}_l\ )$$

is said to specify an **inherited attribute** of *B*. The inherited attributes of a symbol are computed from information held in the environment of the symbol in the parse tree.

As before, our formal notation needs modification to reflect the different forms and flows of attributes. A notation often used employs arrows $\uparrow$ and $\downarrow$ in conjunction with the parameters mentioned in the < > metabrackets. Inherited attributes are marked with $\downarrow$, and synthesized attributes with $\uparrow$. In terms of actual coding, $\uparrow$ attributes correspond to "reference" parameters, while $\downarrow$ attributes correspond to "value" parameters. In practice, reference parameters may also be used to manipulate features (such as an environment) that are inherited, modified, and then returned; these are sometimes called **transmitted attributes**, and are marked with $\downarrow\uparrow$ or $\updownarrow$.

11.4 Classes of attribute grammars

Attribute grammars have other important features. If the action of a parser is in some way to construct a tree whose nodes are decorated with semantic attributes relevant to each node, then "walking" this tree after it has been constructed should constitute a possible mechanism for developing the synthetic aspects of a translator, such as code generation. If this is this case, then the order in which the tree is walked becomes crucially important, since attributes can depend on one another. The simplest tree walk - the depth-first, left-to-right method - may not suffice. Indeed, we have a situation completely analogous to that which arises in attempting single-pass assembly and discovering forward references to labels. In principle we can, of course, perform multiple tree walks, just as we can perform multiple-pass assembly. There are, however, two types of attribute grammars for which this is not necessary.

- An **S-attributed grammar** is one that uses only synthesized attributes. For such a grammar the attributes can obviously be correctly evaluated using a bottom-up walk of the parse tree. Furthermore, such a grammar is easily handled by parsing algorithms (such as recursive descent) that do not explicitly build the parse tree.

- An **L-attributed grammar** is one in which the inherited attributes of a particular symbol in any given production are restricted in certain ways. For each production of the general form

  $$A \rightarrow B_1\ B_2\ ...\ B_n$$

  the inherited attributes of B_k may depend only on the inherited attributes of *A* or synthesized attributes of $B_1, B_2\ ...\ B_{k-1}$. For such a grammar the attributes can be correctly evaluated using a left- to-right depth-first walk of the parse tree, and such grammars are usually easily handled by recursive descent parsers, which implicitly walk the parse tree in this way.

We have already pointed out that there are various aspects of computer languages that involve context sensitivity, even though the general form of the syntax might be expressed in a context-free way. Context-sensitive constraints on such languages - often called *context conditions* - are often conveniently expressed by conditions included in its attribute grammar, specifying relations that must be satisfied between the attribute values in the parse tree of a valid program. For example, we might have a production like

```
Assignment =  VarDesignator < TypeV↑ > ":=" Expression < TypeE↑ >
              (. where AssignmentCompatible(TypeV↓, TypeE↓) .) .
```

Alternatively, and more usefully in the construction of real parsers, the context conditions might be expressed in the same notation as for semantic actions, for example

```
Assignment =  VarDesignator < TypeV↑ > ":=" Expression < TypeE↑ >
              (. if (Incompatible(TypeV↓, TypeE↓))
                     SemanticError("Incompatible types"); .) .
```

Finally, we should note that the concept of an attribute grammar may be formally defined in several ways. Waite and Goos (1984) and Rechenberg and Mössenböck (1989) suggest:

> An attribute grammar is a quadruple { G, A, R, K }, where G = { N, T, S, P } is a reduced context-free grammar, A is a finite set of attributes, R is a finite set of semantic actions, and K is a finite set of context conditions. Zero or more attributes from A are associated with each symbol $X \in N \cup T$, and zero or more semantic actions from R and zero or more context conditions from K are associated with each production in P. For each occurrence of a non-terminal X in the parse tree of a sentence in $L(G)$ the attributes of X can be computed in at most one way by semantic actions.

Further reading

Good treatments of the material discussed in this section can be found in the books by Gough (1988), Bennett (1990), and Rechenberg and Mössenböck (1989). As always, the text by Aho, Sethi and Ullman (1986) is a mine of information.

11.5 Case study - a small student database

As another example of using an attribute grammar to construct a system, consider the problem of constructing a database of the members of a student group. In particular, we wish to record their names, along with their intended degrees, after extracting information from an original data file that has records like the following:

```
CompScience3
    BSc  : Mike, Juanito, Rob, Keith, Bruce ;
    BScS : Erik, Arne, Paul, Rory, Andrew, Carl, Jeffrey ;
    BSc  : Nico, Kirsten, Peter, Luanne, Jackie, Mark .
```

Although we are not involved with constructing a compiler in this instance, we still have an example of a syntax directed computation. This data can be described by the context-free productions

```
ClassList  =  ClassName [ Group { ";" Group  } ]  "." .
Group      =  Degree  ":"  Student { "," Student } .
Degree     =  "BSc" | "BScS" .
ClassName  =  identifier .
Student    =  identifier .
```

The attributes of greatest interest are, probably, those that relate to the students' names and degree codes. An attribute grammar, with semantic actions that define how the database could be set up, is as follows:

```
ClassList
=  ClassName                (. OpenDataBase .)
   [ Group { ";" Group } ]  (. CloseDataBase .)
   "." .
Group
=  Degree < DegreeCode↑ >
   ":"  Student < DegreeCode↓ >
   { "," Student < DegreeCode↓ > } .
Degree < DegreeCode ↑ >
=   "BSc"                   (. DegreeCode := bsc .)
  | "BScS"                  (. DegreeCode := bscs .) .
ClassName
=  identifier .
Student < DegreeCode↓ >
=  identifier < Name↑ >     (. AddToDataBase(Name↓, DegreeCode↓) .) .
```

It should be easy to see that this can be used to derive code on the lines of

```
void Student(codes DegreeCode)
{ if (SYM.sym == identifier)
    { AddToDataBase(SYM.name, DegreeCode); getsym(); }
  else
    { printf(" error - student name expected\n"); exit(1); }
}

void Degree(codes &DegreeCode)
{ switch (SYM.sym)
  { case bscsym  : DegreeCode = bsc; break;
    case bscssym : DegreeCode = bscs; break;
    default : printf(" error - invalid degree\n"); exit(1);
  }
  getsym();
}

void Group(void)
{ codes DegreeCode;
  Degree(DegreeCode);
  accept(colon, " error - ':' expected");
  Student(DegreeCode);
  while (SYM.sym == comma)
  { getsym(); Student(DegreeCode); }
}

void ClassName(void)
{ accept(identifier, " error - class name expected"); }

void ClassList(void)
{ ClassName();
  OpenDataBase();
  if (SYM.sym == bscsym || SYM.sym == bscssym)
  { Group();
    while (SYM.sym == semicolon) { getsym(); Group(); }
  }
  CloseDataBase();
  accept(period, " error - '.' expected");
}
```

Although all the examples so far have lent themselves to very easy implementation by a recursive descent parser, it is not difficult to find an example where difficulties arise. Consider the *ClassList* example again, but suppose that the input data had been of a form like

```
CompScience3
   Mike, Juanito, Rob, Keith, Bruce             : BSc ;
   Erik, Arne, Paul, Rory, Andrew, Carl, Jeffrey : BScS ;
   Nico, Kirsten, Peter, Luanne, Jackie, Mark    : BSc .
```

This data can be described by the context-free productions

```
ClassList  =  ClassName [ Group { ";" Group } ]  "." .
Group      =  Student { "," Student } ":" Degree .
Degree     =  "BSc" | "BScS" .
ClassName  =  identifier .
Student    =  identifier .
```

Now a moment's thought should convince the reader that attributing the grammar as follows

```
Group
=  Student < Name↑ >          (. AddToDataBase(Name↓, DegreeCode↓) .)
   { "," Student < Name↑ >    (. AddToDataBase(Name↓, DegreeCode↓) .)
   } ":" Degree < DegreeCode↑ > .
Student < Name↑ >
=  identifier < Name↑ >
```

does not create an L-attributed grammar, but has the unfortunate effect that at the point where it seems natural to add a student to the database, his or her degree has not yet been ascertained.

Just as we did for the one-pass assembler, so here we can sidestep the problem by creating a local forward reference table. It is not particularly difficult to handle this grammar with a recursive descent parser, as the following amended code will reveal:

```
void Student(names &Name)
{ if (SYM.sym == identifier)
    { Name = SYM.name; getsym(); }
  else
    { printf(" error - student name expected\n"); exit(1); }
}

void Group(void)
{ codes DegreeCode;
  names Name[100];
  int last = 0;
  Student(Name[last]);            // first forward reference
  while (SYM.sym == comma)
  { getsym();
    last++; Student(Name[last]);  // add to forward references
  }
  accept(colon, " error - ':' expected");
  Degree(DegreeCode);
  for (int i = last; i >= 0; i--) // process forward reference list
    AddToDataBase(Name[i], DegreeCode);
}
```

Exercises

11.2 Develop an attribute grammar and corresponding parser to handle the evaluation of an expression where there may be an optional leading + or **-** sign (as exemplified by
`+ 9 * ( - 6 + 5 )`).

11.3 Develop an attribute grammar for the 8-bit ASSEMBLER language used in section 4.3, and use it to build an assembler for this language.

11.4 Develop an attribute grammar for the stack ASSEMBLER language used in section 4.4, and use it to build an assembler for this language.