Macro Generators(1)

The early users of symbolic assembly languages soon found them a great improvement over writing numerical machine code by hand but nonetheless, long winded and tedious to write.

A partial solution to this problem was found in another piece of symbolism; names were given to frequently used instruction sequences and these names used in the source program in place of the sequences for which they stood. The source text was subsequently fed through a program which performed the reverse substitution, replacing names with the appropriate instruction text before processing by a conventional assembler. The 'names' came to be known as macros, as an abbreviation of 'macro-instructions', and the program as a macro-generator. In operation, a macro-generator reads in and stores macro-definitions which specify the substitutions that are to be performed then carries out the required transformations on the input file; output may be stored or, where macrogenerator and assembler proper are combined into a single program(a MACRO-ASSEMBLER), passed directly to the assembler as it is generated.

An obvious extension to the simple scheme considered so far is to allow formal parameters to be specified in the macro definitions and replaced by actual parameters in the macro call; this addition greatly increases the power of the method.

A possibility which it is normal to cater for is that one macro may be defined in terms of one or more others; when this occurs, and especially where recursive calling sequences are involved, it becomes easier to regard a macro definition as defining a string-valued function whose value is successively substituted at each point of call rather

/than

than seeing the substitution as a text editing operation.

Pure string-function evaluation systems have been implemented, notably Christopher Strachey's General Purpose Macrogenerator(GPM), in which all operations, including numerical ones, can, in principle at least, be defined as string operations. In practice, these are unsuitable for use as assembly language preprocessers because of their very generality; any system which permits its users to do almost anything can of necessity almost never detect errors. Furthermore, since the basic facilities they offer are truly basic, the path connecting the immediate cause of failure with the true error may be very long and difficult to follow.

Macrogenerators intended for use with assemblers, aim to get around the problems mentioned above, chiefly by applying restrictions as to what the least element is that can be substituted; typically, this is a single 'symbol', in the assembly language sense, or operator. Where the macro-processer is actually built into the assembler, features already there can be used; this will normally allow a facility to be provided more efficiently and in a less obscure manner than could be done using only string operations. Examples are arithmetic operations, saving of temporary values in the assembler tables, conditional assembly to terminate loops and automatic generation of distinct tags.

Note that though it may be convenient for operational reasons to combine the macrogenerator and the assembler proper into one program, the substitutions performed by the macro-generator are completed before the resulting source line(s) are passed over to the assembler proper which replaces symbolic operation and address fields with numerical quantities.

Macrogenerators provide a simple way of writing machine

/independent

independent programs; if a program consists entirely of macro calls it can be translated into any language by supplying appropriate macro definitions at assembly time though producing efficient object code by this means is rather more of a problem.

Predefined system macros can be used for example, to apply register usage and parameter passing conventions; the user always goes via a macro, thus any changed conventions are applied automatically whenever the program is reassembled. All PDP9-15 supervisor calls under the manufacturer's operating system are generated in this way.

As compared with compiler-based high-level languages, macro-based systems are invariably weak at detecting errors in the source text supplied to them; the usual response is to mistranslate it without informing the user. In particular, they are normally incapable of distinguishing between 'name' and 'value'; in PDP9-15 terminology, if one types ADD P,1 when really ADD P,(1) was meant, the macro generator will neither correct it nor detect it as an error. In their favour, given a source program which is known to be correct, say a compiler written in its own language, a good macro system would provide a quick way to translate it into machine code.

C.Whitfield 23/11/71

Macro Generators(2)

The first half of this document deliberately avoided going into any topic in detail in order to be as discursive as possible; now however we must crystallise our ideas and this will require that we fix on one particular macro system. The one chosen, for obvious reasons, is Macro9-15, the manufacturer's assembly language for the DEC PDP9-15 series of computers. The reader is referred to the Macro-15 manual, especially chapter 4, for details regarding the syntax of the language.

Briefly, Macro9-15 is a fairly typical small-machine, two-pass macro-assembler; it expands all macros on both passes whereas on a larger machine with fast backing store, say a disk, the expanded text would be saved from the first pass for the second, thus saving processor time. As is common with such devices, the standard translator imposes fairly arbitrary restrictions as to what is permitted and what is not, these being as much for the convenience of the implementer as to save the user from himself. For example, recursive macros are permitted provided not more than three levels of recursion occur; on this topic the manual says: '...this process(recursive calling) would never cease if more than three levels were allowed'. Despite restrictions of this sort, the macro processor allows one to write far more readable programs than would be possible without it. The rest of this document is an example intended to provide something a little more substantial than is available in the assembler manual.

The example chosen is a reduced version of an operation required in manipulating DECtape directories; the method used is certainly not the most efficient.

'Given a 256 word block of store which we consider to be made up of eight consecutive 32 word fields, accumulate the inclusive-or of all eight fields into one of them.'

The following three lines describe one way of achieving this:

```
OR 200,(BLOCK+200),(BLOCK)
OR 100,(BLOCK+100),(BLOCK)
OR 40,(BLOCK+40),(BLOCK)
```

where:

numbers are in octal
the 256 word block starts at tag 'BLOCK'
the result is left in the 32 words starting at BLOCK

On the following page are two alternative ways to translate the above program, giving very different assembly code representations.

```
        .DEFIN ORFN,X,Y,Z
        LAC* X; CMA; AND* Y; XOR* X; DAC* Z
        .ENDM
/
        .DEFIN  OR,N,FROM,TO,?LABEL
X=N;    LAW -X; DAC TO#
        LAC FROM; DAC T1#
        LAC TO; DAC T2#
LABEL   ORFN T1,T2,T2
        ISZ T1; ISZ T2
        ISZ TO; JMP LABEL
        .ENDM
/
/
START   OR 200,(BLOCK+200),(BLOCK)
        OR 100,(BLOCK+100),(BLOCK)
        OR 40,(BLOCK+40),(BLOCK)
        .EXIT
/
BLOCK   .BLOCK 400
        .END START
```

```
        .DEFIN ORFN,X,Y,Z
        LAC* X; CMA; AND* Y; XOR* X; DAC* Z
        .ENDM
/
        .DEFIN OR,N,FROM,TO
X=N;    LAW -X
        JMS ORRTN
        .DSA FROM
        .DSA TO
        .ENDM
/
/
        .DEFIN GETVAL,P,V
        LAC* P; ISZ P; DAC V
        LAC* V; DAC V
        .ENDM
/
/
ORRTN
        DAC TO#
        GETVAL ORRTN,T1#
        GETVAL ORRTN,T2#
ORRTN1  ORFN T1,T2,T2
        ISZ T1; ISZ T2
        ISZ TO; JMP ORRTN1
        JMP* ORRTN
/
/
/
START   OR 200,(BLOCK+200),(BLOCK)
        OR 100,(BLOCK+100),(BLOCK)
        OR 40,(BLOCK+40),(BLOCK)
        .EXIT
/
BLOCK   .BLOCK 400
        .END START
```

```
                                        .DEFIN ORFN,X,Y,Z
                                        LAC* X; CMA; AND* Y; XOR* X; DAC* Z
                                        .ENDM
                                /
                                        .DEFIN  OR,N,FROM,TO,?LABEL
                                X=N;    LAW -X; DAC TO#
                                        LAC FROM; DAC T1#
                                        LAC TO; DAC T2#
                                LABEL   ORFN T1,T2,T2
                                        ISZ T1; ISZ T2
                                        ISZ TO; JMP LABEL
                                        .ENDM
                                /
                                /
00000 R                         START   OR 200,(BLOCK+200),(BLOCK)
        000200  A  *G           X=200;  LAW -X; DAC TO#
00000 R 777600  A  *G
00001 R 040457  R  *G
00002 R 200462  R  *G                   LAC (BLOCK+200); DAC T1#
00003 R 040460  R  *G
00004 R 200463  R  *G                   LAC (BLOCK); DAC T2#
00005 R 040461  R  *G
00006 R           *G            ..0003  ORFN T1,T2,T2
00006 R 220460  R  *G                   LAC* T1; CMA; AND* T2; XOR* T1; DAC* T2
00007 R 740001  A  *G
00010 R 520461  R  *G
00011 R 260460  R  *G
00012 R 060461  R  *G
00013 R 440460  R  *G                   ISZ T1; ISZ T2
00014 R 440461  R  *G
00015 R 440457  R  *G                   ISZ TO; JMP ..0003
00016 R 600006  R  *G
                                        OR 100,(BLOCK+100),(BLOCK)
        000100  A  *G           X=100;  LAW -X; DAC TO#
00017 R 777700  A  *G
00020 R 040457  R  *G
00021 R 200464  R  *G                   LAC (BLOCK+100); DAC T1#
00022 R 040460  R  *G
00023 R 200463  R  *G                   LAC (BLOCK); DAC T2#
00024 R 040461  R  *G
00025 R           *G            ..0007  ORFN T1,T2,T2
00025 R 220460  R  *G                   LAC* T1; CMA; AND* T2; XOR* T1; DAC* T2
00026 R 740001  A  *G
00027 R 520461  R  *G
00030 R 260460  R  *G
00031 R 060461  R  *G
00032 R 440460  R  *G                   ISZ T1; ISZ T2
00033 R 440461  R  *G
00034 R 440457  R  *G                   ISZ TO; JMP ..0007
00035 R 600025  R  *G
                                        OR 40,(BLOCK+40),(BLOCK)
        000040  A  *G           X=40;   LAW -X; DAC TO#
00036 R 777740  A  *G
00037 R 040457  R  *G
00040 R 200465  R  *G                   LAC (BLOCK+40); DAC T1#
```

```
    00041 R 040460 R *G
    00042 R 200463 R *G          LAC (BLOCK); DAC T2#
    00043 R 040461 R *G
    00044 R           *G ..0011  ORFN T1,T2,T2
    00044 R 220460 R *G          LAC* T1; CMA; AND* T2; XOR* T1; DAC* T2
    00045 R 740001 A *G
    00046 R 520461 R *G
    00047 R 260460 R *G
    00050 R 060461 R *G
    00051 R 440460 R *G          ISZ T1; ISZ T2
    00052 R 440461 R *G
    00053 R 440457 R *G          ISZ T0; JMP ..0011
    00054 R 600044 R *G
                                 .EXIT
    00055 R 000000 A *G          CAL
    00056 R 000015 A *G          15
                        /
    00057 R         A    BLOCK   .BLOCK 400
          000000 R               .END START
    00462 R 000257 R *L
    00463 R 000057 R *L
    00464 R 000157 R *L
    00465 R 000117 R *L
SIZE=00470     .NO ERROR LINES

BMACRO1 V4A
>
```