CHAPTER 5

Macros and Macro Processors

Macros are used to provide a *program generation* facility (see Section 1.2.1) through *macro expansion*. Many languages provide built-in facilities for writing macros. Well known examples of these are the higher level languages PL/I, C, Ada and C++. Assembly languages of most computer systems also provide such facilities. When a language does not support built-in macro facilities, a programmer may achieve an equivalent effect by using generalized preprocessors or software tools like Awk of Unix. The discussion in this chapter is confined to macro facilities provided in assembly languages.

Definition 5.1 (Macro) *A* macro *is a unit of specification for program generation through expansion.*

A macro consists of a name, a set of formal parameters and a body of code. The use of a macro name with a set of actual parameters is replaced by some code generated from its body. This is called *macro expansion*. Two kinds of expansion can be readily identified:

1. *Lexical expansion:* Lexical expansion implies replacement of a character string by another character string during program generation. Lexical expansion is typically employed to replace occurrences of formal parameters by corresponding actual parameters.
2. *Semantic expansion:* Semantic expansion implies generation of instructions tailored to the requirements of a specific usage—for example, generation of type specific instructions for manipulation of byte and word operands. Semantic expansion is characterized by the fact that different uses of a macro can lead to codes which differ in the number, sequence and opcodes of instructions.

Example 5.1 The following sequence of instructions is used to increment the value in a memory word by a constant:

1. Move the value from the memory word into a machine register.
2. Increment the value in the machine register.
3. Move the new value into the memory word.

Since the instruction sequence MOVE-ADD-MOVE may be used a number of times in a program, it is convenient to define a macro named INCR. Using lexical expansion, the macro call INCR A, B, AREG can lead to the generation of a MOVE-ADD-MOV instruction sequence to increment A by the value of B using AREG to perform the arithmetic.

Use of semantic expansion can enable the instruction sequence to be adapted to the types of A and B. For example, for Intel 8088, an INC instruction could be generated if A is a byte operand and B has the value '1', while a MOV-ADD-MOV sequence can be generated in all other situations.

Note that macros differ from subroutines in one fundamental respect. Use of a macro name in the mnemonic field of an assembly statement leads to its *expansion*, whereas use of a subroutine name in a call instruction leads to its *execution*. Thus, programs using macros and subroutines differ significantly in terms of program size and execution efficiency. In fact, macros can be said to trade program size for execution efficiency of a program. Other differences between macros and subroutines will be discussed along with the discussion of advanced macro facilities in Section 5.4.

5.1 MACRO DEFINITION AND CALL

Macro definition

A *macro definition* is enclosed between a *macro header* statement and a *macro end* statement. Macro definitions are typically located at the start of a program. A macro definition consists of

1. A *macro prototype* statement
2. One or more *model statements*
3. Macro *preprocessor statements*.

The macro prototype statement declares the name of a macro and the names and kinds of its parameters. A model statement is a statement from which an assembly language statement may be generated during macro expansion. A preprocessor statement is used to perform auxiliary functions during macro expansion.

The macro prototype statement has the following syntax:

$$<\textit{macro name}> \quad [<\textit{formal parameter spec}> [,...]]$$

where <*macro name*> appears in the mnemonic field of an assembly statement and <*formal parameter spec*> is of the form

$$\& <\textit{parameter name}> [<\textit{parameter kind}>] \tag{5.1}$$

Macro call

A macro is called by writing the macro name in the mnemonic field of an assembly statement. The macro call has the syntax

$$<\textit{macro name}> \quad [<\textit{actual parameter spec}> [,..]] \tag{5.2}$$

where an actual parameter typically resembles an operand specification in an assembly language statement.

Example 5.2 Figure 5.1 shows the definition of macro INCR. MACRO and MEND are the macro header and macro end statements, respectively. The prototype statement indicates that three parameters called MEM_VAL, INCR_VAL and REG exist for the macro. Since parameter kind is not specified for any of the parameters, they are all of the default kind 'positional parameter'. Statements with the operation codes MOVER, ADD and MOVEM are model statements. No preprocessor statements are used in this macro.

```
MACRO
INCR        &MEM_VAL, &INCR_VAL, &REG
MOVER       &REG, &MEM_VAL
ADD         &REG, &INCR_VAL
MOVEM       &REG, &MEM_VAL
MEND
```

Fig. 5.1 A macro definition

5.2 MACRO EXPANSION

A macro call leads to *macro expansion*. During macro expansion, the macro call statement is replaced by a sequence of assembly statements. To differentiate between the original statements of a program and the statements resulting from macro expansion, each expanded statement is marked with a '+' preceding its label field.

Two key notions concerning macro expansion are:

1. *Expansion time control flow:* This determines the order in which model statements are visited during macro expansion.
2. *Lexical substitution:* Lexical substitution is used to generate an assembly statement from a model statement.

Flow of control during expansion

The default flow of control during macro expansion is *sequential*. Thus, in the absence of preprocessor statements, the model statements of a macro are visited sequentially starting with the statement following the macro prototype statement and ending with the statement preceding the MEND statement. A preprocessor statement can alter the flow of control during expansion such that some model statements

are either never visited during expansion, or are repeatedly visited during expan
The former results in *conditional expansion* and the latter in *expansion time loop*

The flow of control during macro expansion is implemented using a *macro pansion counter* (MEC).

Algorithm 5.1 (Outline of macro expansion)

1. MEC := statement number of first statement following the prototype statem
2. While statement pointed by MEC is not a MEND statement
   (a) If a model statement then
      (i) Expand the statement.
      (ii) MEC := MEC + 1;
   (b) Else (i.e. a preprocessor statement)
      (i) MEC := new value specified in the statement;
3. Exit from macro expansion.

MEC is set to point at the statement following the prototype statement. It incremented by 1 after expanding a model statement. Execution of a preprocess statement can set MEC to a new value to implement conditional expansion or expa sion time loops. These features are discussed in Section 5.4.

Lexical substitution

A model statement consists of 3 types of strings

1. An ordinary string, which stands for itself.
2. The name of a formal parameter which is preceded by the character '&'.
3. The name of a preprocessor variable, which is also preceded by the charact '&'.

During lexical expansion, strings of type 1 are retained without substitutio Strings of types 2 and 3 are replaced by the 'values' of the formal parameters preprocessor variables. The value of a formal parameter is the corresponding actu parameter string. The rules for determining the value of a formal parameter depen on the kind of parameter.

Positional parameters

A positional formal parameter is written as &<*parameter name*>, e.g. &SAMPL where SAMPLE is the name of a parameter. In other words, <*parameter kind*> o syntax rule (5.1) is omitted. The <*actual parameter spec*> in a call on a macro usin positional parameters [see syntax rule (5.2)] is simply an <*ordinary string*>.

The value of a positional formal parameter XYZ is determined by the rule o *positional association* as follows:

1. Find the ordinal position of XYZ in the list of formai parameters in the macro prototype statement.
2. Find the actual parameter specification occupying the same ordinal position in the list of actual parameters in the macro call statement (see syntax rule (5.2)). Let this be the ordinary string ABC. Then, the value of formal parameter XYZ is ABC.

Example 5.3 Consider the call

```
INCR        A, B, AREG
```

on macro INCR of Fig. 5.1. Following the rule of positional association, values of the formal parameters are:

formal parameter	*value*
MEM_VAL	A
INCR_VAL	B
REG	AREG

Lexical expansion of the model statements now leads to the code

```
+       MOVER       AREG, A
+       ADD         AREG, B
+       MOVEM       AREG, A
```

Keyword parameters

For keyword parameters, <*parameter name*> is an ordinary string and <*parameter kind*> is the string '=' in syntax rule (5.1). The <*actual parameter spec*> is written as <*formal parameter name*> = <*ordinary string*>. The value of a formal parameter XYZ is determined by the rule of *keyword association* as follows:

1. Find the actual parameter specification which has the form XYZ= <*ordinary string*>.
2. Let <*ordinary string*> in the specification be the string ABC. Then the value of formal parameter XYZ is ABC.

Note that the ordinal position of the specification XYZ=ABC in the list of actual parameters is immaterial. This is very useful in situations where long lists of parameters have to be used.

Example 5.4 Figure 5.2 shows macro INCR of Fig. 5.1 rewritten as macro INCR_M using keyword parameters. The following macro calls

```
INCR_M      MEM_VAL=A, INCR_VAL=B, REG=AREG
...
INCR_M      INCR_VAL=B, REG=AREG, MEM_VAL=A
```

are now equivalent.

```
MACRO
INCR_M      &MEM_VAL=, &INCR_VAL=, &REG=
MOVER       &REG, &MEM_VAL
ADD         &REG, &INCR_VAL
MOVEM       &REG, &MEM_VAL
MEND
```

Fig. 5.2 A macro definition using keyword parameters

Default specifications of parameters

A *default* is a standard assumption in the absence of an explicit specification b the programmer. Default specification of parameters is useful in situations where parameter has the same value in most calls. When the desired value is different fro the default value, the desired value can be specified explicitly in a macro call. Th specification overrides the default value of the parameter for the duration of the ca

Default specification of keyword parameters can be incorporated by extendi syntax (5.1), the syntax for formal parameter specification, as follows:

$$\&<\textit{parameter name}>[<\textit{parameter kind}>[<\textit{default value}>]] \qquad (5.$$

Example 5.5 Register AREG is used for all arithmetic in a program. Hence most calls o macro INCR_M contain the specification &REG=AREG. The macro can be redefine to use a default specification for the parameter REG shown in Fig. 5.3 (see macr INCR_D). Consider the following calls

```
INCR_D      MEM_VAL=A, INCR_VAL=B
INCR_D      INCR_VAL=B, MEM_VAL=A
INCR_D      INCR_VAL=B, MEM_VAL=A, REG=BREG
```

The first two calls are equivalent to the calls in Ex. 5.4. The third call override the default value for REG with the value BREG. BREG will be used to perform th arithmetic in its expanded code.

```
MACRO
INCR_D      &MEM_VAL=, &INCR_VAL=, &REG=AREG
MOVER       &REG, &MEM_VAL
ADD         &REG, &INCR_VAL
MOVEM       &REG, &MEM_VAL
MEND
```

Fig. 5.3 A macro definition with default parameter

Macros with mixed parameter lists

A macro may be defined to use both positional and keyword parameters. In such case, all positional parameters must precede all keyword parameters. For exampl in the macro call

```
SUMUP       A,B,G=20,H=X
```

A, B are positional parameters while G, H are keyword parameters. Correspondence between actual and formal parameters is established by applying the rules governing positional and keyword parameters separately.

...s of parameters

The model statements of Examples 5.2-5.5 have used formal parameters only in operand fields. However, use of parameters is not restricted to these fields. Formal parameters can also appear in the label and opcode fields of model statements.

Example 5.6

```
        MACRO
        CALC        &X, &Y, &OP= MULT, &LAB=

&LAB    MOVER       AREG, &X
        &OP         AREG, &Y
        MOVEM       AREG, &X
        MEND
```

Expansion of the call `CALC A, B, LAB=LOOP` leads to the following code:

```
+ LOOP  MOVER       AREG, A
+       MULT        AREG, B
+       MOVEM       AREG, A
```

...NESTED MACRO CALLS

A model statement in a macro may constitute a call on another macro. Such calls are known as *nested macro calls*. We refer to the macro containing the nested call as the *outer* macro and the called macro as the *inner* macro. Expansion of nested macro calls follows the *last-in-first-out* (LIFO) rule. Thus, in a structure of nested macro calls, expansion of the latest macro call (i.e. the innermost macro call in the structure) is completed first.

Example 5.7 Macro COMPUTE of Fig. 5.4 contains a nested call on macro INCR_D of Fig. 5.3. Figure 5.5 shows the expanded code for the call

```
COMPUTE     X, Y
```

After lexical expansion, the second model statement of COMPUTE is recognized to be a call on macro INCR_D. Expansion of this macro is now performed. This leads to generation of statements marked 2, 3 and 4 in Fig. 5.5. The third model statement of COMPUTE is now expanded. Thus the expanded code for the call on COMPUTE is:

```
+          MOVEM     BREG, TMP
+          MOVER     BREG, X
+          ADD       BREG, Y
+          MOVEM     BREG, X
+          MOVER     BREG, TMP

MACRO
COMPUTE    &FIRST, &SECOND
MOVEM      BREG, TMP
INCR_D     &FIRST, &SECOND, REG=BREG
MOVER      BREG, TMP
MEND
```

Fig. 5.4 A nested macro call

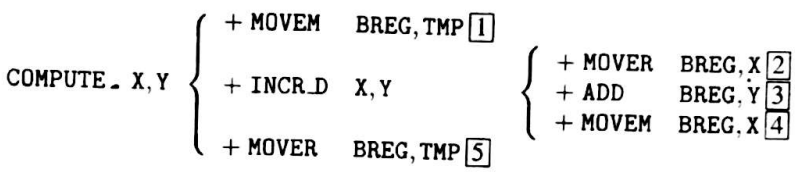


Fig. 5.5 Expanded code for a nested macro call

5.4 ADVANCED MACRO FACILITIES

Advanced macro facilities are aimed at supporting semantic expansion. These facilities can be grouped into

1. Facilities for alteration of flow of control during expansion
2. Expansion time variables
3. Attributes of parameters.

This section describes some advanced facilities and illustrates their use in performing conditional expansion of model statements and in writing expansion time loops.

Alteration of flow of control during expansion

Two features are provided to facilitate alteration of flow of control during expansion:

1. Expansion time sequencing symbols
2. Expansion time statements AIF, AGO and ANOP.

A sequencing symbol (SS) has the syntax

$$. < \textit{ordinary string} > \qquad (5.4)$$

As SS is defined by putting it in the label field of a statement in the macro body. It is used as an operand in an AIF or AGO statement to designate the destination of an expansion time control transfer. It never appears in the expanded form of a model statement.

An AIF statement has the syntax

AIF (<*expression*>) <*sequencing symbol*>

where <*expression*> is a relational expression involving ordinary strings, formal parameters and their attributes, and expansion time variables. If the relational expression evaluates to *true*, expansion time control is transferred to the statement containing <*sequencing symbol*> in its label field. An AGO statement has the syntax

AGO <*sequencing symbol*>

and unconditionally transfers expansion time control to the statement containing <*sequencing symbol*> in its label field. An ANOP statement is written as

<*sequencing symbol*> ANOP

and simply has the effect of defining the sequencing symbol.

nsion time variables

These facil-

Expansion time variables (EV's) are variables which can only be used during the expansion of macro calls. A local EV is created for use only during a particular macro call. A global EV exists across all macro calls situated in a program and can be used in any macro which has a declaration for it. Local and global EV's are created through declaration statements with the following syntax:

LCL <*EV specification*>[,<*EV specification*> ..]
GBL <*EV specification*>[,<*EV specification*> ..]

use in per-
nsion time

and <*EV specification*> has the syntax &<*EV name*>, where <*EV name*> is an ordinary string.

Values of EV's can be manipulated through the preprocessor statement SET. A SET statement is written as

expansion:

<*EV specification*> SET <*SET-expression*>

where <*EV specification*> appears in the label field and SET in the mnemonic field. A SET statement assigns the value of <*SET-expression*> to the EV specified in <*EV specification*>. The value of an EV can be used in any field of a model statement, and in the expression of an AIF statement.

(5.4)

Example 5.8

```
        MACRO
        CONSTANTS
        LCL     &A
&A      SET     1
        DB      &A
&A      SET     &A+1
        DB      &A
        MEND
```

A call on macro CONSTANTS is expanded as follows: The local EV A is created. T first SET statement assigns the value '1' to it. The first DB statement thus declare byte constant '1'. The second SET statement assigns the value '2' to A and the seco DB statement declares a constant '2'.

Attributes of formal parameters

An attribute is written using the syntax

<attribute name>'<formal parameter spec>

and represents information about the value of the formal parameter, i.e. about t corresponding actual parameter. The type, length and size attributes have the nam T, L and S.

Example 5.9

```
        MACRO
        DCL_CONST   &A
        AIF         (L'&A EQ 1) .NEXT
        - -
.NEXT   - -
        - -
        MEND
```

Here expansion time control is transferred to the statement having .NEXT in its lab field only if the actual parameter corresponding to the formal parameter A has t length of '1'.

5.4.1 Conditional Expansion

While writing a general purpose macro it is important to ensure execution efficien of its generated code. Conditional expansion helps in generating assembly cod specifically suited to the parameters in a macro call. This is achieved by ensurin that a model statement is visited only under specific conditions during the expansio of a macro. The AIF and AGO statements are used for this purpose.

Example 5.10 It is required to develop a macro EVAL such that a call

```
        EVAL        A, B, C
```

generates efficient code to evaluate A−B+C in AREG. When the first two parameters of a call are identical, EVAL should generate a single MOVER instruction to load the 3rd parameter into AREG. This is achieved as follows:

```
        MACRO
        EVAL    &X, &Y, &Z
        AIF     (&Y EQ &X) .ONLY
        MOVER   AREG, &X
        SUB     AREG, &Y
        ADD     AREG, &Z
        AGO     .OVER
.ONLY   MOVER   AREG, &Z
.OVER   MEND
```

Since the value of a formal parameter is simply the corresponding actual parameter, the AIF statement effectively compares names of the first two actual parameters. If the names are same, expansion time control is transferred to the model statement MOVER AREG, &Z. If not, the MOVE-SUB-ADD sequence is generated and expansion time control is transferred to the statement .OVER MEND which terminates the expansion. Thus efficient code is generated under all conditions.

Expansion time loops

It is often necessary to generate many similar statements during the expansion of a macro. This can be achieved by writing similar model statements in the macro.

Example 5.11

```
        MACRO
        CLEAR   &A
        MOVER   AREG, ='0'
        MOVEM   AREG, &A
        MOVEM   AREG, &A+1
        MOVEM   AREG, &A+2
        MEND
```

When called as CLEAR B, the MOVER statement puts the value '0' in AREG, while the three MOVEM statements store this value in 3 consecutive bytes with the addresses B, B+1 and B+2.

Alternatively, the same effect can be achieved by writing an expansion time loop which visits a model statement, or a set of model statements, repeatedly during macro expansion. Expansion time loops can be written using expansion time variables (EV's) and expansion time control transfer statements AIF and AGO.

Example 5.12

```
        MACRO
        CLEAR     &X, &N
        LCL       &M
&M      SET       0
        MOVER     AREG, ='0'
.MORE   MOVEM     AREG, &X+&M
&M      SET       &M+1
        AIF       (&M NE N) .MORE
        MEND
```

Consider expansion of the macro call

```
CLEAR     B, 3
```

The LCL statement declares M to be a local EV. At the start of expansion of the call, it is initialized to zero. The expansion of model statement `MOVEM AREG, &X+&M` thus leads to generation of the statement `MOVEM AREG, B`. The value of M is incremented by 1 and the model statement `MOVEM ..` is expanded repeatedly until its value equals the value of N, which is 3 in this case. Thus the macro call leads to generation of the statements

```
+       MOVER     AREG, ='0'
+       MOVEM     AREG, B
+       MOVEM     AREG, B+1
+       MOVEM     AREG, B+2
```

Comparison with execution time loops

Most expansion time loops can be replaced by execution time loops. For example, instead of generating many MOVEM statements as in Ex. 5.12 to clear the memory area starting on B, it is possible to write an execution time loop which moves 0 into B, B+1 and B+2. An execution time loop leads to more compact assembly programs. However, such programs would execute slower than programs containing expansion time loops. Thus a macro can be used to trade program size for execution efficiency.

5.4.2 Other Facilities for Expansion Time Loops

Many assemblers provide other facilities for conditional expansion, an ELSE clause in AIF being an obvious example. The assemblers for M 68000 and Intel 8088 processors provide explicit expansion time looping constructs. We discuss two such facilities here.

The REPT statement

```
REPT      <expression>
```

<*expression*> should evaluate to a numerical value during macro expansion. The statements between REPT and an ENDM statement would be processed for expansion <*expression*> number of times. Example 5.13 illustrates the use of this facility to declare 10 constants with the values 1, 2, .. 10.

Example 5.13

```
        MACRO
        CONST10
        LCL     &M
&M      SET     1
        REPT    10
        DC      '&M'
&M      SETA    &M+1
        ENDM
        MEND
```

IRP statement

```
IRP     <formal parameter>, <argument-list>
```

The formal parameter mentioned in the statement takes successive values from the argument list. For each value, the statements between the IRP and ENDM statements are expanded once.

Example 5.14

```
        MACRO
        CONSTS  &M, &N, &Z
        IRP     &Z, &M, 7, &N
        DC      '&Z'
        ENDM
        MEND
```

A macro call `CONSTS  4, 10` leads to declaration of 3 constants with the values 4, 7 and 10.

Semantic Expansion

Semantic expansion is the generation of instructions tailored to the requirements of a specific usage. It can be achieved by a combination of advanced macro facilities like AIF, AGO statements and expansion time variables. The CLEAR macro of Ex. 5.12 is an instance of semantic expansion. Here, the number of `MOVEM  AREG, ..` statements generated by a call on CLEAR is determined by the value of the second parameter of CLEAR. Macro EVAL of example 5.10 is another instance of conditional expansion wherein one of two alternative code sequences is generated depending on the peculiarities of actual parameters of a macro call. Example 5.15 illustrates semantic expansion using the type attribute.

Example 5.15

```
                MACRO
                CREATE_CONST    &X, &Y
                AIF             (T'&X EQ B) .BYTE
&Y              DW              25
                AGO             .OVER
.BYTE           ANOP
&Y              DB              25
.OVER           MEND
```

This macro creates a constant '25' with the name given by the 2^{nd} parameter. T
type of the constant matches the type of the first parameter.

EXERCISE 5.4

1. Write a macro that moves 8 numbers from the first 8 positions of an array specified
   the first operand into first 8 positions of an array specified as the second operand.
2. Generalize the macro of problem 1 above to move *n* numbers from the first operan
   to the second operand, where *n* is specified as the third operand of the macro.
3. Write a macro which takes A, B, C and D as parameters and calculates A*B + C+D
   AREG.
   (a) Where would you store the temporary result?
   (b) Would you reserve space for the temporary result within the macro body
   outside it (i.e. in the main program)? Why? What are the advantages an
   disadvantages of these alternatives?
   (c) Space reserved within the macro would be replicated in every expansion.
   this space is named then the name would also be repeated in every expansion
   This can lead to duplicate declarations. Can you think of a method to avoid th
   conflict?
4. Many macro processors permit the operation of concatenation within a macro bod
   to form larger strings. For example, if &C is a parameter with the value M2 the
   the string XY.&C has the meaning XYM2. (*note*: '.' stands for the concatenatio
   operation.) Could you use this concept to advantage in problem 3(c)?
5. A global EV carries over its value from one macro expansion to another. Can glob
   EV's be used to advantage in problem 3(c)?
6. Study the conditional expansion and SET variable facilities of a macro language ac
   cessible to you and code problems 1 to 5 in this macro language.
7. A general purpose macro is to be written to move the contents of one area of memory
   into another area of memory. Two key issues here are: If the source area is larger in
   size than the destination area, some of its contents should be ignored (i.e. content
   should be *truncated*). If the destination area is larger in size, some part of it should be
   padded with zeroes or blanks. Both truncation and padding can occur at the start or
   end of the area.

   Code the macro such that user can specify whether truncation and padding should
   occur at the start or end of the areas.

Example 5.15

```
            MACRO
            CREATE_CONST    &X, &Y
            AIF             (T'&X EQ B) .BYTE
&Y          DW              25
            AGO             .OVER
.BYTE       ANOP
&Y          DB              25
.OVER       MEND
```

This macro creates a constant '25' with the name given by the 2nd parameter type of the constant matches the type of the first parameter.

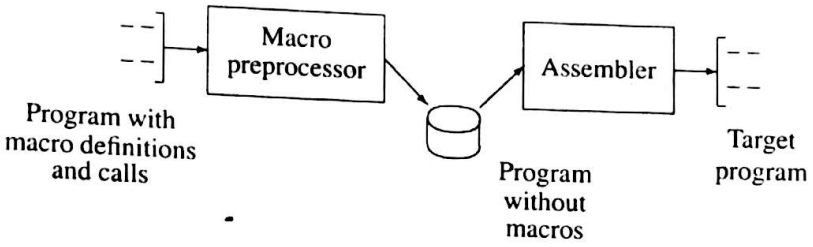
EXERCISE 5.4

1. Write a macro that moves 8 numbers from the first 8 positions of an array specif the first operand into first 8 positions of an array specified as the second operan
2. Generalize the macro of problem 1 above to move *n* numbers from the first op to the second operand, where *n* is specified as the third operand of the macro.
3. Write a macro which takes A, B, C and D as parameters and calculates A*B + C AREG.
   (a) Where would you store the temporary result?
   (b) Would you reserve space for the temporary result within the macro bo outside it (i.e. in the main program)? Why? What are the advantage disadvantages of these alternatives?
   (c) Space reserved within the macro would be replicated in every expansio this space is named then the name would also be repeated in every expan This can lead to duplicate declarations. Can you think of a method to avoid conflict?
4. Many macro processors permit the operation of concatenation within a macro to form larger strings. For example, if &C is a parameter with the value M2 the string XY.&C has the meaning XYM2. (*note:* '.' stands for the concaten operation.) Could you use this concept to advantage in problem 3(c)?
5. A global EV carries over its value from one macro expansion to another. Can gl EV's be used to advantage in problem 3(c)?
6. Study the conditional expansion and SET variable facilities of a macro language cessible to you and code problems 1 to 5 in this macro language.
7. A general purpose macro is to be written to move the contents of one area of mem into another area of memory. Two key issues here are: If the source area is larg size than the destination area, some of its contents should be ignored (i.e. cont should be *truncated*). If the destination area is larger in size, some part of it should *padded* with zeroes or blanks. Both truncation and padding can occur at the start end of the area.

   Code the macro such that user can specify whether truncation and padding sho occur at the start or end of the areas.

N OF A MACRO PREPROCESSOR

The macro preprocessor accepts an assembly program containing definitions and calls and translates it into an assembly program which does not contain any macro definitions or calls. Figure 5.6 shows a schematic of a macro preprocessor. The program form output by the macro preprocessor can now be handed over to an assembler to obtain the target language form of the program.



chematic of a macro preprocessor

Thus the macro preprocessor segregates macro expansion from the process of program assembly. It is economical because it can use an existing assembler. However, it is not as efficient as a macro-assembler, i.e. an assembler that performs macro expansion as well as assembly.

ign Overview

We begin the design by listing all tasks involved in macro expansion.

1. Identify macro calls in the program.
2. Determine the values of formal parameters.
3. Maintain the values of expansion time variables declared in a macro.
4. Organize expansion time control flow.
5. Determine the values of sequencing symbols.
6. Perform expansion of a model statement.

The following 4 step procedure is followed to arrive at a design specification for each task:

1. Identify the information necessary to perform a task.
2. Design a suitable data structure to record the information.
3. Determine the processing necessary to obtain the information.
4. Determine the processing necessary to perform the task.

Application of this procedure to each of the preprocessor tasks is described in the following.

Identify macro calls

A table called the *macro name table* (MNT) is designed to hold the names of macros defined in a program. A macro name is entered in this table when a ma definition is processed. While processing a statement in the source program, preprocessor compares the string found in its mnemonic field with the macro na in MNT. A match indicates that the current statement is a macro call.

Determine values of formal parameters

A table called the *actual parameter table* (APT) is designed to hold the values formal parameters during the expansion of a macro call. Each entry in the table pair

(<*formal parameter name*>, <*value*>)

Two items of information are needed to construct this table, names of formal para eters, and default values of keyword parameters. For this purpose, a table called *parameter default table* (PDT) is used for each macro. This table would be acce ble from the MNT entry of a macro and would contain pairs of the form (<*form parameter name*>, <*default value*>). If a macro call statement does not specif value for some parameter *par*, its default value would be copied from PDT to AP

Maintain expansion time variables

An *expansion time variables' table* (EVT) is maintained for this purpose. The ta contains pairs of the form

(<*EV name*>, <*value*>)

The value field of a pair is accessed when a preprocessor statement or a model sta ment under expansion refers to an EV.

Organize expansion time control flow

The body of a macro, i.e. the set of preprocessor statements and model statements it, is stored in a table called the *macro definition table* (MDT) for use during mac expansion. The flow of control during macro expansion determines when a mod statement is to be visited for expansion. Algorithm 5.1 can be used for this purpo MEC is initialized to the first statement of the macro body in the MDT. It is updat after expanding a model statement or on processing a macro preprocessor statemen

Determine values of sequencing symbols

A *sequencing symbols table* (SST) is maintained to hold this information. The tab contains pairs of the form

(<*sequencing symbol name*>, <*MDT entry #*>)

where <*MDT entry #*> is the number of the MDT entry which contains the model statement defining the sequencing symbol. This entry is made on encountering a statement which contains the sequencing symbol in its label field (in the case of a back reference to the symbol), or on encountering a reference prior to its definition (in the case of a forward reference).

Expansion of a model statement

This is a trivial task given the following:

1. MEC points to the MDT entry containing the model statement.
2. Values of formal parameters and EV's are available in APT and EVT, respectively.
3. The model statement defining a sequencing symbol can be identified from SST.

Expansion of a model statement is achieved by performing a lexical substitution for the parameters and EV's used in the model statement.

Data Structures

Section 5.5.1 has identified the key data structures of the macro preprocessor. To obtain a detailed design of the data structures it is necessary to apply the practical criteria of processing efficiency and memory requirements

The tables APT, PDT and EVT contain pairs which are searched using the first component of the pair as a key—for example, the formal parameter name is used as the key to obtain its value from APT. This search can be eliminated if the position of an entity within a table is known when its value is to be accessed. We will see this in the context of APT.

The value of a formal parameter ABC is needed while expanding a model statement using it, viz.

```
MOVER      AREG, &ABC
```

Let the pair (ABC, ALPHA) occupy entry #5 in APT. The search in APT can be avoided if the model statement appears as

```
MOVER      AREG, (P,5)
```

in the MDT, where (P, 5) stands for the words 'parameter #5'.

Thus, macro expansion can be made more efficient by storing an intermediate code for a statement, rather than its source form, in the MDT. All parameter names could be replaced by pairs of the form (P, *n*) in model statements and preprocessor statements stored in MDT. An interesting offshoot of this decision is that the

first component of the pairs stored in APT is no longer used during macro exp sion, e.g. the information (P, 5) appearing in a model statement is sufficient to cess the value of formal parameter ABC. Hence APT containing (<*formal parame name*>, <*value*>) pairs is replaced by another table called APTAB which only c tains <*value*>'s.

To implement this simplification, ordinal numbers are assigned to all paramet of a macro. A table named *parameter name table* (PNTAB) is used for this purpo PNTAB is used while processing the definition of a macro. Parameter names a entered in PNTAB in the same order in which they appear in the prototype statemen The entry # of a parameter's entry in PNTAB is now its ordinal number. This en is used to replace the parameter name in the model and preprocessor statements the macro while storing it in the MDT. This implements the requirement that t statement `MOVER AREG, &ABC` should appear as `MOVER AREG, (P,5)` in MDT.

In effect, the information (<*formal parameter name*>, <*value*>) in APT h been split into two tables -

PNTAB	which contains formal parameter names
APTAB	which contains formal parameter values (i.e. contains actual parameter)

Note again that PNTAB is used while processing a macro definition while APTAB i used during macro expansion.

Similar analysis leads to splitting of EVT into EVNTAB and EVTAB and SST into SSNTAB and SSTAB. EV names are entered in EVNTAB while processing EV declarations. SS names are entered in SSNTAB while processing an SS reference or definition, whichever occurs earlier.

This arrangement leads to some simplifications concerning PDT. The positional parameters (if any) of a macro appear before keyword parameters in the prototype statement. Hence in the prototype statement for a macro BETA which has p posi tional parameters and k keyword parameters, the keyword parameters have the ordi nal numbers $p+1 \ldots p+k$. Due to this numbering, two kinds of redundancies appear in PDT, the first component of each entry is redundant as in APTAB and EVTAB Further, entries $1 \ldots p$ are redundant since positional parameters cannot have default specifications. Hence entries only need to exist for parameters numbered $p+1 \ldots p+k$. To accommodate these changes, we replace the parameter default table (PDT) by a *keyword parameter default table* (KPDTAB). KPDTAB of macro BETA would only have k entries in it. To note the mapping that the first entry of KPDTAB corre sponds to parameter numbered $p+1$, we store p, the number of positional parameters of macro BETA, in a new field of the MNT entry.

MNT has entries for all macros defined in a program. Each MNT entry contains three pointers MDTP, KPDTP and SSTP, which are pointers to MDT, KPDTAB and SSNTAB for the macro, respectively. Instead of using different MDT's for different macros, for simplicity we can create a single MDT and use different sections of this

Table	*Fields in each entry*
Macro name table (MNT)	*Macro name*, *Number of positional parameters* (#PP), *Number of keyword parameters* (#KP), *Number of expansion time variables* (#EV), *MDT pointer* (MDTP), *KPDTAB pointer* (KPDTP), *SSTAB pointer* (SSTP)
Parameter Name Table (PNTAB)	*Parameter name*
EV Name Table (EVNTAB)	*EV name*
SS Name Table (SSNTAB)	*SS name*
Keyword Parameter Default Table (KPDTAB)	*parameter name*, *default value*
Macro Definition Table (MDT)	*Label*, *Opcode*, *Operands*
Actual Parameter Table (APTAB)	*Value*
EV Table (EVTAB)	*Value*
SS Table (SSTAB)	*MDT entry #*

Fig. 5.7 Tables of the macro preprocessor

table for different macros. A similar arrangement can be used with KPDTAB and SSNTAB. The data structures are now summarized in Fig. 5.7.

Construction and use of the macro preprocessor data structures can be summarized as follows: PNTAB and KPDTAB are constructed by processing the prototype statement. Entries are added to EVNTAB and SSNTAB as EV declarations and SS definitions/references are encountered. MDT entries are constructed while processing the model statements and preprocessor statements in the macro body. An entry is added to SSTAB when the definition of a sequencing symbol is encountered. APTAB

is constructed while processing a macro call. EVTAB is constructed at the start expansion of a macro.

Example 5.16

```
                MACRO
                CLEARMEM    &X, &N, &REG=AREG
                LCL         &M
&M              SET         0
                MOVER       &REG, ='0'
.MORE           MOVEM       &REG, &X+&M
&M              SET         &M+1
                AIF         (&M NE N) .MORE
                MEND
```

Figure 5.8 shows the contents of the data structures for the call

```
CLEARMEM    AREA, 10
```

Data structures above the broken line are used during the processing of a macro definition, while the data structures between the broken and firm lines are construct during macro definition processing and used during macro expansion. Data structur below the firm line are used for the expansion of a macro call.

Note how APTAB has been constructed using the actual parameters in the macro c and the default value AREG of the parameter REG. Note also the SSTAB entry f .MORE.

5.5.3 Processing of Macro Definitions

The following initializations are performed before initiating the processing of mac definitions in a program

KPDTAB_pointer := 1;
SSTAB_ptr := 1;
MDT_ptr := 1;

Algorithm 5.2 is invoked for every macro definition in the program.

Algorithm 5.2 (Processing of a macro definition)

1. *SSNTAB_ptr* := 1;
   PNTAB_ptr := 1;
2. Process the macro prototype statement and form the MNT entry
   (a) *name* := macro name;
   (b) For each positional parameter
      (i) Enter *parameter name* in PNTAB [*PNTAB_ptr*].
      (ii) *PNTAB_ptr* := *PNTAB_ptr* + 1;

PNTAB

X
N
REG

EVNTAB

M

SSNTAB

MORE

name	*#PP*	*#KP*	*#EV*	*MDTP*	*KPDTP*	*SSTP*
CLEARMEM	2	1	1	25	10	5

MNT

KPDTAB

10	REG	AREG

SSTAB

5	28

MDT

25		LCL	(E,1)
26	(E,1)	SET	0
27		MOVER	(P,3), ='0'
28		MOVEM	(P,3), (P,1)+(E,1)
29	(E,1)	SET	(E,1)+1
30		AIF	((E,1) NE (P,2)) (S,1)
31		MEND	

APTAB

AREA
10
AREG

EVTAB

0

Data structures of the macro preprocessor

(iii) #PP := #PP + 1;

(c) KPDTP := *KPDTAB_ptr*;

(d) For each keyword parameter

(i) Enter *parameter name* and *default value* (if any). KPDTAB [*KPDTAB_ptr*].

(ii) Enter *parameter name* in PNTAB [*PNTAB_ptr*].

(iii) *KPDTAB_ptr* := *KPDTAB_ptr* + 1;

(iv) *PNTAB_ptr* := *PNTAB_ptr* + 1;

(v) #KP := #KP + 1;

(e) MDTP := *MDT_ptr*;

(f) #EV := 0;

(g) SSTP := *SSTAB_ptr*;

3. While not a MEND statement

(a) If an LCL statement then

(i) Enter expansion time variable name in EVNTAB.

(ii) #EV := #EV + 1;

(b) If a model statement then

(i) If label field contains a sequencing symbol then

If symbol is present in SSNTAB then

q := entry number in SSNTAB;

else

Enter symbol in SSNTAB [*SSNTAB_ptr*].

q := *SSNTAB_ptr*;

SSNTAB_ptr := *SSNTAB_ptr* + 1;

SSTAB [SSTP + $q - 1$] := *MDT_ptr*;

(ii) For a parameter, generate the specification (P, #*n*).

(iii) For an expansion variable, generate the specification (E, #*m*).

(iv) Record the IC in MDT [*MDT_ptr*];

(v) *MDT_ptr* := *MDT_ptr* + 1;

(c) If a preprocessor statement then

(i) If a SET statement

Search each expansion time variable name used in the statement EVNTAB and generate the spec (E, #*m*).

(ii) If an AIF or AGO statement then

If sequencing symbol used in the statement is present in SSNTAB then

q := entry number in SSNTAB;

else
Enter symbol in SSNTAB [*SSNTAB_ptr*].
q := *SSNTAB_ptr*;
SSNTAB_ptr := *SSNTAB_ptr* + 1;
Replace the symbol by (S, SSTP + q − 1).

(iii) Record the IC in MDT [*MDT_ptr*].

(iv) *MDT_ptr* := *MDT_ptr* + 1;

4. (MEND statement)
If *SSNTAB_ptr* = 1 (i.e. SSNTAB is empty) then
SSTP := 0;
Else *SSTAB_ptr* := *SSTAB_ptr* + *SSNTAB_ptr* − 1;
If #KP = 0 then KPDTP = 0;

Macro Expansion

We use the following data structures to perform macro expansion:

APTAB	Actual parameter table
EVTAB	EV table
MEC	Macro expansion counter
APTAB_ptr	APTAB pointer
EVTAB_ptr	EVTAB pointer

The number of entries in APTAB (i.e., $\#e_{APTAB}$) equals the sum of values in the #PP and #KP fields of the MNT entry of a macro. Number of entries in EVTAB (i.e., $\#e_{EVTAB}$) is given by the value in #EV field of the MNT. APTAB and EVTAB are constructed when a macro call is recognized. *APTAB_ptr* and *EVTAB_ptr* are set to point at these tables. As in Algorithm 5.1, MEC always points to the next statement to be expanded.

Algorithm 5.3 (Macro expansion)

1. Perform initializations for the expansion of a macro
   (a) MEC := MDTP field of the MNT entry;
   (b) Create EVTAB with #EV entries and set *EVTAB_ptr*.
   (c) Create APTAB with #PP+#KP entries and set *APTAB_ptr*.
   (d) Copy keyword parameter defaults from the entries KPDTAB [KPDTP] ... KPDTAB [KPDTP+#KP−1] into APTAB [#PP+1] ... APTAB [#PP+#KP].
   (e) Process positional paremeters in the actual parameter list and copy them into APTAB [1] ... APTAB [#PP].

(f) For keyword parameters in the actual parameter list
Search the keyword name in *parameter name* field of KPDTAB [KPDTP] ... KPDTAB [KPDTP+#KP−1]. Let KPTDAB [*q*] contain a matching entry. Enter value of the keyword parameter in the call (if any) in APTAB [#PP+*q*−KPDTP+1].

2. While statement pointed by MEC is not MEND statement
   (a) If a model statement then
      (i) Replace operands of the form (P, #*n*) and (E, #*m*) by values APTAB [*n*] and EVTAB [*m*] respectively.
      (ii) Output the generated statement.
      (iii) MEC := MEC + 1;
   (b) If a SET statement with the specification (E, #*m*) in the label field then
      (i) Evaluate the expression in the operand field and set an appropriate value in EVTAB [*m*].
      (ii) MEC := MEC + 1;
   (c) If an AGO statement with (S, #*s*) in operand field then
      MEC := SSTAB [SSTP+*s* − 1];
   (d) If an AIF statement with (S, #*s*) in operand field then
      If condition in the AIF statement is *true* then
         MEC := SSTAB [SSTP+*s* − 1];
3. Exit from macro expansion.

Figure 5.9 shows the data structures at some point during the expansion of macro CLEARMEM of Ex. 5.16.

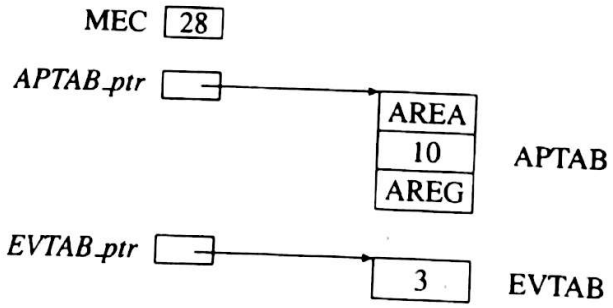


Fig. 5.9 Data structures during macro expansion

5.5.5 Nested Macro Calls

Figure 5.5 in Section 5.3 illustrates nested macro calls. Two basic alternatives exist for processing nested macro calls. We can apply the macro expansion schematic of

Section 5.5.4 to obtain the first level expanded code, viz.

```
                - - -
+               MOVEM       BREG, TMP
+               INCR_D      X, Y, REG=BREG
+               MOVER       BREG, TMP
                - - -
```

In this code macro calls appearing in the source program have been expanded but statements resulting from the expansion may themselves contain macro calls. The macro expansion schematic can be applied to the first level expanded code to expand these macro calls and so on, until we obtain a code form which does not contain any macro calls. This scheme would require a number of passes of macro expansion, which makes it quite expensive.

A more efficient alternative would be to examine each statement generated during macro expansion to see if it is itself a macro call. If so, a provision can be made to expand this call before continuing with the expansion of the parent macro call. This avoids multiple passes of macro expansion, thus ensuring processing efficiency. This alternative requires some extensions in the macro expansion scheme of Section 5.5.4.

Consider the situation when the ADD statement marked [3] in Fig. 5.5 is being generated. Expansion of two macro calls is in progress at this moment. This happened because the outer macro COMPUTE gave rise to a macro call on macro INCR_D during the expansion of its current model statement. The model statements of INCR_D are currently being expanded using the expansion time data structures MEC, APTAB, EVTAB, *APTAB_ptr* and *EVTAB_ptr*. A MEND statement encountered during the expansion must lead to resumption of expansion of the outer macro. This requires that MEC, APTAB, EVTAB, *APTAB_ptr* and *EVTAB_ptr* should be restored to the values contained in them while the macro COMPUTE was being expanded. Control returns to the processing of the source program when the MEND statement is encountered during the processing of COMPUTE.

Thus, two provisions are required to implement the expansion of nested macro calls:

1. Each macro under expansion must have its own set of data structures, viz. MEC, APTAB, EVTAB, *APTAB_ptr* and *EVTAB_ptr*.
2. An *expansion nesting counter* (*Nest_cntr*) is maintained to count the number of nested macro calls. *Nest_cntr* is incremented when a macro call is recognized and decremented when a MEND statement is encountered. Thus *Nest_cntr* > 1 indicates that a nested macro call is under expansion, while *Nest_cntr* = 0 implies that macro expansion is not in progress currently.

The first provision can be implemented by creating many copies of the expansion time data structures. These can be stored in the form of an array. For example, we can

have an array called APTAB_ARRAY, each element of which is an APTAB. APTAB for the innermost macro call would be given by APTAB_ARRAY[*Nest_cntr*]. This arrangement provides access efficiency. However, it is expensive in terms of memory requirements. It also involves a difficult design decision—how many copies of the data structures should be created? If too many copies are created, some may not be used. If too few are created, some assembly programs may have to be rejected because their macro call nesting depth exceeds the number of copies of the data structures.

Since macro calls are expanded in a LIFO manner, a practical solution is to use a stack to accommodate the expansion time data structures. The stack consists of *expansion records*, each expansion record accommodating one set of expansion time data structures. The expansion record at the top of the stack corresponds to the macro call currently being expanded. When a nested macro call is recognized, a new expansion record is pushed on the stack to hold the data structures for the call. At MEND an expansion record is popped off the stack. This would uncover the previous expansion record in the stack which houses the expansion time data structures of the outer macro. The extended stack model of Section 2.2.1 is used for this purpose.

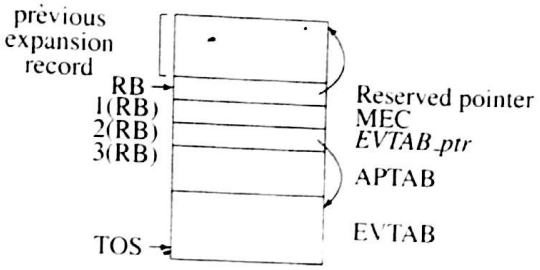


Fig. 5.10 Use of stack for macro preprocessor data structures

Figure 5.10 illustrates the stack oriented management of expansion time data structures. The expansion record on top of the stack contains the data structures in current use. *Record base* (RB) is a pointer pointing to the start of this expansion record. TOS points to the last occupied entry in stack. When a nested macro call is detected, another set of data structures is allocated on the stack. RB is now set to point to the start of the new expansion record. MEC, *EVTAB_ptr*, APTAB and EVTAB are allocated on the stack in that order. During macro expansion, the various data structures are accessed with reference to the value contained in RB. This is performed using the following addresses:

Data structure	*Address*
Reserved pointer	0(RB)
MEC	1(RB)

EVTAB_ptr	2(RB)
APTAB	3(RB) to e_{APTAB}+2(RB)
EVTAB	contents of *EVTAB_ptr*

where 1(RB) stands for 'contents of RB+1'. Note that the first entry of APTAB always has the address 3(RB). This eliminates the need for *APTAB_ptr*.

At a MEND statement, a record is popped off the stack by setting TOS to the end of the previous record. It is now necessary to set RB to point to the start of previous record in stack. This is achieved by using the entry marked 'reserved pointer' in the expansion record. This entry always points to the start of the previous expansion record in stack. While popping off a record, the value contained in this entry can be loaded into RB. This has the effect of restoring access to the expansion time data structures used by the outer macro.

Actions at start and end of a macro expansion are based on the extended stack model of Section 2.2.1.

expansion

Actions at the start of expansion are summarized in Table 5.1.

Actions at start of macro expansion

No.			*Statement*
1.	TOS	:=	TOS+1;
2.	TOS*	:=	RB;
3.	RB	:=	TOS;
4.	1(RB)	:=	MDTP entry of MNT;
5.	2(RB)	:=	RB+3+#e_{APTAB};
6.	TOS	:=	TOS + #e_{APTAB} + #e_{EVTAB}+2;

The first statement increments TOS to point at the first word of the new expansion record. This is the *reserved pointer*. The '*' mark in the second statement TOS* := RB indicates indirection. This statement deposits the address of the previous record base into this word. New RB is now established in statement 3. Statements 4 and 5 set MEC and *EVTAB_ptr* respectively. Statement 6 sets TOS to point to the last entry of the expansion record.

expansion

Actions at the end of expansion are summarized in Table 5.2.

The first statement pops an expansion record off the stack by resetting TOS to the value it had while the outer macro was being expanded. RB is then made to point at the base of the previous record. Data structures in the old expansion record are now accessible as displacements from the new value in RB, e.g. MEC is 1(RB).

Table 5.2 Actions at end of macro expansion

No.	*Statement*
1.	TOS := RB−1;
2.	RB := RB*;

Example 5.17 Figure 5.11 illustrates the stack during the expansion of nested macro c of Fig. 5.4. The statement `INCR_D &FIRST, &SECOND` of macro COMPUT assumed to occupy entry 18 in MDT. The body of macro INCR_D is assume occupy entries 73-75. Parts (a), (b) and (c) of the figure show the stack before, du and after the expansion of the nested macro call `INCR_D X,Y`. In part (a) of figure, RB points at the bottom-most entry of the stack and TOS points at the entry of the expansion record, i.e. at the entry of `Y` in APTAB. When the nested is encountered, the *reserved pointer* of the new record would be made to point at start of the previous expansion record. MEC and APTAB for the call would now established.

After completing the expansion of the call on INCR_D, RB would be reset to bottom-most entry of the stack, and TOS would once again point at the entry in APTAB of the first record. MEC now contains the value 18, which would be cremented to 19. Macro expansion thus continues with the next statement in mi COMPUTE.

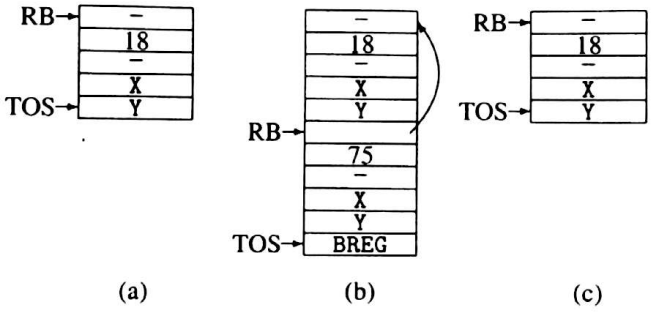


Fig. 5.11 An illustration of the expansion of nested macro calls

EXERCISE 5.5

1. Modify Algorithm 5.3 to enable expansion of nested macro calls.

5.5.6 Design of a Macro-assembler

As mentioned in Section 5.5, use of a macro preprocessor followed by a conventiona assembler (see Fig. 5.6) is an expensive way of handling macros since the numbe of passes over the source program is large and many functions get duplicated. Fo example, analysis of a source statement to detect macro calls requires us to proce

the mnemonic field. A similar function is required in the first pass of the assembler. Similar functions of the preprocessor and the assembler can be merged if macros are handled by a macro assembler which performs macro expansion and program assembly simultaneously. This may also reduce the number of passes.

The discussion in the previous section may give rise to the impression that it is always possible to perform macro expansion in a single pass. This is not true, as certain kinds of forward references in macros cannot be handled in a single pass.

Example 5.18 Consider the assembly program of Fig. 5.12. Here the type attribute T'&X in macro CREATE_CONST is a forward reference to symbol A. Processing of the type attribute cannot be postponed because it determines the nature of the constant defined by the macro. In turn, this affects memory allocation and hence the address of A !

```
                MACRO
                CREATE_CONST    &X, &Y
                AIF             (T'&X EQ B) .BYTE
&Y              DW              25
                AGO             .OVER
.BYTE           ANOP
&Y              DB              25
.OVER           MEND

                CREATE_CONST    A, NEW_CON
                :
A               DB              ?
                END
```

Forward reference in type attribute of a parameter

This problem leads to the classical two pass organization for macro expansion. The first pass collects information about the symbols defined in a program and the second pass performs macro expansion.

Pass structure of a macro-assembler

To design the pass structure of a macro-assembler we identify the functions of a macro preprocessor and the conventional assembler which can be merged to advantage. After merging, the functions can be structured into passes of the macro-assembler. This process leads to the following pass structure:

1. Macro definition processing
2. SYMTAB construction.

1. Macro expansion

2. Memory allocation and LC processing
3. Processing of literals
4. Intermediate code generation.

Pass III

1. Target code generation.

Pass II is large in size since it performs many functions. Further, since it perfo macro expansion as well as Pass I of a conventional assembler, all the data structu of the macro preprocessor and the conventional assembler need to exist during pass.

The pass structure can be simplified if attributes of actual parameters are not to supported. The macro preprocessor would then be a single pass program. Integrati Pass I of the assembler with the preprocessor would give us the following two p structure:

Pass I

1. Macro definition processing
2. Macro expansion
3. Memory allocation, LC processing and SYMTAB construction
4. Processing of literals
5. Intermediate code generation.

Pass II

1. Target code generation.

There is obvious imbalance between the sizes of the two passes. A 3-pass struc ture might be preferred for this reason alone.

EXERCISE 5.5

1. Compare and contrast the properties of macros and subroutines with respect to th following:
   (a) code space requirements
   (b) execution speed
   (c) processing required by the assembler
   (d) flexibility and generality.
2. In an assembly language program, a certain action is required at 10 places in the program. Under what conditions would you code this action as
   (a) a macro?
   (b) a subroutine?

   Justify your answer with the help of appropriate examples.
3. Solve the problems of exercise 5.4.3 using REPT and IRP statements.

4. Extend the macro preprocessor described in this chapter to support the following features:
   (a) REPT and IRP statements discussed in section 5.4.2
   (b) Global EV's
   (c) Nested macro calls.

BIBLIOGRAPHY

Flores (1971), Donovan (1972) and Cole (1981) are few of the texts covering macro processors and macro assemblers in detail. Macros have been used as a tool for writing portable programs. Brown (1974) and Wallis (1982) discuss these aspects in detail.

1. Brown, P.J. (1974): *Macro Processors and Techniques for Portable Software*, Wiley, London.
2. Cole, A.J. (1981): *Macro Processors*, Cambridge University Press, Cambridge.
3. Donovan, J.J. (1972): *Systems Programming*, McGraw-Hill Kogakusha, Tokyo.
4. Flores, I. (1971): *Assemblers and BAL*, Prentice-Hall, Englewood Cliffs.
5. Wallis, P.J.L. (1982): *Portable Programming*, Macmillan, London, 1982.