14 A SIMPLE COMPILER - THE FRONT END

At this point it may be of interest to consider the construction of a compiler for a simple programming language, specifically that of section 8.7. In a text of this nature it is impossible to discuss a full-blown compiler, and the value of our treatment may arguably be reduced by the fact that in dealing with toy languages and toy compilers we shall be evading some of the real issues that a compiler writer has to face. However, we hope the reader will find the ensuing discussion of interest, and that it will serve as a useful preparation for the study of much larger compilers. The technique we shall follow is one of slow refinement, supplementing the discussion with numerous asides on the issues that would be raised in compiling larger languages. Clearly, we could opt to develop a completely hand-crafted compiler, or simply to use a tool like Coco/R. We shall discuss both approaches. Even when a compiler is constructed by hand, having an attributed grammar to describe it is very worthwhile.

On the source diskette can be found a great deal of code, illustrating different stages of development of our system. Although some of this code is listed in appendices, its volume precludes printing all of it. Some of it has deliberately been written in a way that allows for simple modification when attempting the exercises, and is thus not really of "production quality". For example, in order to allow components such as the symbol table handler and code generator to be used either with hand-crafted or with Coco/R generated systems, some compromises in design have been necessary.

Nevertheless, the reader is urged to study the code along with the text, and to attempt at least some of the many exercises based on it. A particularly worthwhile project is to construct a similar compiler, based on a language whose syntax resembles C++ rather more than it does Pascal, and whose development, like that of C++, will be marked by the steady assimilation of extra features. This language we shall name "Topsy", after the little girl in Harriet Beecher Stowe’s story who knew little of her genealogy except a suspicion that she had "grow’d". A simple Topsy program was illustrated in Exercise 8.25, where the reader was invited to create an initial syntactic specification in Cocol.

14.1 Overall compiler structure

In Chapter 2 we commented that a compiler is often developed as a sequence of phases, of which syntactic analysis is only one. Although a recursive descent parser is easily written by applying the ideas of earlier chapters, it should be clear that consideration will have to be given to the relationship of this to the other phases. We can think of a compiler with a recursive descent parser at its core as having the structure depicted in Figure 14.1.

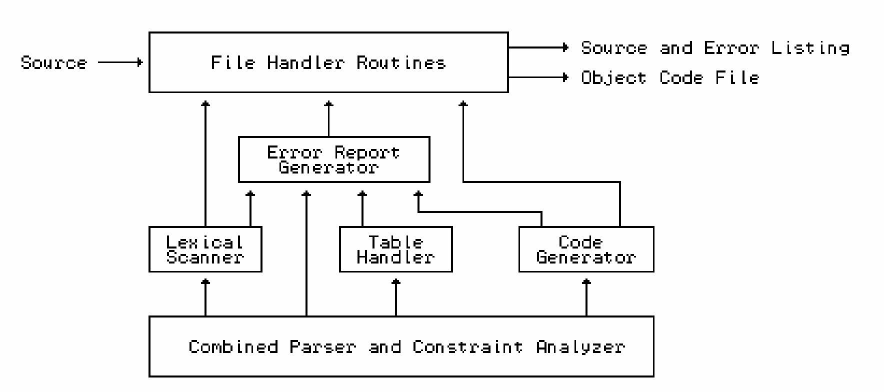


Figure 14.1 Relationship between the main components of a simple compiler

We emphasize that phases need not be sequential, as passes would be. In a recursive descent compiler the phases of syntax analysis, semantic analysis and code generation are very often interleaved, especially if the source language is designed in such a way as to permit one-pass compilation. Nevertheless, it is useful to think of developing modular components to handle the various phases, with clear simple interfaces between them.

In our Modula-2 implementations, the various components of our diagram have been implemented as separate modules, whose `DEFINITION MODULE` components export only those facilities that the clients need be aware of. The corresponding C++ implementations use classes to achieve the same sort of abstraction and protection.

In principle, the main routine of our compiler must resemble something like the following

```
void main(int argc, char *argv[])
{ char SourceName[256], ListName[256];

  // handle command line parameters
  strcpy(SourceName, argv[1]);
  if (argc > 2) strcpy(ListName, argv[2]);
  else appendextension(SourceName, ".lst", ListName);

  // instantiate compiler components
  SRCE   *Source  = new SRCE(SourceName, ListName, "Compiler Version 1", true);
  REPORT *Report  = new REPORT(Source);
  SCAN   *Scanner = new SCAN(Source, Report);
  CGEN   *CGen    = new CGEN(Report);
  TABLE  *Table   = new TABLE(Report);
  PARSER *Parser  = new PARSER(CGen, Scanner, Table, Report);

  // start compilation
  Parser->parse();
}
```

where we notice that instances of the various classes are constructed dynamically, and that their constructors establish links between them that correspond to those shown in Figure 14.1.

In practice our compilers do not look exactly like this. For example, Coco/R generates only a scanner, a parser, a rudimentary error report generator and a driver routine. The scanner and the source handling section of the file handler are combined into one module, and the routines for producing the error listing are generated along with the main driver module. The C++ version of Coco/R makes use of a standard class hierarchy involving parser, scanner and error reporter classes, and establishes links between the various instances of these classes as they are constructed. This gives the flexibility of having multiple instances of parsers or scanners within one system (however, our case studies will not exploit this power).

14.2 Source handling

Among the file handling routines is to be found one that has the task of transmitting the source, character by character, to the scanner or lexical analyser (which assembles it into symbols for subsequent parsing by the syntax analyser). Ideally this source handler should have to scan the program text only once, from start to finish, and in a one-pass compiler this should always be possible.

14.2.1 A hand-crafted source handler

The interface needed between source handler and lexical analyser is straightforward, and can be supplied by a routine that simply extracts the next character from the source each time it is invoked. It is convenient to package this with the routines that assume responsibility for producing a source listing, and, where necessary, producing an error message listing, because all these requirements will be input/output device dependent. It is useful to add some extra functionality, and so the public interface to our source handling class is defined by

```
  class SRCE {
    public:
      FILE *lst;                    // listing file
      char ch;                      // latest character read

      void nextch(void);
      // Returns ch as the next character on this source line, reading a new
      // line where necessary.  ch is returned as NUL if src is exhausted.

      bool endline(void);
      // Returns true when end of current line has been reached

      void listingon(void);
      // Requests source to be listed as it is read

      void listingoff(void);
      // Requests source not to be listed as it is read

      void reporterror(int errorcode);
      // Points out error identified by errorcode with suitable message

      virtual void startnewline()  {;}
      // Called at start of each line

      int getline(void);
      // Returns current line number

      SRCE(char *sourcename, char *listname, char *version, bool listwanted);
      // Opens src and lst files using given names.
      // Resets internal state in readiness for starting to scan.
      // Notes whether listwanted.  Displays version information on lst file.

      ~SRCE();
      // Closes src and lst files
  };
```

Some aspects of this interface deserve further comment:

- We have not shown the private members of the class, but of course there are several of these.

- The `startnewline` routine has been declared virtual so that a simple class can be derived from this one to allow for the addition of extra material at the start of each new line on the listing - for example, line numbers or object code addresses. In Modula-2, Pascal or C, the same sort of functionality may be obtained by manipulating a procedure variable or function pointer.

- Ideally, both source and listing files should remain private. This source handler declares the listing file public only so that we can add trace or debugging information while the system is being developed.

- The class constructor and destructor assume responsibility for opening and closing the files, whose names are passed as arguments.

The implementation of this class is fairly straightforward, and has much in common with the similar class used for the assemblers of Chapter 6. The code appears in Appendix B, and the following implementation features are worthy of brief comment:

- The source is scanned and reflected a whole line at a time, as this makes subsequent error reporting much easier.

- The handler effectively inserts an extra blank at the end of each line. This decouples the rest of the system from the vagaries of whatever method the host operating system uses to represent line ends in text files. It also ensures that no symbol may extend over a line break.

- It is not possible to read past the end of file - attempts to do so simply return a `NUL` character.

- The `reporterror` routine will not display an error message unless a minimum number of characters have been scanned since the last error was reported. This helps suppress the cascade of error messages that might otherwise appear at any one point during error recovery of the sort discussed in sections 10.3 and 14.6.

- Our implementation has chosen to use the `stdio` library, rather than `iostreams`, mainly to take advantage of the concise facilities provided by the `printf` routine.

Exercises

14.1 The `nextch` routine will be called once for every character in the source text. This can represent a considerable bottleneck, especially as programs are often prepared with a great many blanks at the starts of indented lines. Some editors also pad out the ends of lines with unnecessary blanks. Can you think of any way in which this overhead might be reduced?

14.2 Some systems allowing an ASCII character set (with ordinal values in the range 0 ... 127) are used with input devices which generate characters having ordinal values in the range 0 ... 255 - typically the "eighth bit" might always be set, or used or confused with parity checking. How and where could this bit be discarded?

14.3 A source handler might improve on efficiency rather dramatically were it able to read the entire source file into a large memory buffer. Since modern systems are often blessed with relatively huge amounts of RAM, this is usually quite feasible. Develop such a source handler, compatible with the class interface suggested above, bearing in mind that we wish to be able to reflect the source line by line as it is read, so that error messages can be appended as exemplified in section 14.6.

14.4 Develop a source handler implementation that uses the C++ stream-based facilities from the `iostreams` library.

14.2.2 Source handling in Coco/R generated systems

As we have already mentioned, Coco/R integrates the functions of source handler and scanner, so as to be able to cut down on the number of files it has to generate. The source and listing files have to be opened before making the call to instantiate or initialize the scanner, but this is handled automatically by the generated driver routine. It is of interest that the standard frame files supplied with Coco/R arrange for this initialization to read the entire source file into a buffer, as suggested in Exercise 14.3.

14.3 Error reporting

As can be seen from Figure 14.1, most components of a compiler have to be prepared to signal that something has gone awry in the compilation process. To allow all of this to take place in a uniform way, we have chosen to introduce a base class with a very small interface:

```
  class REPORT {
    public:
      REPORT();
      // Initializes error reporter

      virtual void error(int errorcode);
      // Reports on error designated by suitable errorcode number

      bool anyerrors(void);
      // Returns true if any errors have been reported

    protected:
      bool errors;
  };
```

Error reporting is then standardized by calling on the `error` member of this class whenever an error is detected, passing it a unique number to distinguish the error.

The base class can choose simply to abort compilation altogether. Although at least one highly successful microcomputer Pascal compiler uses this strategy (Turbo Pascal, from Borland International), it tends to become very annoying when one is developing large systems. Since the `error` member is virtual, it is an easy matter to derive a more suitable class from this one, without, of course, having to amend any other part of the system. For our hand-crafted system we can do this as follows:

```
  class clangReport : public REPORT {
    public:
      clangReport(SRCE *S)   { Srce = S; }
      virtual void error(int errorcode)
        { Srce->reporterror(errorcode); errors = true; }
    private:
      SRCE *Srce;
  };
```

and the same technique can be used to enhance Coco/R generated systems. The Modula-2 and Pascal implementations achieve the same functionality through the use of procedure variables.

14.4 Lexical analysis

The main task of the scanner is to provide some way of uniquely identifying each successive token or symbol in the source code that is being compiled. Lexical analysis was discussed in section 10.4, and presents few problems for a language as simple as ours.

14.4.1 A hand-crafted scanner

The interface between the scanner and the main parser is conveniently provided by a routine `getsym` for returning a parameter `SYM` of a record or structure type assembled from the source text. This can be achieved by defining a class with a public interface as follows:

```
enum SCAN_symtypes {
  SCAN_unknown, SCAN_becomes, SCAN_lbracket, SCAN_times, SCAN_slash, SCAN_plus,
  SCAN_minus, SCAN_eqlsym, SCAN_neqsym, SCAN_lsssym, SCAN_leqsym, SCAN_gtrsym,
  SCAN_geqsym, SCAN_thensym, SCAN_dosym, SCAN_rbracket, SCAN_rparen, SCAN_comma,
  SCAN_lparen, SCAN_number, SCAN_stringsym, SCAN_identifier, SCAN_coendsym,
  SCAN_endsym, SCAN_ifsym, SCAN_whilesym, SCAN_stacksym, SCAN_readsym,
  SCAN_writesym, SCAN_returnsym, SCAN_cobegsym, SCAN_waitsym, SCAN_signalsym,
  SCAN_semicolon, SCAN_beginsym, SCAN_constsym, SCAN_varsym, SCAN_procsym,
  SCAN_funcsym, SCAN_period, SCAN_progsym, SCAN_eofsym
};

const int lexlength = 128;
typedef char lexeme[lexlength + 1];

struct SCAN_symbols {
  SCAN_symtypes sym;    // symbol type
  int num;              // value
  lexeme name;          // lexeme
};

class SCAN {
  public:
    void getsym(SCAN_symbols &SYM);
    // Obtains the next symbol in the source text

    SCAN(SRCE *S, REPORT *R);
    // Initializes scanner
};
```

Some aspects of this interface deserve further comment:

- `SCAN_symbols` makes provision for returning not only a unique symbol type, but also the corresponding textual representation (known as a **lexeme**), and also the numeric value when a symbol is recognized as a number.

- `SCAN_unknown` caters for erroneous characters like # and ? which do not really form part of the terminal alphabet. Rather than take action, the humble scanner returns the symbol without comment, and leaves the parser to cope. Similarly, an explicit `SCAN_eofsym` is always returned if `getsym` is called after the source code has been exhausted.

- The ordering of the `SCAN_symtypes` enumeration is significant, and supports an interesting form of error recovery that will be discussed in section 14.6.1.

- The enumeration has also made provision for a few symbol types that will be used in the extensions of later chapters.

A scanner for Clang is readily programmed in an *ad hoc* manner, driven by a selection statement, and an implementation can be found in Appendix B. As with source handling, some implementation issues call for comment:

- Some ingenuity has to be applied to the recognition of literal strings. A repeated quote within a string is used (as in Pascal) to denote a single quote, so that the end of a string can only be detected when an odd number of quotes is followed by a non-quote.

- The scanner has assumed responsibility for a small amount of semantic activity, namely the evaluation of a number. Although it may seem a convenient place to do this, such analysis is not always as easy as it might appear. It becomes slightly more difficult to develop scanners that have to distinguish between numbers represented in different bases, or between real and integer numbers.

- There are two areas where the scanner has been given responsibility for detecting errors:

  Although the syntactic description of the language does not demand it, practical considerations require that the value of a numeric constant should be within the range of the machine. This is somewhat tricky to ensure in the case of cross-compilers, where the range on the host and target machines may be different. Some authors go so far as to suggest that this semantic activity be divorced from lexical analysis for that reason. Our implementation shows how range checking can be handled for a self-resident compiler.

  Not only do many languages insist that no identifier (or any other symbol) be carried across a line break, they usually do this for strings as well. This helps to guard against the chaos that would arise were a closing quote to be omitted - further code would become string text, and future string text would become code! The limitation that a string be confined to one source line is, in practice, rarely a handicap, and the restriction is easily enforced.

- We have chosen to use a binary search to recognize the reserved keywords. Tables of symbol types that correspond to keywords, and symbols that correspond to single character terminals, are initialized as the scanner is instantiated. The idioms of C++ programming suggest that such activities are best achieved by having static members of the class, set up by a "initializer" that forms part of their definition. For a binary search to function correctly it is necessary that the table of keywords be in alphabetic order, and care must be taken if and when the scanner is extended.

Exercises

14.5 The only screening this scanner does is to strip blanks separating symbols. How would you arrange for it to strip comments

(a) of the form `{ comment in curly braces }`
(b) of the form `(* comment in Modula-2 braces *)`
(c) of the form `// comment to end of the line as in C++`
(d) of either or both of forms (a) and (b), allowing for nesting?

14.6 Balanced comments are actually dangerous. If not properly closed, they may consume valid source code. One way of assisting the coder is to issue a warning if a semicolon is found within a comment. How could this be implemented as part of the answer to Exercise 14.5?

14.7 The scanner does not react sensibly to the presence of tab or formfeed characters in the source. How can this be improved?

14.8 Although the problem does not arise in the case of Clang, how do you suppose a hand-crafted scanner is written for languages like Modula-2 and Pascal that must distinguish between `REAL` literals of the form `3.4` and subrange specifiers of the form `3..4`, where no spaces delimit the "..", as is quite legal? Can you think of an alternative syntax which avoids the issue altogether? Why do you suppose Modula-2 and Pascal do not use such a syntax?

14.9 Modula-2 allow string literals to be delimited by either single or double quotes, but not to contain the delimiter as a member of the string. C and C++ use single and double quotes to distinguish between character literals and string literals. Develop scanners that meet such requirements.

14.10 In C++, two strings that appear in source with nothing but white space between them are automatically concatenated into a single string. This allows long strings to be spread over several lines, if necessary. Extend your scanner to support this feature.

14.11 Extend the scanner to allow escape sequences like the familiar `\n` (newline) or `\t` (tab) to represent "control" characters in literal strings, as in C++.

14.12 Literal strings present other difficulties to the rest of the system that may have to process them. Unlike identifiers (which usually find their way into a symbol table), strings may have to be stored in some other way until the code generator is able to handle them. Consider extending the `SCAN_symbols` structure so that it contains a member that points to a dynamically allocated array of exactly the correct length for storing any string that has been recognized (and is a null pointer otherwise).

14.13 In our compiler, a diagnostic listing of the symbol table will be provided if the name `Debug` is used for the main program. Several compilers make use of pragmatic comments as compiler directives, so as to make such demands of the system - for example a comment of the form `(*$L- *)` might request that the listing be switched off, and one of the form `(*$L+ *)` that it be reinstated. These requests are usually handled by the scanner. Implement such facilities for controlling listing of the source program, and listing the symbol table (for example, using `(*$T+ *)` to request a symbol table listing). What action should be taken if a source listing has been suppressed, and if errors are discovered?

14.14 The restriction imposed on the recognizable length of a lexeme, while generous, could prove embarrassing at some stage. If, as suggested in Exercise 14.3, a source handler is developed that stores the entire source text in a memory buffer, it becomes possible to use a less restrictive structure for `SCAN_symbols`, like that defined by

```
struct SCAN_symbols {
  SCAN_symtypes sym;  // symbol type
  int num;            // value
  long pos, length;   // starting position and length of lexeme
};
```

Develop a scanner based on this idea. While this is easy to do, it may have ramifications on other parts of the system. Can you predict what these might be?

14.15 Develop a hand-crafted scanner for the Topsy language of Exercise 8.25. Incorporate some of the features suggested in Exercises 14.5 to 14.14.

14.4.2 A Coco/R generated scanner

A Cocol specification of the token grammar for our language is straightforward, and little more need be said. In C++, the generated scanner class is derived from a standard base class that assumes that the source file has already been opened; its constructor takes an argument specifying the corresponding "file handle". As we have already noted in Chapter 12, calls to the `Get` routine of this scanner simply return a token number. If we need to determine the text of a string, the name of an identifier, or the value of a numeric literal, we are obliged to write appropriately attributed productions into the phrase structure grammar. This is easily done, as will be seen by studying these productions in the grammars to be presented later.

Exercises

14.16 Is it possible to write a Cocol specification that generates a scanner that can handle the suggestions made in Exercises 14.10 and 14.11 (allowing strings that immediately follow one another to be automatically concatenated, and allowing for escape sequences like "`\n`" to appear within strings to have the meanings that they do in C++)? If not, how else might such features be incorporated into Coco/R generated systems?

14.4.3 Efficient keyword recognition

The subject of keyword recognition is important enough to warrant further comment. It is possible to write a FSA to do this directly (see section 10.5). However, in most languages, including Clang and Topsy, identifiers and keywords have the same basic format, suggesting the construction of scanners that simply extract a "word" into a string, which is then tested to see whether it is, in fact, a keyword. Since string comparisons are tedious, and since typically 50%-70% of program text consists of either identifiers or keywords, it makes sense to be able to perform this test as quickly as possible. The technique used in our hand-crafted scanner of arranging the keywords in an alphabetically ordered table and then using a binary search is only one of several ideas that strive for efficiency. At least three other methods are often advocated:

- The keywords can be stored in a table in length order, and a sequential search used among those that have the same length as the word just assembled.

- The keywords can be stored in alphabetic order, and a sequential search used among those that have the same initial letter as the word just assembled. This is the technique employed by Coco/R.

- A "perfect hashing function" can be derived for the keyword set, allowing for a single string comparison to distinguish between all identifiers and keywords.

A hashing function is one that is applied to a string so as to extract particular characters, map these onto small integer values, and return some combination of those. The function is usually kept very simple, so that no time is wasted in its computation. A *perfect* hash function is one chosen to be clever enough so that its application to each of the strings in a set of keywords results in a unique value for each keyword. Several such functions are known. For example, if we use an ASCII character set, then the C++ function

```
int hash (char *s)
{ int L = strlen(s); return (256 * s[0] + s[L-1] + L) % 139; }
```

will return 40 unique values in the range 0 ... 138 when applied to the 40 strings that are the keywords of Modula-2 (Gough and Mohay, 1988). Of course, it will return some of these values for non-keywords as well (for example the keyword "`VAR`" maps to the value 0, as does any other three letter word starting with "V" and ending with "R"). To use this function one would first construct a 139 element string table, with the appropriate 40 elements initialized to store the keywords, and the rest to store null strings. As each potential identifier is scanned, its hash value is computed using the above formula. A single probe into the table will then ascertain whether the word just recognized is a keyword or not.

Considerable effort has gone into the determination of "minimal perfect hashing functions" - ones in which the number of possible values that the function can return is exactly the same as the number of keywords. These have the advantage that the lookup table can be kept small (Gough’s function would require a table in which nearly 60% of the space was wasted).

For example, when applied to the 19 keywords used for Clang, the C++ function

```
    int hash (char *s)
    { int L = strlen(s); return Map[s[0]] + Map[s[L-2]] + L - 2; }
```

will return a unique value in the range 0 ... 18 for each of them. Here the mapping is done via a 256 element array `Map`, which is initialized so that all values contain zero save for those shown below:

```
   Map[’B’] = 6; Map[’D’] = 8; Map[’E’] =  5; Map[’L’] = 9;
   Map[’M’] = 7; Map[’N’] = 8; Map[’O’] = 12; Map[’P’] = 3;
   Map[’S’] = 3; Map[’T’] = 8; Map[’W’] =  1;
```

Clearly this particular function cannot be applied to strings consisting of a single character, but such strings can easily be recognized as identifiers anyway. It is one of a whole class of similar functions proposed by Cichelli (1980), who also developed a backtracking search technique for determining the values of the elements of the `Map` array.

It must be emphasized that if a perfect hash function technique is used for constructing scanners for languages like Clang and Topsy that are in a constant state of flux as new keywords are proposed, then the hash function has to be devised afresh with each language change. This makes it an awkward technique to use for prototyping. However, for production quality compilers for well established languages, the effort spent in finding a perfect hash function can have a marked influence on the compilation time of tens of thousands of programs thereafter.

Exercises

To assist with these exercises, a program incorporating Cichelli’s algorithm, based on the one published by him in 1979, appears on the source diskette. Another well known program for the construction of perfect hash functions is known as `gperf`. This is written in C, and is available from various Internet sites that mirror the extensive GNU archives of software distributed by the Free Software Foundation (see Appendix A).

14.17 Develop hand-crafted scanners that make use of the alternative methods of keyword identification suggested here.

14.18 Carry out experiments to discover which method seems to be best for the reserved word lists of languages like Clang, Topsy, Pascal, Modula-2 or C++. To do so it is not necessary to develop a full scale parser for each of these languages. It will suffice to invoke the `getsym` routine repeatedly on a large source program until all symbols have been scanned, and to time how long this takes.

Further reading

Several texts treat lexical analysis in far more detail than we have done; justifiably, since for larger languages there are considerably more problem areas than our simple one raises. Good discussions are found in the books by Gough (1988), Aho, Sethi and Ullman (1986), Welsh and Hay (1986) and Elder (1994). Pemberton and Daniels (1982) give a very detailed discussion of the lexical analyser found in the Pascal-P compiler.

Discussion of perfect hash function techniques is the source of a steady stream of literature. Besides the papers by Cichelli (1979, 1980), the reader might like to consult those by Cormack, Horspool and Kaiserwerth (1985), Sebesta and Taylor (1985), Panti and Valenti (1992), and Trono (1995).

14.5 Syntax analysis

For languages like Clang or Topsy, which are essentially described by LL(1) grammars, construction of a simple parser presents few problems, and follows the ideas developed in earlier chapters.

14.5.1 A hand-crafted parser

Once again, if C++ is the host language, it is convenient to define a hand-crafted parser in terms of its own class. If all that is required is syntactic analysis, the public interface to this can be kept very simple:

```
  class PARSER {
    public:
      PARSER(SCAN *S, REPORT *R);
      // Initializes parser

      void parse(void);
      // Parses the source code
  };
```

where we note that the class constructor associates the parser instance with the appropriate instances of a scanner and error reporter. Our complete compiler will need to go further than this - an association will have to be made with at least a code generator and symbol table handler. As should be clear from Figure 14.1, in principle no direct association need be made with a source handler (in fact, our system makes such an association, but only so that the parser can direct diagnostic output to the source listing).

An implementation of this parser, devoid of any attempt at providing error recovery, constraint analysis or code generation, is provided on the source diskette. The reader who wishes to see a much larger application of the methods discussed in section 10.2 might like to study this. In this connection it should be noted that Modula-2 and Pascal allow for procedures and functions to be nested. This facility (which is lacking in C and C++) can be used to good effect when developing compilers in those languages, so as to mirror the highly embedded nature of the phrase structure grammar.

14.5.2 A Coco/R generated parser

A parser for Clang can be generated immediately from the Cocol grammar presented in section 8.7.2. At this stage, of course, no attempt has been made to attribute the grammar to incorporate error recovery, constraint analysis, or code generation.

Exercises

Notwithstanding the fact that the construction of an parser that does little more than check syntax is still some distance away from having a complete compiler, the reader might like to turn his or her attention to some of the following exercises, which suggest extensions to Clang or Topsy, and to construct grammars, scanners and parsers for recognizing such extensions.

14.19 Compare the hand-crafted parser found on the source diskette with the source code that is produced by Coco/R.

14.20 Develop a hand-crafted parser for Topsy as suggested by Exercise 8.25.

14.21 Extend your parser for Clang to accept the `REPEAT ... UNTIL` loop as it is found in Pascal or Modula-2, or add an equivalent `do` loop to Topsy.

14.22 Extend the `IF ... THEN` statement to provide an `ELSE` clause.

14.23 How would you parse a Pascal-like `CASE` statement? The standard Pascal `CASE` statement does not have an `ELSE` or `OTHERWISE` option. Suggest how this could be added to Clang, and modify the parser accordingly. Is it a good idea to use `OTHERWISE` or `ELSE` for this purpose - assuming that you already have an `IF ... THEN ... ELSE` construct?

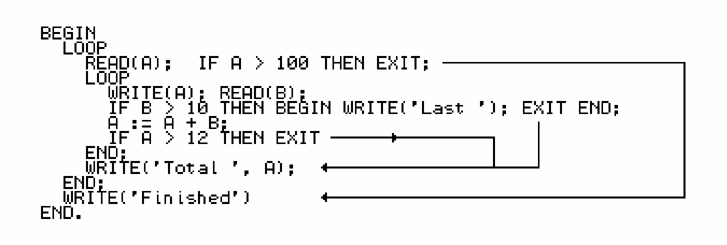
14.24 What advantages does the Modula-2 `CASE` statement have over the Pascal version? How would you parse the Modula-2 version?

14.25 The C++ `switch` statement bears some resemblance to the `CASE` statement, although its semantics are rather different. Add the `switch` statement to Topsy.

14.26 How would you add a Modula-2 or Pascal-like `FOR` loop to Clang?

14.27 The C++ `for` statement is rather different from the Pascal one, although it is often used in much the same way. Add a `for` statement to Topsy.

14.28 The `WHILE, FOR` and `REPEAT` loops used in Wirth's languages are *structured* - they have only one entry point, and only one exit point. Some languages allow a slightly less structured loop, which has only one entry point, but which allows exit from various places within the loop body. An example of this might be as follows



```
BEGIN
  LOOP
    READ(A);  IF A > 100 THEN EXIT;
    LOOP
      WRITE(A); READ(B);
      IF B > 10 THEN BEGIN WRITE('Last '); EXIT END;
      A := A + B;
      IF A > 12 THEN EXIT
    END;
    WRITE('Total ', A);
  END;
  WRITE('Finished')
END.
```

Like others, `LOOP` statements can be nested. However, `EXIT` statements may only appear within `LOOP` sequences. Can you find context-free productions that will allow you to incorporate these statements into Clang?

14.29 If you are extending Topsy to make it resemble C++ as closely as possible, the equivalent of the `EXIT` statement would be found in the `break` or `continue` statements that C++ allows within its various structured statements like `switch`, `do` and `while`. How would you extend the grammar for Topsy to incorporate these statements? Can the restrictions on their placement be expressed in a context-free grammar?

14.30 As a more challenging exercise, suppose we wished to extend Clang or Topsy to allow for variables and expressions of other types besides integer (for example, Boolean). Various approaches might be taken, as exemplified by the following

(a) Replacing the Clang keyword `VAR` by a set of keywords used to introduce variable lists:

```
int  X, Y, Z[4];
bool InTime, Finished;
```

(b) Retention of the `VAR` symbol, along with a set of standard type identifiers, used after variable lists, as in Pascal or Modula-2:

```
VAR
  X, Y, Z[4] : INTEGER;
  InTime, Finished : BOOLEAN;
```

Develop a grammar (and parser) for an extended version of Clang or Topsy that uses one or other of these approaches. The language should allow expressions to use Boolean operators (`AND, OR, NOT`) and Boolean constants (`TRUE` and `FALSE`). Some suggestions were made in this regard in Exercise 13.17.

14.31 The approach used in Pascal and Modula-2 has the advantage that it extends seamlessly to the more general situations in which users may introduce their own type identifiers. In C++ one finds a hybrid: variable lists may be preceded either by special keywords or by user defined type names:

```
typedef bool sieve[1000];
int X, Y;      // introduced by keyword
sieve Primes;  // introduced by identifier
```

Critically examine these alternative approaches, and list the advantages and disadvantages either seems to offer. Can you find context-free productions for Topsy that would allow for the introduction of a simple `typedef` construct?

A cynic might contend that if a language has features which are the cause of numerous beginners' errors, then one should redesign the language. Consider a selection of the following:

14.32 Bailes (1984) made a plea for the introduction of a "Rational Pascal". According to him, the keywords `DO` (in `WHILE` and `FOR` statements), `THEN` (in `IF` statements) and the semicolons which are used as terminators at the ends of declarations and as statement separators should all be discarded. (He had a few other ideas, some even more contentious). Can you excise semicolons from Clang and Topsy, and then write a recursive descent parser for them? If, indeed, semicolons seem to serve no purpose other than to confuse learner programmers, why do you suppose language designers use them?

14.33 The problems with `IF ... THEN` and `IF ... THEN ... ELSE` statements are such that one might be tempted to try a language construct described by

```
IfStatement =  "IF" Condition "THEN" Statement
                  { "ELSIF" Condition "THEN" Statement }
                  [ "ELSE Statement ] .
```

Discuss whether this statement form might easily be handled by extensions to your parser. Does it have any advantages over the standard `IF ... THEN ... ELSE` arrangement - in particular, does it resolve the "dangling else" problem neatly?

14.34 Extend your parser to accept structured statements on the lines of those used in Modula-2, for example

```
IfStatement        =  "IF" Condition "THEN" StatementSequence
                         { "ELSIF" Condition "THEN" StatementSequence }
                         [ "ELSE" StatementSequence  ]
                      "END" .
WhileStatement     =  "WHILE" Condition "DO" StatementSequence  "END" .
StatementSequence  =  Statement { ";" Statement } .
```

14.35 Brinch Hansen (1983) did not approve of implicit "empty" statements. How do these appear in our languages, are they ever of practical use, and if so, in what ways would an explicit statement (like the `SKIP` suggested by Brinch Hansen) be any improvement?

14.36 Brinch Hansen incorporated only one form of loop into Edison - the `WHILE` loop - arguing that the other forms of loops were unnecessary. What particular advantages and disadvantages do these loops have from the points of view of a compiler writer and a compiler user respectively? If you were limited to only one form of loop, which would you choose, and why?

14.6 Error handling and constraint analysis

In section 10.3 we discussed techniques for ensuring that a recursive descent parser can recover after detecting a syntax error in the source code presented to it. In this section we discuss how best to apply these techniques to our Clang compiler, and then go on to discuss how the parser can be extended to perform context-sensitive or constraint analysis.

14.6.1 Syntax error handling in hand-crafted parsers

The scheme discussed previously - in which each parsing routine is passed a set of "follower" symbols that it can use in conjunction with its own known set of "first" symbols - is easily applied systematically to hand-crafted parsers. It suffers from a number of disadvantages, however:

- It is quite expensive, since each call to a parsing routine is effectively preceded by two time-consuming operations - the dynamic construction of a set object, and the parameter passing operation itself - operations which turn out not to have been required if the source being translated is correct.

- If, as often happens, seemingly superfluous symbols like semicolons are omitted from the source text, the resynchronization process can be overly severe.

Thus the scheme is usually adapted somewhat, often in the light of experience gained by observing typical user errors. A study of the source code for such parsers on the source diskette will reveal examples of the following useful variations on the basic scheme:

- In those many places where "weak" separators are found in constructs involving iterations, such as

```
        VarDeclarations     =  "VAR" OneVar { "," OneVar } ";"
```

```
CompoundStatement   =  "BEGIN" Statement { ";" Statement } "END" .
Term                =  Factor  { MulOp  Factor } .
```

the iteration is started as long as the parser detects the presence of the weak separator *or* a valid symbol that would follow it in that context (of course, appropriate errors are reported if the separator has been omitted). This has the effect of "inserting" such missing separators into the stream of symbols being parsed, and proves to be a highly effective enhancement to the basic technique.

- Places where likely errors are expected - such as confusion between the "`:=`" and "`=`" operators, or attempting to provide an integer expression rather than a Boolean comparison expression in an *IfStatement* or *WhileStatement* - are handled in an *ad-hoc* way.

- Many sub-parsers do not need to make use of the prologue and epilogue calls to the `test` routine. In particular, there is no need to do this in routines like those for *IfStatement, WhileStatement* and so on, which have been introduced mainly to enhance the modularity of *Statement*.

- The Modula-2 and Pascal implementations nest their parsing routines as tightly as possible. Not only does this match the embedded nature of the grammar very nicely, it also reduces the number of parameters that have to be passed around.

Because of the inherent cost in the follower-set based approach to error recovery, some compiler writers make use of simpler schemes that can achieve very nearly the same degree of success at far less cost. One of these, suggested by Wirth (1986, 1996), is based on the observation that the symbols passed as members of follower sets to high level parsing routines - such as *Block* - effectively become members of every follower set parameter computed thereafter. When one finally gets to parse a *Statement*, for example, the set of stopping symbols used to establish synchronization at the start of the *Statement* routine is the union of FIRST(*Statement*) + FOLLOW(*Program*) + FOLLOW(*Block*), while the set of stopping symbols used to establish synchronization at the end of *Statement* is the union of FOLLOW(*Statement*) + FOLLOW(*Program*) + FOLLOW(*Block*). Furthermore, if we treat the semicolon that separates statements as a "weak" separator, as previously discussed, no great harm is done if the set used to establish synchronization at the end of *Statement* also includes the elements of FIRST(*Statement*).

Careful consideration of the `SCAN_symtypes` enumeration introduced in section 14.4.1 will reveal that the values have been ordered so that the following patterns hold to a high degree of accuracy:

```
SCAN_unknown .. SCAN_lbracket,     Miscellaneous
SCAN_times, SCAN_slash,            FOLLOW(Factor)
SCAN_plus, SCAN_minus,             FOLLOW(Term)
SCAN_eqlsym .. SCAN_geqsym,        FOLLOW(Expression1) in Condition
SCAN_thensym, SCAN_dosym,          FOLLOW(Condition)
SCAN_rbracket .. SCAN_comma,       FOLLOW(Expression)
SCAN_lparen, .. SCAN_identifier,   FIRST(Factor)
SCAN_coendsym, SCAN_endsym,        FOLLOW(Statement)
SCAN_ifsym .. SCAN_signalsym,      FIRST(Statement)
SCAN_semicolon,                    FOLLOW(Block)
SCAN_beginsym .. SCAN_funcsym,     FIRST(Block)
SCAN_period,                       FOLLOW(Program)
SCAN_progsym,                      FIRST(Program)
SCAN_eofsym
```

The argument now goes that, with this carefully ordered enumeration, virtually all of the tests of the form

```
Sym ∈  SynchronizationSet
```

can be accurately replaced by tests of the form

Sym ≥ SmallestElement(SynchronizationSet)

and that synchronization at crucial points in the grammar can be achieved by using a routine developed on the lines of

```
void synchronize(SCAN_symtypes SmallestElement, int errorcode)
{ if (SYM.sym >= SmallestElement) return;
  reporterror(errorcode);
  do { getsym(); } while (SYM.sym < SmallestElement);
}
```

The way in which this idea could be used is exemplified in a routine for parsing Clang statements.

```
void Statement(void)
// Statement = [ CompoundStatement | Assignment | IfStatement
//             | WhileStatement | WriteStatement | ReadStatement ] .
{ synchronize(SCAN_identifier, 15);
  // We shall return correctly if SYM.sym is a semicolon or END (empty statement)
  // or if we have synchronized (prematurely) on a symbol that really follows
  // a Block
  switch (SYM.sym)
  { case SCAN_identifier: Assignment(); break;
    case SCAN_ifsym:      IfStatement(); break;
    case SCAN_whilesym:   WhileStatement(); break;
    case SCAN_writesym:   WriteStatement(); break;
    case SCAN_readsym:    ReadStatement(); break;
    case SCAN_beginsym:   CompoundStatement(); break;
    default:              return;
  }
  synchronize(SCAN_endsym, 32);
  // In some situations we shall have synchronized on a symbol that can start
  // a further Statement, but this should be handled correctly from the call
  // made to Statement from within CompoundStatement
}
```

It turns out to be necessary to replace some other set inclusion tests, by providing predicate functions exemplified by

```
bool inFirstStatement(SCAN_symtypes Sym)
// Returns true if Sym can start a Statement
{ return (Sym == SCAN_identifier || Sym == SCAN_beginsym ||
          Sym >= SCAN_ifsym && Sym <= SCAN_signalsym);
}
```

Complete parsers using this ingenious scheme are to be found on the source diskette. However, the idea is fairly fragile. Symbols do not always fall uniquely into only one of the FIRST or FOLLOW sets, and in large languages there may be several keywords (like `END`, `CASE` and `OF` in Pascal) that can appear in widely different contexts. If new keywords are added to an evolving language, great care has to be taken to maintain the optimum ordering; if a token value is misplaced, error recovery would be badly affected.

The scheme can be made more robust by declaring various synchronization set constants, without requiring their elements to have contiguous values. This is essentially the technique used in Coco/R generated recovery schemes, and adapting it to hand-crafted parsers is left as an interesting exercise for the reader.

14.6.2 Syntax error handling in Coco/R generated parsers

The way in which a Cocol description of a grammar is augmented to indicate where synchronization should be attempted has already been discussed in section 12.4.2. To be able to achieve optimal use of the basic facilities offered by the use of the `SYNC` and `WEAK` directives calls for some ingenuity. If too many `SYNC` directives are introduced, the error recovery achievable with the use of `WEAK` can actually deteriorate, since the union of all the `SYNC` symbol sets tends to become the entire universe. Below we show a modification of the grammar in section 8.7.2 that has been found to work quite well, and draw attention to the use of two places (in the productions for *Condition* and *Term*) where an explicit call to the error reporting interface has been used to handle situations where one wishes to be lenient in the treatment of missing symbols.

```
PRODUCTIONS /* some omitted to save space */
  Clang             = "PROGRAM" identifier WEAK ";" Block "." .
  Block             = SYNC { ( ConstDeclarations | VarDeclarations ) SYNC }
                      CompoundStatement .
  OneConst          = identifier WEAK "=" number ";" .
  VarDeclarations   = "VAR" OneVar { WEAK "," OneVar } ";" .
  CompoundStatement = "BEGIN" Statement { WEAK ";" Statement } "END" .
  Statement         = SYNC [   CompoundStatement | Assignment
                             | IfStatement       | WhileStatement
                             | ReadStatement     | WriteStatement ] .
  Assignment        = Variable ":=" Expression SYNC .
  Condition         = Expression ( RelOp Expression | (. SynError(91); .) ) .
  ReadStatement     = "READ" "(" Variable { WEAK "," Variable } ")" .
  WriteStatement    = "WRITE"
                      [ "(" WriteElement { WEAK "," WriteElement }  ")" ] .
  Term              = Factor { ( MulOp | (. SynError(92); .) ) Factor } .
```

14.6.3 Constraint analysis and static semantic error handling

We have already had cause to remark that the boundary between syntactic and semantic errors can be rather vague, and that there are features of real computer languages that cannot be readily described by context-free grammars. To retain the advantages of simple one-pass compilation when we include semantic analysis, and start to attach meaning to our identifiers, usually requires that the "declaration" parts of a program come before the "statement" parts. This is easily enforced by a context-free grammar, all very familiar to a Modula-2, Pascal or C programmer, and seems quite natural after a while. But it is only part of the story. Even if we insist that declarations precede statements, a context-free grammar is still unable to specify that only those identifiers which have appeared in the declarations (so-called *defining occurrences*) may appear in the statements (in so-called *applied occurrences*). Nor is a context-free grammar powerful enough to specify such constraints as insisting that only a variable identifier can be used to denote the target of an assignment statement, or that a complete array cannot be assigned to a scalar variable. We might be tempted to write productions that seem to capture these constraints:

```
 Clang             =  "PROGRAM" ProgIdentifier ";" Block "." .
 Block             =  { ConstDeclarations | VarDeclarations }
                      CompoundStatement .
 ConstDeclarations =  "CONST" OneConst { OneConst } .
 OneConst          =  ConstIdentifier "=" number ";" .
 VarDeclarations   =  "VAR" OneVar { "," OneVar } ";" .
 OneVar            =  ScalarVarIdentifier | ArrayVarIdentifier UpperBound .
 UpperBound        =  "[" number "]" .
 Assignment        =  Variable ":=" Expression .
 Variable          =  ScalarVarIdentifier | ArrayVarIdentifier "[" Expression "]" .
 ReadStatement     =  "READ" "(" Variable { "," Variable } ")" .
 Expression        =  ( "+" Term | "-" Term | Term ) { AddOp Term } .
 Term              =  Factor { MulOp Factor } .
 Factor            =    ConstIdentifier | Variable | number
                      | "(" Expression ")" .
```

This would not really get us very far, since all identifiers are lexically equivalent! We could attempt to use a context-sensitive grammar to overcome such problems, but that turns out to be unnecessarily complicated, for they are easily solved by leaving the grammar as it was, adding attributes in the form of context conditions, and using a symbol table.

Demanding that identifiers be declared in such a way that their static semantic attributes can be recorded in a symbol table, whence they can be retrieved at any future stage of the analysis, is not nearly as tedious as users might at first imagine. It is clearly a semantic activity, made easier by a syntactic association with keywords like `CONST`, `VAR` and `PROGRAM`.

Setting up a symbol table may be done in many ways. If one is interested merely in performing the sort of constraint analysis suggested earlier for a language as simple as Clang we may begin by noting that identifiers designate objects that are restricted to one of three simple varieties - namely *constant*, *variable* and *program*. The only apparent complication is that, unlike the other two, a variable identifier can denote either a simple scalar, or a simple linear array. A simple table handler can then be developed with a class having a public interface like the following:

```
  const int TABLE_alfalength = 15; // maximum length of identifiers
  typedef char TABLE_alfa[TABLE_alfalength + 1];

  enum TABLE_idclasses { TABLE_consts, TABLE_vars, TABLE_progs };

  struct TABLE_entries {
    TABLE_alfa name;             // identifier
    TABLE_idclasses idclass;     // class
    bool scalar;                 // distinguish arrays from scalars
  };

  class TABLE {
    public:
      TABLE(REPORT *R);
      // Initializes symbol table

      void enter(TABLE_entries &entry);
      // Adds entry to symbol table

      void search(char *name, TABLE_entries &entry, bool &found);
      // Searches table for presence of name.  If found then returns entry

      void printtable(FILE *lst);
      // Prints symbol table for diagnostic purposes
  };
```

An augmented parser must construct appropriate `entry` structures and enter these into the table when identifiers are first recognized by the routine that handles the productions for *OneConst* and *OneVar*. Before identifiers are accepted by the routines that handle the production for *Designator* they are checked against this table for non-declaration of `name`, abuse of `idclass` (such as trying to assign to a constant) and abuse of `scalar` (such as trying to subscript a constant, or a scalar variable). The interface suggested here is still inadequate for the purposes of code generation, so we delay further discussion of the symbol table itself until the next section.

However, we may take the opportunity of pointing out that the way in which the symbol table facilities are used depends rather critically on whether the parser is being crafted by hand, or by using a parser generator. We draw the reader's attention to the production for *Factor*, which we have written

```
  Factor = Designator | number | "(" Expression ")" .
```

rather than the more descriptive

```
  Factor = ConstIdentifier | Variable | number | "(" Expression ")" .
```

which does not satisfy the LL(1) constraints. In a hand-crafted parser we are free to use semantic information to drive the parsing process, and to break this LL(1) conflict, as the following extract from such a parser will show.

```
  void Factor(symset followers)
  // Factor = Variable | ConstIdentifier | Number | "(" Expression ")" .
  // Variable = Designator .
  { TABLE_entries entry;
    bool found;
    test(FirstFactor, followers, 14);              // Synchronize
    switch (SYM.sym)
    { case SCAN_identifier:
        Table->search(SYM.name, entry, found);     // Look it up
        if (!found) Report->error(202);            // Undeclared identifier
        if (entry.idclass = TABLE_consts) GetSym(); // ConstIdentifier
        else Designator(entry, followers, 206);     // Variable
        break;
      case SCAN_number:
        GetSym(); break;
      case SCAN_lparen:
        GetSym(); Expression(symset(SCAN_rparen) + followers);
        accept(SCAN_rparen, 17); break;
      default:                                      // Synchronized on a
        Report->error(14); break;                   // follower instead
    }
  }
```

In a Coco/R generated parser some other way must be found to handle the conflict. The generated parser will always set up a call to parse a *Designator*, and so the distinction must be drawn at that stage. The following extracts from an attributed grammar shows one possible way of doing this.

```
  Factor
  =                           (. int value; TABLE_entries entry; .)
       Designator<classset(TABLE_consts, TABLE_vars), entry>
     | Number<value>
     | "(" Expression ")" .
```

Notice that the *Designator* routine is passed the set of `idclasses` that are acceptable in this context. The production for *Designator* needs to check quite a number of context conditions:

```
  Designator<classset allowed, TABLE_entries &entry>
  =                           (. TABLE_alfa name;
                                 bool isvariable, found; .)
     Ident<name>              (. Table->search(name, entry, found);
                                 if (!found) SemError(202);
                                 if (!allowed.memb(entry.idclass)) SemError(206);
                                 isvariable = entry.idclass == TABLE_vars; .)
     ( "["                    (. if (!isvariable || entry.scalar) SemError(204); .)
       Expression "]"
       |                      (. if (isvariable && !entry.scalar) SemError(205); .)
     ) .
```

Other variations on this theme are possible. One of these is interesting in that it effectively uses semantic information to drive the parser, returning prematurely if it appears that a subscript should not be allowed:

```
  Designator<classset allowed, TABLE_entries &entry>
  =                           (. TABLE_alfa name;
                                 bool found; .)
     Ident<name>              (. Table->search(name, entry, found);
                                 if (!found) SemError(202);
                                 if (!allowed.memb(entry.idclass)) SemError(206);
                                 if (entry.idclass != TABLE_vars) return; .)
     ( "["                    (. if (entry.scalar) SemError(204); .)
       Expression "]"
       |                      (. if (!entry.scalar) SemError(205); .)
     ) .
```

As an example of how these ideas combine in the reporting of incorrect programs we present a source listing produced by the hand-crafted parser found on the source diskette:

```
   1 : PROGRAM Debug
   2 :   CONST
   2 :        ^; expected
   3 :     TooBigANumber = 328000;
   3 :                           ^Constant out of range
   4 :     Zero := 0;
   4 :            ^:= in wrong context
   5 :   VAR
   6 :     Valu, Smallest, Largest, Total;
   7 :   CONST
   8 :     Min = Zero;
   8 :               ^Number expected
   9 :   BEGIN
  10 :     Total := Zero;
  11 :     IF Valu THEN;
  11 :                 ^Relational operator expected
  12 :     READ (Valu); IF Valu > Min DO WRITE(Valu);
```

```
12 :                                  ^THEN expected
13 :     Largest := Valu; Smallest = Valu;
13 :                                ^:= expected
14 :     WHILE Valu <> Zero DO
15 :       BEGIN
16 :         Total := Total + Valu
17 :         IF Valu > = Largest THEN Largest := Value;
17 :           ^; expected
17 :                    ^Invalid factor
17 :                                                      ^Undeclared identifier
18 :         IF Valu < Smallest THEN Smallest := Valu;
19 :         READLN(Valu); IF Valu > Zero THEN WRITE(Valu)
19 :               ^Undeclared identifier
20 :       END;
21 :     WRITE('TOTAL:', Total, ' LARGEST:', Largest);
22 :     WRITE('SMALLEST: , Smallest)
22 :                                 ^Incomplete string
23 :   END.
23 :      ^) expected
```

Exercises

14.37 Submit the incorrect program given above to a Coco/R generated parser, and compare the quality of error reporting and recovery with that achieved by the hand-crafted parser.

14.38 At present the error messages for the hand-crafted system are reported one symbol *after* the point where the error was detected. Can you find a way of improving on this?

14.39 A disadvantage of the error recovery scheme used here is that a user may not realize which symbols have been skipped. Can you find a way to mark some or all of the symbols skipped by `test`? Has `test` been used in the best possible way to facilitate error recovery?

14.40 If, as we have done, all error messages after the first at a given point are suppressed, one might occasionally find that the quality of error message deteriorates - "early" messages might be less apposite than "later" messages might have been. Can you implement a better method than the one we have? (Notice that the *Followers* parameter passed to a sub-parser for *S* includes not only the genuine FOLLOW(*S*) symbols, but also further *Beacons*.)

14.41 If you study the code for the hand-crafted parser carefully you will realize that *Identifier* effectively appears in all the *Follower* sets? Is this a good idea? If not, what alterations are needed?

14.42 Although not strictly illegal, the appearance of a semicolon in a program immediately following a `DO` or `THEN`, or immediately preceding an `END` may be symptomatic of omitted code. Is it possible to warn the user when this has occurred, and if so, how?

14.43 The error reporter makes no real distinction between context-free and semantic or context-sensitive errors. Do you suppose it would be an improvement to try to do this, and if so, how could it be done?

14.44 Why does this parser not allow you to assign one array completely to another array? What modifications would you have to make to the context-free grammar to permit this? How would the constraint analysis have to be altered?

14.45 In Topsy - at least as it is used in the example program of Exercise 8.25 - all "declarations" seem to precede "statements". In C++ it is possible to declare variables at the point where they are first needed. How would you define Topsy to support the mingling of declarations and statements?

14.46 One school of thought maintains that in a statement like a Modula-2 `FOR` loop, the control variable should be implicitly declared at the start of the loop, so that it is truly local to the loop. It should also not be possible to alter the value of the control variable within the loop. Can you extend your parser and symbol table handler to support these ideas?

14.47 Exercises 14.21 through 14.36 suggested many syntactic extensions to Clang or Topsy. Extend your parsers so that they incorporate error recovery and constraint analysis for all these extensions.

14.48 Experiment with error recovery mechanisms that depend on the ordering of the `SCAN_symtypes` enumeration, as discussed in section 14.6.1. Can you find an ordering that works adequately for Topsy?

14.7 The symbol table handler

In an earlier section we claimed that it would be advantageous to split our compiler into distinct phases for syntax/constraint analysis and code generation. One good reason for doing this is to isolate the machine dependent part of compilation as far as possible from the language analysis. The degree to which we have succeeded may be measured by the fact that we have not yet made any mention of what sort of object code we are trying to generate.

Of course, any interface between source and object code must take cognizance of data-related concepts like *storage*, *addresses* and *data representation*, as well as control-related ones like *location counter, sequential execution* and *branch instruction*, which are fundamental to nearly all machines on which programs in our imperative high-level languages execute. Typically, machines allow some operations which simulate arithmetic or logical operations on data bit patterns which simulate numbers or characters, these patterns being stored in an array-like structure of *memory*, whose elements are distinguished by *addresses*. In high-level languages these addresses are usually given mnemonic names. The context-free syntax of many high-level languages, as it happens, rarely seems to draw a distinction between the "address" for a variable and the "value" associated with that variable, and stored at its address. Hence we find statements like

```
          X  :=  X + 4
```

in which the `X` on the left of the `:=` operator actually represents an address, (sometimes called the *L-value* of `X`) while the `X` on the right (sometimes called the *R-value* of `X`) actually represents the value of the quantity currently residing at the same address. Small wonder that mathematically trained beginners sometimes find the assignment notation strange! After a while it usually becomes second nature - by which time notations in which the distinction is made clearer possibly only confuse still further, as witness the problems beginners often have with pointer types in C++ or Modula-2, where `*P` or `P^` (respectively) denote the explicit value residing at the explicit address `P`. If we relate this back to the productions used in our grammar, we would find that each `X` in the above assignment was syntactically a *Designator*. Semantically these two designators are very different - we shall refer to the one that represents an address as a *Variable Designator*, and to the one that represents a value as a *Value Designator*.

To perform its task, the code generation interface will require the extraction of further information associated with user-defined identifiers and best kept in the symbol table. In the case of constants we need to record the associated values, and in the case of variables we need to record the associated addresses and storage demands (the elements of array variables will occupy a contiguous block of memory). If we can assume that our machine incorporates a "linear array" model of memory, this information is easily added as the variables are declared.

Handling the different sorts of entries that need to be stored in a symbol table can be done in various ways. In a object-oriented class-based implementation one might define an abstract base class to represent a generic type of entry, and then derive classes from this to represent entries for variables or constants (and, in due course, records, procedures, classes and any other forms of entry that seem to be required). The traditional way, still required if one is hosting a compiler in a language that does not support inheritance as a concept, is to make use of a variant record (in Modula-2 terminology) or union (in C++ terminology). Since the class-based implementation gives so much scope for exercises, we have chosen to illustrate the variant record approach, which is very efficient, and quite adequate for such a simple language. We extend the declaration of the `TABLE_entries` type to be

```
struct TABLE_entries {
  TABLE_alfa name;          // identifier
  TABLE_idclasses idclass;  // class
  union {
    struct {
      int value;
    } c;                    // constants
    struct {
      int size, offset;     // number of words, relative address
      bool scalar;          // distinguish arrays
    } v;                    // variables
  };
};
```

The way in which the symbol table is constructed can be illustrated with reference to the relevant parts of a Cocol specification for handling *OneConst* and *OneVar*:

```
OneConst
=                             (. TABLE_entries entry; .)
   Ident<entry.name>          (. entry.idclass = TABLE_consts; .)
   WEAK "="
   Number<entry.c.value> ";"  (. Table->enter(entry); .) .

OneVar<int &framesize>
=                             (. TABLE_entries entry;
                                 entry.idclass = TABLE_vars;
                                 entry.v.size = 1; entry.v.scalar = true;
                                 entry.v.offset = framesize + 1; .)
   Ident<entry.name>
   [ UpperBound<entry.v.size> (. entry.v.scalar = false; .)
   ]                          (. Table->enter(entry);
                                 framesize += entry.v.size; .) .

UpperBound<int &size>
=  "[" Number<size> "]"       (. size++; .) .

Ident<char *name>
=  identifier                 (. LexName(name, TABLE_alfalength); .) .
```

Here `framesize` is a simple count, which is initialized to zero at the start of parsing a *Block*. It keeps track of the number of variables declared, and also serves to define the addresses which these variables will have relative to some known location in memory when the program runs. A trivial modification gets around the problem if it is impossible or inconvenient to use zero-based addresses in the real machine.

Programming a symbol table handler for a language as simple as ours can be correspondingly simple. On the source diskette can be found such implementations, based on the idea that the symbol table can be stored within a fixed length array. A few comments on implementation techniques will guide the reader who wishes to study this code:

- The table is set up so that the entry indexed by zero can be used as a sentinel in a simple sequential search by `search`. Although this is inefficient, it is adequate for prototyping the system.

- A call to `Table->search(name, entry, found)` will always return with a well defined value for `entry`, even if the `name` had never been declared. Such undeclared identifiers will seem to have an effective `idclass = TABLE_progs`, which will be semantically unacceptable everywhere, thus ensuring that incorrect code can never be generated.

Exercises

14.49 How would you check that no identifier is declared more than once?

14.50 Identifiers that are undeclared by virtue of mistyped declarations tend to be annoying, for they result in many subsequent errors being reported. Perhaps in languages as simple as ours one could assume that all undeclared identifiers should be treated as variables, and entered as such in the symbol table at the point of first reference. Is this a good idea? Can it easily be implemented? What happens if arrays are undeclared?

14.51 Careful readers may have noticed that a Clang array declaration is different from a C++ one - the bracketed number in Clang specifies the highest permitted index value, rather than the array length. This has been done so that one can declare variables like

```
        VAR Scalar, List[10], VeryShortList[0];
```

How would you modify Clang and Topsy to use C++ semantics, where the declaration of `VeryShortList` would have to be forbidden?

14.52 The names of identifiers are held within the symbol table as fixed length strings, truncated if necessary. It may seem unreasonable to expect compilers (especially written in Modula-2 or Pascal, which do not have dynamic strings as standard types) to cater for identifiers of any length, but too small a limitation on length is bound to prove irksome sooner or later, and too generous a limitation simply wastes valuable space when, as so often happens, users choose very short names. Develop a variation on the symbol table handler that allocates the name fields dynamically, to be of the correct size. (This can, of course, also be done in Modula-2.) Making table entries should be quite simple; searching for them may call for a little more ingenuity.

14.53 A simple sequential search algorithm is probably perfectly adequate for the small Clang programs that one is likely to write. It becomes highly inefficient for large applications. It is far more efficient to store the table in the form of a binary search tree, of the sort that you may have encountered in other courses in Computer Science. Develop such an implementation, noting that it should not be necessary to alter the public interface to the table class.

14.54 Yet another approach is to construct the symbol table using a hash table, which probably yields the shortest times for retrievals. Hash tables were briefly discussed in Chapter 7, and should also be familiar from other courses you may have taken in Computer Science. Develop a hash table implementation for your Clang or Topsy compiler.

14.55 We might consider letting the scanner interact with the symbol table. Consider the implications of developing a scanner that stores the strings for identifiers and string literals in a string table, as suggested in Exercise 6.6 for the assemblers of Chapter 6.

14.56 Develop a symbol table handler that utilizes a simple class hierarchy for the possible types of entries, inheriting appropriately from a suitable base class. Once again, construction of such a table should prove to be straightforward, regardless of whether you use a linear array, tree, or hash table as the underlying storage structure. Retrieval might call for more ingenuity, since C++ does not provide syntactic support for determining the exact class of an object that has been statically declared to be of a base class type.

14.8 Other aspects of symbol table management - further types

It will probably not have escaped the reader’s attention, especially if he or she has attempted the exercises in the last few sections, that compilers for languages which handle a wide variety of types, both "standard" and "user defined", must surely take a far more sophisticated approach to constructing a symbol table and to keeping track of storage requirements than anything we have seen so far. Although the nature of this text does not warrant a full discussion of this point, a few comments may be of interest, and in order.

In the first place, a compiler for a block-structured language will probably organize its symbol table as a collection of dynamically allocated trees, with one root for each level of nesting. Although using simple binary trees runs the risk of producing badly unbalanced trees, this is unlikely. Except for source programs which are produced by program generators, user programs tend to introduce identifiers with fairly random names; few compilers are likely to need really sophisticated tree constructing algorithms.

Secondly, the nodes in the trees will be fairly complex record structures. Besides the obvious links to other nodes in the tree, there will probably be pointers to other dynamically constructed nodes, which contain descriptions of the types of the identifiers held in the main tree.

Thus a Pascal declaration like

```
    VAR
      Matrix : ARRAY [1 .. 10, 2 .. 20] OF SET OF CHAR;
```

might result in a structure that can be depicted something like that of Figure 14.2.

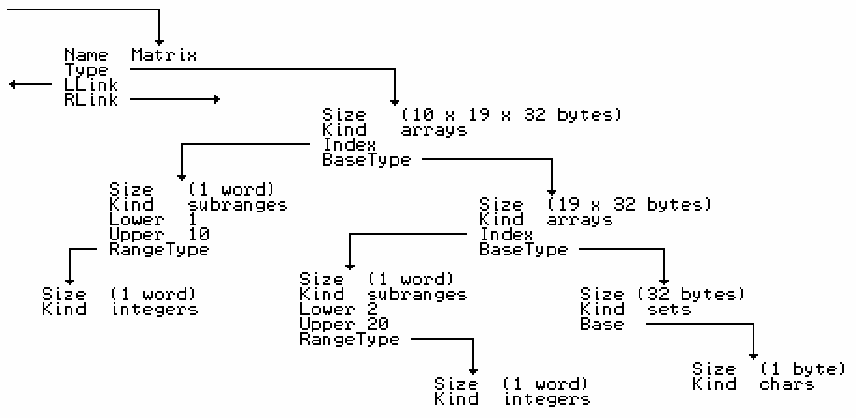


Figure 14.2 Symbol table entry for
VAR Matrix : ARRAY [1 .. 10, 2 .. 20] OF SET OF CHAR;

We may take this opportunity to comment on a rather poorly defined area in Pascal, one that came in for much criticism. Suppose we were to declare

```
  TYPE
    LISTS = ARRAY [1 .. 10] OF CHAR;
  VAR
    X : LISTS;
    A : ARRAY [1 .. 10] OF CHAR;
    B : ARRAY [1 .. 10] OF CHAR;
    Z : LISTS;
```

`A` and `B` are said to be of **anonymous type**, but to most people it would seem obvious that `A` and `B` are of the same type, implying that an assignment of the form `A := B` should be quite legal, and, furthermore, that `X` and `Z` would be of the same type as `A` and `B`. However, some compilers will be satisfied with mere **structural equivalence** of types before such an assignment would be permitted, while others will insist on so-called **name equivalence**. The original Pascal Report did not specify which was standard.

In this example `A, B, X` and `Z` all have structural equivalence. `X` and `Z` have name equivalence as well, as they have been specified in terms of a named type `LISTS`.

With the insight we now have we can see what this difference means from the compiler writer’s viewpoint. Suppose `A` and `B` have entries in the symbol table pointed to by `ToA` and `ToB` respectively. Then for name equivalence we should insist on `ToA^.Type` and `ToB^.Type` being the same (that is, their `Type` pointers address the same descriptor), while for structural equivalence we should insist on `ToA^.Type^` and `ToA^.Type^` being the same (that is, their `Type` pointers address descriptors that have the same structure).

Further reading and exploration

We have just touched on the tip of a large and difficult iceberg. If one adds the concept of types and type constructors into a language, and insists on strict type-checking, the compilers become much larger and harder to follow than we have seen up till now. The energetic reader might like to follow up several of the ideas which should now come to mind. Try a selection of the following, which are deliberately rather vaguely phrased.

14.57 Just how do real compilers deal with symbol tables?

14.58 Just how do real compilers keep track of type checking? Why should name equivalence be easier to handle than structural equivalence?

14.59 Why do some languages simply forbid the use of "anonymous types", and why don’t more languages forbid them?

14.60 How do you suppose compilers keep track of storage allocation for `struct` or `RECORD` types, and for `union` or variant record types?

14.61 Find out how storage is managed for dynamically allocated variables in language like C++, Pascal, or Modula-2.

14.62 How does one cope with arrays of variable (dynamic) length in subprograms?

14.63 Why can we easily allow the declaration of a pointer type to precede the definition of the type it points to, even in a one-pass system? For example, in Modula-2 we may write

```
TYPE
  LINKS = POINTER TO NODES (* NODES not yet seen *);
  NODES = RECORD
            ETC  : JUNK;
            Link : LINKS;
            . . .
```

14.64 Brinch Hansen did not like the Pascal subrange type because it seems to lead to ambiguities (for example, a value of `34` can be of type `0 .. 45`, and also of type `30 .. 90` and so on), and so omitted them from Edison. Interestingly, a later Wirth language, Oberon, omits them as well. How might Pascal and Modula-2 otherwise have introduced the subrange concept, how could we overcome Brinch Hansen’s objections, and what is the essential point that he seems to have overlooked in discarding them?

14.65 One might accuse the designers of Pascal, Modula-2 and C of making a serious error of judgement - they do not introduce a string type as standard, but rely on programmers to manipulate arrays of characters, and to use error prone ways of recognizing the end, or the length, of a string. Do you agree? Discuss whether what they offer in return is adequate, and if not, why not. Suggest why they might deliberately not have introduced a string type.

14.66 Brinch Hansen did not like the Pascal variant record (or `union`). What do such types allow one to do in Pascal which is otherwise impossible, and why should it be necessary to provide the facility to do this? How else are these facilities catered for in Modula-2, C and C++? Which is the better way, and why? Do the ideas of type extension, as found in Oberon, C++ and other "object oriented" languages provide even better alternatives?

14.67 Many authors dislike pointer types because they allow "insecure" programming". What is meant by this? How could the security be improved? If you do not like pointer types, can you think of any alternative feature that would be more secure?

There is quite a lot of material available on these subjects in many of the references cited previously. Rather than give explicit references, we leave the Joys of Discovery to the reader.