6 SIMPLE ASSEMBLERS

In this chapter we shall be concerned with the implementation of simple assembler language translator programs. We assume that the reader already has some experience in programming at the assembler level; readers who do not will find excellent discussions of this topic in the books by Wakerly (1981) and MacCabe (1993). To distinguish between programs written in "assembler code", and the "assembler program" which translates these, we shall use the convention that ASSEMBLER means the language and "assembler" means the translator.

The basic purpose of an assembler is to translate ASSEMBLER language mnemonics into binary or hexadecimal machine code. Some assemblers do little more than this, but most modern assemblers offer a variety of additional features, and the boundary between assemblers and compilers has become somewhat blurred.

6.1 A simple ASSEMBLER language

Rather than use an assembler for a real machine, we shall implement one for a rudimentary ASSEMBLER language for the hypothetical single-accumulator machine discussed in section 4.3.

An example of a program in our proposed language is given below, along with its equivalent object code. We have, as is conventional, used hexadecimal notation for the object code; numeric values in the source have been specified in decimal.

```
Assembler 1.0 on 01/06/96 at 17:40:45

00                   BEG                     ; count the bits in a number
00    0A             INI                     ; Read(A)
01          LOOP                             ; REPEAT
01    16             SHR                     ;  A := A DIV 2
02    3A 0D          BCC     EVEN            ;  IF A MOD 2 # 0 THEN
04    1E 13          STA     TEMP            ;    TEMP := A
06    19 14          LDA     BITS
08    05             INC
09    1E 14          STA     BITS            ;    BITS := BITS + 1
0B    19 13          LDA     TEMP            ;    A := TEMP
0D    37 01 EVEN     BNZ     LOOP            ; UNTIL A = 0
0F    19 14          LDA     BITS            ;
11    0E             OTI                     ; Write(BITS)
12    18             HLT                     ; terminate execution
13          TEMP     DS      1               ; VAR TEMP : BYTE
14    00    BITS     DC      0               ;     BITS : BYTE
15                   END
```

ASSEMBLER programs like this usually consist of a sequence of statements or instructions, written one to a line. These statements fall into two main classes.

Firstly, there are the **executable instructions** that correspond directly to executable code. These can be recognized immediately by the presence of a distinctive **mnemonic** for an **opcode**. For our machine these executable instructions divide further into two classes: there are those that require an **address** or operand as part of the instruction (as in `STA TEMP`) and occupy two bytes of object code, and there are those that stand alone (like `INI` and `HLT`). When it is necessary to refer to such statements elsewhere, they may be labelled with an introductory distinctive **label** identifier of the programmer's choice (as in `EVEN BNZ LOOP`), and may include a **comment**, extending from an introductory semicolon to the end of a line.

The address or operand for those instructions that requires them is denoted most simply by either a numeric literal, or by an identifier of the programmer's choice. Such identifiers usually correspond to the ones that are used to label statements - when an identifier is used to label a statement itself we speak of a **defining occurrence** of a label; when an identifier appears as an address or operand we speak of an **applied occurrence** of a label.

The second class of statement includes the **directives**. In source form these appear to be deceptively similar to executable instructions - they are often introduced by a label, terminated with a comment, and have what may appear to be mnemonic and address components. However, directives have a rather different role to play. They do not generally correspond to operations that will form part of the code that is to be executed at *run-time*, but rather denote actions that direct the action of the assembler at *compile-time* - for example, indicating where in memory a block of code or data is to be located when the object code is later loaded, or indicating that a block of memory is to be preset with literal values, or that a name is to be given to a literal to enhance readability.

For our ASSEMBLER we shall introduce the following directives and their associated compile-time semantics, as a representative sample of those found in more sophisticated languages:

```
   Label Mnemonic   Address    Effect

  not used   BEG    not used   Mark the beginning of the code
  not used   END    not used   Mark the end of the code
  not used   ORG    location   Specify location where the following code
                               is to be loaded
  optional   DC     value      Define an (optionally labelled) byte,
                               to have a specified initial value
  optional   DS     length     Reserve length bytes (optional label associated
                               with the first byte)
  name       EQU    value      Set name to be a synonym for the given value
```

Besides lines that contain a full statement, most assemblers usually permit incomplete lines. These may be completely blank (so as to enhance readability), or may contain only a label, or may contain only a comment, or may contain only a label and a comment.

Our first task might usefully be to try to find a grammar that describes this (and similar) programs. This can be done in several ways. Our informal description has already highlighted various syntactic classes that will be useful in specifying the phrase structure of our programs, as well as various token classes that a scanner may need to recognize as part of the assembly process. One possible grammar - which leaves the phrase structure very loosely defined - is given below. This has been expressed in Cocol, the EBNF variant introduced in section 5.9.5.

```
  COMPILER ASM

    CHARACTERS
      eol        = CHR(13) .
      letter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
      digit      = "0123456789" .
      printable  = CHR(32) .. CHR(127) .

    IGNORE  CHR(9) .. CHR(12)

    TOKENS
      number     = digit { digit } .
      identifier = letter { letter | digit } .
      EOL        = eol .
      comment    = ";" { printable } .

    PRODUCTIONS
      ASM        = { Statement } EOF .
      Statement  = [ Label ] [ Mnemonic [ Address ] ] [ comment ] EOL .
      Address    = Label | number .
      Mnemonic   = identifier .
      Label      = identifier .
  END ASM.
```

This grammar has the advantage of simplicity, but makes no proper distinction between directives and executable statements, nor does it indicate which statements *require* labels or address fields. It is possible to draw these distinctions quite easily if we introduce a few more non-terminals into the phrase structure grammar:

```
COMPILER ASM

  CHARACTERS
    eol        = CHR(13) .
    letter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
    digit      = "0123456789" .
    printable  = CHR(32) .. CHR(127) .

  IGNORE  CHR(9) .. CHR(12)

  TOKENS
    number     = digit { digit } .
    identifier = letter { letter | digit } .
    EOL        = eol .
    comment    = ";" { printable } .

  PRODUCTIONS
    ASM               = StatementSequence "END" EOF .
    StatementSequence = { Statement [ comment ] EOL } .
    Statement         = Executable | Directive .
    Executable        = [ Label ] [ OneByteOp | TwoByteOp Address ] .
    OneByteOp         = "HLT" | "PSH" | "POP"  (* | . . . . etc *) .
    TwoByteOp         = "LDA" | "LDX" | "LDI"  (* | . . . . etc *) .
    Address           = Label | number .
    Directive         =   Label "EQU" KnownAddress
                        | [ Label ] ( "DC" Address | "DS" KnownAddress )
                        | "ORG" KnownAddress | "BEG" .
    Label             = identifier .
    KnownAddress      = Address .
END ASM.
```

When it comes to developing a practical assembler, the first of these grammars appears to have the advantage of simplicity so far as syntax analysis is concerned - but this simplicity comes at a price, in that the static semantic constrainer would have to expend effort in distinguishing the various statement forms from one another. An assembler based on the second grammar would not leave so much to the semantic constrainer, but would apparently require a more complex parser. In later sections, using the simpler description as the basis of a parser, we shall see how both it and the constrainer are capable of development in an *ad hoc* way.

Neither of the above syntactic descriptions illustrates some of the pragmatic features that may beset a programmer using the ASSEMBLER language. Typical of these are restrictions or relaxations on case-sensitivity of identifiers, or constraints that labels may have to appear immediately at the start of a line, or that identifiers may not have more than a limited number of significant characters. Nor, unfortunately, can the syntactic description enforce some essential static semantic constraints, such as the requirement that each alphanumeric symbol used as an address should also occur uniquely in a label field of an instruction, or that the values of the address fields that appear with directives like `DS` and `ORG` must have been defined before the corresponding directives are first encountered. The description may *appear* to enforce these so-called *context-sensitive* features of the language, because the non-terminals have been given suggestive names like `KnownAddress`, but it turns out that a simple parser will not be able to enforce them on its own.

As it happens, neither of these grammars yet provides an adequate description for a compiler generator like Coco/R, for reasons that will become apparent after studying Chapter 9. The modifications needed for driving Coco/R may be left as an interesting exercise when the reader has had more experience in parsing techniques.

6.2 One- and two-pass assemblers, and symbol tables

Readers who care to try the assembly translation process for themselves will realize that this cannot easily be done on a single pass through the ASSEMBLER source code. In the example given earlier, the instruction

```
          BCC   EVEN
```

cannot be translated completely until one knows the address of `EVEN`, which is only revealed when the statement

```
   EVEN   BNZ   LOOP
```

is encountered. In general the process of assembly is always non-trivial, the complication arising - even with programs as simple as this one - from the inevitable presence of **forward references**.

An assembler may solve these problems by performing two distinct passes over the user program. The primary aim of the first pass of a **two-pass assembler** is to draw up a **symbol table**. Once the first pass has been completed, all necessary information on each user defined identifier should have been recorded in this table. A second pass over the program then allows full assembly to take place quite easily, referring to the symbol table whenever it is necessary to determine an address for a named label, or the value of a named constant.

The first pass can perform other manipulations as well, such as some error checking. The second pass depends on being able to rescan the program, and so the first pass usually makes a copy of this on some backing store, usually in a slightly altered form from the original.

The behaviour of a two-pass assembler is summarized in Figure 6.1.

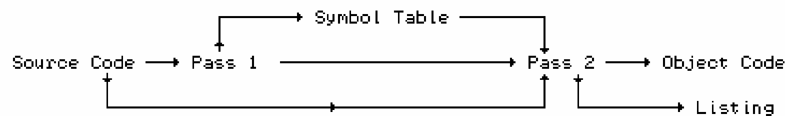

Figure 6.1 Flow of information through a two-pass assembler

The other method of assembly is via a **one-pass assembler**. Here the source is scanned but once, and the construction of the symbol table is rather more complicated, since outstanding references must be recorded for later *fixup* or *backpatching* once the appropriate addresses or values are revealed. In a sense, a two-pass assembler may be thought of as making two passes over the source program, while a one-pass assembler makes a single pass over the source program, followed by a later partial pass over the object program.

As will become clear, construction of a sophisticated assembler, using either approach, calls for a fair amount of ingenuity. In what follows we shall illustrate several principles rather simply and naïvely, and leave the refinements to the interested reader in the form of exercises.

Assemblers all make considerable use of tables. There are always (conceptually at least) two of these:

- The *Opcode Translation Table*. In this will be found matching pairs of mnemonics and their numerical equivalents. This table is of fixed length in simple assemblers.

- The *Symbol Table*. In this will be entered the user defined identifiers, and their corresponding addresses or values. This table varies in length with the program being assembled.

Two other commonly found tables are:

- The *Directive Table*. In this will be found mnemonics for the directives or pseudo-operations. The table is of fixed length, and is usually incorporated into the opcode translation table in simple assemblers.

- The *String Table*. As a space saving measure, the various user-defined names are often gathered into one closely packed table - effectively being stored in one long string, with some distinctive separator such as a `NUL` character between each sub-string. Each identifier in the symbol table is then cross-linked to this table. For example, for the program given earlier we might have a symbol table and string table as shown in Figure 6.2.

Name	Address or Value	String table position
BITS	14 (hex) 20 (decimal)	0
TEMP	13 (hex) 19 (decimal)	5
EVEN	0D (hex) 13 (decimal)	10
LOOP	01 (hex) 1 (decimal)	15

B	I	T	S	■	T	E	M	P	■	E	V	E	N	■	L	O	O	P	■
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Figure 6.2 Symbol table and string table for a simple assembler program

More sophisticated macro-assemblers need several other tables, so as to be able to handle user-defined opcodes, their parameters, and the source text which constitutes the definition of each macro. We return to a consideration of this point in the next chapter.

The first pass, as has been stated, has as its primary aim the creation of a symbol table. The "name" entries in this are easily made as the label fields of the source are read. In order to be able to complete the "address" entries, the first pass has to keep track, as it scans the source, of the so-called **location counter** - that is, the address at which each code and data value will later be located (when the code generation takes place). Such addresses are controlled by the directives `ORG` and `DS` (which affect the location counter explicitly), as well as by the directive `DC`, and, of course, by the opcodes which will later result in the creation of one or two machine words. The directive `EQU` is a special case; it simply gives a naming facility.

Besides constructing the symbol table, this pass must supervise source handling, and lexical, syntactic and semantic analysis. In essence it might be described by something on the lines of the following, where, we hasten to add, considerable liberties have been taken with the pseudo-code used to express the algorithm.

```
Initialize tables, and set Assembling := TRUE; Location := 0;
WHILE Assembling DO
  Read line of source and unpack into constituent fields
     Label, Mnemonic, AddressField (* which could be a Name or Number *)
  Use Mnemonic to identify Opcode from OpTable
  Copy line of source to work file for later use by pass two
  CASE Mnemonic OF
    "BEG" : Location := 0
    "ORG" : Location := AddressField.Number
    "DS " : IF Line.Labelled THEN SymbolTable.Enter(Label, Location)
            Location := Location + AddressField.Number
    "EQU" : SymbolTable.Enter(Label, AddressField.Number)
    "END" : Assembling := FALSE
    all others (* including DC *):
            IF Line.Labelled THEN SymbolTable.Enter(Label, Location)
            Location := Location + number of bytes to be generated
  END
```

```
END
```

The second pass is responsible mainly for code generation, and may have to repeat some of the source handling and syntactic analysis.

```
Rewind work file, and set Assembling := TRUE
WHILE Assembling DO
  Read a line from work file and unpack Mnemonic, Opcode, AddressField
  CASE Mnemonic OF
    "BEG" : Location := 0
    "ORG" : Location := AddressField.Number
    "DS " : Location := Location + AddressField.Number
    "EQU" : no action (* EQU dealt with on pass one *)
    "END" : Assembling := FALSE
    "DC " : Mem[Location] := ValueOf(AddressField); INC(Location)
    all others:
            Mem[Location] := Opcode;  INC(Location)
            IF two-byte Opcode THEN
              Mem[Location] := ValueOf(AddressField); INC(Location)
            END
  END
  Produce source listing of this line
END
```

6.3 Towards the construction of an assembler

The ideas behind assembly may be made clearer by slowly refining a simple assembler for the language given earlier, allowing only for the creation of fixed address, as opposed to relocatable code. We shall assume that the assembler and the assembled code can co-reside in memory. We are confined to write a cross-assembler, not only because no such real machine exists, but also because the machine is far too rudimentary to support a resident assembler - let alone a large C++ or Modula-2 compiler.

In C++ we can define a general interface to the assembler by introducing a class with a public interface on the lines of the following:

```
class AS {
  public:
    void assemble(bool &errors);
    // Assembles and lists program.
    // Assembled code is dumped to file for later interpretation, and left
    // in pseudo-machine memory for immediate interpretation if desired.
    // Returns errors = true if assembly fails

    AS(char *sourcename, char *listname, char *version, MC *M);
    // Instantiates version of the assembler to process sourcename, creating
    // listings in listname, and generating code for associated machine M
};
```

This public interface allows for the development of a variety of assemblers (simple, sophisticated, single-pass or multi-pass). Of course there are private members too, and these will vary somewhat depending on the techniques used to build the assembler. The constructor for the class creates a link to an instance of a machine class `MC` - we are aiming at the construction of an assembler for our hypothetical single-accumulator machine that will leave assembled code in the pseudo-machine's memory, where it can be interpreted as we have already discussed in Chapter 4. The main program for our system will essentially be developed on the lines of the following code:

```
void main(int argc, char *argv[])
{ bool errors;
  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 assembler components
    MC *Machine   = new MC();
    AS *Assembler = new AS(SourceName, ListName, "Assembler version 1", Machine);

    // start assembly
    Assembler->assemble(errors);

    // examine outcome and interpret if possible
    if (errors) { printf("\nAssembly failed\n"); }
    else { printf("\nAssembly successful\n"); Machine->interpret(); }
    delete Machine;
    delete Assembler;
  }
```

This driver routine has made provision for extracting the file names for the source and listing files from command line parameters set up when the assembler program is invoked.

In using a language like C++ or Modula-2 to implement the assembler (or rather assemblers, since we shall develop both one-pass and two-pass versions of the assembler class), it is convenient to create classes or modules to be responsible for each of the main phases of the assembly process. In keeping with our earlier discussion we shall develop a source handler, scanner, and simple parser. In a two-pass assembler the parser is called from a first pass that follows parsing with static semantic analysis; control then passes to the second pass that completes code generation. In a one-pass assembler the parser is called in combination with semantic analysis and code generation.

On the source diskette that accompanies this book can be found a great deal of code illustrating this development, and the reader is urged to study this as he or she reads the text, since there is too much code to justify printing it all in this chapter. Appendix D contains a complete listing of the source code for the assembler as finally developed by the end of the next chapter.

6.3.1 Source handling

In terms of the overall translator structure illustrated in Figure 2.4, the first phase of an assembler will embrace the source character handler, which scans the source text, and analyses it into lines, from which the scanner will be then able to extract tokens or symbols. The public interface to a class for handling this phase might be:

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

      void nextch(void);
      // Returns ch as the next character on current 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

      bool startline(void);
      // Returns true if current ch is the first on a line

      void writehex(int i, int n);
      // Writes (byte valued) i to lst file as hex pair, left-justified in n spaces

      void writetext(char *s, int n);
      // Writes s to lst file, left-justified in n spaces

      SH();
      // Default constructor

      SH(char *sourcename, char *listname, char *version);
      // Opens src and lst files using given names
      // Initializes source handler, and displays version information on lst file

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

Some aspects of this interface deserve further comment:

- It is probably bad practice to declare variables like `ch` as public, as this leaves them open to external abuse. However, we have compromised here in the interests of efficiency.

- Client routines (like those which call `nextch`) should not have to worry about anything other than the values provided by `ch`, `startline()` and `endline()`. The main client routine is, of course, the lexical analyser.

- Little provision has been made here for producing a source listing, other than to export the file on which the listing might be made, and the mechanism for writing some version information and hexadecimal values to this file. A source line might be listed immediately it is read, but in the case of a two-pass assembler the listing is usually delayed until the second pass, when it can be made more complete and useful to the user. Furthermore, a free-format input can be converted to a fixed-format output, which will probably look considerably better.

The implementation of this class is straightforward and can be studied in Appendix D. As with the interface, some aspects of the implementation call for comment:

- `nextch` has to provide for situations in which it might be called after the input file has been exhausted. This situation should only arise with erroneous source programs, of course.

- Internally the module stores the source on a line-buffered basis, and adds a blank character to the end of each `line` (or a `NUL` character in the case where the source has ended). This is useful for ensuring that a symbol that extends to the end of a line can easily be recognized.

Exercises

6.1 A source handler implemented in this way will be found to be very slow on many systems, where each call to a routine to read a single character may involve a call to an underlying operating system. Experiment with the idea that the source handler first reads the entire source into a large memory buffer in one fell swoop, and then returns characters by extracting them from this buffer. Since memory (even on microcomputers) now tends to be measured in megabytes, while source programs are rather small, this idea is usually quite feasible. Furthermore, this suggestion overcomes the problem of using a line buffer of restricted size, as is used in our simple implementation.

6.3.2 Lexical analysis

The next phase to be tackled is that of lexical analysis. For our simple ASSEMBLER language we recognize immediately that source characters can only be assembled into numbers, alphanumeric names (as for labels or opcodes) or comment strings. Accordingly we adopt the following public interface to our scanner class:

```
enum LA_symtypes {
  LA_unknown, LA_eofsym, LA_eolsym, LA_idsym, LA_numsym, LA_comsym
};


struct LA_symbols {
  bool islabel;          // if in first column
  LA_symtypes sym;       // class
```

```
    ASM_strings str;         // lexeme
    int num;                 // value if numeric
  };


  class LA {
    public:
      void getsym(LA_symbols &SYM, bool &errors);
      // Returns the next symbol on current source line.
      // Sets errors if necessary and returns SYM.sym = unknown if no
      // valid symbol can be recognized

      LA(SH *S);
      // Associates scanner with its source handler S
  };
```

where we draw the reader's attention to the following points:

- The `LA_symbols` structure allows the client to recognize that the first symbol found on a line has defined a label if it began in the very first column of the line - a rather messy feature of our ASSEMBLER language.

- In ASSEMBLER programs, the ends of lines become significant (which is not the case with languages like C++, Pascal or Modula-2), so that it is useful to introduce `LA_eolsym` as a possible symbol type.

- Similarly, we must make provision for not being able to recognize a symbol (by returning `LA_unknown`), or not finding a symbol (`LA_eofsym`).

Developing the `getsym` routine for the recognition of these symbols is quite easy. It is governed essentially by the lexical grammar (defined in the `TOKENS` section of our Cocol specification given earlier), and is sensibly driven by a `switch` or `CASE` statement that depends on the first character of the token. The essence of this - again taking considerable liberties with syntax - may be expressed

```
BEGIN
  skip leading spaces, or to end of line
  recognize end-of-line and start-of-line conditions, else
  CASE CH OF
    letters: SYM.Sym := LA_idsym;  unpack word;
    digits : SYM.Sym := LA_numsym; unpack number;
    ';'    : SYM.Sym := LA_comsym; unpack comment;
    ELSE   : SYM.Sym := LA_unknown
  END
END
```

A detailed implementation may be found on the source diskette. It is worth pointing out the following:

- All fields (attributes) of `SYM` are well defined after a call to `getsym`, even those of no immediate interest.

- While determining the value of `SYM.num` we also copy the digits into `SYM.name` for the purposes of later listing. At this stage we have assumed that overflow will not occur in the computation of `SYM.num`.

- Identifiers and comments that are too long are ruthlessly truncated.

- Identifiers are converted to upper case for consistency. Comments are preserved unchanged.

Exercises

6.2 First extend the lexical grammar, and then extend the lexical analyser to allow hexadecimal constants as alternatives in addresses, for example

```
LAB  LDI  $0A       ; 0A(hex) = 10(decimal)
```

6.3 Another convention is to allow hexadecimal constants like 0FFh or 0FFH, with the trailing H implying hexadecimal. A hex number must, however, start with a digit in the range '0' .. '9', so that it can be distinguished from an identifier. Extend the lexical grammar, and then implement this option. Why is it harder to handle than the convention suggested in Exercise 6.2?

6.4 Extend the grammar and the analyser to allow a single character as an operand or address, for example

```
LAB    LDI  'A'       ; load immediate 'A' (ASCII 041H)
```

The character must, of course, be converted into the corresponding ordinal value by the assembler. How can one allow the quote character itself to be used as an address?

6.5 If the entire source of the program were to be read into memory as suggested in Exercise 6.1 it would no longer be necessary to copy the `name` field for each symbol. Instead, one could use two numeric fields to record the starting position and the length of each name. Modify the lexical analyser to use such a technique. Clearly this will impact the detailed implementation of some later phases of assembly as well - see Exercise 6.8.

6.6 As an alternative to storing the entire source program in memory, explore the possibility of constructing a string table on the lines of that discussed in section 6.2.

6.3.3 Syntax analysis

Our suggested method of syntax analysis requires that each free format source line be decomposed in a consistent way. A suitable public interface for a simple class that handles this phase is given below:

```
enum SA_addresskinds { SA_absent, SA_numeric, SA_alphameric };

struct SA_addresses {
  SA_addresskinds kind;
  int number;     // value if known
  ASM_alfa name;  // character representation
};

struct SA_unpackedlines {
  // source text, unpacked into fields
  bool labelled, errors;
  ASM_alfa labfield, mnemonic;
  SA_addresses address;
  ASM_strings comment;
};

class SA {
  public:
    void parse(SA_unpackedlines &srcline);
    // Analyses the next source line into constituent fields

    SA(LA * L);
    // Associates syntax analyser with its lexical analyser L
};
```

and, as before, some aspects of this deserve further comment:

- The `SA_addresses` structure has been introduced to allow for later extensibility.

- The `SA_unpackedlines` structure makes provision for recording whether a source line has been labelled. It also makes provision for recording that the line is erroneous. Some errors might be detected when the syntax analysis is performed; others might only be detected when the constraint analysis or code generation are attempted.

- Not only does syntax analysis in the first pass of a two-pass assembler require that we unpack a source line into its constituent fields, using the `getsym` routine, the first pass also has to be able to write the source line information to a work file for later use by the second pass. It is convenient to do this *after* unpacking, to save the necessity of re-parsing the source on the second pass.

The routine for unpacking a source line is relatively straightforward, but has to allow for various combinations of present or absent fields. The syntax analyser can be programmed by following the EBNF productions given in Cocol under the `PRODUCTIONS` section of the simpler grammar in section 6.1, and the implementation on the source diskette is worthy of close study, bearing in mind the following points:

- The analysis is very *ad hoc*. This is partly because it has to take into account the possibility of errors in the source. Later in the text we shall look at syntax analysis from a rather more systematic perspective, but it is usually true that syntax analysers incorporate various messy devices for side-stepping errors.

- Every field is well defined when analysis is complete - default values are inserted where they are not physically present in the source.

- Should the source text become exhausted, the syntax analyser performs "error correction", effectively by creating a line consisting only of an `END` directive.

- When an unrecognizable symbol is detected by the scanner, the syntax analyser reacts by recording that the line is in error, and then copies the rest of the line to the `comment` field. In this way it is still possible to list the offending line in some form at a later stage.

- The simple routine for `getaddress` will later be modified to allow expressions as addresses.

Exercises

6.7 At present mnemonics and user defined identifiers are both handled in the same way. Perhaps a stronger distinction should be drawn between the two. Then again, perhaps one should allow mnemonics to appear in address fields, so that an instruction like

```
LAB   LDI   LDI      ;  A := 27
```

would become legal. What modifications to the underlying grammar and to the syntax analyser would be needed to implement any ideas you may have on these issues?

6.8 How would the syntax analyser have to be modified if we were to adopt the suggestion that all the source code be retained in memory during the assembly process? Would it be necessary to unpack each line at all?

6.3.4 The symbol table interface

We define a clean public interface to a symbol table handler, thus allowing us to implement various strategies for symbol table construction without disturbing the rest of the system. The interface chosen is

```
typedef void (*ST_patch)(MC_bytes mem[], MC_bytes b, MC_bytes v);

class ST {
  public:
    void printsymboltable(bool &errors);
    // Summarizes symbol table at end of assembly, and alters errors
    // to true if any symbols have remained undefined

    void enter(char *name, MC_bytes value);
    // Adds name to table with known value

    void valueofsymbol(char *name, MC_bytes location, MC_bytes &value,
                       bool &undefined);
    // Returns value of required name, and sets undefined if not found.
    // location is the current value of the instruction location counter

    void outstandingreferences(MC_bytes mem[], ST_patch fix);
    // Walks symbol table, applying fix to outstanding references in mem

    ST(SH *S);
    // Associates table handler with source handler S (for listings)
};
```

6.4 Two-pass assembly

For the moment we shall focus attention on a two-pass assembler, and refine the code from the simple algorithms given earlier. The first pass is mainly concerned with static semantics, and with constructing a symbol table. To be able to do this, it needs to keep track of a location counter, which is updated as opcodes are recognized, and which may be explicitly altered by the directives `ORG`, `DS` and `DC`.

6.4.1 Symbol table structures

A simple implementation of the symbol table handler outlined in the last section, suited to two-pass assembly, is to be found on the source diskette. It uses a dynamically allocated stack, in a form that should readily be familiar to students of elementary data structures. More sophisticated table handlers usually employ a so-called **hash table**, and are the subject of later discussion. The reader should note the following:

- For a two-pass assembler, labels are entered into the table (by making calls on `enter`) only when their *defining occurrences* are encountered during the first pass.

- On the second pass, calls to `valueofsymbol` will be made when *applied occurrences* of labels are encountered.

- For a two-pass assembler, function type `ST_patch` and function `outstandingreferences` are irrelevant - as, indeed, is the `location` parameter to `valueofsymbol`.

- The symbol table entries are very simple structures defined by

```
struct ST_entries {
  ASM_alfa name;      // name
```

```
    MC_bytes value;     // value once defined
    bool defined;       // true after defining occurrence encountered
    ST_entries *slink;  // to next entry
  };
```

6.4.2 The first pass - static semantic analysis

Even though no code is generated until the second pass, the location counter (marking the address of each byte of code that is to be generated) must be tracked on both passes. To this end it is convenient to introduce the concept of a *code line* - a partial translation of each *source line*. The fields in this structure keep track of the location counter, opcode value, and address value (for two-byte instructions), and are easily assigned values after extending the analysis already performed by the syntax analyser. These extensions effectively constitute static semantic analysis. For each unpacked source line the analysis is required to examine the `mnemonic` field and - if present - to attempt to convert this to an opcode, or to a directive, as appropriate. The `opcode` value is then used as the despatcher in a switching construct that keeps track of the location counter and creates appropriate entries in the symbol table whenever defining occurrences of labels are met.

The actual code for the first pass can be found on the source diskette, and essentially follows the basic algorithm outlined in section 6.2. The following points are worth noting:

- Conversion from `mnemonic` to `opcode` requires the use of some form of opcode table. In this implementation we have chosen to construct a table that incorporates both the machine opcodes and the directive pseudo-opcodes in one simple sorted list, allowing a simple binary search to locate a possible opcode entry quickly.

  An alternative strategy might be to incorporate the opcode table into the scanner, and to handle the conversion as part of the syntax analysis, but we have chosen to leave that as the subject of an exercise.

- The attempt to convert a mnemonic may fail in two situations. In the case of a line with a blank opcode field we may sensibly return a fictitious legal empty opcode. However, when an opcode is present, but cannot be recognized (and must thus be assumed to be in error) we return a fictitious illegal opcode `err`.

- The system makes use of an intermediate work file for communicating between the two passes. This file can be discarded after assembly has been completed, and so can, in principle, remain hidden from the user.

- The arithmetic on the location counter `location` must be done *modulo* 256 because of the limitations of the target machine.

- Our assembler effectively requires that all identifiers used as labels must be "declared". In this context this means that all the identifiers in the symbol table must have appeared in the label field of some source line, and should all have been entered into the symbol table by the end of the first pass. When appropriate, we determine the value of an address, either directly, or from the symbol table, by calling the table handler routine `valueofsymbol`, which returns a parameter indicating the success of the search. It might be thought that failure is ruled out, and that calls to this routine are made only in the second pass. However, source lines using the directives `EQU`, `DS` and `ORG` may have address fields specified in terms of labels, and so even on the first pass the assembler may have to refer to the values of these labels. Clearly chaos will arise if the labels used in the address fields for these directives are not declared before use, and the assembler must be prepared to flag violations of this principle as errors.

6.4.3 The second pass - code generation

The second pass rescans the program by extracting partially assembled lines from the intermediate file, and then passing each of these to the code generator. The code generator has to keep track of further errors that might arise if any labels were not properly defined on the first pass. Because of the work already done in the first pass, handling the directives is now almost trivial in this pass.

Once again, complete code for a simple implementation is to be found on the source diskette, and it should be necessary only to draw attention to the following points:

- For our simple machine, all the generated objected code can be contained in an array of length 256. A more realistic assembler might not be able to contain the entire object code in memory, because of lack of space. For a two-pass assembler few problems would arise, as the code could be written out to a file as soon as it was generated.

- Exactly how the object code is finally to be treated is a matter for debate. Here we have called on the `listcode` routine from the class defining the pseudo-machine, which dumps the 256 bytes in a form that is suitable for input to a simple loader. However, the driver program suggested earlier also allows this code to be interpreted immediately after assembly has been successful.

- An assembler program typically gives the user a listing of the source code, usually with assembled code alongside it. Occasionally extra frills are provided, like cross reference tables for identifiers and so on. Our one is quite simple, and an example of a source listing produced by this assembler was given earlier.

Exercises

6.9 Make an extension to the ASSEMBLER language, to its grammar, and to the assembler program, to allow a character string as an operand in the `DC` directive. For example

```
TYRANT  DC  "TERRY"
```

should be treated as equivalent to

```
TYRANT  DC  'T'
        DC  'E'
        DC  'R'
        DC  'R'
        DC  'Y'
```

Is it desirable or necessary to delimit strings with different quotes from those used by single characters?

6.10 Change the table handler so that the symbol table is stored in a binary search tree, for efficiency.

6.11 The assembler will accept a line consisting only of a non-empty `LABEL` field. Is there any advantage in being able to do this?

6.12 What would happen if a label were to be *defined* more than once?

6.13 What would happen if a label were left undefined by the end of the first pass?

6.14 How would the symbol table handling alter if the source code were all held in memory throughout assembly (see Exercise 6.1), or if a string table were used (see Exercise 6.6)?

6.5 One-pass assembly

As we have already mentioned, the main reason for having the second pass is to handle the problem of *forward references* - that is, the use of labels before their locations or values have been defined. Most of the work of lexical analysis and assembly can be accomplished directly on the first pass, as can be seen from a close study of the algorithms given earlier and the complete code used for their implementation.

6.5.1 Symbol table structures

Although a one-pass assembler not always be able to determine the value of an address field immediately it is encountered, it is relatively easy to cope with the problem of forward references. We create an additional field `flink` in the symbol table entries, which then take the form

```
struct ST_entries {
  ASM_alfa name;           // name
  MC_bytes value;          // value once defined
  bool defined;            // true after defining occurrence encountered
  ST_entries *slink;       // to next entry
  ST_forwardrefs *flink;   // to forward references
};
```

The `flink` field points to entries in a **forward reference table**, which is maintained as a set of linked lists, with nodes defined by

```
struct ST_forwardrefs {    // forward references for undefined labels
  MC_bytes byte;           // to be patched
  ST_forwardrefs *nlink;   // to next reference
};
```

The `byte` fields of the `ST_forwardrefs` nodes record the addresses of as yet incompletely defined object code bytes.

6.5.2 The first pass - analysis and code generation

When reference is made to a label in the address field of an instruction, the `valueofsymbol` routine searches the symbol table for the appropriate entry, as before. Several possibilities arise:

- If the label has already been defined, it will already be in the symbol table, marked as `defined = true`, and the corresponding address or value can immediately be obtained from the `value` field.

- If the label is not yet in the symbol table, an entry is made in this table, marked as `defined = false`. The `flink` field is then initialized to point to a newly created entry in the forward reference table, in the `byte` field of which is recorded the address of the object byte whose value has still to be determined.

- If the label is already in the symbol table, but still flagged as `defined = false`, then a further entry is made in the forward reference table, linked to the earlier entries for this label.

This may be made clearer by considering the same program as before (shown fully assembled, for convenience).

```
00                    BEG                   ; count the bits in a number
00    0A              INI                   ; Read(A)
01           LOOP                           ; REPEAT
01    16              SHR                   ;  A := A DIV 2
02    3A 0D           BCC      EVEN         ;  IF A MOD 2 # 0 THEN
04    1E 13           STA      TEMP         ;    TEMP := A
06    19 14           LDA      BITS
08    05              INC
09    1E 14           STA      BITS         ;    BITS := BITS + 1
0B    19 13           LDA      TEMP         ;    A := TEMP
0D    37 01  EVEN     BNZ      LOOP         ; UNTIL A = 0
0F    19 14           LDA      BITS         ;
11    0E              OTI                   ; Write(BITS)
12    18              HLT                   ; terminate execution
13           TEMP     DS       1            ; VAR TEMP : BYTE
14    00     BITS     DC       0            ;     BITS : BYTE
15                    END
```

When the instruction at 02h (`BCC EVEN`) is encountered, `EVEN` is entered in the symbol table, undefined, linked to an entry in the forward reference table, which refers to 03h. Assembly of the next instruction enters `TEMP` in the symbol table, undefined, linked to a new entry in the forward reference table, which refers to 05h. The next instruction adds `BITS` to the symbol table, and when the instruction at 09h (`STA BITS`) is encountered, another entry is made to the forward reference table, which refers to 0Ah, itself linked to the entry which refers to 07h. This continues in the same vein, until by the time the instruction at 0Dh (`EVEN BNZ LOOP`) is encountered, the tables are as shown in Figure 6.3.

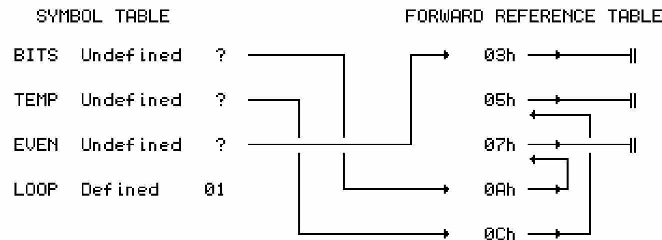

Figure 6.3 Symbol table and forward references part way through assembly

In passing, we might comment that in a real system this strategy might lead to extremely large structures. These can, fairly obviously, be kept smaller if the bytes labelled by the `DC` and `DS` instructions are all placed before the "code" which manipulates them, and some assemblers might even insist that this be done.

Since we shall also have to examine the symbol table whenever a label is defined by virtue of its appearance in the label field of an instruction or directive, it turns out to be convenient to introduce a private routine `findentry`, internal to the table handler, to perform the symbol table searching.

```
void findentry(ST_entries *&symentry, char *name, bool &found);
```

This involves a simple algorithm to scan through the symbol table, being prepared for either finding or not finding an entry. In fact, we go further and code the routine so that it *always* finds an appropriate entry, if necessary creating a new node for the purpose. Thus, `findentry` is a routine with side-effects, and so might be frowned upon by the purists. The parameter `found` records whether the entry refers to a previously created node or not.

The code for `enter` also changes somewhat. As already mentioned, when a non-blank `label` field

is encountered, the symbol table is searched. Two possibilities arise:

- If the `label` was not previously there, the new entry is completed, flagged as `defined = true`, and its `value` field is set to the now known value.

- If it was previously there, but flagged `defined = false`, the extant symbol table entry is updated, with `defined` set to `true`, and its `value` field set to the now known value.

At the end of assembly the symbol table will, in most situations, contain entries in the forward reference lists. Our table handler exports an `outstandingreferences` routine to allow the assembler to walk through these lists. Rather than have the symbol table handler interact directly with the code generation process, this pass is accomplished by applying a procedural parameter as each node of the forward reference lists is visited. In effect, rather than making a second pass over the source of the program, a partial second pass is made over the object code.

This may be made clearer by considering the same program fragment as before. When the definition of `BITS` is finally encountered at 14h, the symbol table and forward reference table will effectively have become as shown in Figure 6.4.

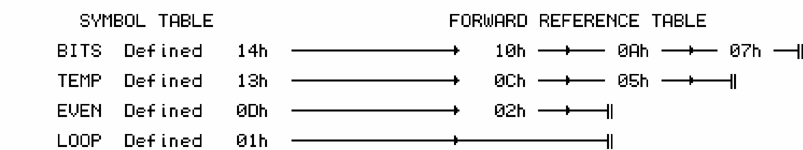

Figure 6.4 Symbol table and forward references at the end of assembly

---

Exercises

6.15 What modifications (if any) are needed to incorporate the extensions suggested as exercises at the end of the last section into the simple one-pass assembler?

6.16 We mentioned in section 6.4.3 that there was no great difficulty in assembling large programs with a two-pass assembler. How does one handle programs too large to co-reside with a one-pass assembler?

6.17 What currently happens in our one-pass assembler if a label is redefined? Should one be allowed to do this (that is, is there any advantage to be gained from being allowed to do so), and if not, what should be done to prevent it?

6.18 Constructing the forward reference table as a dynamic linked structure may be a little grander than it needs to be. Explore the possibility of holding the forward reference chains within the code being assembled. For example, if we allow the symbol table entries to be defined as follows

```
struct ST_entries {
  ASM_alfa name;       // name
  MC_bytes value;      // value once defined
  bool defined;        // true after defining occurrence encountered
  ST_entries *slink;   // to next entry
  MC_bytes flink;      // to forward references
};
```

we can arrange that the latest forward reference "points" to a byte in memory in which will be

stored the label's `value` once this is known. In the interim, this byte contains a pointer to an earlier byte in memory where the same `value` has ultimately to be recorded. For the same program fragment as was used in earlier illustrations, the code would be stored as follows, immediately before the final `END` directive is encountered. Within this code the reader should note the chain of values 0Ah, 07h, 00h (the last of which marks the end of the list) giving the list of bytes where the value of `BITS` is yet to be stored.

```
00                   BEG           ; count the bits in a number
00  0A               INI           ; read(A)
01         LOOP                    ; REPEAT
01  16               SHR           ;  A := A DIV 2
02  3A 00            BCC EVEN      ;  IF A MOD 2 # 0 THEN
04  1E 00            STA TEMP      ;    TEMP := A
06  19 00            LDA BITS
08  05               INC
09  1E 07            STA BITS      ;    BITS := BITS + 1
0B  19 05            LDA TEMP      ;    A := TEMP
0D  37 01  EVEN      BNZ LOOP      ; UNTIL A = 0
0F  19 0A            LDA BITS      ;
11  0E               OTI           ; Write(BITS)
12  18               HLT           ; terminate execution
13         TEMP      DS  1         ; VAR TEMP : BYTE
14  00     BITS      DC  0         ;     BITS : BYTE
15                   END
```

By the end of assembly the forward reference table effectively becomes as shown below. The outstanding references may be fixed up in much the same way as before, of course.

```
Name    Defined    Value   FLink

BITS     true       14h     0Ah
TEMP     true       13h     0Ch
EVEN     true       0Dh     00h
LOOP     true       01h     00h
```