10 PARSER AND SCANNER CONSTRUCTION

In this chapter we aim to show how parsers and scanners may be synthesized once appropriate grammars have been written. Our treatment covers the manual construction of these important components of the translation process, as well as an introduction to the use of software tools that help automate the process.

10.1 Construction of simple recursive descent parsers

For the kinds of language that satisfy the rules discussed in the last chapter, parser construction turns out to be remarkably easy. The syntax of these languages is governed by production rules of the form

non-terminal $\rightarrow$ allowable string

where the allowable string is a concatenation derived from

- the basic symbols or terminals of the language
- other non-terminals
- the actions of meta-symbols such as { }, [ ], and | .

We express the effect of applying each production by writing a procedure (or *void function* in C++ terminology) to which we give the name of the non-terminal that appears on its left side. The purpose of this routine is to analyse a sequence of symbols, which will be supplied on request from a suitable scanner (lexical analyser), and to verify that it is of the correct form, reporting errors if it is not. To ensure consistency, the routine corresponding to any non-terminal *S*:

- may assume that it has been called *after* some (globally accessible) variable *Sym* has been found to contain one of the terminals in FIRST(*S*).

- will then parse a complete sequence of terminals which can be derived from *S*, reporting an error if no such sequence is found. (In doing this it may have to call on similar routines to handle sub-sequences.)

- will relinquish parsing after leaving *Sym* with the first terminal that it finds which cannot be derived from *S*, that is to say, a member of the set FOLLOW(*S*).

The shell of each parsing routine is thus

```
PROCEDURE S;
(* S  →  string *)
  BEGIN
    (* we assert Sym  ∈  FIRST(S) *)
    Parse(string)
    (* we assert  Sym  ∈  FOLLOW(S) *)
  END S;
```

where the transformation *Parse(string)* is governed by the following rules:

(a) If the production yields a single terminal, then the action of *Parse* is to report an error if an unexpected terminal is detected, or (more optimistically) to accept it, and then to scan to the next symbol.

```
Parse (terminal) →
  IF IsExpected(terminal)
    THEN Get(Sym)
    ELSE ReportError
  END
```

(b) If we are dealing with a "single" production (that is, one of the form *A* = *B*), then the action of *Parse* is a simple invocation of the corresponding routine

```
Parse(SingleProduction A)  →  B
```

This is a rather trivial case, just mentioned here for completeness. Single productions do not really need special mention, except where they arise in the treatment of longer strings, as discussed below.

(c) If the production allows a number of alternative forms, then the action can be expressed as a selection

```
Parse (α1  |  α2  |  ... αn ) →
  CASE Sym OF
    FIRST(α1 ) : Parse(α1 );
    FIRST(α2 ) : Parse(α2 );
    ......
    FIRST(αn ) : Parse(αn )
  END
```

in which we see immediately the relevance of Rule 1. In fact we can go further to see the relevance of Rule 2, for to the above we should add the action to be taken if one of the alternatives of *Parse* is empty. Here we do nothing to advance *Sym* - an action which must leave *Sym*, as we have seen, as one of the set FOLLOW(*S*) - so that we may augment the above *in this case* as

```
Parse (α1  |  α2  | ... αn  |  ε ) →
  CASE Sym OF
    FIRST(α1 ) : Parse(α1 );
    FIRST(α2 ) : Parse(α2 );
    ......
    FIRST(αn ) : Parse(αn );
    FOLLOW(S)  :  (* do nothing *)
    ELSE ReportError
  END
```

(d) If the production allows for a nullable option, the transformation involves a decision

```
Parse ( [ α ] ) →
  IF Sym ∈ FIRST(α) THEN Parse(α) END
```

(e) If the production allows for possible repetition, the transformation involves a loop, often of the form

```
Parse ( { α } ) →
  WHILE Sym ∈ FIRST(α) DO Parse(α) END
```

Note the importance of Rule 2 here again. Some repetitions are of the form

```
S  →  α  { α }
```

which transforms to

```
Parse(α);  WHILE Sym ∈ FIRST(α) DO Parse(α)  END
```

On occasions this may be better written

```
          REPEAT  Parse(α) UNTIL Sym ∉ FIRST(α)
```

(f) Very often, the production generates a sequence of terminal and non-terminals. The action is then a sequence derived from (a) and (b), namely

```
                 Parse (α_1 α_2 ... α_n ) →
                   Parse(α_1 ); Parse(α_2 ); ... Parse(α_n )
```

10.2 Case studies

To illustrate these ideas further, let us consider some concrete examples.

The first involves a rather simple grammar, chosen to illustrate the various options discussed above.

```
            G = { N , T , S , P }
            N = { A , B , C , D }
            T = { "("  ,  ")"  ,  "+"  ,  "a"  ,  "["  ,  "]"  ,  "." }
            S = A
            P =
                A  →  B  "."
                B  →  [  "a"   |   "("  C  ")"   |   "["  B  "]"  ]
                C  →  B D
                D  →  { "+"  B }
```

We first check that this language satisfies the requirements for LL(1) parsing. We can easily see that Rule 1 is satisfied. As before, in order to apply our rules more easily we first rewrite the productions to eliminate the EBNF metasymbols:

```
                A  →  B  "."                                        (1)
                B  →  "a"   |   "("  C  ")"   |   "["  B  "]"  |  ε   (2, 3, 4, 5)
                C  →  B D                                           (6)
                D  →  "+"  B  D  |  ε                               (7, 8)
```

The only productions for which there are alternatives are those for *B* and *D*, and each non-nullable alternative starts with a different terminal. However, we must continue to check Rule 2. We note that *B* and *D* can both generate the null string. We readily compute

FIRST(*B*) = { "*a*" , "(" , "[" }
FIRST(*D*) = { "+" }

The computation of the FOLLOW sets is a little trickier. We need to compute FOLLOW(*B*) and FOLLOW(*D*).

For FOLLOW(*D*) we use the rules of section 9.2. We check productions that generate strings of the form $\alpha D \zeta$. These are the ones for *C* (6) and for *D* (7). Both of these have *D* as their rightmost symbol; (7) in fact tells us nothing of interest, and we are lead to the result that

FOLLOW(*D*) = FOLLOW(*C*) = { ")" }.

(FOLLOW(*C*) is determined by looking at production (3)).

For FOLLOW(*B*) we check productions that generate strings of the form $\alpha B \zeta$. These are the ones for *A* (1) and *C* (6), the third alternative for *B* itself (4), and the first alternative for *D* (7). This seems to indicate that

FOLLOW(B) = { "." , "]" } $\cup$ FIRST(D) = { "." , "]" , "+" }

We must be more careful. Since the production for D can generate a null string, we must augment FOLLOW(B) by FOLLOW(C) to give

FOLLOW(B) = { "." , "]" , "+" } $\cup$ { ")" } = { "." , "]" , "+" , ")" }

Since FIRST(B) $\cap$ FOLLOW(B) = Ø and FIRST(D) $\cap$ FOLLOW(D) = Ø, Rule 2 is satisfied for both the non-terminals that generate alternatives, both of which are nullable.

A C++ program for a parser follows. The terminals of the language are all single characters, so that we do not have to make any special arrangements for character handling (a simple `getchar` function call suffices) or for lexical analysis.

The reader should note that, because the grammar is strictly LL(1), the function that parses the non-terminal B may discriminate between the genuine followers of B (thereby effectively recognizing where the ε-production needs to be applied) and any spurious followers of B (which would signal a gross error in the parsing process).

```
// Simple Recursive Descent Parser for the language defined by the grammar
//         G = { N , T , S , P }
//         N = { A , B , C , D }
//         T = { "(" , ")" , "+" , "a" , "[" , "]" , "." }
//         S = A
//         P =
//             A  =  B "." .
//             B  =  "a"   |   "(" C ")"   |   "[" B "]"   |   .
//             C  =  B D .
//             D  =  { "+" B } .
// P.D. Terry,  Rhodes University, 1996

#include <stdio.h>
#include <stdlib.h>

char sym;  // Source token

void getsym(void)
{ sym = getchar(); }

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

void A(void);  // prototypes
void B(void);
void C(void);
void D(void);

void A(void)
//  A = B "." .
{ B(); accept('.', " Error - '.' expected"); }

void B(void)
//  B  =  "a"   |   "(" C ")"   |   "[" B "]"   |   .
{ switch (sym)
  { case 'a':
      getsym(); break;
    case '(':
      getsym(); C(); accept(')', " Error - ')' expected"); break;
    case '[':
      getsym(); B(); accept(']', " Error - ']' expected"); break;
    case ')':
    case ']':
    case '+':
    case '.':
      break;  // no action for followers of B
    default:
      printf("Unknown symbol\n"); exit(1);
  }
}

void C(void)
//  C = B D .
{ B(); D(); }

void D(void)
//  D  =  { "+" B } .
{ while (sym == ’+’) { getsym(); B(); } }

void main()
{ sym = getchar(); A();
  printf("Successful\n");
}
```

Some care may have to be taken with the relative ordering of the declaration of the functions, which in this example, and in general, are recursive in nature. (These problems do not occur if the functions have "prototypes" like those illustrated here.)

It should now be clear why this method of parsing is called *Recursive Descent*, and that such parsers are most easily implemented in languages which directly support recursive programming. Languages like Modula-2 and C++ are all very well suited to the task, although they each have their own particular strengths and weaknesses. For example, in Modula-2 one can take advantage of other organizational strategies, such as the use of nested procedures (which are not permitted in C or C++), and the very tight control offered by encapsulating a parser in a module with a very thin interface (only the routine for the goal symbol need be exported), while in C++ one can take advantage of OOP facilities (both to encapsulate the parser with a thin public interface, and to create hierarchies of specialized parser classes).

A little reflection shows that one can often combine routines (this corresponds to reducing the number of productions used to define the grammar). While this may produce a shorter program, precautions must be taken to ensure that the grammars, and any implicit semantic overtones, are truly equivalent. An equivalent grammar to the above one is

```
G = { N , T , S , P }
N = { A , B }
T = { "("  ,  ")"  ,  "+"  ,  "a"  ,  "["  ,  "]"  ,  "." }
S = A
P =
    A  →  B "."                                                (1)
    B  →  "a"   |   "("  B  { "+"  B }  ")"   |   "["  B  "]"   |  ε (2, 3, 4, 5)
```

leading to a parser

```
  // Simple Recursive Descent Parser for the same language
  // using an equivalent but different grammar
  // P.D. Terry,  Rhodes University, 1996

  #include <stdio.h>
  #include <stdlib.h>

  char sym;  // Source token

  void getsym(void)
  { sym = getchar(); }

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

  void B(void)
  //  B = "a"  |  "(" B { "+" B } ")"  |  "[" B "]"  |   .
  { switch (sym)
    { case ’a’:
        getsym(); break;
      case ’(’:
        getsym(); B(); while (sym == ’+’) { getsym(); B(); }
        accept(’)’, " Error - ’)’ expected"); break;
      case ’[’:
        getsym(); B(); accept(’]’, " Error - ’]’ expected"); break;
      case ’)’:
      case ’]’:
      case ’+’:
      case ’.’:
        break;    // no action for followers of B
      default:
        printf("Unknown symbol\n"); exit(1);
    }
  }

  void A(void)
  //  A = B "." .
  { B(); accept(’.’, " Error - ’.’ expected"); }

  void main(void)
  { sym = getchar(); A();
    printf("Successful\n");
  }
```

Although recursive descent parsers are eminently suitable for handling languages which satisfy the LL(1) conditions, they may often be used, perhaps with simple modifications, to handle languages which, strictly, do not satisfy these conditions. The classic example of a situation like this is provided by the `IF ... THEN ... ELSE` statement. Suppose we have a language in which statements are defined by

```
     Statement      =  IfStatement  |  OtherStatement .
     IfStatement    =  "IF"  Condition  "THEN"  Statement [ "ELSE" Statement  ] .
```

which, as we have already discussed, is actually ambiguous as it stands. A grammar defined like this is easily parsed deterministically with code like

```
  void Statement(void); // prototype

  void OtherStatement(void);
  // handle parsing of other statement - not necessary to show this here

  void IfStatement(void)
  { getsym(); Condition();
    accept(thensym, " Error - ’THEN’ expected");
    Statement();
    if (sym == elsesym) { getsym(); Statement(); }
  }

  void Statement(void)
  { switch(sym)
    { case ifsym : IfStatement(); break;
      default :    OtherStatement(); break;
    }
  }
```

The reader who cares to trace the function calls for an input sentence of the form

```
      IF  Condition  THEN  IF  Condition  THEN  OtherStatement  ELSE  OtherStatement
```

will note that this parser has the effect of recognizing and handling an `ELSE` clause as soon as it can - effectively forcing an *ad hoc* resolution of the ambiguity by coupling each `ELSE` to the closest unmatched `THEN`. Indeed, it would be far more difficult to design a parser that implemented the other possible disambiguating rule - no wonder that the semantics of this statement are those which correspond to the solution that becomes easy to parse!

As a further example of applying the LL(1) rules and considering the corresponding parsers, consider how one might try to describe variable designators of the kind found in many languages to denote elements of record structures and arrays, possibly in combination, for example `A[B.C.D]`. One set of productions that describes some (although by no means all) of these constructions might appear to be:

```
     Designator     =  identifier  Qualifier .                (1)
     Qualifier      =  Subscript | FieldSpecifier .           (2, 3)
     Subscript      =   "["  Designator  "]" | ε  .           (4, 5)
     FieldSpecifier =   "."  Designator  | ε .                (6, 7)
```

This grammar is not LL(1), although it may be at first difficult to see this. The production for *Qualifier* has alternatives, and to check Rule 1 for productions 2 and 3 we need to consider FIRST($Qualifier_1$) and FIRST($Qualifier_2$). At first it appears obvious that

FIRST($Qualifier_1$) = FIRST(*Subscript*) = { "[" }

but we must be more careful. *Subscript* is nullable, so to find FIRST($Qualifier_1$) we must augment this singleton set with FOLLOW(*Subscript*). The calculation of this requires that we find productions with *Subscript* on the right side - there is only one of these, production (2). From this we see that FOLLOW(*Subscript*) = FOLLOW(*Qualifier*), which from production (1) is FOLLOW(*Designator*). To determine FOLLOW(*Designator*) we must examine productions (4) and (6). Only the first of these contributes anything, namely { "]" }. Thus we eventually conclude that

FIRST($Qualifier_1$) = { "[", "]" }.

Similarly, the obvious conclusion that

FIRST($Qualifier_2$) = FIRST(*FieldSpecifier*) = { "." }

is also too naïve (since *FieldSpecifier* is also nullable); a calculation on the same lines leads to the result that

FIRST($Qualifier_2$) = { "." , "]" }

Rule 1 is thus broken; the grammar is not LL(1).

The reader will complain that this is ridiculous. Indeed, rewriting the grammar in the form

```
     Designator     =  identifier Qualifier .               (1)
     Qualifier      =  Subscript  |  FieldSpecifier | ε .   (2, 3, 4)
     Subscript      =   "[" Designator "]" .                (5)
     FieldSpecifier =   "." Designator .                    (6)
```

leads to no such transgressions of Rule 1, or, indeed of Rule 2 (readers should verify this to their own satisfaction). Once again, a recursive descent parser is easily written:

```
  void Designator(void); // prototype

  void Subscript(void)
  { getsym(); Designator(); accept(rbracket, " Error - ’]’ expected"); }

  void FieldSpecifier(void)
  { getsym(); Designator(); }

  void Qualifier(void)
  { switch(sym)
    { case lbracket : Subscript(); break;
      case period   : FieldSpecifier(); break;
      case rbracket : break; // FOLLOW(Qualifier) is empty
      default :       printf("Unknown symbol\n"); exit(1);
    }
  }

  void Designator(void)
  { accept(identifier, " Error - identifier expected");
    Qualifier();
  }
```

In this case there is an easy, if not even obvious way to repair the grammar, and to develop the parser. However, a more realistic version of this problem leads to a situation that cannot as easily be resolved. In Modula-2 a *Designator* is better described by the productions

```
        Designator           =  QualifiedIdentifier  {  Selector  } .
        QualifiedIdentifier  =  identifier  { "."  identifier } .
        Selector             =  "."  identifier  |  "["  Expression  "]"  | "^" .
```

It is left as an exercise to demonstrate that this is not LL(1). It is left as a harder exercise to come to a formal conclusion that one cannot find an LL(1) grammar that describes *Designator* unambiguously. The underlying reason is that "." is used in one context to separate a module identifier from the identifier that it qualifies (as in `Scanner.SYM`) and in a different context to separate a record identifier from a field identifier (as in `SYM.Name`). When these are combined (as in `Scanner.SYM.Name`) the problem becomes more obvious.

The reader may have wondered at the fact that the parsing methods we have advocated all look "ahead", and never seem to make use of what has already been achieved, that is, of information which has become embedded in the previous history of the parse. All LL(1) grammars are, of course, context-free, yet we pointed out in Chapter 8 that there are features of programming languages which cannot be specified in a context-free grammar (such as the requirement that variables must be declared before use, and that expressions may only be formed when terms and factors are of the correct types). In practice, of course, a parser is usually combined with a semantic analyser; in a sense some of the past history of the parse is recorded in such devices as symbol tables which the semantic analysis needs to maintain. The example given here is not as serious as it may at first appear. By making recourse to the symbol table, a Modula-2 compiler will be able to resolve the potential ambiguity in a static semantic way (rather than in an *ad hoc* syntactic way as is done for the "dangling else" situation).

Exercises

10.1 Check the LL(1) conditions for the equivalent grammar used in the second of the programs above.

10.2 Rework Exercise 10.1 by checking the director sets for the productions.

10.3 Suppose we wished the language in the previous example to be such that spaces in the input file were irrelevant. How could this be done?

10.4 In section 8.4 an unambiguous set of productions was given for the `IF ... THEN ... ELSE` statement. Is the corresponding grammar LL(1)? Whatever the outcome, can you construct a recursive descent parser to handle such a formulation of the grammar?

10.3 Syntax error detection and recovery

Up to this point our parsers have been content merely to stop when a syntactic error is detected. In the case of a real compiler this is probably unacceptable. However, if we modify the parser as given above so as simply not to stop after detecting an error, the result is likely to be chaotic. The analysis process will quickly get out of step with the sequence of symbols being scanned, and in all likelihood will then report a plethora of spurious errors.

One useful feature of the compilation technique we are using is that the parser can detect a syntactically incorrect structure after being presented with its first "unexpected" terminal. This will not necessarily be at the point where the error really occurred. For example, in parsing the sequence

```
BEGIN IF A > 6 DO B := 2; C := 5 END END
```

we could hope for a sensible error message when `DO` is found where `THEN` is expected. Even if parsing does not get out of step, we would get a less helpful message when the second `END` is found - the compiler can have little idea where the missing `BEGIN` should have been.

A production quality compiler should aim to issue appropriate diagnostic messages for all the "genuine" errors, and for as few "spurious" errors as possible. This is only possible if it can make some likely assumption about the nature of each error and the probable intention of the author, or if it skips over some part of the malformed text, or both. Various approaches may be made to handling the problem. Some compilers go so far as to try to correct the error, and continue to produce object code for the program. Error correction is a little dangerous, except in some trivial cases, and we shall discuss it no further here. Many systems confine themselves to attempting **error recovery**, which is the term used to describe the process of simply trying to get the parser back into step with the source code presented to it. The art of doing this for hand-crafted compilers is rather intricate, and relies on a mixture of fairly well defined methods and intuitive experience, both with the language being compiled, and with the class of user of the same.

Since recursive descent parsers are constructed as a set of routines, each of which tackles a sub-goal on behalf of its caller, a fairly obvious place to try to regain lost synchronization is at the entry to and exit from these routines, where the effects of getting out of step can be confined to examining a small range of known FIRST and FOLLOW symbols. To enforce synchronization at the entry to the routine for a non-terminal *S* we might try to employ a strategy like

```
IF Sym ∉ FIRST(S) THEN
  ReportError; SkipTo(FIRST(S))
END
```

where *SkipTo* is an operation which simply calls on the scanner until it returns a value for *Sym* that is a member of FIRST(*S*). Unfortunately this is not quite adequate - if the leading terminal has been omitted we might then skip over symbols that should be processed later, by the routine which called *S*.

At the exit from *S*, we have postulated that *Sym* should be a member of FOLLOW(*S*). This set may not be known to *S*, but it should be known to the routine which calls *S*, so that it may conveniently be passed to *S* as a parameter. This suggests that we might employ a strategy like

```
IF Sym ∉ FOLLOW(S) THEN
  ReportError; SkipTo(FOLLOW(S))
END
```

The use of FOLLOW(*S*) also allows us to avoid the danger mentioned earlier of skipping too far at routine entry, by employing a strategy like

```
IF Sym ∉ FIRST(S) THEN
  ReportError; SkipTo(FIRST(S) | FOLLOW(S))
END;
IF SYM.Sym ∈ FIRST(S) THEN
```

```
        Parse(S);
        IF SYM.Sym ∉ FOLLOW(S) THEN
          ReportError; SkipTo(FOLLOW(S))
        END
      END
```

Although the FOLLOW set for a non-terminal is quite easy to determine, the legitimate follower may itself have been omitted, and this may lead to too many symbols being skipped at routine exit. To prevent this, a parser using this approach usually passes to each sub-parser a *Followers* parameter, which is constructed so as to include

- the minimally correct set FOLLOW(*S*), augmented by
- symbols that have already been passed as *Followers* to the calling routine (that is, later followers), and also
- so-called **beacon symbols**, which are on no account to be passed over, even though their presence would be quite out of context. In this way the parser can often avoid skipping large sections of possibly important code.

On return from sub-parser *S* we can then be fairly certain that *Sym* contains a terminal which was either expected (if it is in FOLLOW(*S*)), or can be used to regain synchronization (if it is one of the beacons, or is in FOLLOW(*Caller(S)*)). The caller may need to make a further test to see which of these conditions has arisen.

In languages like Modula-2 and Pascal, where set operations are directly supported, implementing this scheme is straightforward. C++ does not have "built-in" set types. Their implementation in terms of a template class is easily achieved, and operator overloading can be put to good effect. An interface to such a class, suited to our applications in this text, can be defined as follows

```
  template <int maxElem>
  class Set {                         // { 0 .. maxElem }
    public:
      Set();                          // Construct { }
      Set(int e1);                    // Construct { e1 }
      Set(int e1, int e2);            // Construct { e1, e2 }
      Set(int e1, int e2, int e3);    // Construct { e1, e2, e3 }
      Set(int n, int e[]);            // Construct { e[0] .. e[n-1] }
      void incl(int e);               // Include e
      void excl(int e);               // Exclude e
      int memb(int e);                // Test membership for e
      Set operator + (const Set &s)   // Union with s               (OR)
      Set operator * (const Set &s)   // Intersection with s        (AND)
      Set operator - (const Set &s)   // Difference with s
      Set operator / (const Set &s)   // Symmetric difference with s (XOR)
    private:
      unsigned char bits[(maxElem + 8) / 8];
      int length;
      int wrd(int i);
      int bitmask(int i);
      void clear();
  };
```

The implementation is realized by treating a large set as an array of small bitsets; full details of this can be found in the source code supplied on the accompanying diskette and in Appendix B.

Syntax error recovery is then conveniently implemented by defining functions on the lines of

```
  typedef Set<lastDefinedSym> symset;

  void accept(symtypes expected, int errorcode)
  { if (Sym == expected) getsym(); else reporterror(errorcode); }

  void test(symset allowed, symset beacons, int errorcode)
  { if (allowed.memb(Sym)) return;
```

```
    reporterror(errorcode);
    symset stopset = allowed + beacons;
    while (!stopset.memb(Sym)) getsym();
  }
```

where we note that the amended `accept` routine does not try to regain synchronization in any way. The way in which these functions could be used is exemplified in a routine for handling variable declarations for Clang:

```
  void VarDeclarations(symset followers);
  // VarDeclarations = "VAR" OneVar { "," OneVar } ";" .
  { getsym();                               // accept "var"
    test(symset(identifier), followers, 6); // FIRST(OneVar)
    if (Sym == identifier)                  // we are in step
    { OneVar(symset(comma, semicolon) + followers);
      while (Sym == comma)                  // more variables follow
      { getsym(); OneVar(symset(comma, semicolon) + followers); }
      accept(semicolon, 2);
      test(followers, symset(), 34);
    }
  }
```

The `followers` passed to `VarDeclarations` should include as "beacons" the elements of FIRST(*Statement*) - symbols which could start a *Statement* (in case `BEGIN` was omitted) - and the symbol which could follow a *Block* (period, and end-of-file). Hence, calling `VarDeclarations` might be done from within `Block` on the lines of

```
  if (Sym == varsym)
    VarDeclarations(FirstBlock + FirstStatement + followers);
```

Too rigorous an adoption of this scheme will result in some spurious errors, as well as an efficiency loss resulting from all the set constructions that are needed. In hand-crafted parsers the ideas are often adapted somewhat. As mentioned earlier, one gains from experience when dealing with learners, and some concession to likely mistakes is, perhaps, a good thing. For example, beginners are likely to confuse operators like "`:=`", "`=`" and "`==`", and also `THEN` and `DO` after `IF`, and these may call for special treatment. As an example of such an adaptation, consider the following variation on the above code, where the parser will, in effect, handle variable declarations in which the separating commas have been omitted. This is strategically a good idea - variable declarations that are not properly processed are likely to lead to severe difficulties in handling later stages of a compilation.

```
  void VarDeclarations(symset followers);
  // VarDeclarations = "VAR" OneVar { "," OneVar } ";" .
  { getsym()                                    // accept "var"
    test(symset(identifier), followers, 6);     // FIRST(OneVar)
    if (Sym == identifier)                      // we are in step
    { OneVar(symset(comma, semicolon) + followers);
      while (Sym == comma || Sym == identifier) // only comma is legal
      { accept(comma), 31); OneVar(symset(comma, semicolon) + followers); }
      accept(semicolon, 2);
      test(followers, symset(), 34);
    }
  }
```

Clearly it is impossible to recover from all possible contortions of code, but one should guard against the cardinal sins of not reporting errors when they are present, or of collapsing completely when trying to recover from an error, either by giving up prematurely, or by getting the parser caught in an infinite loop reporting the same error.

Exercises

10.5 Extend the parsers developed in section 10.2 to incorporate error recovery.

10.6 Investigate the efficacy of the scheme suggested for parsing variable declarations, by tracing the way in which parsing would proceed for incorrect source code such as the following:

```
          VAR A, B C , , D; E, F;
```

Further reading

Error recovery is an extensive topic, and we shall have more to say on it in later chapters. Good treatments of the material of this section may be found in the books by Welsh and McKeag (1980), Wirth (1976b), Gough (1988) and Elder (1994). A much higher level treatment is given by Backhouse (1979), while a rather simplified version is given by Brinch Hansen (1983, 1985). Papers by Pemberton (1980) and by Topor (1982), Stirling (1985) and Grosch (1990b) are also worth exploring, as is the bibliographical review article by van den Bosch (1992).

10.4 Construction of simple scanners

In a sense, a scanner or lexical analyser may be thought of as just another syntax analyser. It handles a grammar with productions relating non-terminals such as *identifier*, *number* and *Relop* to terminals supplied, in effect, as single characters of the source text. When used in conjunction with a higher level parser a subtle shift in emphasis comes about: there is, in effect, no special goal symbol. Each invocation of the scanner is very much bottom-up rather than top-down; its task ends when it has reduced a string of characters to a token, without preconceived ideas of what that should be. These tokens or non-terminals are then regarded as terminals by the higher level recursive descent parser that analyses the phrase structure of *Block*, *Statement*, *Expression* and so on.

There are at least five reasons for wishing to decouple the scanner from the main parser:

- The productions involved are usually very simple. Very often they amount to regular expressions, and then a scanner may be programmed without recourse to methods like recursive descent.

- A symbol like an *identifier* is lexically equivalent to a "reserved word"; the distinction may sensibly be made as soon as the basic token has been synthesized.

- The character set may vary from machine to machine, a variation easily isolated in this phase.

- The semantic analysis of a numeric literal constant (deriving the internal representation of its value from the characters) is easily performed in parallel with lexical analysis.

- The scanner can be made responsible for screening out superfluous separators, like blanks and comments, which are rarely of interest in the formulation of the higher level grammar.

In common with the parsing strategy suggested earlier, development of the routine or function responsible for token recognition

- may assume that it is always called *after* some (globally accessible) variable *CH* has been found to contain the next character to be handled in the source

- will then read a complete sequence of characters that form a recognizable token
- will relinquish scanning after leaving *CH* with the first character that does not form part of this token (so as to satisfy the precondition for the next invocation of the scanner).

A scanner is necessarily a top-down parser, and for ease of implementation it is desirable that the productions defining the token grammar also obey the LL(1) rules. However, checking these is much simpler, as token grammars are almost invariably regular, and do not display self-embedding (and thus can be almost always easily be transformed into LL(1) grammars).

There are two main strategies that are employed in scanner construction:

- Rather than being decomposed into a set of recursive routines, simple scanners are often written in an *ad hoc* manner, controlled by a large `CASE` or `switch` statement, since the essential task is one of choosing between a number of tokens, which are sometimes distinguishable on the basis of their initial characters.
- Alternatively, since they usually have to read a number of characters, scanners are often written in the form of a **finite state automaton** (FSA) controlled by a loop, on each iteration of which a single character is absorbed, the machine moving between a number of "states", determined by the character just read. This approach has the advantage that the construction can be formalized in terms of an extensively developed automata theory, leading to algorithms from which scanner generators can be constructed automatically.

A proper discussion of automata theory is beyond the scope of this text, but in the next section we shall demonstrate both approaches to scanner construction by means of some case studies.

10.5 Case studies

To consider a concrete example, suppose that we wish to extend the grammar used for earlier demonstrations into one described in Cocol as follows:

```
COMPILER A
  CHARACTERS
    digit  = "0123456789" .
    letter = "abcdefgefghijklmnopqrstuvwxyz" .
  TOKENS
    number      = digit { digit } .
    identifier  = "a" { letter } .
  PRODUCTIONS
    A = B "." .
    B = identifier | number | "(" C ")" | "(." B ".)" | .
    C = B D .
    D = { "+" B } .
END A.
```

Combinations like `(.` and `.)` are sometimes used to represent the brackets [and] on machines with limited character sets. The tokens we need to be able to recognize are definable by an enumeration:

```
TOKENS = { number, lbrack, lparen, rbrack, rparen, plus, period, identifier }
```

It should be easy to see that these tokens are not uniquely distinguishable on the basis of their leading characters, but it is not difficult to write a set of productions for the token grammar that obeys the LL(1) rules:

```
     token  =    digit { digit }   (* number *)
              | "(" [ "." ]        (* lparen, lbrack *)
              | "." [ ")" ]        (* period, rbrack *)
              | ")"                (* rparen *)
              | "+"                (* plus *)
              | "a" { letter }     (* identifier *) .
```

from which an *ad hoc* scanner algorithm follows very easily on the lines of

```
  TOKENS FUNCTION GetSym;
  (* Precondition:  CH is already available
     Postcondition: CH is left as the character following token *)
    BEGIN
      IgnoreCommentsAndSeparators;
      CASE CH OF
        'a' :
          REPEAT Get(CH) UNTIL CH ∉ {'a' .. 'z'};
          RETURN identifier;
        '0' .. '9' :
          REPEAT Get(CH) UNTIL CH ∉ {'0' .. '9'};
          RETURN number;
        '(' :
          Get(CH);
          IF CH = '.'
            THEN Get(CH); RETURN lbrack
            ELSE RETURN lparen
          END;
        '.' :
          Get(CH);
          IF CH = ')'
            THEN Get(CH); RETURN rbrack
            ELSE RETURN period
          END;
        '+' :
          Get(CH); RETURN plus
        ')' :
          Get(CH); RETURN rparen
        ELSE
          Get(CH); RETURN unknown
      END
    END
```

A characteristic feature of this algorithm - and of most scanners constructed in this way - is that they are governed by a selection statement, within the alternatives of which one frequently finds loops that consume sequences of characters. To illustrate the FSA approach - in which the algorithm is inverted to be governed by a single loop - let us write our grammar in a slightly different way, in which the comments have been placed to reflect the state that a scanner can be thought to possess at the point where a character has just been read.

```
     token  =   (* unknown *) digit (* number *) { digit (* number *) }
              | (* unknown *) "(" (* lparen *) [ "." (* lbrack *) ]
              | (* unknown *) "." (* period *) [ ")" (* rbrack *) ]
              | (* unknown *) ")" (* rparen *)
              | (* unknown *) "+" (* plus *)
              | (* unknown *) "a" (* identifier *) { letter (* identifier *)  }
```

Another way of representing this information is in terms of a transition diagram like that shown in Figure 10.1, where, as is more usual, the states have been labelled with small integers, and where the arcs are labelled with the characters whose recognition causes the automaton to move from one state to another.

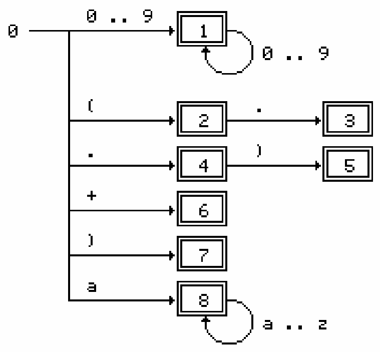


Figure 10.1 A transition diagram for a simple FSA

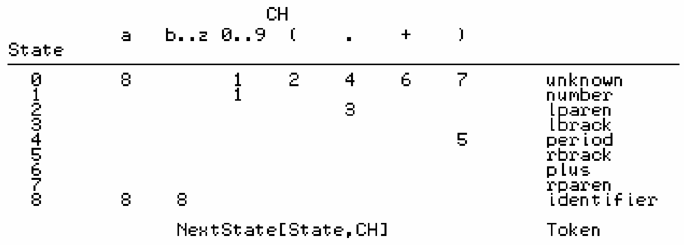
There are many ways of developing a scanner from these ideas. One approach, using a table-driven scanner, is suggested below. To the set of states suggested by the diagram we add one more, denoted by `finished`, to allow the postcondition to be easily realized.

```
TOKENS FUNCTION GetSym;
(* Preconditions: CH is already available,
                  NextState, Token mappings defined
   Postcondition: CH is left as the character following token *)
  BEGIN
    State := 0;
    WHILE state ≠ finished DO
      LastState := State;
      State := NextState[State, CH];
      Get(CH);
    END;
    RETURN Token[LastState];
  END
```

Here we have made use of various mapping functions, expressed in the form of arrays:

```
Token[s]          is defined to be the token recognized when the machine has reached state s
NextState[s, x]   indicates the transition that must be taken when the machine
                  is currently in state s, and has just recognized character x.
```

For our example, the arrays `Token` and `NextState` would be set up as in the table below. For clarity, the many transitions to the `finished` state have been left blank.



	CH							
State	a	b..z	0..9	(	.	+	)	
0	8		1	2	4	6	7	unknown
1			1					number
2					3			lparen
3								lbrack
4							5	period
5								rbrack
6								plus
7								rparen
8	8	8						identifier
		NextState[State,CH]						Token

A table-driven algorithm is efficient in time, and effectively independent of the token grammar, and thus highly suited to automated construction. However it should not take much imagination to see that it is very hungry and wasteful of storage. A complex scanner might run to dozens of states, and many machines use an ASCII character set, with 256 values. For each character a column would be needed in the matrix, yet most of the entries (as in the example above) would be identical. And although we may have given the impression that this method will always succeed, this is not necessarily so. If the underlying token grammar were not LL(1) it might not be possible to define an unambiguous transition matrix - some entries might appear to require two or more values. In this situation we speak of requiring a **non-deterministic finite automaton** (NDFA) as opposed to the **deterministic finite automaton** (DFA) that we have been considering up until now.

Small wonder that considerable research has been invested in developing variations on this theme. The code below shows one possible variation, for our specimen grammar, in the form of a complete C++ function. In this case it is necessary to have but one static array (denoted by `state0`), initialized so as to map each possible character into a single state.

```
TOKENS getsym(void)
// Preconditions: First character ch has already been read
//                state0[] has been initialized
{ IgnoreCommentsAndSeparators();
  int state = state0[ch];
  while (1)
  { ch = getchar();
    switch (state)
    { case 1 :
        if (!isdigit(ch)) return number;
        break; // state unchanged
      case 2 :
        if (ch = '.') state = 3; else return lparen;
        break;
      case 3 :
        return lbrack;
      case 4 :
        if (ch = ')') state = 5; else return period;
        break;
      case 5 :
        return rbrack;
      case 6 :
        return plus;
      case 7 :
        return rparen;
      case 8 :
        if (!isletter(ch)) return identifier;
        break; // state unchanged
      default :
        return unknown;
    }
  }
}
```

Our scanner algorithms are as yet immature. Earlier we claimed that scanners often incorporated such tasks as the recognition of keywords (which usually resemble identifiers), the evaluation of constant literals, and so on. There are various ways in which these results can be achieved, and in later case studies we shall demonstrate several of them. In the case of the state machine it may be easiest to build up a string that stores all the characters scanned, a task that requires minimal perturbation to the algorithms just discussed. Subsequent processing of this **lexeme** can then be done in an application-specific way. For example, searching for a string in a table of keywords will easily distinguish between keywords and identifiers.

Exercises

10.7 Our scanner algorithms have all had the property that they consume at least one character. Suppose that the initial character could not form part of a token (that is, did not belong to the vocabulary of the language). Would it not be better *not* to consume it?

10.8 Similarly, we have made no provision for the very real possibility that the scanner may not find any characters when it tries to read them, as would happen if it tried to read past the end of the source. Modify the algorithm so that the scanner can recognize this condition, and return a distinctive `eof` token when necessary. Take care to get this correct: the solution may not be as obvious as it at first appears.

10.9 Suppose that our example language was extended to recognize `abs` as a keyword. We could accomplish this by extending the last part of the transition diagram given earlier to that shown in Figure 10.2.

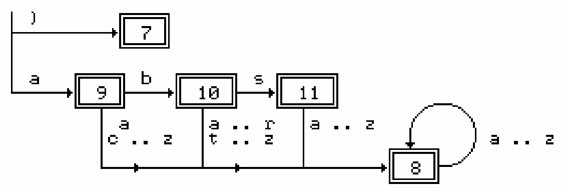


Figure 10.2 Part of a transition diagram for an extended FSA

What corresponding changes would need to be made to the tables needed to drive the parser? In principle one could, of course, handle any number of keywords in a similar fashion. The number of states would grow very rapidly to the stage where manual construction of the table would become very tedious and error-prone.

10.10 How could the C++ code given earlier be modified to handle the extension suggested in Exercise 10.9?

10.11 Suppose our scanner was also required to recognize quoted strings, subject to the common restriction that these should not be allowed to carry across line breaks in the source. How could this be handled? Consider both the extensions that would be needed to the *ad hoc* scanner given earlier, and also to the table driven scanner.

Further reading

Automata theory and the construction of finite state automata are discussed in most texts on compiler construction. A particularly thorough treatment is is to be found in the book by Gough (1988); those by Holub (1990), Watson (1989) and Fischer and LeBlanc (1988, 1991) are also highly readable.

Table driven parsers may also be used to analyse the higher level phrase structure for languages which satisfy the LL(k) conditions. Here, as in the FSA discussed above, and as in the LR parser to be discussed briefly later, the parser itself becomes essentially language independent. The automata have to be more sophisticated, of course. They are known as "push down automata", since they generally need to maintain a stack, so as to be able to handle the self-embedding found in the productions of the grammar. We shall not attempt to discuss such parsers here, but refer the interested reader to the books just mentioned, which all treat the subject thoroughly.

10.6 LR parsing

Although space does not permit of a full description, no modern text on translators would be complete without some mention of so-called **LR(k)** parsing. The terminology here comes from the notion that we scan the input string from **L**eft to right (the L), applying reductions so as to yield a **R**ightmost parse (the R), by looking as far ahead as the next *k* terminals to help decide which production to apply. (In practice *k* is never more than 1, and may be zero.)

The technique is **bottom-up** rather than **top-down**. Starting from the input sentence, and making **reductions**, we aim to end up with the goal symbol. The reduction of a sentential form is achieved by substituting the left side of a production for a string (appearing in the sentential form) which matches the right side, rather than by substituting the right side of a production whose left side appears as a non-terminal in the sentential form.

A bottom-up parsing algorithm might employ a **parse stack**, which contains part of a possible sentential form of terminals and/or non terminals. As we read each terminal from the input string we push it onto the parse stack, and then examine the top elements of this to see whether we can make a reduction. Some terminals may remain on the parse stack quite a long time before they are finally pushed off and discarded. (By way of contrast, a top- down parser can discard the terminals immediately after reading them. Furthermore, a recursive descent parser stores the non-terminal components of the partial sentential form only implicitly, as a chain of as yet uncompleted calls to the routines which handle each non-terminal.)

Perhaps an example will help to make this clearer. Suppose we have a highly simplified (non-LL(1)) grammar for expressions, defined by

```
Goal        =  Expression "." .                    (1)
Expression  =  Expression "-" Term  |  Term .      (2, 3)
Term        =  "a"                                 (4)
```

and are asked to parse the string "*a - a - a* ." .

The sequence of events could be summarized

```
Step    Action     Using production      Stack

  1      read   a                        a
  2      reduce           4              Term
  3      reduce           3              Expression
  4      read   -                        Expression -
  5      read   a                        Expression - a
  6      reduce           4              Expression - Term
  7      reduce           2              Expression
  8      read   -                        Expression -
  9      read   a                        Expression - a
 10      reduce           4              Expression - Term
 11      reduce           2              Expression
 12      read   .                        Expression .
 13      reduce           1              Goal
```

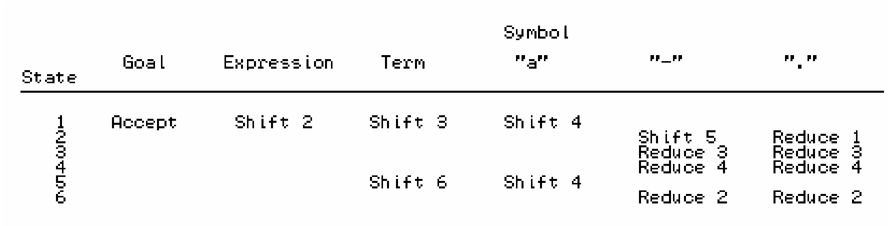
We have reached *Goal* and can conclude that the sentence is valid.

The careful reader may declare that we have cheated! Why did we not use the production *Goal = Expression* when we had reduced the string "*a*" to *Expression* after step 3? To apply a reduction it is, of course necessary that the right side of a production be currently on the parse stack, but this in itself is insufficient. Faced with a choice of right sides which match the top elements on the parse stack, a practical parser will have to employ some strategy, perhaps of looking ahead in the input string, to decide which to apply.

Such parsers are invariably table driven, with the particular strategy at any stage being determined by looking up an entry in a rectangular matrix indexed by two variables, one representing the current "state" of the parse (the position the parser has reached within the productions of the grammar) and the other representing the current "input symbol" (which is one of the terminal or non-terminals of the grammar). The entries in the table specify whether the parser is to *accept* the input string as correct, *reject* as incorrect, *shift* to another state, or *reduce* by applying a particular production. Rather than stack the symbols of the grammar, as was implied by the trace above, the parsing algorithm pushes or pops elements representing states of the parse - a *shift* operation pushing the newly reached state onto the stack, and a *reduce* operation popping as many elements as there are symbols on the right side of the production being applied. The algorithm can be expressed:

```
BEGIN
  GetSYM(InputSymbol); (* first Sym in sentence *)
  State := 1; Push(State); Parsing := TRUE;
  REPEAT
    Entry := Table[State, InputSymbol];
    CASE Entry.Action OF
      shift:
        State := Entry.NextState; Push(State);
        IF IsTerminal(InputSymbol) THEN
          GetSYM(InputSymbol) (* accept *)
        END
      reduce:
        FOR I := 1 TO Length(Rule[Entry].RightSide) DO Pop END;
        State := Top(Stack);
        InputSymbol := Rule[Entry].LeftSide;
      reject:
        Report(Failure); Parsing := FALSE
      accept:
        Report(Success); Parsing := FALSE
    END
  UNTIL NOT Parsing
END
```

Although the algorithm itself is very simple, construction of the parsing table is considerably more difficult. Here we shall not go into how this is done, but simply note that for the simple example given above the parsing table might appear as follows (we have left the `reject` entries blank for clarity):



	Symbol					
State	Goal	Expression	Term	"a"	"-"	"."
1	Accept	Shift 2	Shift 3	Shift 4		
2					Shift 5	Reduce 1
3					Reduce 3	Reduce 3
4					Reduce 4	Reduce 4
5			Shift 6	Shift 4		
6					Reduce 2	Reduce 2

Given this table, a parse of the string "*a* - *a* - *a* ." would proceed as follows. Notice that the period has been introduced merely to make recognizing the end of the string somewhat easier.

```
State      Symbol    Stack      Action

  1            a     1          Shift to state 4, accept  a
  4            -     1 4        Reduce by (4) Term = a
  1         Term     1          Shift to state 3
  3            -     1 3        Reduce by (3) Expression = Term
  1     Expression   1          Shift to state 2
  2            -     1 2        Shift to state 5, accept  -
  5            a     1 2 5      Shift to state 4, accept  a
  4            -     1 2 5 4    Reduce by (4) Term = a
  5         Term     1 2 5      Shift to state 6
  6            -     1 2 5 6    Reduce by (2) Expression = Expression - Term
  1     Expression   1          Shift to state 2
  2            -     1 2        Shift to state 5, accept  -
  5            a     1 2 5      Shift to state 4, accept  a
  4            .     1 2 5 4    Reduce by (4) Term = a
  5         Term     1 2 5      Shift to state 6
  6            .     1 2 5 6    Reduce by (2) Expression = Expression - Term
  1     Expression   1          Shift to state 2
  2            .     1 2        Reduce by (1) Goal = Expression
  1         Goal     1          Accept as completed
```

The reader will have noticed that the parsing table for the toy example is very sparsely filled. The use of fixed size arrays for this, for the production lists, or for the parse stack is clearly non-optimal.

One of the great problems in using the LR method in real applications is the amount of storage which these structures require, and considerable research has been done so as to minimize this.

As in the case of LL(1) parsers it is necessary to ensure that productions are of the correct form before we can write a deterministic parser using such algorithms. Technically one has to avoid what are known as "shift/reduce conflicts", or ambiguities in the action that is needed at each entry in the parse table. In practice the difficult task of producing the parse table for a large grammar with many productions and many states, and of checking for such conflicts, is invariably left to parser generator programs, of which the best known is probably **yacc** (Johnson, 1975). A discussion of yacc, and of its underlying algorithms for LR(k) parsing is, regrettably, beyond the scope of this book.

It turns out that LR(k) parsing is much more powerful than LL(k) parsing. Before an LL(1) parser can be written it may be necessary to transform an intuitively obvious grammar into one for which the LL(1) conditions are met, and this sometimes leads to grammars that look unnaturally complicated. Fewer transformations of this sort are needed for LR(k) parsers - for example, left recursion does not present a problem, as can be seen from the simple example discussed earlier. On the other hand, when a parser is extended to handle constraint analysis and code generation, an LL(1)-based grammar presents fewer problems than does an LR(1)-based one, where the extensions are sometimes found to introduce violations of the LR(k) rules, resulting in the need to transform the grammar anyway.

The rest of our treatment will all be presented in terms of the recursive descent technique, which has the great advantage that it is intuitively easy to understand, is easy to incorporate into hand-crafted compilers, and leads to small and efficient compilers.

Further reading

On the accompanying diskette will be found source code for a demonstration program that implements the above algorithm in the case where the symbols can be represented by single characters. The reader may like to experiment with this, but be warned that the simplicity of the parsing algorithm is rather overwhelmed by all the code required to read in the productions and the elements of the parsing tables.

In the original explanation of the method we demonstrated the use of a stack which contained symbols; in the later discussion we commented that the algorithm could merely stack states. However, for demonstration purposes it is convenient to show both these structures, and so in the program we have made use of a **variant record** or **union** for handling the parse stack, so as to accommodate elements which represent symbols as well as ones which represent parse states. An alternative method would be to use two separate stacks, as is outlined by Hunter (1981).

Good discussions of LR(k) parsing and of its variations such as SLR (Simple LR) and LALR (Look Ahead LR) appear in many of the sources mentioned earlier in this chapter. (These variations aim to reduce the size of the parsing tables, at the cost of being able to handle slightly less general grammars.) The books by Gough (1988) and by Fischer and LeBlanc (1988, 1991) have useful comparisons of the relative merits of LL(k) and LR(k) parsing techniques.

10.7 Automated construction of scanners and parsers

Recursive descent parsers are easily written, provided a satisfactory grammar can be found. Since the code tends to match the grammar very closely, they may be developed manually quickly and accurately. Similarly, for many applications the manual construction of scanners using the techniques demonstrated in the last section turns out to be straightforward.

However, as with so many "real" programming projects, when one comes to develop a large compiler, the complexities of scale raise their ugly heads. An obvious course of action is to interleave the parser with the semantic analysis and code generation phases. Even when modular techniques are used - such as writing the system to encapsulate the phases in well-defined separate classes or modules - real compilers all too easily become difficult to understand, or to maintain (especially in a "portable" form).

For this reason, among others, increasing use is now made of **parser generators** and **scanner generators** - programs that take for their input a system of productions and create the corresponding parsers and scanners automatically. We have already made frequent reference to one such tool, Coco/R (Mössenböck, 1990a), which exists in a number of versions that can generate systems, embodying recursive descent parsers, in either C, C++, Java, Pascal, Modula-2 or Oberon. We shall make considerable use of this tool in the remainder of this text.

Elementary use of a tool like Coco/R is deceptively easy. The user prepares a Cocol grammar description of the language for which the scanner and parser are required. This grammar description forms the most obvious part of the input to Coco/R. Other parts come in the form of so-called **frame files** that give the skeleton of the common code that is to be generated for any scanner, parser or driver program. Such frame files are highly generic, and a user can often employ a standard set of frame files for a wide number of applications.

The tool is typically invoked with a command like

```
cocor -c -l -f grammarName
```

where `grammarName` is the name of the file containing the Cocol description. The arguments prefixed with hyphens are used in the usual way to select various options, such as the generation of a driver module (`-c`), the production of a detailed listing (`-l`), a summary of the FIRST and FOLLOW sets for each non-terminal (`-f`), and so on.

After the grammar has been analysed and tested for self-consistency and correctness (ensuring, for example, that all non-terminals have been defined, that there are no circular derivations, and that all tokens can be distinguished), a recursive descent parser and complementary FSA scanner are generated in the form of highly readable source code. The exact form of this depends on the version of Coco/R that is being used. The Modula-2 version, for example, generates `DEFINITION MODULES` specifying the interfaces, along with `IMPLEMENTATION MODULES` detailing the implementation of each component, while the C++ version produces separate header and implementation files that define a hierarchical set of classes.

Of course, such tools can only be successfully used if the user understands the premises on which they are based (for example, Coco/R can guarantee real success only if it is presented with an underlying grammar that is LL(1)). Their full power comes about when the grammar descriptions are extended further in ways to be described in the next chapter, allowing for the construction of complete compilers incorporating constraint analysis, error recovery, and code generation, and so we delay further discussion for the present.

Exercises

10.12 On the accompanying diskette will be found implementations of Coco/R for C/C++, Turbo Pascal, and Modula-2. Submit the sample grammar given earlier to the version of your choice, and compare the code generated with that produced by hand in earlier sections.

10.13 Exercises 5.11 through 5.21 required you to produce Cocol descriptions of a number of grammars. Submit these to Coco/R and explore its capabilities for testing grammars, listing FIRST and FOLLOW sets, and constructing scanners and parsers.

Further reading

Probably the most famous parser generator is **yacc**, originally developed by Johnson (1975). There are several excellent texts that describe the use of **yacc** and its associated scanner generator **lex** (Lesk, 1975), for example those by Aho, Sethi and Ullman (1986), Bennett (1990), Levine, Mason and Brown (1992), and Schreiner and Friedman (1985).

The books by Fischer and LeBlanc (1988) and Alblas and Nymeyer (1996) describe other generators written in Pascal and in C respectively.

There are now a great many compiler generating toolkits available. Many of them are freely available from one or other of the large repositories of software on the Internet (some of these are listed in Appendix A). The most powerful are more difficult to use than Coco/R, offering, as they do, many extra features, and, in particular, incorporating more sophisticated error recovery techniques than are found in Coco/R. It will suffice to mention three of these.

Grosch (1988, 1989, 1990a), has developed a toolkit known as Cocktail, with components for generating LALR based parsers (LALR), recursive descent parsers (ELL), and scanners (REX), in a variety of languages.

Grune and Jacobs (1988) describe their LL(1)-based tool (LLGen), as a "programmer friendly LL(1) parser". It incorporates a number of interesting techniques for helping to resolve LL(1) conflicts, improving error recovery, and speeding up the development of large grammars.

A toolkit for generating compilers written in C or C++ that has received much attention is PCCTS, the Purdue University Compiler Construction Tool Set (Parr, Dietz and Cohen (1992), Parr (1996)). This is comprised of a parser generator (ANTLR), a scanner generator (DLG) and a tree-parser generator (SORCERER). It provides internal support for a number of frequently needed operations (such as abstract syntax tree construction), and is particularly interesting in that it uses LL(k) parsing with k > 1, which its authors claim give it a distinct edge over the more traditional LL(1) parsers (Parr and Quong, 1995, 1996).