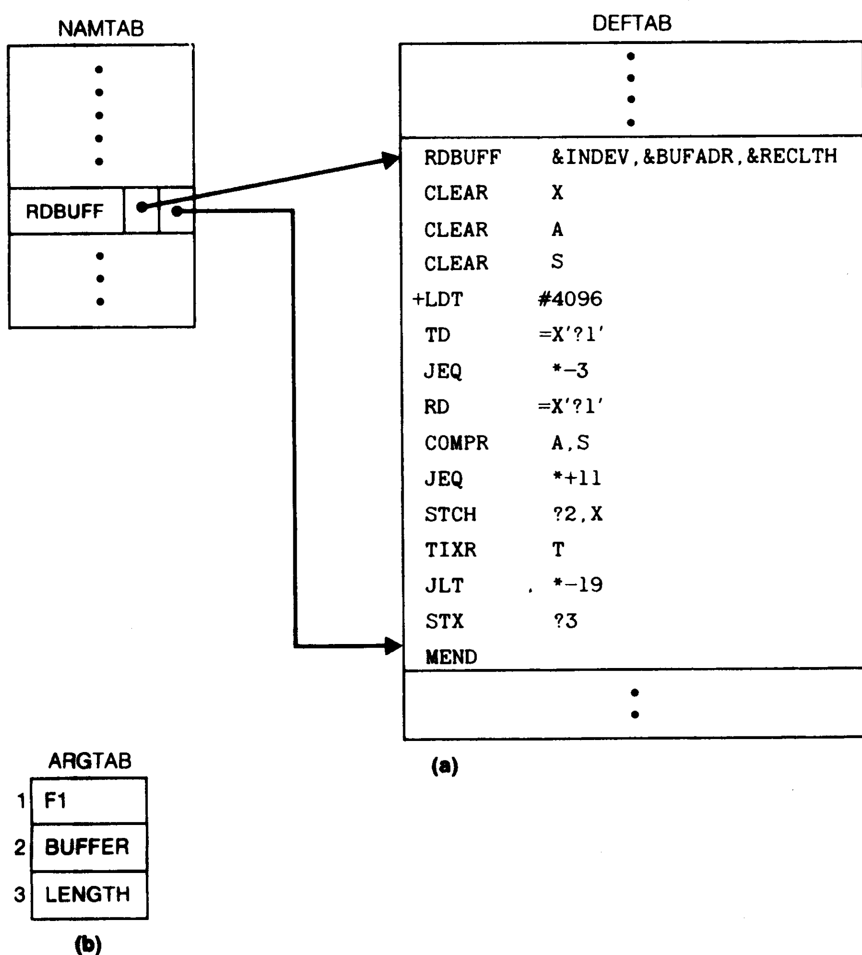

Figure 4.4 Contents of macro processor tables for the program in Fig. 4.1: (a) entries in NAMTAB and DEFTAB defining macro RDBUFF, (b) entries in ARGTAB for invocation of RDBUFF on line 190.

called to set up the argument values in ARGTAB and expand a macro invocation statement. The procedure GETLINE, which is called at several points in the algorithm, gets the next line to be processed. This line may come from DEFTAB (the next line of a macro being expanded), or from the input file, depending upon whether the Boolean variable EXPANDING is set to TRUE or FALSE.

One aspect of this algorithm deserves further comment: the handling of macro definitions within macros (as illustrated in Fig. 4.3). When a macro definition is being entered into DEFTAB, the normal approach would be to continue until an MEND directive is reached. This would not work for the example in

```
begin {macro processor}
    EXPANDING := FALSE
    while OPCODE ≠ 'END' do
        begin
            GETLINE
            PROCESSLINE
        end {while}
end {macro processor}


procedure PROCESSLINE
    begin
        search NAMTAB for OPCODE
        if found then
            EXPAND
        else if OPCODE = 'MACRO' then
            DEFINE
        else write source line to expanded file
    end {PROCESSLINE}


procedure DEFINE
    begin
        enter macro name into NAMTAB
        enter macro prototype into DEFTAB
        LEVEL := 1
        while LEVEL > 0 do
            begin
                GETLINE
                if this is not a comment line then
                    begin
                        substitute positional notation for parameters
                        enter line into DEFTAB
                        if OPCODE = 'MACRO' then
                            LEVEL := LEVEL + 1
                        else if OPCODE = 'MEND' then
                            LEVEL := LEVEL - 1
                    end {if not comment}
            end {while}
        store in NAMTAB pointers to beginning and end of definition
    end {DEFINE}
```

Figure 4.5 Algorithm for a one-pass macro processor.

```
procedure EXPAND
    begin
       EXPANDING  := TRUE
       get first line of macro definition {prototype} from DEFTAB
       set up arguments from macro invocation in ARGTAB
       write macro invocation to expanded file as a comment
       while not end of macro definition do
          begin
             GETLINE
             PROCESSLINE
          end {while}
       EXPANDING := FALSE
    end {EXPAND}

procedure GETLINE
    begin
       if EXPANDING then
          begin
             get next line of macro definition from DEFTAB
             substitute arguments from ARGTAB for positional notation
          end {if}
       else
          read next line from input file
    end {GETLINE}
```

Figure 4.5 *(cont'd)*

Fig. 4.3, however. The MEND on line 3 (which actually marks the end of the definition of RDBUFF) would be taken as the end of the definition of MACROS. To solve this problem, our DEFINE procedure maintains a counter named LEVEL. Each time a MACRO directive is read, the value of LEVEL is increased by 1; each time an MEND directive is read, the value of LEVEL is decreased by 1. When LEVEL reaches 0, the MEND that corresponds to the original MACRO directive has been found. This process is very much like matching left and right parentheses when scanning an arithmetic expression.

You may want to apply this algorithm by hand to the program in Fig. 4.1 to be sure you understand its operation. The result should be the same as shown in Fig. 4.2.

Most macro processors allow the definitions of commonly used macro instructions to appear in a standard system library, rather than in the source program. This makes the use of such macros much more convenient. Definitions are retrieved from this library as they are needed during macro processing. The extension of the algorithm in Fig. 4.5 to include this sort of processing appears as an exercise at the end of this chapter.

4.2 MACHINE-INDEPENDENT MACRO PROCESSOR FEATURES

In this section we discuss several extensions to the basic macro processor functions presented in Section 4.1. As we have mentioned before, these extended features are not directly related to the architecture of the computer for which the macro processor is written. Section 4.2.1 describes a method for concatenating macro instruction parameters with other character strings. Section 4.2.2 discusses one method for generating unique labels within macro expansions, which avoids the need for extensive use of relative addressing at the source statement level. Section 4.2.3 introduces the important topic of conditional macro expansion and illustrates the concepts involved with several examples. This ability to alter the expansion of a macro by using control statements makes macro instructions a much more powerful and useful tool for the programmer. Section 4.2.4 describes the definition and use of keyword parameters in macro instructions.

4.2.1 Concatenation of Macro Parameters

Most macro processors allow parameters to be concatenated with other character strings. Suppose, for example, that a program contains one series of variables named by the symbols XA1, XA2, XA3, ..., another series named by XB1, XB2, XB3, ..., etc. If similar processing is to be performed on each series of variables, the programmer might want to incorporate this processing into a macro instruction. The parameter to such a macro instruction could specify the series of variables to be operated on (A, B, etc.). The macro processor would use this parameter to construct the symbols required in the macro expansion (XA1, XB1, etc.).

Suppose that the parameter to such a macro instruction is named &ID. The body of the macro definition might contain a statement like

```
LDA     X&ID1
```

in which the parameter &ID is concatenated after the character string X and before the character string 1. Closer examination, however, reveals a problem with such a statement. The beginning of the macro parameter is identified by the starting symbol &; however, the end of the parameter is not marked. Thus the operand in the foregoing statement could equally well represent the character string X followed by the parameter &ID1. In this particular case, the macro processor could potentially deduce the meaning that was intended. However, if the macro definition contained both &ID and &ID1 as parameters, the situation would be unavoidably ambiguous.

Most macro processors deal with this problem by providing a special *concatenation operator.* In the SIC macro language, this operator is the character →. Thus the previous statement would be written as

```
LDA     X&ID→1
```

so that the end of the parameter &ID is clearly identified. The macro processor deletes all occurrences of the concatenation operator immediately after performing parameter substitution, so the character → will not appear in the macro expansion.

Figure 4.6(a) shows a macro definition that uses the concatenation operator as previously described. Figure 4.6(b) and (c) shows macro invocation

```
1 SUM     MACRO   &ID
2         LDA     X&ID→1
3         ADD     X&ID→2
4         ADD     X&ID→3
5         STA     X&ID→S
6         MEND
```

(a)

```
SUM     A
```

```
LDA     XA1
ADD     XA2
ADD     XA3
STA     XAS
```

(b)

```
SUM     BETA
```

```
LDA     XBETA1
ADD     XBETA2
ADD     XBETA3
STA     XBETAS
```

(c)

Figure 4.6 Concatenation of macro parameters.

statements and the corresponding macro expansions. You should work through the generation of these macro expansions for yourself to be sure you understand how the concatenation operators are handled. You are also encouraged to think about how the concatenation operator would be handled in a macro processing algorithm like the one given in Fig. 4.5.

4.2.2 Generation of Unique Labels

As we discussed in Section 4.1, it is in general not possible for the body of a macro instruction to contain labels of the usual kind. This leads to the use of relative addressing at the source statement level. Consider, for example, the definition of WRBUFF in Fig. 4.1. If a label were placed on the TD instruction on line 135, this label would be defined twice—once for each invocation of WRBUFF. This duplicate definition would prevent correct assembly of the resulting expanded program.

Because it was not possible to place a label on line 135 of this macro definition, the Jump instructions on lines 140 and 155 were written using the relative operands *–3 and *–14. This sort of relative addressing in a source statement may be acceptable for short jumps such as "JEQ *–3." However, for longer jumps spanning several instructions, such notation is very inconvenient, error-prone, and difficult to read. Many macro processors avoid these problems by allowing the creation of special types of labels within macro instructions.

Figure 4.7 illustrates one technique for generating unique labels within a macro expansion. A definition of the RDBUFF macro is shown in Fig. 4.7(a). Labels used within the macro body begin with the special character $. Figure 4.7(b) shows a macro invocation statement and the resulting macro expansion. Each symbol beginning with $ has been modified by replacing $ with $AA. More generally, the character $ will be replaced by $xx, where xx is a two-character alphanumeric counter of the number of macro instructions expanded. For the first macro expansion in a program, xx will have the value AA. For succeeding macro expansions, xx will be set to AB, AC, etc. (If only alphabetic and numeric characters are allowed in xx, such a two-character counter provides for as many as 1296 macro expansions in a single program.) This results in the generation of unique labels for each expansion of a macro instruction. For further examples, see Figs. 4.8 and 4.10.

The SIC assembler language allows the use of the character $ in symbols; however, programmers are instructed not to use this character in their source programs. This avoids any possibility of conflict between programmer-generated symbols and those created by the macro processor.

```
25    RDBUFF    MACRO    &INDEV,&BUFADR,&RECLTH
30              CLEAR    X              CLEAR LOOP COUNTER
35              CLEAR    A
40              CLEAR    S
45             +LDT      #4096          SET MAXIMUM RECORD LENGTH
50    $LOOP     TD       =X'&INDEV'     TEST INPUT DEVICE
55              JEQ      $LOOP          LOOP UNTIL READY
60              RD       =X'&INDEV'     READ CHARACTER INTO REG A
65              COMPR    A,S            TEST FOR END OF RECORD
70              JEQ      $EXIT          EXIT LOOP IF EOR
75              STCH     &BUFADR,X      STORE CHARACTER IN BUFFER
80              TIXR     T              LOOP UNLESS MAXIMUM LENGTH
85              JLT      $LOOP            HAS BEEN REACHED
90    $EXIT     STX      &RECLTH        SAVE RECORD LENGTH
95              MEND
```

(a)

```
                RDBUFF   F1,BUFFER,LENGTH

30              CLEAR    X              CLEAR LOOP COUNTER
35              CLEAR    A
40              CLEAR    S
45             +LDT      #4096          SET MAXIMUM RECORD LENGTH
50    $AALOOP   TD       =X'F1'         TEST INPUT DEVICE
55              JEQ      $AALOOP        LOOP UNTIL READY
60              RD       =X'F1'         READ CHARACTER INTO REG A
65              COMPR    A,S            TEST FOR END OF RECORD
70              JEQ      $AAEXIT        EXIT LOOP IF EOR
75              STCH     BUFFER,X       STORE CHARACTER IN BUFFER
80              TIXR     T              LOOP UNLESS MAXIMUM LENGTH
85              JLT      $AALOOP          HAS BEEN REACHED
90    $AAEXIT   STX      LENGTH         SAVE RECORD LENGTH
```

(b)

Figure 4.7 Generation of unique labels within macro expansion.

4.2.3 Conditional Macro Expansion

In all of our previous examples of macro instructions, each invocation of a particular macro was expanded into the same sequence of statements. These statements could be varied by the substitution of parameters, but the form of

```
25   RDBUFF   MACRO   &INDEV,&BUFADR,&RECLTH,&EOR,&MAXLTH
26            IF      (&EOR NE '')
27   &EORCK   SET     1
28            ENDIF
30            CLEAR   X               CLEAR LOOP COUNTER
35            CLEAR   A
38            IF      (&EORCK EQ 1)
40            LDCH    =X'&EOR'        SET EOR CHARACTER
42            RMO     A,S
43            ENDIF
44            IF      (&MAXLTH EQ '')
45           +LDT     #4096           SET MAX LENGTH = 4096
46            ELSE
47           +LDT     #&MAXLTH        SET MAXIMUM RECORD LENGTH
48            ENDIF
50   $LOOP    TD      =X'&INDEV'      TEST INPUT DEVICE
55            JEQ     $LOOP           LOOP UNTIL READY
60            RD      =X'&INDEV'      READ CHARACTER INTO REG A
63            IF      (&EORCK EQ 1)
65            COMPR   A,S             TEST FOR END OF RECORD
70            JEQ     $EXIT           EXIT LOOP IF EOR
73            ENDIF
75            STCH    &BUFADR,X       STORE CHARACTER IN BUFFER
80            TIXR    T               LOOP UNLESS MAXIMUM LENGTH
85            JLT     $LOOP             HAS BEEN REACHED
90   $EXIT    STX     &RECLTH         SAVE RECORD LENGTH
95            MEND
```

(a)

```
              RDBUFF  F3,BUF,RECL,04,2048

30            CLEAR   X               CLEAR LOOP COUNTER
35            CLEAR   A
40            LDCH    =X'04'          SET EOR CHARACTER
42            RMO     A,S
47           +LDT     #2048           SET MAXIMUM RECORD LENGTH
50   $AALOOP  TD      =X'F3'          TEST INPUT DEVICE
55            JEQ     $AALOOP         LOOP UNTIL READY
60            RD      =X'F3'          READ CHARACTER INTO REG A
65            COMPR   A,S             TEST FOR END OF RECORD
70            JEQ     $AAEXIT         EXIT LOOP IF EOR
75            STCH    BUF,X           STORE CHARACTER IN BUFFER
80            TIXR    T               LOOP UNLESS MAXIMUM LENGTH
85            JLT     $AALOOP           HAS BEEN REACHED
90   $AAEXIT  STX     RECL            SAVE RECORD LENGTH
```

(b)

Figure 4.8 Use of macro-time conditional statements.

```
              RDBUFF   0E,BUFFER,LENGTH,,80

30                     CLEAR    X                 CLEAR LOOP COUNTER
35                     CLEAR    A
47                    +LDT      #80               SET MAXIMUM RECORD LENGTH
50         $ABLOOP     TD       =X'0E'            TEST INPUT DEVICE
55                     JEQ      $ABLOOP           LOOP UNTIL READY
60                     RD       =X'0E'            READ CHARACTER INTO REG A
75                     STCH     BUFFER,X          STORE CHARACTER IN BUFFER
80                     TIXR     T                 LOOP UNLESS MAXIMUM LENGTH
87                     JLT      $ABLOOP             HAS BEEN REACHED
90         $ABEXIT     STX      LENGTH            SAVE RECORD LENGTH
```

(c)

```
              RDBUFF   F1,BUFF,RLENG,04

30                     CLEAR    X                 CLEAR LOOP COUNTER
35                     CLEAR    A
40                     LDCH     =X'04'            SET EOR CHARACTER
42                     RMO      A,S
45                    +LDT      #4096             SET MAX LENGTH = 4096
50         $ACLOOP     TD       =X'F1'            TEST INPUT DEVICE
55                     JEQ      $ACLOOP           LOOP UNTIL READY
60                     RD       =X'F1'            READ CHARACTER INTO REG A
65                     COMPR    A,S               TEST FOR END OF RECORD
70                     JEQ      $ACEXIT           EXIT LOOP IF EOR
75                     STCH     BUFF,X            STORE CHARACTER IN BUFFER
80                     TIXR     T                 LOOP UNLESS MAXIMUM LENGTH
85                     JLT      $ACLOOP             HAS BEEN REACHED
90         $ACEXIT     STX      RLENG             SAVE RECORD LENGTH
```

(d)

Figure 4.8 *(cont'd)*

the statements, and the order in which they appeared, were unchanged. A simple macro facility such as this can be a useful tool. Most macro processors, however, can also modify the sequence of statements generated for a macro expansion, depending on the arguments supplied in the macro invocation. Such a capability adds greatly to the power and flexibility of a macro language. In this section we present a typical set of conditional macro expansion statements. Other examples are found in the macro processor descriptions in Section 4.4.

The term *conditional assembly* is commonly used to describe features such as those discussed in this section. However, there are applications of macro processors that are not related to assemblers or assembler language programming. For this reason, we prefer to use the term *conditional macro expansion*.

The use of one type of conditional macro expansion statement is illustrated in Fig. 4.8. Figure 4.8(a) shows a definition of a macro RDBUFF, the logic and functions of which are similar to those previously discussed. However, this definition of RDBUFF has two additional parameters: &EOR, which specifies a hexadecimal character code that marks the end of a record, and &MAXLTH, which specifies the maximum length record that can be read. (As we shall see, it is possible for either or both of these parameters to be omitted in an invocation of RDBUFF.)

The statements on lines 44 through 48 of this definition illustrate a simple macro-time conditional structure. The IF statement evaluates a Boolean expression that is its operand. If the value of this expression is TRUE, the statements following the IF are generated until an ELSE is encountered. Otherwise, these statements are skipped, and the statements following the ELSE are generated. The ENDIF statement terminates the conditional expression that was begun by the IF statement. (As usual, the ELSE clause can be omitted entirely.) Thus if the parameter &MAXLTH is equal to the null string (that is, if the corresponding argument was omitted in the macro invocation statement), the statement on line 45 is generated. Otherwise, the statement on line 47 is generated.

A similar structure appears on lines 26 through 28. In this case, however, the statement controlled by the IF is not a line to be generated into the macro expansion. Instead, it is another macro processor directive (SET). This SET statement assigns the value 1 to &EORCK. The symbol &EORCK is a *macro-time variable* (also often called a *set symbol*), which can be used to store working values during the macro expansion. Any symbol that begins with the character & and that is not a macro instruction parameter is assumed to be a macro-time variable. All such variables are initialized to a value of 0. Thus if there is an argument corresponding to &EOR (that is, if &EOR is not null), the variable &EORCK is set to 1. Otherwise, it retains its default value of 0. The value of this macro-time variable is used in the conditional structures on lines 38 through 43 and 63 through 73.

In the previous example the value of the macro-time variable &EORCK was used to store the result of the comparison involving &EOR (line 26). The IF statements that use this value (lines 38 and 63) could, of course, simply have repeated the original test. However, the use of a macro-time variable makes it clear that the same logical condition is involved in both IF statements. Examining the value of the variable may also be faster than repeating the original test, especially if the test involves a complicated Boolean expression rather than just a single comparison.

Figure 4.8(b–d) shows the expansion of three different macro invocation statements that illustrate the operation of the IF statements in Fig. 4.8(a). You should carefully work through these examples to be sure you understand how the given macro expansion was obtained from the macro definition and the macro invocation statement.

The implementation of the conditional macro expansion features just described is relatively simple. The macro processor must maintain a symbol table that contains the values of all macro-time variables used. Entries in this table are made or modified when SET statements are processed. The table is used to look up the current value of a macro-time variable whenever it is required.

When an IF statement is encountered during the expansion of a macro, the specified Boolean expression is evaluated. If the value of this expression is TRUE, the macro processor continues to process lines from DEFTAB until it encounters the next ELSE or ENDIF statement. If an ELSE is found, the macro processor then skips lines in DEFTAB until the next ENDIF. Upon reaching the ENDIF, it resumes expanding the macro in the usual way. If the value of the specified Boolean expression is FALSE, the macro processor skips ahead in DEFTAB until it finds the next ELSE or ENDIF statement. The macro processor then resumes normal macro expansion.

The implementation outlined above does not allow for nested IF structures. You are encouraged to think about how this technique could be modified to handle such nested structures (see Exercise 4.2.10).

It is extremely important to understand that the testing of Boolean expressions in IF statements occurs at the time macros are *expanded*. By the time the program is assembled, all such decisions have been made. There is only one sequence of source statements [for example, the statements in Fig. 4.8(c)], and the conditional macro expansion directives have been removed. Thus macro-time IF statements correspond to options that might have been selected by the programmer in writing the source code. They are fundamentally different from statements such as COMPR (or IF statements in a high-level programming language), which test data values during program *execution*. The same applies to the assignment of values to macro-time variables, and to the other conditional macro expansion directives we discuss.

The macro-time IF-ELSE-ENDIF structure provides a mechanism for either generating (once) or skipping selected statements in the macro body. A different type of conditional macro expansion statement is illustrated in Fig. 4.9. Figure 4.9(a) shows another definition of RDBUFF. The purpose and function of the macro are the same as before. With this definition, however, the programmer can specify a list of end-of-record characters. In the macro invocation statement in Fig. 4.9(b), for example, there is a list (00,03,04) corresponding to the parameter &EOR. Any one of these characters is to be

```
 25  RDBUFF   MACRO   &INDEV,&BUFADR,&RECLTH,&EOR
 27  &EORCT   SET     %NITEMS(&EOR)
 30           CLEAR   X                 CLEAR LOOP COUNTER
 35           CLEAR   A
 45          +LDT     #4096             SET MAX LENGTH = 4096
 50  $LOOP    TD      =X'&INDEV'        TEST INPUT DEVICE
 55           JEQ     $LOOP             LOOP UNTIL READY
 60           RD      =X'&INDEV'        READ CHARACTER INTO REG A
 63  &CTR     SET     1
 64           WHILE   (&CTR LE &EORCT)
 65           COMP    =X'0000&EOR[&CTR]'
 70           JEQ     $EXIT
 71  &CTR     SET     &CTR+1
 73           ENDW
 75           STCH    &BUFADR,X         STORE CHARACTER IN BUFFER
 80           TIXR    T                 LOOP UNLESS MAXIMUM LENGTH
 85           JLT     $LOOP               HAS BEEN REACHED
 90  $EXIT    STX     &RECLTH           SAVE RECORD LENGTH
100           MEND
```

(a)

```
              RDBUFF  F2,BUFFER,LENGTH,(00,03,04)

 30           CLEAR   X                 CLEAR LOOP COUNTER
 35           CLEAR   A
 45          +LDT     #4096             SET MAX LENGTH = 4096
 50  $AALOOP  TD      =X'F2'            TEST INPUT DEVICE
 55           JEQ     $AALOOP           LOOP UNTIL READY
 60           RD      =X'F2'            READ CHARACTER INTO REG A
 65           COMP    =X'000000'
 70           JEQ     $AAEXIT
 65           COMP    =X'000003'
 70           JEQ     $AAEXIT
 65           COMP    =X'000004'
 70           JEQ     $AAEXIT
 75           STCH    BUFFER,X          STORE CHARACTER IN BUFFER
 80           TIXR    T                 LOOP UNLESS MAXIMUM LENGTH
 85           JLT     $AALOOP             HAS BEEN REACHED
 90  $AAEXIT  STX     LENGTH            SAVE RECORD LENGTH
```

(b)

Figure 4.9 Use of macro-time looping statements.

interpreted as marking the end of a record. To simplify the macro definition, the parameter &MAXLTH has been deleted; the maximum record length will always be 4096.

The definition in Fig. 4.9(a) uses a macro-time looping statement WHILE. The WHILE statement specifies that the following lines, until the next ENDW statement, are to be generated repeatedly as long as a particular condition is true. As before, the testing of this condition, and the looping, are done while the macro is being expanded. The conditions to be tested involve macro-time variables and arguments, not run-time data values.

The use of the WHILE-ENDW structure is illustrated on lines 63 through 73 of Fig. 4.9(a). The macro-time variable &EORCT has previously been set (line 27) to the value %NITEMS(&EOR). %NITEMS is a macro processor function that returns as its value the number of members in an argument list. For example, if the argument corresponding to &EOR is (00,03,04), then %NITEMS(&EOR) has the value 3.

The macro-time variable &CTR is used to count the number of times the lines following the WHILE statement have been generated. The value of &CTR is initialized to 1 (line 63), and incremented by 1 each time through the loop (line 71). The WHILE statement itself specifies that the macro-time loop will continue to be executed as long as the value of &CTR is less than or equal to the value of &EORCT. This means that the statements on lines 65 and 70 will be generated once for each member of the list corresponding to the parameter &EOR. The value of &CTR is used as a subscript to select the proper member of the list for each iteration of the loop. Thus on the first iteration the expression &EOR[&CTR] on line 65 has the value 00; on the second iteration it has the value 03, and so on.

Figure 4.9(b) shows the expansion of a macro invocation statement using the definition in Fig. 4.9(a). You should examine this example carefully to be sure you understand how the WHILE statements are handled.

The implementation of a macro-time looping statement such as WHILE is also relatively simple. When a WHILE statement is encountered during macro expansion, the specified Boolean expression is evaluated. If the value of this expression is FALSE, the macro processor skips ahead in DEFTAB until it finds the next ENDW statement, and then resumes normal macro expansion. If the value of the Boolean expression is TRUE, the macro processor continues to process lines from DEFTAB in the usual way until the next ENDW statement. When the ENDW is encountered, the macro processor returns to the preceding WHILE, re-evaluates the Boolean expression, and takes action based on the new value of this expression as previously described.

This method of implementation does not allow for nested WHILE structures. You are encouraged to think about how such nested structures might be supported (see Exercise 4.2.14).

4.2.4 Keyword Macro Parameters

All the macro instruction definitions we have seen thus far used *positional parameters*. That is, parameters and arguments were associated with each other according to their positions in the macro prototype and the macro invocation statement. With positional parameters, the programmer must be careful to specify the arguments in the proper order. If an argument is to be omitted, the macro invocation statement must contain a null argument (two consecutive commas) to maintain the correct argument positions. [See, for example, the macro invocation statement in Fig. 4.8(c).]

Positional parameters are quite suitable for most macro instructions. However, if a macro has a large number of parameters, and only a few of these are given values in a typical invocation, a different form of parameter specification is more useful. (Such a macro may occur in a situation in which a large and complex sequence of statements—perhaps even an entire operating system—is to be generated from a macro invocation. In such cases, most of the parameters may have acceptable default values; the macro invocation specifies only the changes from the default set of values.)

For example, suppose that a certain macro instruction GENER has 10 possible parameters, but in a particular invocation of the macro, only the third and ninth parameters are to be specified. If positional parameters were used, the macro invocation statement might look like

```
GENER   ,,DIRECT,,,,,,3.
```

Using a different form of parameter specification, called *keyword parameters*, each argument value is written with a keyword that names the corresponding parameter. Arguments may appear in any order. If the third parameter in the previous example is named &TYPE and the ninth parameter is named &CHANNEL, the macro invocation statement would be

```
GENER   TYPE=DIRECT,CHANNEL=3.
```

This statement is obviously much easier to read, and much less error-prone, than the positional version.

Figure 4.10(a) shows a version of the RDBUFF macro definition using keyword parameters. Except for the method of specification, the parameters are the same as those in Fig. 4.8(a). In the macro prototype, each parameter name is followed by an equal sign, which identifies a keyword parameter. After the equal sign, a default value is specified for some of the parameters. The parameter is assumed to have this default value if its name does not appear in the macro invocation statement. Thus the default value for the parameter &INDEV is F1. There is no default value for the parameter &BUFADR.

```
25    RDBUFF   MACRO   &INDEV=F1,&BUFADR=,&RECLTH=,&EOR=04,&MAXLTH=4096
26             IF       (&EOR NE '')
27    &EORCK   SET      1
28             ENDIF
30             CLEAR    X              CLEAR LOOP COUNTER
35             CLEAR    A
38             IF       (&EORCK EQ 1)
40             LDCH    =X'&EOR'        SET EOR CHARACTER
42             RMO      A,S
43             ENDIF
47            +LDT     #&MAXLTH        SET MAXIMUM RECORD LENGTH
50    $LOOP    TD      =X'&INDEV'      TEST INPUT DEVICE
55             JEQ      $LOOP          LOOP UNTIL READY
60             RD      =X'&INDEV'      READ CHARACTER INTO REG A
63             IF       (&EORCK EQ 1)
65             COMPR    A,S            TEST FOR END OF RECORD
70             JEQ      $EXIT          EXIT LOOP IF EOR
73             ENDIF
75             STCH     &BUFADR,X      STORE CHARACTER IN BUFFER
80             TIXR     T              LOOP UNLESS MAXIMUM LENGTH
85             JLT      $LOOP            HAS BEEN REACHED
90    $EXIT    STX      &RECLTH        SAVE RECORD LENGTH
95             MEND
```

(a)

```
               RDBUFF   BUFADR=BUFFER,RECLTH=LENGTH

30             CLEAR    X              CLEAR LOOP COUNTER
35             CLEAR    A
40             LDCH    =X'04'          SET EOR CHARACTER
42             RMO      A,S
47            +LDT     #4096           SET MAXIMUM RECORD LENGTH
50    $AALOOP  TD      =X'F1'          TEST INPUT DEVICE
55             JEQ      $AALOOP        LOOP UNTIL READY
60             RD      =X'F1'          READ CHARACTER INTO REG A
65             COMPR    A,S            TEST FOR END OF RECORD
70             JEQ      $AAEXIT        EXIT LOOP IF EOR
75             STCH     BUFFER,X       STORE CHARACTER IN BUFFER
80             TIXR     T              LOOP UNLESS MAXIMUM LENGTH
85             JLT      $AALOOP          HAS BEEN REACHED
90    $AAEXIT  STX      LENGTH         SAVE RECORD LENGTH
```

(b)

Figure 4.10 Use of keyword parameters in macro instructions.

```
              RDBUFF    RECLTH=LENGTH,BUFADR=BUFFER,EOR=,INDEV=F3

30              CLEAR    X              CLEAR LOOP COUNTER
35              CLEAR    A
47             +LDT     #4096           SET MAXIMUM RECORD LENGTH
50    $ABLOOP   TD       =X'F3'         TEST INPUT DEVICE
55              JEQ      $ABLOOP        LOOP UNTIL READY
60              RD       =X'F3'         READ CHARACTER INTO REG A
75              STCH     BUFFER,X       STORE CHARACTER IN BUFFER
80              TIXR     T              LOOP UNLESS MAXIMUM LENGTH
85              JLT      $ABLOOP          HAS BEEN REACHED
90    $ABEXIT   STX      LENGTH         SAVE RECORD LENGTH
```

(c)

Figure 4.10 *(cont'd)*

Default values can simplify the macro definition in many cases. For example, the macro definitions in Figs. 4.10(a) and 4.8(a) both provide for setting the maximum record length to 4096 unless a different value is specified by the user. The default value established in Fig. 4.10(a) takes care of this automatically. In Fig. 4.8(a), an IF-ELSE-ENDIF structure is required to accomplish the same thing.

The other parts of Fig. 4.10 contain examples of the expansion of keyword macro invocation statements. In Fig. 4.10(b), all the default values are accepted. In Fig. 4.10(c), the value of &INDEV is specified as F3, and the value of &EOR is specified as null. These values override the corresponding defaults. Note that the arguments may appear in any order in the macro invocation statement. You may want to work through these macro expansions for yourself, concentrating on how the default values are handled.

4.3 MACRO PROCESSOR DESIGN OPTIONS

In this section we discuss some major design options for a macro processor. The algorithm presented in Fig. 4.5 does not work properly if a macro invocation statement appears within the body of a macro instruction. However, it is often desirable to allow macros to be used in this way. Section 4.3.1 examines the problems created by such macro invocation statements, and suggests some possibilities for the solution of these problems.

Although the most common use of macro instructions is in connection with assembler language programming, there are other possibilities.

Section 4.3.2 discusses general-purpose macro processors that are not tied to any particular language. An example of such a macro processor can be found in Section 4.4.3. Section 4.3.3 examines the other side of this issue: the integration of a macro processor with a particular assembler or compiler. We discuss the possibilities for cooperation between the macro processor and the language translator, and briefly indicate some of the potential benefits and problems of such integration.

4.3.1 Recursive Macro Expansion

In Fig. 4.3 we presented an example of the *definition* of one macro instruction by another. We have not, however, dealt with the *invocation* of one macro by another. Figure 4.11 shows an example of such a use of macros. The definition of RDBUFF in Fig. 4.11(a) is essentially the same as the one in Fig. 4.1. The order of the parameters has been changed to make the point of the example clearer. In this case, however, we have assumed that a related macro instruction (RDCHAR) already exists. The purpose of RDCHAR is to read one character from a specified device into register A, taking care of the necessary test-and-wait loop. The definition of this macro appears in Fig. 4.11(b). It is convenient to use a macro like RDCHAR in the definition of RDBUFF so that the programmer who is defining RDBUFF need not worry about the details of device access and control. (RDCHAR might be written at a different time, or even by a different programmer.) The advantages of using RDCHAR in this way would be even greater on a more complex machine, where the code to read a single character might be longer and more complicated than our simple three-line version.

Unfortunately, the macro processor design we have discussed previously cannot handle such invocations of macros within macros. For example, suppose that the algorithm of Fig. 4.5 were applied to the macro invocation statement in Fig. 4.11(c). The procedure EXPAND would be called when the macro was recognized. The arguments from the macro invocation would be entered into ARGTAB as follows:

Parameter	Value
1	BUFFER
2	LENGTH
3	F1
4	(unused)
.	.

```
10   RDBUFF   MACRO    &BUFADR,&RECLTH,&INDEV
15   .
20   .        MACRO TO READ RECORD INTO BUFFER
25   .
30            CLEAR    X              CLEAR LOOP COUNTER
35            CLEAR    A
40            CLEAR    S
45           +LDT      #4096          SET MAXIMUM RECORD LENGTH
50   $LOOP    RDCHAR   &INDEV         READ CHARACTER INTO REG A
65            COMPR    A,S            TEST FOR END OF RECORD
70            JEQ      $EXIT          EXIT LOOP IF EOR
75            STCH     &BUFADR,X      STORE CHARACTER IN BUFFER
80            TIXR     T              LOOP UNLESS MAXIMUM LENGTH
85            JLT      $LOOP            HAS BEEN REACHED
90   $EXIT    STX      &RECLTH        SAVE RECORD LENGTH
95            MEND
```

(a)

```
 5   RDCHAR   MACRO    &IN
10   .
15   .        MACRO TO READ CHARACTER INTO REGISTER A
20   .
25            TD       =X'&IN'        TEST INPUT DEVICE
30            JEQ      *-3            LOOP UNTIL READY
35            RD       =X'&IN'        READ CHARACTER
40            MEND
```

(b)

```
              RDBUFF   BUFFER,LENGTH,F1
```

(c)

Figure 4.11 Example of nested macro invocation.

The Boolean variable EXPANDING would be set to TRUE, and expansion of the macro invocation statement would begin. The processing would proceed normally until line 50, which contains a statement invoking RDCHAR. At that point, PROCESSLINE would call EXPAND again. This time, ARGTAB would look like

Parameter	Value
1	F1
2	(unused)
.	.

The expansion of RDCHAR would also proceed normally. At the end of this expansion, however, a problem would appear. When the end of the definition of RDCHAR was recognized, EXPANDING would be set to FALSE. Thus the macro processor would "forget" that it had been in the middle of expanding a macro when it encountered the RDCHAR statement. In addition, the arguments from the original macro invocation (RDBUFF) would be lost because the values in ARGTAB were overwritten with the arguments from the invocation of RDCHAR.

The cause of these difficulties is the recursive call of the procedure EXPAND. When the RDBUFF macro invocation is encountered, EXPAND is called. Later, it calls PROCESSLINE for line 50, which results in another call to EXPAND before a return is made from the original call. A similar problem would occur with PROCESSLINE since this procedure too would be called recursively. For example, there might be confusion about whether the return from PROCESSLINE should be made to the main (outermost) loop of the macro processor logic or to the loop within EXPAND.

These problems are not difficult to solve if the macro processor is being written in a programming language (such as Pascal or C) that allows recursive calls. The compiler would be sure that previous values of any variables declared within a procedure were saved when that procedure was called recursively. It would also take care of other details involving return from the procedure. (In Chapter 5 we consider in detail how such recursive calls are handled by a compiler.)

If a programming language that supports recursion is not available, the programmer must take care of handling such items as return addresses and values of local variables. In such a case, PROCESSLINE and EXPAND would probably not be procedures at all. Instead, the same logic would be incorporated into a looping structure, with data values being saved on a stack.

The algorithm for implementing the recursive macro call is same as the algorithm for a one-pass macro processor (Fig. 4.5) except the EXPAND and GETLINE procedures. The EXPAND and GETLINE procedures are as follows (Fig. 4.12).

```
procedure EXPAND
level = 0; SP = -1
begin
  set S(SP + N + 2) = SP
  set SP = SP + N + 2
  set S(SP + 1) = DEFTAB index from NAMTAB
  set up macro call argument list array in
    S(SP + 2)...S(SP + N + 1) where N = total number of
      arguments
    while not end of macro detinition and Level ! = 0 do
```

```
      begin
        GETLINE
        PROCESSLINE
      end {while}
  set N = SP - S(SP) - 2
  set SP = S(SP)
end {EXPAND}

procedure GETLINE
begin if SP! = -1 then
  begin
    increment DEFTAB pointer to next entry
    set S(SP + 1) = S(SP + 1) + 1
    get the line from DEFTAB with the pointer
    S(SP + 1)
    substitute arguments from macro call
    S(SP + 2)...S(SP + N + 1)
  end
else
    read next line from input file
end {GETLINE}
```

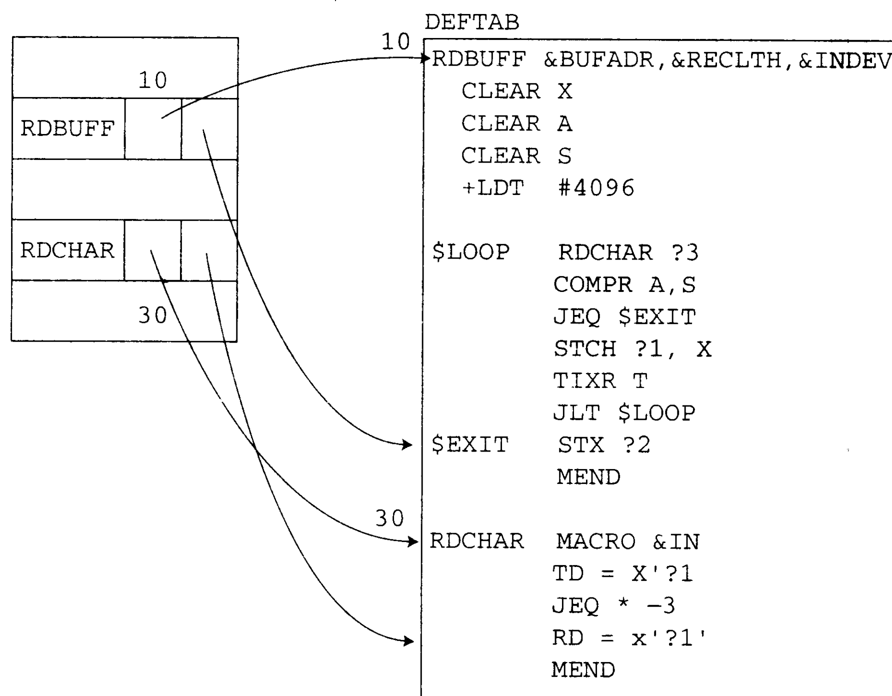

Figure 4.12 Recursive macro expansion for the example in Fig. 4.11.

```
1. SP = -1
2. Call RDBUFF BUFFER, LENGTH, F1
   SP = 1
```

S(1)	-1
S(2)	10+1+1+1+1+1
S(3)	
S(4)	BUFFER
S(5)	LENGTH
S(6)	F1

Macros expanded

```
CLEAR X
CLEAR A
CLEAR S
+LDT  #4096
```

```
3. SP = 7
```

S(1)	-1
S(2)	15
S(3)	
S(4)	BUFFER
S(5)	LENGTH
S(6)	F1
S(7)	1
S(8)	30+1+1+1+1
S(9)	
S(10)	F1

```
TD = X'F1'
JEQ * -3
RD = X'F1'
```

```
N  = SP - S(SP) - 2
   = 7 - 1 - 2
   = 4
SP = S(SP) = 1
```

```
4. SP = 1
```

S(1)	-1
S(2)	15+1+1+1+1+1+1
S(3)	
S(4)	BUFFER
S(5)	LENGTH
S(6)	F1

```
      COMPR A,S
      STCH  BUFFER, X
      TIXR  T
      JLT   $LOOP
$EXIT STX   LENGTH
```

```
N  = 1 + 1 - 2
   = 0
SP = S(SP) = S(1) = 1
```

Figure 4.12 *(cont'd)*

4.3.2 General-Purpose Macro Processors

The most common use of macro processors is as an aid to assembler language programming. Often such macro processors are combined with, or closely related to, the assembler. Macro processors have also been developed for some high-level programming languages. (One example of this kind of macro processor is discussed in Section 4.4.2.) These special-purpose macro processors are similar in general function and approach; however, the details differ from language to language. In this section we discuss general-purpose macro processors. These are not dependent on any particular programming language, but can be used with a variety of different languages.

The advantages of such a general-purpose approach to macro processing are obvious. The programmer does not need to learn about a different macro facility for each compiler or assembler language, so much of the time and expense involved in training are eliminated. The costs involved in producing a general-purpose macro processor are somewhat greater than those for developing a language-specific processor. However, this expense does not need to be repeated for each language; the result is a substantial overall saving in software development cost. Similar savings in software maintenance effort should also be realized. Over a period of years, these maintenance costs may be even more significant than the original cost for software development.

In spite of the advantages noted, there are still relatively few general-purpose macro processors. One of the reasons for this situation is the large number of details that must be dealt with in a real programming language. A special-purpose macro processor can have these details built into its logic and structure. A general-purpose facility, on the other hand, must provide some way for a user to define the specific set of rules to be followed.

In a typical programming language, there are several situations in which normal macro parameter substitution should not occur. For example, comments should usually be ignored by a macro processor (at least in scanning for parameters). However, each programming language has its own methods for identifying comments. Some languages (such as Pascal and C) use special characters to mark the start and end of a comment. Others (such as Ada) use a special character to mark only the start of a comment; the comment is automatically terminated at the end of the source line. Some languages (such as FORTRAN) use a special symbol to flag an entire line as a comment. In most assembler languages, any characters on a line following the end of the instruction operand field are automatically taken as comments. Sometimes comments are recognized partly by their position in the source line.

Another difference between programming languages is related to their facilities for grouping together terms, expressions, or statements. A general-purpose macro processor may need to take these groupings into account in scanning the source statements. Some languages use keywords such as **begin** and **end** for grouping statements. Others use special characters such as { and }. Many languages use parentheses for grouping terms and expressions. However, the rules for doing this may vary from one language to another.

A more general problem involves the *tokens* of the programming language—for example, identifiers, constants, operators, and keywords. Languages differ substantially in their restrictions on the length of identifiers and the rules for the formation of constants. Sometimes the rules for such tokens are different in certain parts of the program (for example, within a FORMAT statement in FORTRAN or a DATA DIVISION in COBOL). In some

languages, there are multiple-character operators such as ** in FORTRAN and := in Pascal. Problems may arise if these are treated by a macro processor as two separate characters rather than as a single operator. Even the arrangement of the source statements in the input file may create difficulties. The macro processor must be concerned with whether or not blanks are significant, with the way statements are continued from one line to another, and with special statement formatting conventions such as those found in FORTRAN and COBOL.

Another potential problem with general-purpose macro processors involves the syntax used for macro definitions and macro invocation statements. With most special-purpose macro processors, macro invocations are very similar in form to statements in the source programming language. (For example, the invocation of RDBUFF in Fig. 4.1 has the same form as a SIC assembler language statement.) This similarity of form tends to make the source program easier to write and read. However, it is difficult to achieve with a general-purpose macro processor that is to be used with programming languages having different basic statement forms.

In Section 4.4.3 we briefly describe one example of a general-purpose macro processor. Other discussions of general-purpose macro processors and macro processors for high-level languages can be found in Cole (1981), Kernighan and Plauger (1976), Brown (1974), and Campbell-Kelley (1973).

4.3.3 Macro Processing within Language Translators

The macro processors that we have discussed so far might be called *preprocessors*. That is, they process macro definitions and expand macro invocations, producing an expanded version of the source program. This expanded program is then used as input to an assembler or compiler. In this section we discuss an alternative: combining the macro processing functions with the language translator itself.

The simplest method of achieving this sort of combination is a *line-by-line* macro processor. Using this approach, the macro processor reads the source program statements and performs all of its functions as previously described. However, the output lines are passed to the language translator as they are generated (one at a time), instead of being written to an expanded source file. Thus the macro processor operates as a sort of input routine for the assembler or compiler.

This line-by-line approach has several advantages. It avoids making an extra pass over the source program (writing and then reading the expanded source file), so it can be more efficient than using a macro preprocessor. Some

of the data structures required by the macro processor and the language translator can be combined. For example, OPTAB in an assembler and NAMTAB in the macro processor could be implemented in the same table. In addition, many utility subroutines and functions can be used by both the language translator and the macro processor. These include such operations as scanning input lines, searching tables, and converting numeric values from external to internal representations. A line-by-line macro processor also makes it easier to give diagnostic messages that are related to the source statement containing the error (i.e., the macro invocation statement). With a macro preprocessor, such an error might be detected only in relation to some statement in the macro expansion. The programmer would then need to backtrack to discover the original source of trouble.

Although a line-by-line macro processor may use some of the same utility routines as the language translator, the functions of macro processing and program translation are still relatively independent. The main form of communication between the two functions is the passing of source statements from one to the other. It is possible to have even closer cooperation between the macro processor and the assembler or compiler. Such a scheme can be thought of as a language translator with an *integrated* macro processor.

An integrated macro processor can potentially make use of any information about the source program that is extracted by the language translator. The actual degree of integration varies considerably from one system to another. At a relatively simple level of cooperation, the macro processor may use the results of such translator operations as scanning for symbols, constants, etc. Such operations must be performed by the assembler or compiler in any case; the macro processor can simply use the results without being involved in such details as multiple-character operators, continuation lines, and the rules for token formation. This is particularly useful when the rules for such details vary from one part of the program to another (for example, within FORMAT statements and character string constants in FORTRAN).

The sort of token scan just mentioned is conceptually quite simple. However, many real programming languages have certain characteristics that create unpleasant difficulties. One classic example is the FORTRAN statement

```
DO 100 I = 1,20
```

This is a DO statement: DO is recognized as a keyword, 100 as a statement number, I as a variable name, etc. However, blanks are not significant in FORTRAN statements (except within character string constants). Thus the similar statement

```
DO 100 I = 1
```

has a quite different meaning. This is an assignment statement that gives the value 1 to the variable DO100I. Thus the proper interpretation of the characters DO, 100, etc., cannot be decided until the rest of the statement is examined. Such interpretations would be very important if, for example, a macro involved substituting for the variable name I. A FORTRAN compiler must be able to recognize and handle situations such as this. However, it would be very difficult for an ordinary macro processor (not integrated with a compiler) to do so. Such a macro processor would be concerned only with character strings, not with the interpretation of source statements.

With an even closer degree of cooperation, an integrated macro processor can support macro instructions that depend upon the context in which they occur. For example, a macro could specify a substitution to be applied only to variables or constants of a certain type, or only to variables appearing as loop indices in DO statements. The expansion of a macro could also depend upon a variety of characteristics of its arguments.

There are, of course, disadvantages to integrated and line-by-line macro processors. They must be specially designed and written to work with a particular implementation of an assembler or compiler (not just with a particular programming language). The costs of macro processor development must therefore be added to the cost of the language translator, which results in a more expensive piece of software. In addition, the assembler or compiler will be considerably larger and more complex than it would be if a macro preprocessor were used. The size may be a problem if the translator is to run on a computer with limited memory. In any case, the additional complexity will add to the overhead of language translation. (For example, some assemblers with integrated macro processors consume more time per line of source code than do some compilers on the same computing system.) Decisions about what type of macro processor to use should be based on considerations such as the frequency and complexity of macro processing that is anticipated, and other characteristics of the computing environment.

4.4 IMPLEMENTATION EXAMPLES

In this section we briefly present three examples of actual macro processors. As before, we do not attempt to cover all the characteristics of each system. Instead, we focus on the more interesting or unusual features. The first example is a macro processor that is integrated with the MASM assembler (see also Section 2.5.1). The second is a macro facility that is part of the ANSI C preprocessor. The third example is a general-purpose macro processor that is not associated with any particular programming language.

4.4.1 MASM Macro Processor

This section describes some of the macro processing features of the Microsoft MASM assembler. Further information about MASM can be found in Barkakati (1992).

The macro processor of MASM is integrated with Pass 1 of the assembler. It supports all of the main macro processor functions that we have discussed, including the definition and invocation of macro instructions within macros. Macros may be redefined during a program, without causing an error. The new definition of the macro simply replaces the first one. However, this practice can be very confusing to a person reading the program—it should probably be avoided.

One of the main differences between the MASM macro processor and the one we discussed for SIC lies in the nature of the conditional macro expansion statements. MASM calls these *conditional assembly* statements. Although the main use of these statements is in connection with macro instructions, they can also appear outside of macros.

Figure 4.13 illustrates some of the MASM macro and conditional assembly statements. The macro instruction defined in Fig. 4.13(a) computes the absolute difference between the values of its first two parameters. These parameters may be either words or doublewords. If they are doublewords, the third parameter has the value E and the calculation uses the doubleword register EAX. If the first two parameters are words, the third parameter is omitted. In that case, the calculation uses the word-length portion of the register, which is designated by AX.

The MACRO header on line 1 gives the name of the macro and its parameters. Notice that macro parameters in MASM need not begin with & or any other special character. The end of the macro is marked by the ENDM on line 15.

Line 2 declares that EXIT is a local label. When the macro is expanded, each local label is replaced by a unique name. MASM generates these unique names in the form *??n*, where *n* is a hexadecimal number in the range 0000 to FFFF. See the macro expansions in Fig. 4.13(b) and (c) for an example of this.

The IFNB on line 3 evaluates to "true" if its operand is not blank. If the parameter SIZE is not blank (that is, if it is present in the macro invocation), lines 4 through 8 are processed. Otherwise, these lines are skipped. Lines 4 through 8 contain a nested conditional statement. The IFDIF on line 4 is true if the string represented by SIZE is different from the string E. In that case, lines 5 through 7 are processed. Line 5 generates a comment that will appear on the assembly listing. The .ERR on line 6 signals to MASM that an error has been detected, and the EXITM directs MASM to terminate the expansion of the macro. Figure 4.13(d) illustrates the result.

```
1   ABSDIF    MACRO     OP1,OP2,SIZE
2             LOCAL     EXIT
3             IFNB      <SIZE>        ;; IF SIZE IS NOT BLANK
4             IFDIF     <SIZE>,<E>    ;;   THEN IT MUST BE E
5             ; ERROR -- SIZE MUST BE E OR BLANK
6             .ERR
7             EXITM
8             ENDIF                   ;; END OF IFDIF
9             ENDIF                   ;; END OF IFNB
10            MOV       SIZE&AX,OP1   ; COMPUTE ABSOLUTE DIFFERENCE
11            SUB       SIZE&AX,OP2   ;; SUBTRACT OP2 FROM OP1
12            JNS       EXIT          ;; EXIT IF RESULT GE 0
13            NEG       SIZE&AX       ;;   OTHERWISE CHANGE SIGN
14    EXIT:
15            ENDM
```

(a)

```
          ABSDIF    J,K

            ↓

          MOV       AX,J          ; COMPUTE ABSOLUTE DIFFERENCE
          SUB       AX,K
          JNS       ??0000
          NEG       AX
??0000:
```

(b)

```
          ABSDIF    M,N,E

            ↓

          MOV       EAX,M         ; COMPUTE ABSOLUTE DIFFERENCE
          SUB       EAX,N
          JNS       ??0001
          NEG       EAX
??0001:
```

(c)

```
          ABSDIF    P,Q,X

            ↓

          ; ERROR -- SIZE MUST BE E OR BLANK
```

(d)

Figure 4.13 Examples of MASM macro and conditional statements.

The & on line 10 is a concatenation operator, used to combine the value of the parameter SIZE with the string AX. If SIZE has the value E, the result is EAX. If SIZE is blank, the result is simply AX. [Compare the macro expansions in Fig. 4.13(b) and (c).]

Notice the difference between the comments on lines 9 and 10. The comment on line 9, which begins with *;;* , is a *macro comment.* It serves only as documentation for the macro definition; it is ignored when the macro is expended. The comment on line 10, which begins with *;* , is an ordinary assembler language comment. It is included as part of the macro expansion.

Figure 4.14(a) illustrates one of the iteration statements available in MASM. The IRP on line 2 sets the macro-time variable S to a sequence of values specified in < ... >. The statements between the IRP and the matching ENDM on line 4 are generated once for each such value of S. Figure 4.14(b) shows an example of the resulting macro expansion.

4.4.2 ANSI C Macro Language

This section describes some of the macro processing features of the ANSI C programming language. Section 5.5.1 discusses the structure of a typical compiler and preprocessor that implement these features. Further information can be found in Schildt (1990), as well as in many C language reference books.

```
1      NODE      MACRO     NAME
2                IRP       S,<'LEFT','DATA','RIGHT'>
3      NAME&S    DW        0
4                ENDM                          ;; END OF IRP
5                ENDM                          ;; END OF MACRO
```

(a)

```
                 NODE      X
                  ↓
       XLEFT     DW        0
       XDATA     DW        0
       XRIGHT    DW        0
```

(b)

Figure 4.14 Example of MASM iteration statement.

In the ANSI C language, definitions and invocations of macros are handled by a preprocessor. This preprocessor is generally not integrated with the rest of the compiler. Its operation is similar to the macro processor we discussed in Section 4.1. The preprocessor also performs a number of other functions, some of which are discussed in Section 5.5.1.

Here are two simple (and commonly used) examples of ANSI C macro definitions.

```
#define  NULL    0
#define  EOF     (-1)
```

After these definitions appear in the program, every occurrence of NULL will be replaced by 0, and every occurrence of EOF will be replaced by (–1). It is also possible to use macros like this to make limited changes in the syntax of the language. For example, after defining the macro

```
#define  EQ      ==
```

a programmer could write

```
while (I EQ 0)...
```

The macro processor would convert this into

```
while (I == 0)...
```

which is the correct C syntax. This could help avoid the common C error of writing = in place of ==. (However, many people consider such syntactic modifications to be a poor programming practice.)

ANSI C macros can also be defined with parameters. Consider, for example, the macro definition

```
#define  ABSDIFF(X,Y)  ((X) > (Y) ? (X) - (Y) : (Y) - (X))
```

In this case, the macro name is ABSDIFF; the parameters are named X and Y. The body of the macro makes use of a special C language conditional expression. If the condition (X) > (Y) is true, the value of this expression is the first alternative specified, (X) – (Y). If the condition is false, the value of the expression is the second alternative, (Y) – (X).

A macro invocation consists of the name of the macro followed by a parenthesized list of parameters separated by commas. When the macro is

expanded, each occurrence of a macro parameter is replaced by the corresponding argument. For example,

```
ABSDIFF(I+1, J-5)
```

would be converted by the macro processor into

```
((I+1) > (J-5) ? (I+1) - (J-5) : (J-5) - (I+1))
```

Notice the similarity between this macro invocation and a function call. Clearly, we could write a function ABSDIFF to perform this same operation. However, the macro is more efficient, because the amount of computation required to compute the absolute difference is quite small—much less than the overhead of calling a function. The macro version can also be used with different types of data. For example, we could invoke the macro as

```
ABSDIFF(I, 3.14159)
```

or

```
ABSDIFF('D', 'A')
```

Because C macros are handled by a preprocessor, it is necessary to be very careful in writing macro definitions with parameters. The macro processor simply makes string substitutions, without considering the syntax of the C language. For example, if we had written the definition of ABSDIFF as

```
#define  ABSDIFF(X,Y)  X > Y ? X - Y : Y - X
```

the macro invocation

```
ABSDIFF(3 + 1, 10 - 8)
```

would be expanded into

```
3 + 1 > 10 - 8 ? 3 + 1 - 10 - 8 : 10 - 8 - 3 + 1
```

which would not produce the intended result. (The first alternative in this case has the value –14 instead of 2, as it should be.)

In ANSI C, parameter substitutions are not performed within quoted strings. For example, consider the macro definition

```
#define  DISPLAY(EXPR)  printf("EXPR = %d\n", EXPR)
```

The macro invocation

```
DISPLAY(I*J+1)
```

would be expanded into

```
printf("EXPR = %d\n", I*J+1)
```

(However, some C compilers *would* perform the substitution for EXPR inside the quoted string, possibly with a warning message to the programmer.)

To avoid this problem, ANSI C provides a special "stringizing" operator #. When the name of a macro parameter is preceded by #, argument substitution is performed in the usual way. After the substitution, however, the resulting string is enclosed in quotes. For example, if we define

```
#define  DISPLAY(EXPR)  printf(#EXPR "= %d\n", EXPR)
```

then the invocation

```
DISPLAY(I*J+1)
```

would be expanded into

```
printf("I*J+1" "= %d\n", EXPR)
```

Macros in ANSI C may contain definitions or invocations of other macros. After a macro is expanded, the macro processor rescans the text that has been generated, looking for more macro definitions or invocations. For example, the invocation

```
DISPLAY(ABSDIFF(3,8))
```

would be expanded into

```
printf("ABSDIFF(3,8)" "= %d\n", ABSDIFF(3,8))
```

After rescanning, this would become

```
printf("ABSDIFF(3,8)" "= %d\n",
                        ((3) > (8) ? (3) - (8) : (8) - (3))
```

(Notice that the ABSDIFF within the quoted string is not treated as a macro invocation.) When executed, this statement would produce the output

```
ABSDIFF(3,8) = 5
```

The rescanning process behaves somewhat differently from the macro processing we discussed earlier in this chapter. If the body of a macro contains a token that happens to match the name of the macro, this token is not replaced during rescanning. Thus a macro cannot invoke (or define) itself recursively.

The ANSI C preprocessor also provides conditional compilation statements. These statements can be used to be sure that a macro (or other name) is defined at least once. For example, in the sequence

```
#ifndef BUFFER_SIZE
#define BUFFER_SIZE 1024
#endif
```

the #define will be processed only if BUFFER_SIZE has not already been defined.

Conditionals are also often used to control the inclusion of debugging statements in a program. Consider, for example, the sequence

```
#define DEBUG 1
.
.
.
#if DEBUG == 1
printf(...)      /* debugging output */
#endif
```

In this case, the printf statement will be included in the output from the preprocessor (and therefore compiled into the program). If the first line were changed to

```
#define DEBUG 0
```

the printf would not be included in the program. The same thing could also be accomplished by writing

```
#ifdef DEBUG
printf(...)      /* debugging output */
#endif
```

In this case, the printf would be included if a #define statement for DEBUG appeared in the source program.

4.4.3 The ELENA Macro Processor*

This section describes some of the features of the ELENA general-purpose macro processor. ELENA was developed as a research tool, not as a commercial software product. However, the same design and implementation techniques could be used in developing other general-purpose macro processors. Further information about ELENA can be found in Barcucci and Pelacani (1984).

Macro definitions in ELENA are composed of a header and a body, as with most macro processors. However, the header is not required to have any special form. It consists of a sequence of keywords and parameter markers (which are identified by the special character %). The only restriction is that at least one of the first two tokens in a macro header must be a keyword, not a parameter marker. A macro invocation is a sequence of tokens that matches the macro header. For example, a macro with the header

```
%1 = %2 + %3
```

could be invoked as

```
ALPHA = BETA + GAMMA
```

and a macro with header

```
ADD %1 TO THE VALUE OF %2
```

could be invoked as

```
ADD 10 TO THE VALUE OF INDEX
```

Figure 4.15 illustrates how ELENA could be used with different languages. Consider the macro header shown in Fig. 4.15(a). If this macro is to be used with the C language, its body might be defined as shown in Fig. 4.15(b). An example of a macro invocation and expansion using this body appears in Fig. 4.15(c).

On the other hand, suppose that the macro is to be used in an x86 assembler language program. In that case, the body might be defined as shown in Fig. 4.15(d). An example of a macro invocation and expansion using this body appears in Fig. 4.15(e). Notice that in this expansion the label &STOR is changed to &STOR0001. The character & identifies &STOR as a local label within the macro definition. The macro processor appends a numeric value to create unique labels each time the macro is expanded.

*Adapted from "A Software Development System Based on a Macroprocessor" by Elena Barcucci and Gianluca Pelacani, from Software: *Practice and Experience*, Vol. 14, pp. 519–531 (June 1984). ©John Wiley & Sons Ltd.

```
%1 := ABSDIFF(%2,%3)
```

(a)

```
%1 = (%2) > (%3) ? (%2) - (%3) : (%3) - (%2)
```

(b)

```
Z := ABSDIFF(X,Y)
```

```
Z = (X) > (Y) ? (X) - (Y) : (Y) - (X)
```

(c)

```
         MOV     EAX,%2
         SUB     EAX,%3
         JNS     &STOR
         NEG     EAX
&STOR    MOV     EAX,%1
```

(d)

```
Z := ABSDIFF(X,Y)
```

```
         MOV     EAX,X
         SUB     EAX,Y
         JNS     STOR0001
         NEG     EAX
STOR0001 MOV     EAX,Z
```

(e)

Figure 4.15 Examples of ELENA macro definition and invocation.

ELENA also provides macro-time variables and macro-time instructions that can be used to control the macro expansion. Consider the macro header shown in Fig. 4.16(a) and the associated body in Fig. 4.16(b). The .SET statement on the first line of the macro body sets the macro-time variable .LAA to 1. The next line is a statement to be generated as part of the macro expansion. After this line is generated, the following .SET statement adds 1 to the value of .LAA. If this new value is less than or equal to the value of the second parameter,

```
ADD %1 TO THE FIRST %2 ELEMENTS OF V
```

(a)

```
        .SET .LAA = 1
.E      V(.LAA) = V(.LAA) + %1
        .SET .LAA = .LAA + 1
        .IF .LAA LE %2 .JUMP .E
```

(b)

```
ADD 5 TO THE FIRST 3 ELEMENTS OF V
```

```
V(1) = V(1) + 5
V(2) = V(2) + 5
V(3) = V(3) + 5
```

(c)

Figure 4.16 Example of ELENA macro-time instructions.

the .IF macro-time instruction causes the macro processor to jump back to the line with the macro-time label .E. Figure 4.16(c) shows an example of a macro invocation and expansion using this body.

The macro-time instructions in ELENA represent a different type of approach to conditional macro expansion. The .IF statement in Fig. 4.16(b) is a macro-time conditional "go to" statement. In the SIC macro language, we would have written this definition using the WHILE-ENDW structure, which is a higher-level macro-time instruction.

The ELENA macro processor uses a macro definition table that is similar to the one we discussed for SIC. However, the process of matching a macro invocation with a macro header is more complicated. Notice that there is no single token that constitutes the macro "name." Instead, the macro is identified by the sequence of keywords that appear in its header. Consider, for example, the two macro headers

```
ADD %1 TO %2
ADD %1 TO THE FIRST ELEMENT OF %2
```

Furthermore, it is not even clear from a macro invocation statement which tokens are keywords and which are parameters. A sequence of tokens like

```
DISPLAY TABLE
```

could be an invocation of a macro with header

```
DISPLAY %1
```

(where the parameter specifies what to display). On the other hand, it could also be an invocation of a macro with header

```
%1 TABLE
```

(where the parameter specifies what operation to perform on TABLE). Notice that it is not possible for both DISPLAY and TABLE to be parameters, because at least one of the first two tokens in a macro header is required to be a keyword.

ELENA deals with this problem by constructing an index of all macro headers according to the keywords in the first two tokens of the header. Potential macro invocations are compared against all headers with keywords that match at least one of their first two tokens. For example, the sequence

```
A SUM B,C
```

would be compared against all macro headers in which the first token is A or the second token is SUM.

During this process, it is possible that a sequence of tokens could match more than one macro header. For example, the sequence

```
A = B + 1
```

might match both of the headers

```
%1 = %2 + %3
```

and

```
%1 = %2 + 1
```

In this situation, ELENA selects the header with the fewest parameters (i.e., the second of the two headers just mentioned). If there are two or more matching headers with the same number of parameters, the most recently defined macro is selected.

EXERCISES

Section 4.1

1. Apply the algorithm in Fig. 4.5 to process the source program in Fig. 4.1; the results should be the same as shown in Fig. 4.2.
2. Using the methods outlined in Chapter 8, develop a modular design for a one-pass macro processor.
3. Macro invocation statements are a part of the source program. In many cases, the programmer may not be concerned with the statements in the macro expansion. How could the macro processor and assembler cooperate to list only the macro invocation, and not the expanded version?
4. Suppose we want macro definitions to appear as a part of the assembly listing. How could the macro processor and the assembler accomplish this?
5. In most cases, character strings that occur in comments should not be replaced by macro arguments, even if they happen to match a macro parameter. How could parameter substitution in comments be prevented?
6. How should a programmer decide whether to use a macro or a subroutine to accomplish a given logical function?
7. Suppose that a certain logical task must be performed at 10 different places in an assembler language program. This task could be implemented either as a macro or as a subroutine. Describe a situation where using a macro would take *less* central memory than using a subroutine.
8. Some macros simply expand into instructions that call a subroutine. What are the advantages of this approach, as compared to using either a "pure" macro or a "pure" subroutine?
9. Write an algorithm for a two-pass macro processor in which all macro definitions are processed in the first pass, and all macro invocations are expanded in the second pass. You do not need to allow for macro definitions or invocations within macros.
10. Modify the algorithm in Fig. 4.5 to allow macro definitions to be retrieved from a library if they are not specified by the programmer.

11. Suggest appropriate ways of organizing and accessing the tables DEFTAB and NAMTAB.
12. Suppose that the occurrences of macro parameters in DEFTAB were not replaced by the positional notation *?n*. What changes would be required in the macro processor algorithm of Fig. 4.5?
13. Suppose that we have a two-pass macroassembler—that is, a one-pass macro processor built into Pass 1 of a two-pass assembler. The macro processor uses the general scheme described in Section 4.1.
    a. Suppose that we want the programmer to be able to invoke macros without having to include the macro definitions in the source program. For example, a programmer might simply write a RDBUFF statement in his or her program—the macro processor would automatically retrieve the definition of RDBUFF from a standard macro library. Briefly describe how this feature could be implemented efficiently.
    b. Suppose that macros are *not* allowed to have the same name as machine instructions. For example, if the programmer tries to define a macro named CLEAR, the macroassembler should give an error message (because CLEAR is a SIC/XE machine instruction). Describe how this restriction could be implemented.
    c. Now suppose that we *do* want to allow macros to have the same name as machine instructions. For example, if the programmer defines a macro named CLEAR, then any CLEAR statements in the program being assembled should be expanded as macro invocations. If the programmer does not define a macro named CLEAR, then any CLEAR statements should be assembled as machine instructions. Describe how this could be implemented.
    d. Suppose, as in part (c), that macros are allowed to have the same name as machine instructions. Suppose that we want to have the definitions of such macros retrieved automatically from a macro library, as in part (a). What problems arise? How might you solve these problems?

Section 4.2

1. The macro definitions in Fig. 4.1 contain several statements in which macro parameters are concatenated with other characters (for example, lines 50 and 75). Why was it not necessary to use concatenation operators in these statements?

2. Modify the algorithm in Fig. 4.5 to include the handling of concatenation operators.
3. Modify the algorithm in Fig. 4.5 to include the generation of unique labels within macro expansions.
4. Suppose that we want to allow labels within macro expansions without requiring them to have any special form (such as beginning with $). Each such label would be considered to be defined only within the macro expansion in which it occurs; this would eliminate the problem caused by duplicate labels. How could the macro processor and the assembler work together to allow this?
5. What is the most important difference between the following two sequences of statements?

a.
```
            LDA     ALPHA
            COMP    #0
            JEQ     SKIP
            LDA     #3
            STA     BETA
   SKIP     ...
```

b.
```
            IF      (&ALPHA NE 0)
   &BETA    SET     3
            ENDIF
```

6. Expand the following macro invocation statements, using the macro definition in Fig. 4.8(a):

```
a.          RDBUFF  F1,BUFFER,LENGTH,00,1024
b.  LOOP    RDBUFF  F2,BUFFER,LTH
```

7. Suppose that you have a simple one-pass macro processor like the one described in Section 4.1. Now you want to add a conditional macro expansion statement IFDEF to the macro processor. The following example illustrates the use of IFDEF:

```
        IFDEF   ALPHA
        .
        .
        .
        ENDIF
```

The statements between IFDEF and ENDIF are to be generated as part of the macro expansion if (and only if) the label ALPHA is defined somewhere in the assembly language program being processed. (It is not necessary that the definition of ALPHA appear before the macro invocation whose expansion contains the IFDEF.) Notice that ALPHA is an ordinary label, not a macro-time variable.

What changes would you have to make to the macro processor in order to implement IFDEF? Explain how your new macro processor would handle the IFDEF... ENDIF conditional statements.

8. Suppose that you have a simple one-pass macro processor like the one described in Section 4.1. Now you want to add a built-in function named %SIZEOF to the macro processor. This function can be applied to macro parameters, and returns the number of bytes occupied by the corresponding argument. Consider, for example, the following program:

```
P8        START   0
MOVE      MACRO   &FROM,&TO
&LENGTH   SET     %SIZEOF(&FROM)
          IF      (&LENGTH EQ 1)
          LDCH    &FROM
          STCH    &TO
          ELSE
          LDX     #&LENGTH
          LDS     #FROM
          LDT     #TO
          JSUB    MOVERTN
          ENDIF
          MEND
FIRST     MOVE    A,B
          MOVE    C,D
          RSUB
A         RESB    1
B         RESB    1
C         RESB    500
D         RESB    500
          END
```

In the first invocation of MOVE, %SIZEOF(A) returns 1; in the second, %SIZEOF(C) returns 500. Thus the macro invocations would be expanded as follows:

```
MOVE    A,B              MOVE    C,D
  ↓                        ↓
LDCH    A                LDX     #500
STCH    B                LDS     #C
                         LDT     #D
                         JSUB    MOVERTN
```

What changes would you have to make in the macro processor to implement the function %SIZEOF? Explain how your new macro processor would work when processing the program shown above.

9. Modify the algorithm in Fig. 4.5 to include SET statements and the IF-ELSE-ENDIF structure. You do not need to allow for nested IFs.
10. Modify your answer to Exercise 9 to allow nested IFs.
11. What is the most important difference between the following two control structures?

a.
```
              LDT      #8
              CLEAR    X
      LOOP    .
              .
              .
              TIXR     T
              JLT      LOOP
```

b.
```
      &CTR    SET      0
              WHILE    (&CTR LT 8)
              .
              .
              .
      &CTR    SET      &CTR+1
              ENDW
```

12. Using the definition in Fig. 4.9(a), expand the following macro invocation statements:

```
a.            RDBUFF   F1,BUFFER,LENGTH,(04,12)
b.    LABEL   RDBUFF   F1,BUFFER,LENGTH,00
c.            RDBUFF   F1,BUFFER,LENGTH
```