SLAC-175

SLAC-175

MAC74, A General Purpose Macro Processor

Joseph C. H. Park

SLAC Report No. 175

September 1974

Under Contract with the U. S. Atomic Energy Commission

Contract AT(04-3)-515

SLAC-175
UC-32
(E/I)

MAC74, A GENERAL PURPOSE MACRO PROCESSOR

JOSEPH C. H. PARK
STANFORD LINEAR ACCELERATOR CENTER
STANFORD UNIVERSITY
Stanford, California 94305

PREPARED FOR THE U. S. ATOMIC ENERGY
COMMISSION UNDER CONTRACT NO. AT(04-3)-515

September 1974

Printed in the United States of America. Available from National Technical Information Service, U. S. Department of Commerce, 5285 Port Royal Road, Springfield, Virginia 22151.
Price: Printed Copy $5.45; Microfiche $1.45.

Abstract

We describe a general purpose macro processor, which is easy to use, has many versatile features and is designed to be macro-processed for easy modifications and extensions.

Part I shows how the processor is used for various purposes in actual examples of programming. Part II is a user's guide and reference containing a detailed description of the processor in this aspect.

Acknowledgement

This processor grew because of needs. Whenever such occasion arose, features were implemented or modified in a way that was the easiest and shortest that I could think of without belaboring on how others do. Books and papers were consulted whenever basic principles were concerned. They will be acknowledged in a subsequent report, describing the processor in such aspects.

I owe the concept of REMOTE to the RMT pseudo-op of the assembler, SCATRE, for the IBM 7090-7094 at the University of Illinois, Urbana, Illinois, in the early 1960's. I learned the idea of using macros to "talk in a concise and invariant way" with programs that are beyond users' control and subject to change, from the IBM OS for 360 series relating to their Supervisor and Data Management Macros.

The early form of the processor is due to Mr. John Ahern in 1968, when he was also associated with Experimental Group D at SLAC. I would like to thank him for his inspiring and farsighted way of programming. There are other programs that he wrote "for us", all of which consistently met such a standard.

I had numerous occasions to talk with Dr. John Ehrman of the Stanford Center for Information Processing, Stanford University, during many years, dating back to the Illinois days. I would particularly like to thank him for making available his "huge collection" of papers dealing with macro processing.

I am grateful to Professor Robert F. Mozley of Experimental Group D at SLAC for his support in many ways throughout these years.

Table of Contents.

Introduction.

This report describes a general purpose macro processor, which is easy to use, has many versatile features, and is designed so that it is easily modified and extended by a user. The processor at various stages of development has been successfully used as a general purpose programming tool (mainly privately) for many years at SLAC.

The first version dating back to early 1968 was written by J. Ahern at SLAC. As documented elsewhere[1] the processor in this early form had the basic facility of macro definition and expansion as well as replacement features (which are after all macros in disguise). The processor even in this state was extremely useful such as in preparing data cards for SUMX, which in fact was one of the original motivating needs to be met.

Later more higher-level-language-like features, expressions (for macro-time evaluation), built-in functions, if-then-else statements, branching directives, statement labels and the like were implemented by the present author in order to satisfy a wider varitety of probramming needs.

Soon the processor began to take on a shape of a full fledged language and modifying existing features or adding new ones became progressively more cumbersone and error prone. Eventually it dawned on the author that the "mechanical" part of such programming should be "macro processed". At this time the processor was reworked in its entirety so that the parts likely to be subject

to further extension are all together "generated" by macro processing.

Another parallel direction pursued was developing an operator INVOKE for a truly "do-anything" purpose and additional features to aid in debugging existing parts of or trying out new ideas for the processor itself.

By this time the processor was in such a constantly changing state that it was almost impossible to continue without a doucment on a working version. On the other hand the present state of the processor is such that for the sake of further work as well as thorough testing it is desirable to subject it to use under more diverse circumstances by more people. It is chiefly for these reasons that this report was prepared.

Part I contains introductory remarks about a macro processing in general. The main purpose of this part is to illustrate what our macro processor does by simple examples, most of which are based on actual applications.

Part II is chiefly a (passive) users's guide and reference for our processor. More elaborate examples are also given in this part.

In the above two parts we dealt chiefly with describing the processor in a user's point of view. In a subsequent report we intend to continue the description in another aspect, namely that of a programmer. This forthcoming report will consist of the following parts.

Part III describes basic design principles employed in a macro processor and the inner workings of our program. The aim here is to help an active user in prusuing a further modification and extension of this processor.

Part IV shows how we have, by means of our macro processing, "mechanized", to an extent, the act of modifying and extending the processor itself. The extent here is obviously dynamic (constantly expanding) in that through this process we generate a more powerful processor with which to further mechanize this process. This part also serves to illustrate non-trivial macro processing in the language of our processor.

In Part V the reader may find actual listings of the processor and the processor-processor (both of which may already be obsolete).

PART I. What Does the Processor Do ?

Our intention here is to give an introductory description of our macro processor. Therefore we should begin by answering what is meant by "macro processing".

(1) What is macro processing ?

The idea of a macro processing in its primitive form probably originated in an assembly language programming, in which the macro feature allows a programmer to define a group of instructions to be a macro "instruction" and use it as if it were a single machine instruction. The macro facility thus enables a programmer to invent more powerful "instructions" of his own design out of basic machine operations as well as to avoid the boredom of repeating a block of codes.

A macro processor in this conventional sense is deemed to be an extension of the basic assembler and may be added to it as a preprocessor. The principal task of such a preprocessor is to copy the input text to an output file replacing ("expanding") each macro instruction by its definition. The output from this pass, consisting entirely of instructions known to the basic assembler, is then fed to it. In practice such macro processing is implemented directly into the initial pass of an assembler eliminating the overhead of an intermediate file and combining with other initial functions of the assemlby. A macro processor in this form is such an integral part of the assembler that it is very likely impossible to isolate it for the purpose

of using it elsewhere.

As the art of macro processing developed further by abstracting (generalizing) basic ideas and by acquiring more features, it soon became to be recognized as a language in its own right, divorced from the assembler and designed to process any language including high level compiler languages. A macro processor in this sense is better named a general language preprocessor.

In addition to the basic facility of macro (with arguments, to broaden its usefulness) definition and expansion the processor must be able to evaluate arithmetic or logical expressions and execute branching, conditional or otherwise. In short it must be powerful and versatile enough so that a user can easily and compactly express in this language how to process a given text (in a base language).

(2) Our processor.

Our processor is a general purpose (that is, base language independent) macro processor. It has the advantage over other existing macro processors[2] in the following major aspects.

a. It is easy to learn.

Rules used are whenever possible the same as those of PL/I or FORTRAN. Statements that look familiar do in fact the "expected". Quite often the outside appearance can be made exactly alike by

renaming (through a processor command) such as SUBROUTINE for MACRO, END for MEND, etc.

b. It is versatile.

In addition to basic features to be expected for a macro processor it has a versatility naturally inherited from PL/I (in which the bulk of the processor is written) and the IBM O(perating) S(ystem) support, such as built-in functions based on PL/I functions identically named.

c. It is easy to modify and extend.

Having written the basic components, such as the lexical scanner, symbol table manager, stacking routine for expressions, etc., in PL/I, which may have longer life-expectancy, other areas of the processor, which are likely to be subject to modification and expansion, such as the part dealing with processor commands, built-in functions, etc., are generated by the macro processing. That is, through the processor language we tell a user in a precise and mechanical manner how these features are implemented. This perhaps is the chief advantage of our processor (for who can write a macro processor to everybody's satisfaction ?)

(3) Illustrations.

We continue with our explanation of what the processor does by illustrating how it is used to meet various needs. Most of these examples come from actual applications, some of which served

in fact to motivate new features in our processor.

The examples to follow in this part are deliberately chosen to be simple and easy to understand. Also the explanation of the processor commands used in these examples are kept to be brief. A full description of features in our processor and their rules is given in Part II. More complicated use (in that they are almost full fledged programs in the language) of our macro processor may be found in Part IV.

a. COMMON declarations in FORTRAN.

In FORTRAN the scope of a variable is ordinarily limited to one SUBROUTINE. "Global" variables to be shared among several SUBROUTINES are to be declared under COMMON with each SUBROUTINE involved having an exact copy of the shared COMMON.

The following example, which is taken from the kinematic fitting routine, FIT73[3], shows how the macro processing solves this problem.

We first define a macro named DCLFIT which has all the COMMON declarations to be shared in one place.

```
):
):  COMMON DECLARATIONS FOR FIT AND BLOCKS. IF EITHER A1 OR A2
):  IS 'T', THEN THE UNLABELLED COMMON FOR TEMPORARY SAPCE IS
):  DELETED. SIMILARLY 'G' FOR THE STATEMENT FUNCTION DEALING
):  WITH THE ERROR MATIRX.
):
)         MACRO DCLFIT(A1,A2)
          IMPLICIT REAL*&L (A-H,O-Z)
          COMMON /$BLK1/ MMD,NOT4,NCALL,NP,SN(2),TMASS,RMASS(&NPX),
     1 XMEAS(&NVX),GINA(&NV2)
     2 /$BLK2/ NV, MISSN,MP,MV,NMISS(4),NMISSP(4),LCR(&NPX),
     3 X(&NVX), XOLD(&NVX),DX(&NVX),P(4,&NPX),PSQ(&NPX)
     4 /$BLK3/ LC,MQ, ALFA(4),F(4),B(4),HIN(16),PLI(&NV4),
```

```
     5 FIL(&NV4),ELI(&NV4),V(&NV4),C(4,4),UT(3,3),Q(4)
     6 /$BLK45/KI(4),CI(6),NRJCT,NSTEPS,MODE,IPUSH,IWASCT,IARECT,ICHEAT,
     7 ISNEAK,MARK,NROOT,LIGHT,IL,IM,ML,CHIOLD,CHINEW,DCHI,DFOLD,DELTAE,
     8 DEL,AQ,BQ,CQ,EL,EM,PL,UL(3),PIL,DISQ,ROOT
     9 /$BLK6/ STR(&NVX),GINB(&NV2)
)         IF TIMING THEN CALL TAUC : COMMON /$TAU/ IF NEEDED
)         IF A1='T' | A2='T' THEN GOTO IG : SKIP TEMP
          COMMON    TEMP,TMPI(4),TMPJ(&NVX)
) IG    : IF A1='G' | A2='G' THEN GOTO MEXIT : SKIP IG
                    IG(I,J)=NV*(J-1)+I
) MEXIT : MEND
):
```

Subsequently in each SUBROUTINE, where these COMMONs are needed, we insert a processor statement calling for the macro DCLFIT.

```
          BLOCKDATA
)         CALL DCLFIT('T','G')
           .........
           .........
          END
          SUBROUTINE BLOCK1(*)
)         CALL DCLFIT
           .........
           .........
          END
          SUBROUTINE BLOCK2(*,*)
)         CALL DCLFIT
           .........
              etc.
```

Because the user's FORTRAN (base language) statements and the processor language statements that look alike are intermixed, we require as a distinguishing feature that each processor statement must have a marker ")" (or anything else the user wishes) in the first column.

Note that several dimensioned variables in DCLFIT are declared with processor variable names like &NPX, &NVX, etc., which are to be substituted by the processor with values assigned by the user. In this particular example the user need only to specify

one parameter, viz, the maximum number of particles to be fitted at a vertex, say 11,

```
) NPX=11          : MAXIMUM NUMBER OF OUTGOINT PARTICLES
);                  INCLUDING NEUTRAL AND INCIDENT IF ANY
);                  NOTE THAT ALL OTHER DIMENSIONS (VARIABLE WRT
);                  NO OF PARTICLES) ARE DERIVED FROM THIS.
```

All the remaining varaibles are then derived by the processor as below.

```
) NVX=3*NPX       : NUMBER OF VARIABLES
) NV4=4*NVX       : SIZE OF FLI AND SO ON
) NV2=NVX*NVX     : SIZE OF ERROR MATRICES
```

Assignment statements as above automatically declare the receiver variables to be eligible for replacement anywhere, both inside and outside a macro. Variables to be replaced, if used in a base language statement, are prefixed by an "escape" character "&" (or any other character or null) to distinguish them from those in the base language. The "&" in this context is another form of macro CALL.

b. "Firm" storage allocation in FORTRAN.

Example a. also indicates how through the macro processing array sizes are easily and consistently changed throughout a program. Remember that a size parameter may arise in different context than the declaration anywhere in the text. The effect is somewhere between the completely fixed method (the only one) in FORTRAN and the automatic one in PL/I (with the most OS overhead).

On a 360 or 370 series of computers, if a length of a COMMON is too large, then the performance of the program itself may be

adversely affected (because of the additional base register loading needed) so that for a production program like that of kinematic fitting the "size" needs to be readjusted frequently. In our case it ranged from NPX=11 for a photoproduction experiment to NPX=3 for a K decay experiment, which is a significant reduction in COMMON area.

c. Better readability.

The following is a segment of a FORTRAN program, (which we happen to come across) purporting to calculate the square of the matrix element in the decay,

K -> PI MU NU GAMMA

```
CCC    TAKE A DEEP BREATH !!!!
       ATT1= -(AP/DOT(1)**2+AM/DOT(2)**2+2.*DOT(7)/(DOT(1)*DOT(2)))
      1       *(4.*AFP**2*(2.*DOT(10)*DOT(5)+AK*DOT(6))-AF2**2*AM*DOT(6)
      2   +4.*AM*AFP*AF2*DOT(5))
          ATT2    =                    -8.*AFP**2*DOT(5)*((DOT(7)/(DOT(2)*DOT(1))
      3 +AM/DOT(2)**2)*DOT(3)-DOT(8)/DOT(1)+DOT(10)/DOT(2))
       ATT3=
      4 -(4.*AK*AFP**2-AM*AF2**2)*((DOT(7)/(DOT(2)*DOT(1))+AM/DOT(2)**2)
      5 *DOT(4)-DOT(9)/DOT(1)+DOT(6)/DOT(2))
             ATT4          =                          -2.*(((DOT(7)/DOT(2)/DOT(1))
      6 +AP/DOT(1)**2)*DOT(3)-DOT(10)/DOT(2)+DOT(8)/DOT(1))
      7 *(8.*AFP*ALAMP*(2.*DOT(10)*DOT(5)+AK*DOT(6))
      8 -2.*AM*AF2*DF2DT*DOT(6)+4.*AM*(AFP*DF2DT+AF2*ALAMP)*DOT(5))
       ATT5 = 4.*AFP**2*(   (DOT(3)/DOT(2)) *  (AK + 2.*DOT(10)  +
      2    2.*DOT(8)  + 2.*DOT(3)  )    +   AK*(1.  +  DOT(1)/DOT(2))  )
       ATT6 = AM*AF2**2*  ( DOT(3)/DOT(2)  -  DOT(1)/DOT(2)  -  1.)
       AMT = ATT1 + ATT2 + ATT3 + ATT4 + ATT5 + ATT6
```

where DOT(1),....,DOT(10) are Lorentz products of two four-vectors in various combinations of this decay.

For improved readability the author might have macro processed

this part as follows

```
)       SET ESCAPE ''                                   ; Line 1

 ATT1=-(AP/PIGAM**2+AM/MUGAM**2+2.*MUPI/(PIGAM*MUGAM))
1       *(4.*AFP**2*(2.*KMU*KNU+AK*MUNU)-AF2**2*AM*MUNU
2       +4.*AM*AFP*AF2*KNU)
 ATT2=-8.*AFP**2*KNU*((MUPI/(MUGAM*PIGAM)
3       +AM/MUGAM**2)*KGAM-KPI/PIGAM+KMU/MUGAM)
 ATT3=
4       -(4.*AK*AFP**2-AM*AF2**2)*((MUPI/(MUGAM*PIGAM)+AM/MUGAM**2)
5       *NUGAM-NUPI/PIGAM+MUNU/MUGAM)
 ATT4=-2.*(((MUPI/MUGAM/PIGAM)
6       +AP/PIGAM**2)*KGAM-KMU/MUGAM+KPI/PIGAM)
7       *(8.*AFP*ALAMP*(2.*KMU*KNU+AK*MUNU)
8       -2.*AM*AF2*DF2DT*MUNU+4.*AM*(AFP*DF2DT+AF2*ALAMP)*KNU)
 ATT5=4.*AFP**2*((KGAM/MUGAM)*(AK+2.*KMU+
2       2.*KPI+2.*KGAM)+AK*(1.+PIGAM/MUGAM))
 ATT6=AM*AF2**2*(KGAM/MUGAM-PIGAM/MUGAM-1.)

)       SET ESCAPE &                                    ; Line 2
```

The escape names used such as PIGAM for DOT(1), MUGAM for DOT(2), etc., might have also been generated using a general purpose macro called NAME as below.

```
): ARGUMENTS BECOME ESCAPE NAMES WITH VALUES ASSIGNED BY
): RHS WITH A RUNNING INDEX I
)       MACRO NAME(A1,A2,A3,A4,A5,A6,A7,A8,A9,A10)
)          I=I+1 ; INCREMENT INDEX USED IN RHS
)          &A1=RHS                                      ; Line 3
)       IF A2='' THEN GOTO MEXIT                        ; Line 4
)       CALL NAME(A2,A3,A4,A5,A6,A7,A8,A9,A10)
)   MEXIT: MEND
```

This macro in our example might have been used as

```
)   RHS='DOT(&I)'    ; Value is a character string
)   I=0 ; Initialize index
)   CALL NAME('PIGAM','MUGAM','KGAM','NUGAM','KNU','MUNU')
)   CALL NAME('MUPI','KPI','NUPI','KMU') ; continuation of above
): How about having some Greek in EBCDIC ?
```

The effect of the command in Line 1. is to declare every

token (word, lexical unit) in the text to be eligible for replacement. A token is replaced only if a value has been previously assigned to it. To minimize subsequent processing time the command in Line 2. restores the prefix & for escape names.

The macro NAME shows one way of using "recursion". Each time NAME CALLs i t s e l f (i.e. at each new level of recursion) the next argument in the original argument list plays the role of the "first" argument &A1. This loop is terminated by the test in Line 4. when the argument list is exhausted.

Since arugments are names for processor variables being made, Line 3. specifies &A1 on the left had side rather than A1 and NAME is CALLed with arguments, each of which is a character string enclosed by quotes.

d. Lexical scanning.

The replacement feature of the macro processing appears so simple that one may ask (in fact, did), can one achieve the same result using a text editor like WYLBUR ?

The answer is with difficulty, because a general purpose (interactive) text editor like WYLBUR functions by unconditionally replacing one character string with another regardless of whether the target string is a word or a subpart of another word, that is, without a lexical analysis of recognizing unbreakable atoms.

The following is a series of WYLBUR commands in an EXEC FILE

to be executed in sequence by WYLBUR assuming the text in which these changes are to be made is in the ACTIVE FILE.

```
CHANGE '&ABS'      TO 'EVBFN(1)'  NOLIST
CHANGE '&CONCAT'   TO 'EVBFN(2)'  NOLIST
CHANGE '&INDEX'    TO 'EVBFN(3)'  NOLIST
CHANGE '&LENGTH'   TO 'EVBFN(4)'  NOLIST
CHANGE '&MAX'      TO 'EVBFN(5)'  NOLIST
CHANGE '&MIN'      TO 'EVBFN(6)'  NOLIST
CHANGE '&MOD'      TO 'EVBFN(7)'  NOLIST
CHANGE '&SUBSTR'   TO 'EVBFN(8)'  NOLIST
CHANGE '&TRANSL'   TO 'EVBFN(9)'  NOLIST
CHANGE '&UNSPEC'   TO 'EVBFN(10)' NOLIST
CHANGE '&VERIFY'   TO 'EVBFN(11)' NOLIST
CHANGE '&MAXLSTR'  TO '80'        NOLIST
```

This example might have worked, were it not for the fact that one escape name is a part of another escape name and therefore the result is sensitive to the order of replacement. If the warning prefix "&" were not used, the situation would have been unpredictably worse. Clearly macro processing with its lexical analysis is better suited for such purpose.

e. Program tailoring.

Quite often a general purpose program needs to be tailored for a particular task. It would be inelegant, to say the least, to make a new copy and introduce permanent changes in it for every such occasion. We would instead like to "derive" a particular version from the original one so that only one version has to be maintained at all times. With macro processing we can solve this problem by collecting particular modifications into macros and conditionally invoking them in the text depending on a key word.

It has been described elsewhere[3] how the macro processing is used to tailor FIT73, a general purpose kinematic fitting routine, to treat the time-of-flight measurement for K in the K-decay experiment. The conditional CALL to TAUC in DCLFIT of Example a. belongs to this modification.

We include here another more straightforward case. A problem arose in generating by Monte Carlo method events of the kind,

K -> PI+ PI- PIO GAMMA

Because the energy spectrum of emitted photon is approximately 1/k, the general method (to be shown below) is absurdly inefficient. We need to modify the program to do an "importance sampling" in the variable k. A part of this program shown below illustrates trivially how this modification was introduced into the general purpose routine without destroying it.

The following is a front half of the SUBROUTINE GENEV of version dated March 74 by J. Park which is due to a version dated June 69 by G. D. Chandler which is due to a version dated April 68 by G. Ascoli which is based on a program of the same name by G. Lynch once upon a time.

```
C
COMPUTE KINETIC ENERGY AVAILABLE
      EKIN=TECM-SMF
      W=-&ONE
                IF (EKIN .LE. &ZERO) GO TO 999
C
CONSTRUCT EMM(1...N)=M1,M12,....,TECM
 11   EMM(1)=EMF(1)
      EMM(NPF)=TECM
```

```
      IF(NPF-2) 999,50,15
);    .....................
)        MACRO GENERAL
);    .....................
C
CHOOSE N-2 RANDOM NUMBERS IN  (0,&ONE)
 15   DO 20  I=3,NPF
 20   RA(I-2)=RANUM(R)
C
C  'FOLD'THEM SO THEY ARE IN ASCENDING ORDER
      DO 40  I=3,NPF
      RMIN=&TWO
      DO 30  J=3,NPF
      IF (RA(J-2)-RMIN) 25,30,30
 25   RMIN=RA (J-2)
      IT=J
 30   CONTINUE
      X(I-2)=RMIN
 40   RA(IT-2)=&TWO
C
C     NOW HAVE  X(1),....,X(N-2) CHOSEN AT RANDOM FROM THE UNIFORMLY
C     POPULATED'TRIANGLE'
C          0  <  X(1) ...  <  X(N-2)  <  &ONE
C     COMPUTE   M12   EMM(2)=EKIN* X(1)+EMF(1)+EMF(2)   AND SO ON.
C
      S=EMM(1)
      DO 45 I=3,NPF
      S=S+EMF(I-1)
 45   EMM(I-1)=EKIN*X(I-2)+S
);    .....................
)        MACRO X21        ; SAMPLING IN K -> PI L NU GAMMA
);    .....................
 15   EMM(3)=SQRT(EM(1)*(EM(1)-&TWO*XKMIN*EXP(ALPHA*RANUM(R))))
      EMM(2)=RANUM(R)*(EMM(3)-EMF(1)-EMF(2)-EMF(3))+EMF(1)+EMF(2)
);    .....................
)        MEND
)        IF X21 THEN CALL X21
)        EL'SE CALL GENERAL
);    .....................
C
C     GET MAGNITUDE OF MOMENTA FOR SEQUENTIAL DECAYS,STARTING FROM LAS
C     PD(1)=MOMENTUM FOR (1+2) TO 1 AND 2              1+2 FRAME
C     PD(2)=MOMENTUM FOR (1+2+3) TO  (1+2) AND 3     1+2+3 FRAME ,ETC.
C     COMPUTE  W=PD(1)...*PD(N-1)*EKIN**(N-2)/TECM
C
 50   W=&ONE/EKIN/TECM
      DO 60 I=2,NPF
)     CALL QCM('PD(I-1)','EMM(I)','EMM(I-1)','EMF(I)')
 60   W=W*PD(I-1)*EKIN
C
      GO TO 999
```

Incidentally with this modification the run time in CPU was

reduced by a factor of 100. The reduction in similar cases with leptons would be even more spectacular because of higher Q values.

f. Generating data cards in SUMX.

SUMX* used to be (at SLAC and is elsewhere) extensively used as a tool for examing a massive collection of data. This is a package of versatile programs, to which user commands are communicated as "Data Cards" satisfying strict rules. For example a histogram requires at least three cards like

```
  INVARIANT MASS OF P PI+ (RHO IN)
100        0.04       1.08       21
722        10
```

Years ago it was not unusual to read in a trayful of such cards for one SUMX run and watch the job abort because of one missing card somewhere. The card might have been inadvertantly left out in hand-editing the deck to make a change, say, in the bin size of each histogram. We owe one of the original motivation for the present macro processor to such frustrations.

With macro processing one might proceed as follows. First define a macro for making a histogram including rapidly changing parameters as arguments and the rest as global escape names.

```
)   MACRO HIST(HZ,LOC,TITLE)
  &TITLE
&&NH       &&DH       &&HZ       &&NPT      &&FAC                  &&LOG
&&LOC      &&WT       &&NT       &&SGM
)   MEND
```

Values to global variables are assigned once for all by invoking a macro like:

```
)   MACRO DEFAULT
) NPT=21   : PRINCIPAL TEST NUMBER (RHO TEST)
) FAC=''   ; FACTOR TO SCALE
) LOG=''   ; 'LOG' OR ANYTHING TO GET LOG HISTOGRAM
) WT=10    ; LOCATION OF EVENT WEIGHT
) NT=''    ; TEST ASSOCIATED WITH MULTIPLICITY ELEMENT
) SGM=''   ; LOCATION OF ERROR FOR IDEOGRAM
) NH=100   ; NUMBER OF BINS IN A HISTOGRAM
) DH=0.04  ; HISTOGRAM BIN WIDTH (MEV)
)   MEND
```

Furthermore the user's event data can also be symbolically addressed by using a macro like;

```
)   MACRO FORMAT
); NUMBERS IN RHS REFER TO LOCATIONS IN BOUT
  ........
) M35=722 ; INVARIANT MASS OF PARTICLE 3 AND 5
  ........
)    MEND
```

Not only are names more convenient to use and less prone to mistake than numbers, but also the use of macro like FORMAT makes the processing deck insensitive to the format change.

The set of three cards for a histogram mentioned above can now be generated by a single line command like

```
) CALL HIST(1.08,M35,'INVARIANT MASS OF P PI+ (RHO IN)')
```

The scheme requires some preliminary work such as defining general purpose macros, defaults for global symbols, etc., (which is equivalent to the SUMX manual as viewed by the processor). But once done, they may be collected into the MACLIB and automatically invoked whenever needed. A more elaborate scheme has been

described elsewhere[5], although the processor described there is now obsolete.

g. Using KIOWA.

Being designed for a similar purpose, KIOWA[6] is superior to SUMX in several hidden aspects as well as obvious, one of which is the flexibility in using it, because a full fledged language, namely FORTRAN, is used to drive it (rather than "barren" data cards as in SUMX).

In this scheme a histogram, for example, requires two separate operations, defining each histogram through COMMON arrays, HZ for lower limit, DH for bin width, NH for number of bins, IH for plot type and ILAB, HLAB for title on the one hand and invoking a SUBROUTINE, CALL HIST(I,A,WT), for each entry into an I-th histogram with quantity A and entry weight WT on the other.

Since it would be wasteful to write executable codes for the defining operations, which need be done only once in the beginning, the only alternative in FORTRAN is to collect these definitions under a BLOCKDATA routine in a place remote from where updating CALLs to the corresponding histograms are made. How can we keep for each histogram these two parts together ?

In order to make this possible a processing feature REMOTE was introduced such that a text following a command of the form

) SET REMOTE BEGIN

and upto another command like

) SET REMOTE END

is temporarily collected by the processor to be put at the end of the main text, when the processing of the main text terminates.

The following macro illustrates how this feature may be used to cause a single CALL to it emit the defining part remotely and generate the updating CALL to HIST locally.

```
);
);   HIST DEFINES A HISTOGRAM, IF NEW, REMOTELY IN A BLOCK DATA,
);   AND GNERATES THE HISTOGRAM UPDATING CALL.  NOTE THE PARAMETERS
);   TEST, IH, AND WT ARE TO BE DEFINED EXTERNALLY RATHER THAN
);   THRU THE PARM LIST, SINCE THEY ARE VARYING LESS FREQUENTLY
);   THAN THE ONES IN THE PARM LIST.
);
);   IN ADDITION NCAL AND NHISTO MUST BE SET. NCAL = 1(0) FOR
);   TRUE(FALSE). IF TRUE, ONLY THE HISTOGRAM DEFINING PART
);   OF THIS MACRO WILL BE EXPANDED. NHISTO IS AN INTGER WHICH
);   GETS ADDED TO THE SPECIFIED SEQUENCE NO TO OBTAIN THE ACTUAL
);   HISTOGRAM SEQUENCE.
);
)         MACRO HIST(I,A,HZ,DH,NH,TITLE)
)    I=I+NHISTO                    ; ADD IN THE ORIGIN
)    IF I < NHIST+1 , GOTO MAKE    ; THIS HIST HAS BEEN DEFINED EARLIER
)    NHIST=NHIST+1                 ; OTHERWISE, DEFINE A NEW ONE
)    SET REMOTE BEGIN              ; EMIT DEFINITION REMOTELY
          DATA HZ(&NHIST)/&HZ/,DH(&NHIST)/&DH/,NH(&NHIST)/&NH/,
      1        IH(&NHIST)/&IH/,HLAB(1,&NHIST)/'&TITLE'/
)    SET REMOTE END                ; COME BACK
)    MAKE: IF NCAL , GOTO MEXIT    ; SKIP CALL IF NO_CALL TRUE
          &TEST CALL HIST(&I,&A,&WT)
)    MEXIT: MEND
```

h. A FORTRAN programming style.

The following program is selected because it is short enough to be included here in its entirety to illustrate a style of FORTRAN programming with macro processing.

The program is self-contained apart from (1) a general purpose

integrator, DSINT for double precision and SINT otherwise, and a mass table, MASS, with the same prefix convention, both of which are on user's "SYSLIB" and (2) general purpose macros, PROLOGUE and RETURN for stated purposes, which are on user's MACLIB.

```
)':  WE WANT TO INTEGRATE (PER BIN IN THE T-HISTOGRAM) A
)':  FUNCTION, THE INVARIANT PHASE SPACE IN THE DECAY,
)':  K -> 3 PI, MULTIPLIED BY A FORM FACTOR, WHICH IS
)':  ASSUMED TO BE VARIOUS POWERS OF THE KINETIC ENERGY T
)':  OF PI0 IN THE K REST FRAME.  WE WANT TO DO THIS FOR
)':  &IMAX CASES STARTING FROM T**0, TO T**(IMAX-1).
)':
)':                                  JULY 29, 1974 J. PARK
)':
)       CALL PROLOGUE(8)  : DEFINE FUNCTION PREFIX &D,
)':                           CONSTANT SUFFIX &S,
)':  PRECISION LENGTH &L AND CONSTANTS. THE ARGUMENT SHOULD BE
)':  4(8) FOR SINGLE(DOUBLE) PRECESION. PROLOGUE ALSO INITIALIZES
)':  CREATED SYMBOLS FOR GENERATED LABELS ETC.
)':
) DELTA='1.&S.0'    : BIN WIDTH 1 MEV
) TMAX='54.&S.0'    : MAX TPI0. MAY NOT BE EXACT
) TEST='1.&S-4'     : % ERROR IN INTEGRATION
) LIM=12            : SEE WRITE-UP OF SINT
) IMAX=5
)':
      IMPLICIT REAL*&L (A-H,O-Z)
C
      DIMENSION SIGMA(&IMAX)
C                   INITIALIZE INTEGRANDS
      CALL INIT($)
C                   LOWER LIMIT
      T1=&ZERO
C                   UPPER LIMIT
10    T2=T1+&DELTA
)':
)     MACRO DO
)     I=1
)     FORM=''       ; SUCCESSIVE POWERS OF T
) MORE :
)':                 DECLARE FUNCTION TO BE INTEGRATED
      EXTERNAL PFSQ&I
      SIGMA(&I)=&D.SINT(T1,T2,&TEST,&LIM,NOI,R,PFSQ&I)
)     SET REMOTE BEGIN ; GENERATE TEXT FOR INTEGRAND
      ENTRY PFSQ&I(T)
         EPI0=MPI0+T
         M12SQ=MKSQ+MPI0SQ-&TWO*MK*EPI0
         PFSQ=T*(EPI0+MPI0)*(&ONE-&FOUR*MPIMSQ/M12SQ)
         IF (PFSQ .LT. &ZERO) PFSQ=&ZERO
```

```
            PFSQ&I=&D.SQRT(PFSQ)&FORM
            RETURN
)       SET REMOTE END ; COME BACK
)       IF I=1 THEN FORM='*T' ; NEXT FORM FACTOR
)       ELSE FORM='*T**&I'
)       I=I+1
)       IF I < IMAX+1 THEN GOTO MORE
)       SET REMOTE BEGIN
        END
)       SET REMOTE END
)       MEND
);
)       CALL DO
C                   ANSWERS, SIGMA, ARE TO BE USED IN
C                   FORTRAN DATA STATEMENTS.
        WRITE(6,11) SIGMA
);  I CAN NEVER GET THE PARENTHESIS IN FORMAT TO WORK RIGHT
)       E=',E14.6,'','''
11      FORMAT(6X &E &E &E &E)
        T1=T1+&DELTA
        IF (T1 .LT. &TMAX) GOTO 10
)       CALL RETURN
);
        FUNCTION INIT(DUMMY)
        IMPLICIT REAL*&L (A-Z)
);
)       MACRO MASSES(A1,A2,A3,A4,A5,A6,A7,A8)
        &A1=&D.MASS(&A2,M)
        &A1.SQ=&A1*&A1
)       IF (A3='')=0 THEN CALL MASSES(A3,A4,A5,A6,A7,A8)
)       MEND
);
)       CALL MASSES('MK',11,'MPIO',8,'MPIM',7)
        RETURN
);  TO BE CONTINUED WITH THE TEXT REMOTELY ASSEMBLED
);
```

i. Miscellaneous.

We describe yet another example of how a macro processing facilitates the use of a program which is inherently complicated and cumbersome to use otherwise. The case deals with using an I/O routine with entry names, JOPEN, JCLOSE, JREAD and JWRITE, and identified under a CSECT name of JHPBPAM, which supports a P(artitioned) D(ata) S(et) in FORTRAN.

The basic function of JHPBPAM is to let a user invoke in FORTRAN various OS data management macros having to do with the B(asic) P(artitioned) A(ccess) M(ethod). The user is asked to supply a work area for each member and for different operations be it read or write, in which JHPBPAM will store and use all things pertaining to his I/O. In this way the user has access to "inside" information ordinarily unavailable in FORTRAN such as DCB and furthermore JHPBPAM becomes reentrant.

Who needs it ?

This routine was originally written as a basic tool in setting up a "generalized" job, that is the JCL leaves undefined to some extent what data the user wants to read or write as well as what program he really wants to run, all of which are to be determined when the job goes into execution. This scheme was successfully used to "beat" the Job Queue delay. We also used JHPBPAM to "simultaneously" (multi-tasking) run several KIOWA jobs accessing different data within the "elapsed time" it would have taken to run one such job.

In order to make life easy a series of macros were written as shown below. They form a part of our MACLIB so that we never have to remember details again.

For instance, having once CALLed DCLWORK and initialized generated labels, SYSLBL1,...,SYSLBL3, one may do

) CALL OPEN('READ') and/or ('WRITE')

for read and/or write once. And subsequently

) CALL READ('member name',...)

to read a record of the specified PDS member, or

) CALL WRITE('member name',...)

to write such a record. Finally

) CALL CLOSE('READ') or ('WRITE')

closes the data set.

```
);
):  THIS MACRO SUPPLIES READ/WRITE WORK AREAS FOR USE WITH
);  JHPBPAM. IT ALSO GENERATES ESCAPE NAMES ADDRESSING
):  ELEMENTS OF THESE AREAS. USE OF THESE NAMES WILL MAKE
);  THE PROGRAM INSENSITIVE TO CHANGES IN WORK AREAS.
);
)          MACRO DCLWORK(MORE)    :   LAST EDITED ON 5/28/74
      COMMON/$WORK/ IRW(70),IWW(66) &MORE
      INTEGER*2 IRWH(140),IWWH(132)
      EQUIVALENCE (IRWH(1),IRW(1)),(IWWH(1),IWW(1))
);        DEFINITION OF WORK AREA FOR WRITING
)         CALL DCLW1('W',2,3,4,9,10,12,7,13,35)
);        DEFINITION OF WORK AREA FOR READING
)         CALL DCLW1('R',4,5,6,17,18,20,11,17,39)
)         MACRO DCLW1(A1,A2,A3,A4,A5,A6,A7,A8,A9,A10)
) &A1.NAME1='I&A1.W(&A2)'     ; FIRST HALF OF MEMBER NAME
) &A1.NAME2='I&A1.W(&A3)'     ; SECOND HALF
) &A1.TTR  ='I&A1.W(&A4)'     ;  TTR
) &A1.CNT  ='I&A1.WH(&A5)'    ; TOTAL NO OF I/O ACCESS
) &A1.ERR  ='I&A1.WH(&A6)'    ; TOTAL NO OF ERRORS
) &A1.SYN2 ='I&A1.WH(&A7)'    ; SYNCHRONOUS ERROR FLAG
) &A1.TOTAL='I&A1.W(&A8)'     ; TOTAL NO OF BYTES READ OR WRITTEN
) &A1.DCB  ='I&A1.W(&A9)'     ; DCB 22 FULL WORDS
) &A1.SYNMSB='I&A1.W(&A10)'   ; MESSAGE BUFFER 32 FULL WORDS
)         MEND
);
);  OPEN('R') OR ('W') OPENS A DATA SET UNDER THE DD NAME OF
);  IN OR OUT RESPECTIVELY SPECIFYING A PDS.
);
)         MACRO OPEN(R,LABEL)
)         R=SUBSTR(R,1,1)     ; USE FIRST CHAR ONLY
&&LABEL   J=JOPEN('&R',I&R.W)
)         CALL TESTJ('OPEN')
)         MEND
);
);  SIMILARLY FOR CLOSE('R') OR ('W').
);
)         MACRO CLOSE(R,LABEL)
)         R=SUBSTR(R,1,1)     ; USE FIRST CHAR ONLY
&&LABEL   J=JCLOSE('&R',I&R.W)
```

```
)          CALL TESTJ('CLOSE')
)          MEND
);
);  MEMBER : HOLDS 8-CHAR. MEMBER NAME (PADDED WITH BLANKS)
);  LENGTH : LENGTH IN BYTES OF DATA TRANSMITTED TO
);  RECORD : TARGET AREA
);  LABEL  : IF NOT GIVEN, GENERATED LABEL SYSLBL1 IS USED
);
)          MACRO READ(MEMBER,LENGTH,RECORD,LABEL)
)          IF (LABEL='')=0 , GOTO GIVEN
) SYSLBL1=SYSLBL1+1            ; OTHERWISE SUPPLY A LABEL
) LABEL=SYSLBL1
) GIVEN :
&&LABEL    J=JREAD(&MEMBER,&LENGTH,&RECORD,IRW)
)          CALL TESTJ('READ',LABEL)
)          MEND
);
)          MACRO WRITE(MEMBER,LENGTH,RECORD,LABEL)
&&LABEL    J=JWRITE(&MEMBER,&LENGTH,&RECORD,IWW)
)          CALL TESTJ('WRITE')
)          MEND
);
);  THIS MACRO EXAMINS THE COMPLETION CODE RETURNED BY JHPBPAM
);  MOST OF WHICH ARE COMPLETION CODES "AS IS" FROM THE RELEVANT
);  DM MACROS.  ALLOWS 20 FATAL READ ERRORS AND 0 WRITE ERRORS
);  BEFORE ABENDING. USES GENERATED LABELS SYSLBL1,...,SYSLBL3.
);  REFERS TO &SYSPRINT FOR ERROR MESSAGES.
);
)          MACRO TESTJ(WHO,AGAIN)
)          CALL SYSLBLS        ; GENERATE LABEL
      IF(J .LE. 0) GOTO &SYSLBL3
      WRITE(&SYSPRINT,&SYSLBL1) J
&&SYSLBL1 FORMAT('- UNSUCCESSFUL RETURN WITH CODE',I5,'FROM J&WHO')
) SYN2=WSYN2                   ; GET READY FOR SYNAD MESSAGE
) ERRCNT=WERR
) MSB='IWW(34+K)'
)          IF WHO='WRITE' THEN GOTO MESSAGE
) SYN2=RSYN2
) ERRCNT=RERR
) MSB='IRW(38+K)'
)          IF WHO='READ'  THEN GOTO MESSAGE
      CALL ABD1
)          GOTO MEXIT
) MESSAGE :
      IF (J .NE. 20) CALL ABD1
      WRITE(&SYSPRINT,&SYSLBL2) &ERRCNT,&SYN2,(&MSB,K=1,32)
&&SYSLBL2 FORMAT(' SYNCHRONOUS ERROR COUNT & CODE =',I5,Z12/1X,32A4)
      IF (&ERRCNT .GT. 20) CALL ABD1
)          IF AGAIN='' THEN GOTO MEXIT
      GOTO &AGAIN
) MEXIT :
&&SYSLBL3 CONTINUE
)          MEND
```

PART II. How to Use the Processor.

This part is intended as a user's guide and reference and contains a detailed description of our macro processor in this aspect.

As discussed elsewhere the processor itself is in a form to be processed by the processor. In particular the processor contains in "unbound form" parameters such as &MAXIDTR for the maximum allowed length of an identifier. Particular values are assigned to these parameters by means of a running processor, when the processor needs to be recompiled and made into a new load module (to be saved into a load module library).

Accordingly this part is written using such escape names. Consequently the reader is asked to mildly "macro process" the text on encountering such names. For this purpose we include here a short table of definitions of parameters appearing in the text including their current values.

Processor parameters and their values used in Part II text.

```
) VERSIONID='MAC74C06AUG74' ;
) ESCAPECH='&' ; PREFIX FOR ESCAPE NAMES
) MAXFATALERR=20 ; MAXIMUM NUMBER OF ON ERROR CONDITIONS ALLOWED.
) MAXIDTL=16 ; MAXIMUM LENGTH FOR IDENTIFIERS, VARIABLE & LABEL
);             NAMES. LEFTSIDE, SYMBOL.NAME, TARGET GLOBALLY
);             AND SNAME, INAME IN LOOKUP.
```

```
) MAXLSTR=80 : MAXIMUM LENGTH FOR VARYING CHAR STRINGS, TOKEN
);              (GLOBAL) , SVALUE,SARG1,SARG2,SARG(&MAXNARG) (IN EVAL)
) MAXMACARG=10 : MAXIMUM NO OF ARGUMENTS ALLOWED IN MACROS
): THE PRACTICAL LIMIT FOR MAXNARG AND MAXMACARG IS SET BY THE
): REQUIREMENT THAT A PP STATEMENT CAN ONLY BE ONE "CARD" LONG
): WHICH CAN BE EXTENDED ON IMPLEMENTING CONTINUATION FEATURE FOR
): PP STATEMENTS AS IN WORKSTR.
);
) MAXNARG=10 ; MAXIMUM NO OF ARGUMENTS ALLOWED IN BUILT-IN FUNCTIONS
) MAXNESTIF   =20 ; MAXIMUM NUMBER OF IF-CONDITIONS CONCURRENT.
) MAXNESTLEVEL=20 ; MAXIMUM LEVEL OF NESTING IN MACRO CALLS
) MAXREPLCOUNT=50 ; MAXIMUM NO OF REPLACEMENT / "LINE"
) PPBRKCH=')' ; MARKER FOR PP STATEMENTS
) PPSEC=80 ; END COLUMN FOR PP STATEMENTS. MAY BE 72 TO EXCLUDE
);              CARD SEQUENCE NUMBER.
) SIZEP=MAXMACARG*MAXNESTLEVEL ; SIZE OF PUSH DOWN STACK, PUSHIP,
);              TO HOLD PREVIOUS FORMAL PARAMETER VALUES IN CALL.
) NCONT=10 ; NO OF FORTRAN CONTINUATIONS. BECAUSE OF PACKING
) WORKSTRLNG=72+NCONT*66 ; PRACTICAL LIMIT IS HIGHER THAN SPECIFIED
):
```

In particular examples in the text are written assuming the values given above are effective.

(1) Characters.

User's text (in whatever base language), character string

data, and comments in processor statements may consist of any characters but a subset of EBCDIC character set is used for the processor statements (commands) and identifiers.

This subset consists of

(a) Letters: A through Z, $, @, #, and _,

(b) Digits: 0 through 9, and

(c) Breaks: the following special characters,

	Blank
=	Equal or assignment
+	Addition (infix) or plus (prefix)
-	Subtraction (infix) or minus (prefix)
*	Multiplication
/	Division
(	Left parenthesis
)	Right parenthesis
,	Comma
.	Period
;	Semicolon
:	Colon
'	Quote
¬	Not (prefix)
&	And (infix)
|	Or (infix)
>	Greater than
<	Less than

In decomposing a statement (lexical analysis) into tokens (lexical units) the special characters, blanks in particular, serve as delimiters. That is, a token, such as identifier, command keyword, data constant, etc., must be delimited by a special character. Special characters themselves do not require delimiters. They may or may not have intervening blanks.

All other characters not listed above, such as %, are treated

as special characters and thus have delimiting power.

Obviously special characters including blanks cannot be embedded in a token unless it is a character string constant. The character _, which is declared to be a part of letters, may be used for better readability in a compound name.

Otherwise, blanks may be freely used in a processor statement. Any number of blanks may appear whereever one is allowed.

(2) Identifiers.

Variable names, labels and macro names must be 1 to &MAXIDTR characters in length. Such an identifier must start with a letter followed by letters and digits in any mix, but must not contain special characters.

(3) Integers.

Integers consist only of digits. An integer is treated just like any other character string, except when it participates in macro time arithmetic. During the evaluation stage an integer may get converted on demand to a signed 31-bit magnitude fixed binary number. The user should insure that no conversion error may result. See the remarks under Data Conversion and Error Handling in the subsequent section named Expression.

In logical context an integer 1 (0) is interpreted as true (false) and conversly.

(4) Character String and Fixed Point Constant.

A character string may consist of 0 to &MAXLSTR characters enclosed in quotes and may contain any character beyond those specified as processor characters. A quote within a string is represented by two consecutive quotes.

A string of digits containing a decimal point, period, is automatically recognized as a string without enclosing quotes.

(5) Processor Statements

A processor statement is marked by the warning marker, &PPBRKCH in the first column. The remaining field in columns 2 to &PPSEC is used for a statement, which may optionally start with a label followed by a colon then commands and may terminate with a semicolon followed by comments, if any.

A label defined inside a macro is known only to that macro. That is, the scope of definition of a macro label is limited to one macro. For example, both called and calling macros may have identically named labels.

The keywords representing processor commands, such as MACRO, CALL, IF, etc., are "reserved" words. They should not be used for any other purpose unless, of course, renamed through SET commands.

A null statement consisting of the marker in column 1 and null or blanks terminated, if desired, by a semicolon, is treated as

no operation.

A processor statment as well as that in the base language (user's text) is subject to replacement via escape names. Furthermore, a processor statement may also be generated by a replacement yielding the marker in the first column. Operators may also be generated in a processor statement with the following exceptions.

1. During macro defining (editing) stage the replacement must be temporarily suspended so that operators like MEND and :, which must be recognized at this time, cannot be generated by replacements.

Consider an example,

```
) COLON=':'
)    MACRO ABC
     ...
) LABEL &COLON
     ...
)    GOTO LABEL
)    MEND
```

Backward GOTO as in this example will not work, but forward branch under similar situation will work.

(6) Assignment Statement.

<label> : <receiver> = <source expression> ;

label

identifies a statement. The colon is used to declare the identidier to its left as a label. It is to be deleted, if

no label is present.

receiver

a variable name, or an escape name which ultimately becomes a variable name.

source expression

a primary which is a variable name, an escape name, an integer, or a string.

arithmetic or logical expressions using primaries, operators, and functions.

The terminator ; is optional but recommended because it improves performance by removing superfluous scan for trailing blanks.

Examples

```
) I='-29' ; quotes are required because of the minus sign
) I='29'  ; quotes could have been left out
) PSEUDO_VECTOR&I='I AM THE VALUE OF &I.-TH ELEMENT'
) YES=I=2.5 ; YES=1(0), if I=2.5 is true (false)
```

(7) GOTO Statement

<label> : GOTO <target label>

label

&SAME_AS_BEFORE

target label

When used outside a macro definition, the target label

must exist in the input subsequent to this GOTO command. That is, only forward reference is allowed outside a macro. This command will skip all subsequent statements upto the statement containg the target label, from which the subsequent input will resume.
No such restriction exists inside a macro. In particular, This command may be used to loop back to an earlier point.

(8) IF, THEN, ELSE Statments

<label> : IF(<qualifier>) <condition> THEN(<qualifier>) <true part>
ELSE(<qualifier>) <false part>

label

&SAME_AS_BEFORE

condition

is a logical expression having the value either true or false.

true part

specifies a command to be performed, if the condition is true.

false part

similarly for false.

qualifier

is optional (defaulting to the value 1) and specifies which if condition the qualified operator is referring

to, in cases where more than one (and up to &MAXNESTIF) if conditions are concurrently active.

Qualifier may be a digit, a variable, an escape name, or an expression.

Consider the following statement,

```
IF <condition 1> THEN IF <condition 2> THEN <true part>
ELSE <false part>
```

The meaning of ELSE in this example is not unique, because the condition it refers to changes depending on whether the first condition is true or not. This statement can be made more unambiguous through qualifiers as below,

```
IF <condition 1> THEN IF(2) <condition 2> THEN(2) <true part>
ELSE.1 <false part>
```

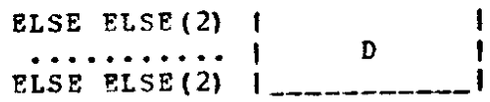
Qualifiers may also be used to "expand" the true or false part into many lines as below,

```
IF <condition 1> ;
IF <condition 2> ;
  THEN THEN(2)  ____________
  THEN THEN(2) |            |
   ........... |     A      |
  THEN THEN(2) |____________|

  THEN ELSE(2)  ____________
  THEN ELSE(2) |            |
   ........... |     B      |
  THEN ELSE(2) |____________|

  ELSE THEN(2)  ____________
  ELSE THEN(2) |            |
   ........... |     C      |
  ELSE THEN(2) |____________|

  ELSE ELSE(2)  ____________
```



where A B C and D are blocks of commands to be performed under four different (mutually exclusive) conditins that may arise.

(9) MACRO Statement

<label> : MACRO <macro name> (<formal parameter list>)

label

&SAME_AS_BEFORE

macro name

is an identifier, or an escape name having identifier value. It is used to name the macro being defined, which consists of a body of statements in base language and/or processor commands and is terminated by the MEND command.

formal parameter list

is an optional list consisting of up to &MAXMACARG parameters separated by comma. Within the macro body the parameters in this list are treated as escape names whose scope is local (i.e. limited) to the present macro. The macro body may contain other escape names not present in the parameter list. The latter are treated as having global scope of definition.

This list is called formal to distinguish it from the actual parameter list to be given in the CALL statements.

(10) MEND Statement.

<label> : MEND

label

&SAME_AS_BEFORE

This statement terminates a macro definition. In the current version (&VERSIONID) no nested macro definition is allowed so that if the processor runs into another MACRO command during the stage of editing the macro body of an earlier macro, then a MEND statement is generated to terminate the ongoing defining process before starting a new one.

(11) CALL Statement.

<label> : CALL <macro name> (<actual parameter list>)

label

&SAME_AS_BEFORE

macro name

is an identifier or an escape name having an identifer as value. The macro so named must have previously been defined in the input stream or must exist on a file named MACLIB. Macros in this macro library become available all at once when the MACLIB is accessed for the first (and last) time, which is triggered by a call

to a heretofore undefined macro. Failure in opening MACLIB (most likely because of its absence) or ultimately undefined macro will be flagged as error by the processor.

actual parameter list

consists of parameters separated by comma. each of which may be a variable, an escape name, or an expression.

On encountering a CALL statement the processor goes through the following actions:

1. If necessary, any values currently assigned to the formal parameters are saved on a pushdown stack.

2. The values of the actual parameters supplied in the CALL statement are then assigned to the corresponding formal parameters in the given order. An omitted parameter is treated as giving null string value to the corresponding formal parameter. Excess actual parameters are ignored.

3. The processor then begins fetching input from the macro body.

4. The input remains switched to the macro body until the MEND statement is reached, at which time the formal parameters are restored to their earlier values from the pushdown stack. The input is switched back to the state prior to the CALL statment.

The macro calls may be nested. That is, a call to a macro may further invlove call to another macro before the original call

expires. The depth of this nesting may not exceed &MAXNESTLEVEL. Recursive calls are also allowed. That is, a call to a macro may further involve call to the same macro. The same restriction as to the depth applies here also.

Below is an example of a recursive call in a macro with variable number of arguments.

```
)  MACRO ABC(A1,A2,....,A10)
   ________________________
   | text using, say, &a1 |   ; Note at each level of recursion
   |______________________|   ; the next argument in the
                              ; original list plays the role
                              ; of the first argument, &A1.
)  IF (A2='')=0 THEN CALL ABC(A2,A3,....,A10)
)  MEND
```

(12) FREE Statement.

<label> : FREE <identifier list>

label

&SAME_AS_BEFORE

identifier list

consists of variable names or escape names separated by a comma.

This statement "removes" the specified names from the table of definitions so that in the subsequent use they appear to be undefined. The primary function of FREE is to cause "delayed binding" by declaring a variable to be ineligible for replacement.

Consider the following illustration,

```
) I='INITIAL'
```

A global symbol I has been declared for the purpose of abbreviation. At some later point the user proceeds to define a macro as below,

```
)     MACRO CASE(MAXCASE)                    ; line 1
)     VALUE='CASE(&I)'                       ; line 2
)     I=1
) MORE : CASE&I=VALUE
)     I=I+1
)     IF I < MAXCASE+1 THEN GOTO MORE
)     MEND
```

the purpose of which seems to be to generate escape names, CASE1, CASE2, ... having values CASE(1), CASE(2), ... respectively.

Now it is obvious from its use within the macro that I was intended to be a temporary variable and ought to be local to this macro. (This is a big assumption and the processor should not do this under all circumstances). But the processor, when it reaches line 2, cannot tell the "local" I from the "global" I and therefore line 2 becomes (during invoking stage, of course)

```
)     VALUE='CASE(INITIAL)'
```

and the subsequent loop assigns this value to CASE1, CASE2,... Worse, at the end the "global" I has acquired a new junky value losing its originally intended meaning.

A key to the solution of this problem is to delay binding for I at line 2 by inserting

```
) ISAVE=I    ; save value
) FREE I     ; make it undefined
```

and just prior to MEND

) I=ISAVE : restore

User may make this solution fool-proof by making ISAVE a "dummy" arugment of the macro (so that the value is saved automatically at entry to this macro and restored at exit), or by inventing SAVE and RESTORE macros with pushdown stacks to hold the value of any local variable during a macro expansion.

(13) SET Statement.

<label> : SET <key> <value>

label

&SAME_AS_BEFORE

key

1. may be any keyword of the processing commands including SET itself, in which case the given value, truncated to 5 charaters, if necessary, becomes the new keyword for the same function. The effect of this command, unlike the others below, is to rename a command keyword so that the old name for it is lost.

2. may be ESCAPE or MARKER, in which case the value truncated to a single character becomes the new escape name prefix or the processor warning marker respectively.

3. may be REMOTE, in which case the value must be either BEGIN, or END. One command is used to switch the output file SYSOUT from "local" (under the DD name of SYSOUT) to "remote" (under the DD name of SYSUT) so that whatever follows this command gets collected at the end. The other command switches "remote" state back to "local".

4. may be any keyword for the "options" that can be specified through the PARM field of JCL EXEC card. This command is used to dynamically (i.e. during run time) reset "options". For example in processing FORTRAN text the option, FORT=1, should be in effect to take care of continuations, but a user insisting on writing Hollerith FORMAT statements may want to suspend this option temporarily.

5. may be TITLE, in which case the value is a character string of length less than 80 (how else ?) enclosed by quotes. This command sets the page heading and ejects to a new page. The command itself does not appear in the SYSPRINT listing.

6. may be PARM, in which case the command syntax is

<label> : SET PARM <name> <value>

name

is any one of the built-in functions.

value

is an integer not greater than &MAXNARG.

This command resets the expected number of arguments.

The key, except in case 4, may be abbreviated to the first three characters.

The primary purpose of SET command is to weaken the constraint that processor keywords are "reserved" words and hence cannot be (strictly speaking, at user's risk) used for any other purpose.

Example.

```
)SET MAR $              ; Because $ is not a special char. col 2
$                       ; must hence be blank in a pp stmt.
$MEND                   ; MEND not recognized.
$ SET FREE DEACTIVATE ; Truncated to DEACT.
$ DEACT A,B,C           ; because a kwyword must be 5 char or less.
$ SET SET @             ; SET is renamed @.
$ @ FORT 0              ; Turn off FORT continuation feature
112     FORMAT(130H  A title consisting of 130 characters
      + including blanks    )
$ @ FORT 1              ; Turn it back on.
$ @ ESC ' '             ; Subject every token with leading blank
$                       ; to replacement.
$ @ ESC ''              ; Every token is eligible for replancement.
```

Incidently reader might wonder why the key words are limited to being 5 character or less. This is so that, if variable names are 6 or more character long, then assignment and label statements are recognized without the overhead of having to check against the table of operations.

A reader might perhaps wonder how come the built-in function names are not allowed to be SET ? This is because the table-

look-up relies on the table being in EBCDIC collating sequence so that SET, if implemented, must reorder in this case. (This is deemed to be a case of diminishing return).

(14) Built-in Functions.

The built-in functions of our macro processor may be divided into three classes: (1) the class of functions which are based on the PL/I built-in functions of the same name, such as SUBSTR and TRANSLATE, (2) that of functions peculiar to our processor such as (token-) TYPE, and (3) the function INVOKE for "dynamic" invocation of user-defined functions or any procedures including those of, say, PL/I library.

Since the processor itself is written in PL/I, it has in principle easy access to all the built-in functions of PL/I (including future ones). Becuase the "basic" data type of our processor is character string, we have selected as useful those dealing with character strings and integers (which are self-defining character strings for our purpose).

From this class we further excluded those functions which require a "constant" argument. Remember that the processor is executing user commands interpretively. In particular the constant arugment is to be specified by the user during run-time and is unknown during the compile-time of the processor. One brute force method of implementing such a function would be to (1) limit the

choice to a few constants, (2) repeat essentially the same coding for each of these values, and (3) branch to one case according to the given value. We chose not to do this deferring this class of functions until a better scheme turns up.

An expected argument must be always given. That is, there is no "optional" argument as in the case of the third arguement of SUBSTR in PL/I. A missing argument in our case is treated as having null value. This is consistent with the rule used in treating arguments of a macro call.

The value of our built-in function is always a character string. If the corresponding PL/I function returns an integer value, then it is converted to a character string with leading blanks suppressed.

Note that the function names are not reserved words. An identifier is first looked up in the table of user-defined-variable names. Only when it is not present in this table, the table of built-in function names is interrogated. If it is still missing, then it is treated as an undefined variable.

Consider the following example: Suppose a user inadvertently used a variable named SUBSTR as in

```
) SUBSTR='THIS IS AN EXAMPLE'
```

Does he then lose SUBSTR built-in function for good ? The answer is no, because all he has to do is to remove the identifier, SUBSTR from the appropriate table before he invokes SUBSTR function by

issuing the following command,

```
) FREE SUBSTR ; (better save the value if needed again )
```

Whenever applicable the PL/I function of the corresponding name is invoked so that the user may consult a PL/I manual for precise definitions of their use. We include here a brief description for convenience and also note special rules whenever they are different from the relevant PL/I specifications.

(a) String Handling PL/I like functions.

The notations i, j, k are used to refer to integers and s, r, t for strings.

1. i=INDEX(s,r)

The value i is an integer giving either (1) location in string s at which string r has been found (scanning from left to right) or (2) 0 for no occurance or if either s or r is a null string.

An example of INDEX in a macro to get rid of blanks.

```
)    MACRO SQUEEZE(A) ; S WILL HAVE BLANKS SQUEEZED OUT
)    L=LENGTH(A)
)    I=INDEX(A,' ')
)    IF L=0 THEN S=''
)    ELSE IF I=0 THEN GOTO MEXIT
)    ELSE S=CONCAT(SUBSTR(A,1,I-1),SUBSTR(A,I+1,L-I))
)    ELSE CALL SQUEEZE(S)
) MEXIT: MEND
```

2. i=LENGTH(s)

Obvious.

3. r=SUBSTR(s,i,j)

r is a substring of s starting at i-th character with the resulting length of j. Let k be the length of s then it must be that

$0 \leq j \leq k$, $1 \leq i \leq k$, $i+j-1 \leq k$

Otherwise r is unpredictable according to the current PL/I manual If j is 0, then r is null.

Another form is r=SUBSTR(s,i) where the remaining length is implied. A user insisting on using this form must SET PARM SUBSTR 2 beforehand making sure the expected argument number is set back to 3 afterward.

4. t=TRANSLATE(s,r,p)

s is a source string to be translated, r is replacement string and p is position string, both of which defines the translation table character-by-character. The function name may be abbreviated to the first six characters.

For example TRANSLATE('COS(C)','SIN','COS') is 'SIN(S)'.

5. s=UNSPEC(x)

returns internal coded representation of x. See PL/I manual. For example UNSPEC('A') is a character string '11000001' representing a hexadecimal digit C1 as EBCDIC code for 'A'. This string may be converted to a decimal digit to be used in an arithmetic expression.

6. i=VERIFY(s,r)

i is 0, if string s is contained in r. Otherwise, position of the first character (from the left) in string s which is not in string r.

An example of VERIFY in a macro to get rid of leading blanks.

```
)     MACRO NOLBL(A) ; S WILL LOSE LEADING BLANKS
)     I=VERIFY(A,' ')
)     IF I=0 THEN S=''
)     ELSE S=SUBSTR(A,I,LENGTH(A)-I+1)
)     MEND
```

We give another illustration in using these functions.

```
);
); LET S BE A STRING OF TOKENS SEPARATED BY COMMA WITH NO
); BLANKS.  GET_TOKEN BREAKS S INTO TOKENS, FOR EACH OF WHICH
); COUNTER, MANY, WILL BE INCREMENTED. THE TOKEN ITSELF IS
); SAVED INTO A PSEUDO-VECTOR TOKEN&I (I=1,MANY).
);
)           MACRO GET_TOKEN(B,S)
)   MANY=B ; STARTING VALUE - 1      ; TEXT HAS BEEN INTENTIONALLY
) MORE: L=LENGTH(S)                  ; PUSHED LEFT, SQUEEZING OUT
)   IF L=0 THEN GOTO MEXIT           ; SUPERFLUOUS BLANKS SO THAT
)   MANY=MANY+1                      ; THE WASTE IN PROCESSING
)   P=INDEX(S,',')                   ; NO INFORMATION IS MINIMIZED.
)   IF P=0 THEN TOKEN&MANY=S
)   ELSE TOKEN&MANY=SUBSTR(S,1,P-1)
)   ELSE S=SUBSTR(S,P+1,L-P)
)   ELSE GOTO MORE
) MEXIT:   MEND
);
); FIND EXAMINS PRESENCE OF TOKENS IN A STRING AND INSERTS
```

```
); SURROUNDING BLANKS FOR EACH TOKEN PRESENT.
);
)           MACRO FIND(A)
)   S=A
)   I=1
) MORE: P=INDEX(S,TOKEN&I)
)   IF P=0 THEN GOTO CONTINUE
)   A=CONCAT(SUBSTR(S,1,P-1),' #TOKEN&I ')
)   SET PARM SUBSTR 2
)   S=CONCAT(A,SUBSTR(S,P+LENGTH(TOKEN&I)))
)   SET PARM SUBSTR 3
)   GOTO MORE ; UNTIL NO MORE OF THE SAME KIND.
) CONTINUE: I=I+1
)   IF I < MANY+1 THEN GOTO MORE
); SET PARM TRANSL 3 ; THIS VERSION HAD 2 BY MISTAKE
);                       ABOVE OBSOLETE AS OF AUG/6/74
```

To a casual reader FIND may perhaps appear to contain needles gyrations. To see their need, consider a situation in which one token is contained in another, as in 'A' and 'RATHER'. We show the string S prior to TRANSLATE,

```
      &S
)   S=TRANSL(S,'&','#') ; CHANGE # TO &
)   S='&S'
)           MEND
);
```

To see how these MACROS work, define some tokens,

```
)   CALL GET_TOKEN(0,'BLANKS,RATHER,STRING,THIS,A')
```

And apply to a string,

```
)   CALL FIND('THISISARATHERLONGSTRINGWITHOUTBLANKS')
```

String S is now &S.

(b) Arithmetic PL/I like functions.

1. i=ABS(j)

2. i=MAX(j,k) Note only two arguments allowed

3. i=MIN(j,k) Same

4. i=MOD(j,k) remainder of j/k

(c) Miscellaneous.

1. s=CONCAT(r,t)

s is the concatenation of two strings, r and t. This has the same effect as the PL/I operator ||, which, incidentally is an infix operator requiring a very different method of parsing as in evaluating arithmetic expressions in infix notations. We made up the CONCAT function so that the alternate method of recursive function call can be used for this operator. (How about, s='&r&t', when r and t are character string constants ?).

(D) INVOKE

This operator was originally invented for the purpose of dynamically invoking (i.e. by bringing a user desired program into main storage and executing it during the run time) any program in the (PDS) load module library under the DD name of SYSLIB. Through concatenation of OS libraries and user libraries in which user written programs are collected, this feature makes available to the processor a potentially huge set of programs,existing procedures, user compiled

"macros".

Soon it became apparent that there is some advantage of INVOKing even PL/I built-in functions, if the latter are used less frequently than the ones described earlier, because of the saving in table look-up overhead. With dormant INVOKE the table of built-in functions looks like a linear, ordered list. Because INVOKE is a member of this list the table becomes tree-like, when INVOKE is active.

This form of INVOKE is used as a function,

```
value=INVOKE(name,arg1,arg2,...)
```

Actually function is a misnomer for INVOKE, because the INVOKed procedure may have the primary purpose of generating more lines (user text) in SYSOUT, processing subsequent lines in SYSIN (to the macro processor itself the effect is as if these lines have been skipped), and what not.

The argument list must, of course, be a variable one. In order to prevent accident and to supply some syntax check it is required that the user tell beforehand what the number of arguments is,

```
)    SET PARM INVOKE <value>
```

where value is the number of arguments to be passed to the procedure INVOKed. Does this requirement prevent using INVOKE recursively ?

The answer is NO, because dummy arguments may be supplied, as in the following example.

Suppose we want to invoke a procedure named NAME1 with 3 arguments, the second of which is a value of another procedure, NAME2, which in turn requires 2 arugments.

```
)     SET PARM INVOKE 3 ; maximum
)     value=INVOKE(NAME1,arg11,INVOKE(NAME2,arg21,arg22,arg23),arg13)
);    arg23 is a dummy
```

1. DATE and TIME

The PL/I built-in functions, DATE and TIME, belong to the category of less frequently used function under normal situations. Therefore they are implemented through INVOKE.

INVOKE('DATE') gives a character string of length 6 in the form YYMMDD, and INVOKE('TIME'), a character string of length 9 in the form HHMMSSTTT, where TTT is in msec, assuming, of course, the number of arguments has been set to 0 beforehand.

(15) Expression.

Using the customary rules as in PL/I or FORTRAN, any expression containing constants, variables, escape names, functions, parentheses, and the following infix operators, is allowed.

operators	priority	operation
* /	highest	integer arithmetic
+ -	↓	" "

> <	|	" "
=	|	string comparison
&	V	logical and
|	lowest	logical or

An expression may be used in an assignment statement, as an argument of a macro or function call, or as a qualifier of IF, THEN, and ELSE.

1. Data conversion.

The internal form (type and attribute) of a data, be it a constant, value of a variable, or of a function, is always a varying length character string (CHARACTER(&MAXLSTR) VARYING). An arithmetic operation is carried out by converting operands to signed 31-bit magnitude fixed binary numbers (FIXED BINARY(31)) The result is converted back to a character string with l e a d i n g b l a n k s s u p p r e s s e d. A logical operation is performed with intermediate conversions to and from a bit string of length one (BIT(1) ALIGNED).

Also function calls involve conversion of arguments to required types, when applicable. The value of an integer function is also converted to a character string with leading blanks removed.

2. Error handling.

Apart from fatal (in the sense that the subsequent performance of the processor itself might be affected) errors like stack overflows, no attempt is made to detect beforehand a potential error condition. An expression is calculated with implicit

conversion without any prior check.

An error condition, when detected by the operating system, is trapped back to the processor so that the processing may continue. This approach is slightly inconvenient for users, because by the time an error condtition is noted the primary cause is often difficult to pinpoint (specially when a statement contains many complex expressions). On the other hand this approach has an advantage of not penalizing a careful user with checking overhead.

Some frequent errors are;

a. overflow in conversion of an integer string to binary form (in an arithmetic operation).
b. conversion of a string with character other than 0 or 1 to a bit string (in a logical expression).
c. unmatched parenthesis.
d. successive infix operators like a*-2 (should have been a*'-2').

(16) Replacement.

A "line" of text, be it a statement in the base language, or a processor command, is scanned for the presence of the escape character, &ESCAPECH, which signals a potential replacement.

If

1. the escape character is followed by an identifier or by

another escape character followed by an identifier,

and

2. the identifier is eligible for replacement by having been assigned a value including null,

then the escape name, the escape(s) and the identifier, is substituted by the corresponding value. Because the lengths of the two strings, name and value, do not in general match, we allow two different modes of replacement, p a c k e d and n o n - p a c k e d, as described later.

Scanning for an escape name proceeds from left to right. The field scanned is the entire "logical line" which

1. if FORT=1, consists of a FORTRAN statement and possible continuation lines,

2. otherwise, columns from BEGC to ENDC of the user' text, (BEGC and ENDC are PARM field options defaulting to 1 and 80 respectively).

3. or, if commands, columns from 2 to &PPSEC, extending beyond the terminator ; (so that even comments are subject to replacements).

When an escape name has successfully been replaced by its value, the scan is restarted from the beginning column. Rescanning continues until no more escapes remain, or only irreplacable escape names remain (which are left "as is", such as &123 in FORTRAN, a label being passed to a subroutine as an argument). The rescanning mechanism allows generating escape names by

replacement. The maximum number of replacements allowed per line is &MAXREPLCOUNT.

The following is an example which otherwise would lead to an infinite replacement loop.

```
) FREE X : MAKE SURE X IN INELIGIBLE FOR REPLACEMENT
) X='&X' ; NOW MAKE IT ELIGIBLE BY GIVING IT A VALUE ?
    &X
```

There are two different modes of replacement:

1. Packed Replacement and Period for Concatenation.

This is indicated by the escape name having a single escape. If the delimiter of the escape name is a period, then the period is removed allowing concatenation of the replacement value with the part to the right of the period. Replacement is done by removing the escape name and inserting in its place the associated value. The remainder of the statement to the right of the escape name is shifted right or left as needed to accomodate the value. Blanks are added at the end of the line, if there is an overall left movement, or truncation, if right. The latter is detected as an error condition.

2. Non-packed Replacement.

This is signalled by a double escape preceding an identifier. A period delimiter is n o t recognized as concatenation. The replacement of the escape name is done without any shifting of the part to the right of the name. If the value is shorter than

the name, blanks are added. If the value is longer, then the blank field, if any, to the right of the name is borrowed to hold the trailing non-blank characters of the value. If there is still no room, even after using the blank field and discounting trailing blanks of the value, then the processor will flag the truncation of the value as an error.

The latter mode of replacement is useful in processing column sensitive data cards. Or in making subsitutuions in the label field of column sensitive language like FORTRAN.

(17) Keywords in the PARM field of the EXEC card.

Some parameters of the processor that are deemed likely to stay unchanged during a run, are passed as "options" to the processor through the JCL PARM field mechanism of the EXEC card. The format of this field is a list of items separated by a comma, each item consisting of a keyword followed by an equal sign and an integer. These options and their default values are listed below. Remember that all except SIZS can be changed during processing time by means of SET commands.

Key	Default	Meaning
BEGC	1	Begin column for replacement scanning of base language statements. FORT=1 option will override this.
ENDC	80	End column likewise.
MACO	1	If 0, no output is written on SYSOUT
PGEN	2	If 0, no output listing is produced on SYSPRINT. If 1, top level statements are listed

		If 2, base language statements from macro expansions, in addition.
FORT	0	If 1, FORTRAN statement conventions become effective. See below.
PRBS	0	If 1, the text before resolving escape names is printed.
SIZS	10000	Number of bytes allocated to store variables names and values. If necessary, may be expanded to 32767.

The FORTRAN statement convention mentioned above is to be understood as follows. The columns 1 through 72 of the first card and columns 7 through 72 of any continuation cards are treated as a single base language statement in carrying out the replacement operation. Trailing blanks at the end of each line are removed unless, of course, they occur as a part of a character string. If each line is full then &NCONT continuation lines are allowed. More, otherwise.

(18) Diagnostic Messages and Severity Codes.

An error condition detected by the processor is flagged by an error message giving a self-explanatory diagnosis together with other helpful information such as the statement number in question. A severity value is also assigned, the highest of which is propagated as a JCL Condition Code to the next job step, when the processing terminates.

Other errors, such as

a. arithmetic overflow and conversion errors in expression,

b. missing SYSUT, when REMOTE is used,

c. too many print lines,

cause Operating System interrupts, at which time the OS diagnostic messages (like PL/I object-time messages headed by IHEnnnI) will appear followed by the calling trace listing all procedures active at the time of interrupt. The severity code 4 is assigned to such error conditions. The processor will regain control in an attempt to continue, but will abort the processing after &MAXFATALERR number of such occurances.

All processor diagnostic messages and their severity values are collected below. The format is a character string being assigned to a variable whose name identifies the procedure involved. The collection is a reproduction of a part of the processor-processor to be described elsewhere. The OS messages can be found in the OS reference manual such as the PL/I(F) Programmer's Guide.

```
);     Messages and Severities.  Version of &VERSIONID
);     ------------------------------------------------
);
) TOKENERR1= '''STRING TOO LONG,TRUNCATED TO &MAXLSTR CHAR'',8'
) TOKENERR2= '''IDENTIFIER TOO LONG,TRUNC TO &MAXIDTL CHAR'',8'
) MACLIBERR1= '''UNABLE TO OPEN FILE MACLIB'',16'
);           The DD card for a user macro library, MACLIB, is
);           probably missing.
) EVALERR1= '''UNASSIGNED VAR'',8'
) EVALERR2= '''MISSING ( IN BUILT-IN FUNCTION'',8'
)        FREE A1
```

```
) EVALERR3= '''EVALUATE STACK EV&A1 FULL'',16'
);          EVSTK is a primary stack sized &EVSTKSIZE. EVOUT is
);          a secondary stack sized &EVOUTSIZE.  The expression
);          probably contains too many nested parentheses.
);          Break the expression into several lines.
) EVALERR4= '''ERROR IN EXPRESSION'',8'
) CLASSIFYERR1= '''MISSING KEYWORD IN PROC STMT'',8'
) CLASSIFYERR2= '''UNDEFINED PPSTMT'',8';
) DOSETERR1= '''ILLEGAL KEY IN SET STMT'',8'
) DOSETERR2= '''NO VALUE IN SET STMT'',8'
) DOSETERR3= '''ILLEGAL VALUE IN SET PARM'',8'
) DOSETERR4= '''FUNCTION MISNAMED IN SET PARM'',8'
) MCALLERR1= '''MISSING MACRO NAME'',8'
) MCALLERR2= '''UNDEF MACRO NAME'',8'
) MCALLERR3= '''MACRO DEPTH EXCEEDS &MAXNESTLEVEL'',16'
) MCALLERR4= '''INVAL DELIM'',8'
);          See comments under MEDITERR4.
) MCALLERR5= '''PUSHDOWN STACK OVERFLOW IN MACRO CALL'',16'
);          PUSHIP size SIZEP is too small.
) MEDITERR1= '''MISSING MACRO NAME'',12'
) MEDITERR2= '''INVALID FORMAL PARM'',12'
);          Parameter name must be an identifier.
) MEDITERR3= '''DUP FORMAL PARM'',12'
) MEDITERR4= '''INVALID DELIM'',12'
);          A parameter must be delimited by either , or ).
) MEDITERR5= '''MORE THAN &MAXMACARG FORMAL PARAMETERS IN MACDEF'',12'
) MEDITERR6= '''DUP MAC DEFINITION'',12'
```

```
) MFDITERR7= '''DUP LABEL DEFINITION'',12'
) EXPANDERR1= '''TRUNC WILL OCCUR IN NON-PACK REPLACEMENT'',8'
) EXPANDERR2= '''SIG PART OF STMT/REPL VALUE LOST IN PACKED REPL'',8'
) EXPANDERR3= '''MORE THAN &MAXREPLCOUNT REPLACEMENTS'',12'
) MAINERR1= '''STRING SPACE EXHAUSTED,ABORTING STMT'',16'
);          Area for value-strings STRINGAREA sized SIZS.
) MAINERR2= '''MACRO DEF SPACE EXHAUSTED,NEW ALLOC MAY FAIL'',16'
);          Area for names and macro body QUIETAREA sized SIZEQ.
) MAINERR3= '''INVALID PARM NAME'',4'
);          See descriptions of options in JCL parm field.
) MAINERR4= '''SYNTAX ERR IN PARM FIELD'',4'
);          The syntax is <key word> = <integer> followed by
);          a comma if more.
) MAINERR5= '''BEGC,ENDC SPEC INVALID'',4'
);          Error detected during parm field processing. Reset
);          to their default values.
) MAINERR6= '''INSUFFICIENT WORK SPACE--WILL ATTEMPT TO CONTINUE'',16'
);          The available space determined by issuing a conditional
);          GETMAIN to the OS (and printed in SYSPRINT under
);          SIZW) is not sufficient to allocate a push down stack
);          (PUSHIP with size 4*SIZEP in bytes), a controlled
);          area for value strings (STRINGAREA,SIZS), a based
);          area for names and macro definitions (QUIETAREA,SIZEQ)
);          and control bytes for each. As a remedy increase
);          the step REGION size in JCL.
) MAINERR7= '''TOO MANY CHAR IN FORT STMT'',8'
);          Too many continuation lines in FORTRAN. That is,
```

```
):          the size &WORKSTRLNG for a work area WORKSTR is
):          not big enough.
) MAINERR8= '''MEND NOT IN MACRO'',8'
) MAINERR9= '''MORE THAN &MAXNESTIF IF-CONDITIONS'',12'
):          Too many concurrently active if-conditions.
) MAINERR10= '''ILLEGAL ADDRESS IN GO TO'',12'
):          A label must be an identifier.
) MAINERR11= '''UNDEFINED ADDRESS IN GOTO'',12'
):          Reference to an undefined label in a macro.
) MAINERR12= '''EOF DURING LABEL SEARCH'',16'
):          Forward reference to a non-existent label outside
):          macros.
) MAINERR13= '''BAD QUALIFIER IN ELSE OR THEN'',8'
):          The qualifier refers to an undefined IF condition.
) MAINERR14= '''*** ILLEGAL USE OF REMOTE. SEVERITY=16'''
):          REMOTE is used pairwise, (BEGIN,END).

):     ------------------------------------------------------
):
):               End of Messages and Severities
```

(19) A JCL Example and a Run.

The following is a typical JCL for a macro processing step.

```
//    JOB card
//MAC EXEC PGM=MAC74C,REGION=300K,PARM='PRBS=1'
//STEPLIB DD DSN=WYL.ED.PUB.PARK,DISP=SHR
```

```
//MACLIB DD (User's macro library, if any)
//SYSLIB DD (User's program library, if any)
//SYSPRINT DD SYSOUT=A
//SYSOUT DD DSN=&MACOUT,DISP=(MOD,PASS),UNIT=SYSDA,
//  SPACE=(TRK,(20,2),RLSE),DCB=(RECFM=FB,LRECL=80,BLKSIZE=1680)
//SYSUT DD DSN=&TEMP,DISP=(MOD,PASS),UNIT=SYSDA,
//  SPACE=(TRK,(20,2),RLSE),DCB=(RECFM=FB,LRECL=80,BLKSIZE=1680)
//SYSUDUMP DD SYSOUT=A,DCB=(RECFM=VBA,LRECL=137,BLKSIZE=1644)
//SYSIN DD *
```

```
 ----------------------
|                      |
| Text to be processed.|
|______________________|
```

The EXEC statement specifies the program to be executed, the main storage requirement (for this step) and options in the processor. This is followed by following D(ata) D(efinition) statements.

STEPLIB identifying a (PDS) load module library containing the processor.

MACLIB optional macro library, which is a sequential data set with a similar DCB as that of SYSOUT. Macros in this library need not be in alphabetic order.

SYSLIB optional program (load module) library which is a partitioned data set containing programs as members to be INVOKed.

SYSPRINT output from the processor to be printed.

SYSOUT processed output to be passed onto a subsequent step (which may not be immediately following). Need not be given, if the option MACO is chosen to be 0.

SYSUT a temporary output from the processor which is generated when REMOTE is used. Not needed otherwise.

SYSUDUMP to get the OS dump if the step terminates abnormally.

Not needed otherwise.

SYSIN input text to be processed.

The next two pages contain a reproduction of the SYSPRINT output from a run in which we have collected various examples used in Part II. The character E under LEVEL means that the corresponding statement is being processed by the procedure handling macro definitions. Level 0 means user's text, otherwise it is the nesting level of macro calls generating the line concerned. REPLC counts the total number of replacements in a line. In this particular run the option PRBS=1 was taken in order to print each line containing escape names before replacement. Such lines are marked BR.

```
STMT  LEVEL PEPLC      SOURCE STATEMENT                                  MAC74C 06AUG 74

  2                  )    MACRO SQUEEZE(A) ; S WILL HAVE BLANKS SQUEEZED OUT
  3     E            )    L=LENGTH(A)
  4     E            )    I=INDEX(A,' ')
  5     E            )    IF L=0 THEN S=''
  6     E            )    ELSE IF I=0 THEN GOTO MEXIT
  7     E            )    ELSE S=CONCAT(SUBSTR(A,1,I-1),SUBSTR(A,I+1,L-I))
  8     E            )    ELSE CALL SQUEEZE(S)
  9     E            ) MEXIT: MEND
 10     0     0
 11     0     0          TO SEE HOW IT WORKS.
 12     0     0
 13     0            )    CALL SQUEEZE('I WILL  HAVE LOST   ALL BLANKS')
 75     0     0
 76     0    8R          STRING $ IS NOW 6$
 76     0     1          STRING $ IS NOW IWILLHAVELOSTALL@LANKS
 77     0     0
 78     0     0
 79                 )    MACRO NOL@L(A) ; S WILL LOSE LEADING BLANKS
 80     E           )    I=VERIFY(A,' ')
 81     E           )    IF I=0 THEN S=''
 82     E           )    ELSE S=SUBSTR(A,I,LENGTH(A)-I+1)
 83     E           )    MEND
 84     0           )    CALL NOL@L('   WATCH LEADING BLANKS VANISH')
 89     0     0
 90     0    8R          STRING $ IS NOW 6$
 90     0     1          STRING $ IS NOW WATCH LEACING BLANKS VANISH
 91     0           );
 92     0           ); LET S BE A STRING OF TOKENS SEPARATED BY COMMA WITH NO
 93     0           ); BLANKS.  GET_TOKEN BREAKS S INTO TOKENS, FOR EACH OF WHICH
 94     0           ); COUNTER, MANY, WILL BE INCREMENTED. THE TOKEN ITSELF IS
 95     0           ); SAVED INTO A PSEUDO-VECTOR TOKEN& (I=1,MANY).
 96     0           );
 97                 )          MACRO GET_TOKEN(@,S)
 98     E           )  MANY=@ ; STARTING VALUE - 1     ; TEXT HAS BEEN INTENTIONALLY
 99     E           ) MORE: L=LENGTH(S)                ; PUSHED LEFT, SQUEEZING OUT
100     E           )  IF L=0 THEN GOTO MEXIT          ; SUPERFLUOUS BLANKS SO THAT
101     E           )  MANY=MANY+1                     ; THE WASTE IN PROCESSING
102     E           )  P=INDEX(S,',')                  ; NO INFORMATION IS MINIMIZED.
103     E           )  IF P=0 THEN TOKEN&MANY=S
104     E           )  ELSE TOKEN&MANY=SUBSTR(S,1,P-1)
105     E           )  ELSE S=SUBSTR(S,P+1,L-P)
106     E           )  ELSE GOTO MORE
107     E           ) MEXIT:   MEND
108     0           );
109     0           ); FIND EXAMINS PRESENCE OF TOKENS IN A STRING AND INSERTS
110     0           ); SURROUNDING BLANKS FOR EACH TOKEN PRESENT.
111     0           );
112                 )          MACRO FIND(A)
113     E           )  S=A
114     E           )  I=1
115     E           ) MORE: P=INDEX(S,TOKEN&I)
116     E           )  IF P=0 THEN GOTO CONTINUE
117     E           )  A=CONCAT(SUBSTR(S,1,P-1),' #TOKEN&I ')
118     E           )  SET PARM SUBSTR 2
```

```
STMT  LEVEL REPLC          SOURCE STATEMENT                                  MAC74C 06AUG74

119     E             )  S=CONCAT(A,SUBSTR(S,P+LENGTH(TOKEN6)))
120     E             )  SET PARM SUBSTR 3
121     E             )  GOTO MORE ; UNTIL NO MORE OF THE SAME KIND.
122     E             ) CONTINUE: I=I+1
123     E             )  IF I < MANY+1 THEN GOTO MORE
124     E             ): SET PARM TRANSL 3 ; THIS VERSION HAD 2 BY MISTAKE
125     E             ):                     ABOVE OBSOLETE AS OF AUG/6/74
126     E
127     F                TO A CASUAL READER FIND MAY PERHAPS APPEAR TO CONTAIN
128     E             NEEDLES GYRATIONS.  TO SEE THEIR NEED, CONSIDER A SITUATION
129     E             IN WHICH ONE TOKEN IS CONTAINED IN ANOTHER, AS IN 'A' AND
130     F             'RATHER'. WE SHOW THE STRING S PRIOR TO TRANSLATE,
131     E                &S
132     E             )  S=TRANSL(S,'&','#') ; CHANGE # TO &
133     E             )          MEND
134     0             ):
135     0     0          TO SEE HOW THESE MACROS WORK, DEFINE SOME TOKENS,
136     0     0
137     0             )  CALL GET_TOKEN(0,'BLANKS,RATHER,STRING,THIS,A')
180     0             ):
181     0     0          AND APPLY TO A STRING,
182     0     0
183     0             )  CALL FIND('THISISARATHERLONGSTRINGWITHOUTBLANKS')
243     1     0
244     1     0          TO A CASUAL READER FIND MAY PERHAPS APPEAR TO CONTAIN
245     1     0       NEEDLES GYRATIONS.  TO SEE THEIR NEED, CONSIDER A SITUATION
246     1     0       IN WHICH ONE TOKEN IS CONTAINED IN ANOTHER, AS IN 'A' AND
247     1     0       'RATHER'. WE SHOW THE STRING S PRIOR TO TRANSLATE,
248     1    BR          &S
249     1     1           #TOKEN4 IS #TOKEN5  #TOKEN2 LONG #TOKEN3 WITHOUT #TOKEN1
251     0     0
252     0    BR          $TRING $ IS NOW &S.
252     0     6          $TRING $ IS NOW  THIS IS A  RATHER LONG STRING WITHOUT BLANKS
253     0     0

                --- MAC74C PROCESSED       253 LINES,HIGHEST SEVERITY= 0
```

References.

1. J. Ahern, MACROS-Statement Oriented Macro Processor,SLAC Computation Group User Note 29 (1969).

2. There is an abundance of literatures dealing with macro processing in various forms ranging from string processors as indepedent systems to definitional facilities in several different phases of compiling process for higher level languages. We include a short list of references for macro processors of the former kind that we have come across from time to time. Further references may be found in those listed here.

   C. Strachey, A General Purpose Macrogenerator, Computer Journal 8, 225 (1965).

   P. J. Brown, The ML/I Macro Processor, CACM 10, 618 (1967).

   G. A. Robinson, MACRO-FORTRAN, A Facility for Programmer-defined Macro-Instructions in FORTRAN Programs, ANL-7309, Applied Math. Division, Argonne National Laboratory (1967).

   S. H. Cain and E. K. Gordon, TTM: A MACRO Language for Batch Processing, Willis H. Booth Computing Center Programming Report No. 8, California Institute of Technology (1969).

   A. D. Hall, The M6 MACRO Processor, Computing Science Technical Report #2, Bell Laboratories, Murray Hill, New Jersey (1972).

3. J. C. H. Park, FIT73-A Kinematic Fitting Routine, to be published as a SLAC Report.

4. The discussion deals with the CERN version of SUMX as

described by J. Zoll, CERN, Track Chamber Program Library Manual (1970).

5. J. C. H. Park, On the Use of a Macro Processor with SUMX, SLAC Report No. 151 (1972).

6. J. Friedman and R. Chaffe, SLAC KIOWA, General Description SLAC Computation Group, CGTM No. 146 (1973).