To: Distribution

From: Marshall Presser

Date: 12/27/79

Subject: PL/I Macro Language Facilities

INTRODUCTION

This MTB proposes macro language facilities to be included in Multics PL/I. The primary purpose of these facilities is as a tool for systems programmers attempting to maintain software intended to operate on a variety of hardware. This MTB does not address the questions of compatibility with other macro processors and customer desires.

The facilities proprosed here have their origin in a macro processor described by Jim Falksen in MTB 345, "GENERALIZED MACRO PROCESSOR", and from other considerations discussed in MTB 426, "A Multics Macro Processor". What follows here is a semi-formal description of the language itself, some examples, a discussion of the effects upon listings, the integration into the PL/I compiler, the interactions with probe and the formatting programs, and a brief discussion of macro libraries.

This MTB is not intended as a final description of the macro language, but as a working document.

PRELIMINARIES

All PL/I macro language constructs will begin with a lexeme whose first character is the percent-sign, i.e. "%", called the trigger character. This trigger character is used for these reasons:

(1) compatibility with the %include macro and the %page and %skip listing control statements presently in Multics PL/I,

(2) clear demarcation of macro language constructs from PL/I constructs,

(3) desirability of insuring that only users of macro facilities pay any signifigant penalty for them.

The macro language is an interpreted set of macro constructs that perform text manipulation at compile-time and whose "natural" operations are the replacement of one text string by another and the concatenation of strings. Various built-ins, pseudo-ops, structured data types, and control structures have been included to allow for ease of expression.

The language is designed to be insensitive to white space within the syntax of macro constructs, but to preserve white space within non-macro constructs and within macro quoted strings.

As the only data recognized by the macro processor is the character string, clearly white space control is an important part of macro construct input and output and a fuller explanation may be valuable. The important white space rules are:

(1) Where white space is explicitly required by the syntax rules, all such white space so included is irrelevant to the output.

```
%macro foo %;
```

is equivalent to:

```
%macro                      foo
                                                      %;
```

(2) Where white space is used to seperate lexemes and is not explicitly required by the syntax rules, it is signifigant to the output, except when noted.

(a) `-%x   %y   -`, if x is a variable whose value is the string "abc" and y is one with value "def", yields "-abc   def   -".

(b) "%call(x,   y   )" is an invocation of the macro named call with 2 arguments, each of which is one character long. To generate the white space in the argument list, one must macro-quote it, e.g.

```
%call ( %"    x%", %"    y      %")
```

(3) The macro construct "%." is the null seperator. It causes no output and is used to terminate lexemes unambiguously. For example "%x%.y" causes the concatenation of the macro variable x with the character y, while %xy causes the output of the macro variable xy.

(4) The macro construct "%+" is called the gobbler. It absorbs all trailing white space and is useful in formatting macro constructs.

(5) The %skip and %page macros only generate white space in the listing and do not affect any strings seen by the compiler.

Finally, as the macro activity procedes within the confines of the PL/I lex, macro constructs in PL/I comments and PL/I quoted strings are ignored, since these are the fundamental lexemes of PL/I. The macro languages can easily generate all such intents with ease.

The following terms occur frequently in the language description following:

<space>::= <blank> | <newline> | <tab> | <formfeed>

<blank>::= ascii blank character (octal 40)

<newline>::= ascii newline character (octal 12)

<tab>::= ascii tab characters (octal 11 and 13)

<formfeed>::= ascii formfeed character (octal 14)

<trigger>::= ascii percent-sign (octal 45)

<macro-terminator>::= %;

<text>::= a string of ascii characters (possibly null) not containing a trigger character. It is the largest such string in a left to right scan.

<macro-integer>::= any macro construct that evaluates to a decimal integer.

<macro-positive-integer>::= any macro construct that evaluates to a positive decimal integer

<macro-identifier>::= <identifier>

<identifier>::= a PL/I identifier

<macro-input>::= <macro-construct> | <text><macro-input><text>

<macro-construct>::= <macro-declaration> | <macro-reference> | <macro-statement>

<macro-declaration>::= <macro-definition> | <macro-variable-declaration>

<macro-reference>::= <macro-call> | <macro-scalar-reference> | <macro-array-reference> | <macro-numbered-parameter-reference> | <macro-variable-parameter-reference> |

```
<macro-command-line-argument-reference>                          |
<macro-multiple-command-line-argument-reference>                 |
<macro-active-function-call>  |  <macro-arithmetic-evaluation>   |
<macro-literal-percent>      |     <macro-protected-string>      |
<macro-builtin>      |    <macro-command-line-argument-count>    |
<macro-numbered-parameter-count>

<macro-statement>::=          <macro-scalar-assignment>          |
<macro-array-assignmnet>    |    <macro-if>    |    <macro-do>   |
<macro-return> | <macro-rescan> | <macro-error> | <macro-comment>
| <macro-white-space-control>
```

INCLUDE, PAGE, SKIP, PRINT, AND NOPRINT MACROS

The include macro is discussed in the PL/I Language Manual AG94.

The page and skip macros are discussed in the pl1_new_features info seg on system M and will not be discussed here.

```
<print-macro>::= %print<space>...;

<noprint-macro>::= %noprint<space>...;
```

The %print and %noprint statements act as global switches to control the contents of a listing segment. When a %noprint is encountered, it will inhibit further listing of the program until a %print statement is encountered. Both these constructs produce no replacement activity. The default value for listing control is %print.

If no listing is requested in the pl1 command, these statements are irrelevant. These macros are terminated with a semicolon for compatibility with %page and %skip.

MACRO DEFINITION AND INVOCATION

```
<macro-definition>::=
%macro<space>...<macro-identifier>[<named-parameter-list>]
<macro-terminator><macro-input>%mend

<named-parameter-list>::=
(<macro-formal-parameter>[,<macro-formal-parameter>]...)

<macro-formal-parameter>::= <macro-identifier>
```

When a macro definition is encountered, its definition is stored verbatim, i.e. no evaluation is done at define time, only at expansion time. A further explanation of macro expansion is given below.

Example:

```
          %macro greetings(name)%;
          Hello there, %name %mend
```

when invoked at %greetings(fred) will yield the string:

```
          Hello there, fred
```

Notes:

(1) In either a named parameter list or the corresponding argument list in the call, white space between the argument serarator's, (i.e. comma's) and the argument character strings is ignored.

(2) Macros may be redefined, in which case the mostly recently encountered definition holds. There is no method of undefining a macro to retrieve its previous definition.

(3) The definition of macros within macros is permitted, with a choice of syntax. This is discussed in the section on %define below and in the example section.

(4) There is no restriction on the invocation of macros within macro definitions, even recursively. The recursive nesting of include files will not be permitted.

(5) To facilitate legibility, there is an implied "gobbler" after the "%;" in the macro definition header. No white space before the start of the macro body is relevant. If some is required, it can be macro-quoted or inserted with one of the white space builtins.

(6) Macro names, as well as variable names, are not allowed to conflict with the macro reserved words. A list of reserved words appears in appendix A.

(7) All macros are global in scope. There are no objects akin to internal procedures in PL/I.

(8) The preceeding two points lead to the following naming conventions:

(a) Reserved words will always contain only lower case alphabetics.

(b) Macro libraries should adopt a convention like that of entry points in a single segment, e.g. libraryname_entryname,

(c) User defined macros are encouraged to consist of upper case letters, numbers, and the underscore character. This makes then easily visable in source segments.

(9) Depending upon interest and available time, a named argument default _may_ be added.

```
<macro-formal-parameter>::=
<macro-identifier>[=<default-value>]

<default-value>::= <macro-construct>
```

where the default value is computed at _define_ time.

When the macro is invoked, if the corresponding argument is the null string, the default value is supplied.

```
<macro-call>::= %<macro-identifier>(<macro-argument-list>)

<macro-argument-list>::= [<macro-argument>[,<macro-argument>]]

<macro-argument>::= <macro-input>
```

When a macro call (or invocation) is encountered, the text of the call is replaced by the text stream produced by the invocation. Macros must be defined before use, either in the source segment, an include file, or in a macro library. See below for a further description of macro libraries and search rules for locating macros.

Examples:

```
%fhu(fred,%hiya,%fhu(%bahr(sam)))
```

is an invocation of the macro "fhu" with three parameters, the first being the character string "fred", the second the value of the variable "hiya", and the third the result of the macro "fhu" invoked with argument the result of the macro "bahr" called with argument "sam".

Notes:

(1) The order of evaluation of parameters is from left to right. Those who use this knowledge to take advantage of side effects should be wary.

(2) Leading and trailing white space between delimiters, i.e. "(", ",", and ")", is ignored.

(3) Unspecified arguments are passed as a null string so that %xxx(a,,%a) is a call to macro "xxx" with three arguments, the first the single character "a", the second the null string, and the third the value of the macro variable "a".

(4) The number of arguments in an invocation can be either less than, equal to, or greater than the number of named parameters in the definition. If less than, the unspecified parameters are considered to be the null string, and if greater than, the excess arguments are referenced as numbered parameters, a description of which is given below.

MACRO VARIABLES

Declarations

<macro-variable-declaration>::=
<macro-scope><space>...<macro-identifier> <space>...
[<macro-variable-type>]<macro-terminator>

<macro scope>::= <macro-external-scope> | <macro-internal-scope>
| <macro-local-scope>

<macro-external-scope>::= %external | %ext

<macro-internal-scope>::= %internal | %int

<macro-local-scope>::= %local | %loc

<macro-variable-type>::= <macro-scalar> | <macro-stack> |
<macro-queue> | <macro-set> | <macro-array>

<macro-scalar>::= [[scalar][<macro-initializer>]]

<macro-initializer>::= =<macro-input>

<macro-stack>::= [{<macro-positive-integer>}] stack

<macro-queue>::= [{<macro-positive-integer>}] queue

<macro-set>::= [{macro-positive-integer}] set

<macro-array>::= {<macro-array-bounds>}[array]

<macro-array-bounds>::= [<macro-lower-bound>:]<macro-upper-bound>

<macro-lower-bound>::= <macro-integer>

<macro-upper-bound>::= <macro-integer>

Before a macro variable can be used it must be declared in a macro variable declaration statement. All macro variables are character strings. They have both scope and type.

The scope rules determine where the names of these variables are known within an invocation of the macro processor.

External variables are known through the entire compilation after the point at which they are declared. If the declaration occurs within a macro definition, that macro must be invoked before the declaration is considered to have occurred. Once declared, any further declaration of an external variable of the same name is ignored. External variables retain their values in the same fashion as PL/I static variables.

Internal and local variables are known only within the macro in which they are declared. Internal variables retain their values from invocation to invocation, while local variables are known only during the invocation of the macro in which they are imbedded. Internal variables declared outside a macro definition, i.e. in loose text are deemed to belong to a macro whose name is the null string, and are not known within the invocation of any macro.

The use of local variables outside macros is not defined. The macro processor may or may not indicate an error and the results can not be guaranteed.

When a macro variable is assigned or referenced, the scope rules for finding that reference are first local, parameter,or internal, then external. Variable names must not conflict with macro names nor with reserved words. Furthermore, there can be no conflict of parameter names with local or internal variable names.

Macro variables can either be scalars or one of a variety of aggregate types. The scalars are much like scalars in other languages. They may be assigned or referenced.

If "xxx" is the name of a scalar which is accessible via the scope rules, then the macro processor replaces the string "%xxx" by the current value of that scalar variable. Scalars not explicitly initialized are implicitly initialized to the null string.

Scalars may be initialized at declaration time to any input the macro processor is capable of exaluating at the time the declaration actually occurs. For internal and external variables, this initalization only occurs the first time the declaration is seen by the macro processor. For local variables, this initialization occurs each time the macro containing it is invoked.

If scalars are initialized, all leading and trailing white space in the initialization is irrelevant. Required white space can be obtained with the white space builtins or through macro-quoting.

Examples

```
     %external TIME scalar = MONEY   %;
     %loc HO scalar %;
     %int FU=%bar(%flu)%;
```

The syntax of aggregate type declaration is very similar to that for scalars, but the semantics are much different.

A macro array is an ordered collection of character strings. If in the declaration a single bound is declared, the array is assumed to have subscripts beginning at 1 and continuing to its upper bound. If bounds are declared they are the lower and upper bounds. At this time arrays are one dimensional. Future releases may provide for multi-dimensional arrays as well as for the initialization of arrays.

Examples:

```
     %local FOO {20} %;

     %ext GORGONZOLA {%1:%2}  array%;
```

The array FOO has 20 elements, referenced as %FOO{1},...%FOO {20} respectively. The array GORGONZOLA will have bounds given by the first and second unnamed arguments in the call to the macro in which it is located. If either of these does not evaluate to an integer or the upper bound is less than the lower bound, it is an error.

Assignments

```
<macro-scalar-assignment>::= %let<space>...<macro-identifier> =
          <space>...<macro-input><space>...%;

<macro-array-assignment>:: %let<space>...<macro-identifier>
          <macro-array-designator>=<space>...<macro-input> %;

<macro-array-designator>::=  {[<macro_integer>][:<macro-integer>]}
```

An assignment to a macro scalar variables causes the value of the right hand side of the assignment statement to be assigned to the macro variable whose name is indicated on the left hand side of the assignment. Variables must be declared before being assigned.

An array assignment is similar but assignment can either be done to a single element or to a slice, i.e. a set of consecutive members of the array. In this latter form of multiple assignment, all elements are assigned the same value. Future refinements may allow component-wise assignment to a slice. In both cases the macro-input on the right hand side is evaluated and the resultant string assigned to the variable.

Leading and trailing white space is insignifigant on the right hand side of the assignment statement.

Examples:

```
    %let foo = %1 %;

    %let name = %mac22(a,b,c) %;

    %let mung_list{%foob(barb)} = %mung_list {%numb_of_blots} %;

    %let mung_list{2:6} = %mung_list{3} %;
```

In the next to last example above, the bound for the assigned elements are first computed, i.e. %foob(barb). This must evaluate to an appropriate array elements designator. It may, as a side effect, alter the value of the macro variable "number_of_blots". This must be kept in mind.

Variable Reference

```
<macro-scalar-reference>::= %<macro-identifier>

<macro-array-reference>::= %<macro-identifier><macro-ref-designator>

<macro-ref-designator>::= {[<macro-bounds>][;<macro-input>]}

<macro-bounds>::= <macro-integer>[:<macro-integer>]
```

The semantics of variable reference are fairly simple. Only previously declared variables may be referenced. Scalar references are replaced by their values, whereas array references are of one of three types:

(1) A single element. If one subscript is given and no optional semi-colon and macro-input, the value of that element of the array is produced. E.g.

```
    %ABC{20}
```

returns the value of the 20'th elelemnt of the array ABC.

(2) The entire array. If no bounds are given, the entire array is returned, each element separated by a single blank. If the semi-colon is present, the string following it, trimmed of leading and trailing white space, is used to seperate the elements of the array.

```
%powers_of_array{;**} might produce something like:

alpha**beta**gamma**delta
```

(3) A slice. If two subscripts of the array are given, and the first is less than or equal to the second, those elements of the array will be returned, seperated by a single blank, or if the semi-colon form is use, the seperator string indicated after the semi-colon.

```
%frammel{%numer1:%numer2;%my_seperator}
```

Structured Data Types

A macro stack has the property that it is assigned like a scalar but referenced like either an array or a scalar. An assignment is equivalent to pushing an element on the top of the stack and a scalar reference is equivalent to popping the top element off the stack. An array reference can be use to examine elements in the stack. The subscript 1 refers to the top of stack element, the subscript 2 to the second element to be popped off the stack, etc. It is invalid to attempt to examine a stack element not present in the stack. Stacks are considered to be unlimited in size unless an upper bound is explicitly given at declaration time. In this case, the evaluated positive integer is the maximal number of elements in the stack. It is an error to push more than that number of elements onto the stack. Stacks may not be initialized.

Examples:

```
%external STAK stack%;
%loc OPTOR_STACK {%STACK_SIZE}  %;
```

STAK is a macro stack of external scope that is unlimited in size, but OPTOR_STACK is a local stack whose size is computed (on each invocation of the macro in which it is contained) to be equal to the value of the macro varible STACK_SIZE.

A macro queue is much like a macro stack, except the discipline for removal is first in-first out whereas for a stack it is last in-first out. In all other respects they are equivalent. An array reference, with subscript n, to a queue returns the n'th element that will be removed.

A macro set is a an aggregate type composed of elements which are distinct character strings. Assignments are made in the scalar fashion. At the time of assignment a check is made to see if the character string assigned to the set is already an element. If so, the assignment has no effect; otherwise the string is made an element of the set and the number of elements in the set incremented by one. Like queues and stacks, the maximal number of elements can be established at declaration time or if no such limit is placed, an unlimited number of elements is allowed. Initialization of sets is not permitted. The only permissible reference to a set is an array reference to all its elements, as neither a scalar nor an array reference to a single element by position is meaningful.

In a later implementation, the set operations union, intersection and difference may be added.

Example:

```
          %int Fixed_bins set%;
          %let Fixed_bins=arg_count%;
          %let Fixed_bins=bat_index%;
          %let  Fixed_bins = arg_count%;
          declare ( %Fixed_bins{;,} ) fixed bin;
```

produces the string:

```
          declare ( arg_count,bat_index ) fixed bin;
```

PARAMETER REFERENCE

In the following discussion it is assumed that there are m parameters in the named parameter list of the macro being invoked, that there were n arguments is the invocation of this macro, and that m and n are non-negative.

Parameters are of two kinds, named and numbered. Neither can be declared, but both are implicitly of type scalar and of local scope. Named parameters are indicated in the head of a macro declaration, e.g.

```
          %macro xyz (foo1, bar23) %;
```

A reference to named parameter has the same syntax and semantics as a scalar reference. If the macro is called with fewer arguments than parameters (i.e. n < m), those parameters with no corresponding argument are assumed to be the null string. Of course, the first argument corresponds to the first parameter, etc. until the m'th argument.

<macro-numbered-parameter-reference>::= %<positive-integer>

There can be no white-space between the percent-sign and the positive-integer. It should be noted that the positive-integer is not a macro-construct whose result is a positive integer. A variable reference to a parameter can be achieved in a manner described below.

If k is a positive integer, then %k is a reference to the (k+m)'th argument in the invocation of the macro. References to %k when k > n - m, i.e. a numbered parameter reference to an argument that does not exist, are treated as null strings.

<%macro-numbered-parameter-count>::= %*

The macro-numbered-parameter-count is the character string representation of the number of optional (numbered) arguments passed in the invocation of the macro. In the terminology given above it is the maximum of 0 and n - m. All such system supplied numbers are trimmed of leading zero's.

Examples:

The following table may be illuminating:

The asterisk indicates that the named parameter does not exist in that row of the table.

<u>Macro Header</u>	<u>Macro call</u>	<u>%x</u>	<u>%y</u>	<u>%1</u>	<u>%2</u>	<u>%3</u>	<u>%*</u>
%macro z()	%z()	*	*				0
%macro z()	%z(a)	*	*	a			1
%macro z()	%z(a,b)	*	*	a	b		2
%macro z()	%z(a,b,c)	*	*	a	b	c	3
%macro z(x)	%z()		*				0
%macro z(x)	%z(a)	a	*				0
%macro z(x)	%z(a,b)	a	*	b			1
%macro z(x,y)	%z()						0
%macro z(x,y)	%z(a)	a					0

%macro z(x,y)	%z(a,b)	a	b		0
%macro z(x,y,w)	%z()				0
%macro z(x,y,w)	%z(a)	a			0
%macro z(x,y,w)	%z(a,b)	a	b		0
%macro z(x,y,w)	%z(a,b,c)	a	b		0
%macro z(x,y,z)	%z(a,b,c,d)	a	b	d	1

<macro-variable-parameter-reference>::= %{<macro-ref-designator>}

Macro multiple parameter reference works by analogy to multiple array referencing. For positive integers n1 and n2 with n1 <= n2, the macro construct

```
     %{n1:n2;<stuff>}

     is equivalent to :

     %{n1}<stuff>%{n1+1}<stuff>...%{n2-1}<stuff>%{n2}.
```

With n2 < n1 the construct yields the null string and with n1 = n2 or n2 absent, the construct is equivalent to %{n1}.

In this context, macro-constructs may be used to designate both n1 and n2. If this is done, it is an error if either construct does not evaluate to a positive integer.

Example:

```
     %macro boo %;
     %local ba1=2%; %local ba2=4%;
     %{%ba1:%ba2;%1} %mend
     %boo(**,wombat,aardvark,flea_bag)

     will produce:

     wombat**aardvark**flea_bag
```

Command Line Arguments

Command line argument passing permits a command line interface with the macro processor environment. The syntax of command line argument input is like that for the alm command (SWG p 6-5)

As there are no equivalens to named parameters to correspond to command line arguments, all such arguments are interpreted as numbered arguments. There is an exact equivalnce between command line argument reference and macro numbered parameter reference with the understanding that the command line arguments are known throughout the entire compilation.

```
<macro-command-line-argument-reference>::= %$<positive-integer>

<macro-command-line-argument-count>::= %$*

<macro-multiple-command-line-argument-reference>::=
%${<macro-ref-designator>}
```

Example:

```
     %ext count=1%;
     %do %while %count<=%$* %;  %+
     %include %${%count}_traps ; %let count=%(%count+1)%; %od
```

If the compiler is invoked with the three command line arguments, e.g.

```
     pl1 trap_handler -map -table -arg cpx fpx mpxr
```

the segments "cpx_traps.incl.pl1", "fpx_traps.incl.pl1", and "mpxr_traps.incl.pl1" will be included in the ususal fashion.

As the command line arguments usually come in no particular order, it is sometimes easier to use the builtin function %clarg to test for expected arguments. This is described elsewhere.

FLOW OF CONTROL

The "natural" macro operations are text replacement and concatenation. Hence the vast body of macro activity will consist of argument replacement and the construction of text strings by concatenating such replacements with non-macro text. This is much like sequential execution of instructions in a normal programming language and is clearly insufficient. As a result some control structures have been provided. One such, the macro invocation, is discussed above. Others follow below.

```
<macro-if>::= %if <space>...<macro-condition> <macro-then-clause>
[<macro-else-clause>]  %fi

<macro-then-clause>::= %then <space>...<macro-input>

<macro-else-clause>::=    [%elseif    <space>...<macro-condition>
<macro-then-clause>]...  %else <space>...<macro-input>
```

```
<macro-condition>::=          <macro-expression>                      |
<macro-input><macro-relop><macro-input> | <macro-input>

<macro-relop>::= < | <= | > | >= | = | ^=
```

The macro-condition evaluated in %if's and %while's can be one of the following forms:

(1) If it is of the form %(<macro-expression>), the macro-expression is evaluated arithmetically. If it evaluates to zero, the condition is false; otherwise it is true.

(2) If it is of form <macro-input><macro-relop><macro-input> the comparison is lexicographic. Either of the macro-input's may be generated by macro activity.

(3) If neither of the above forms hold, the text is evaluated as a character string (with macro replacement activity). In this case, the strings "0", "f", "no", and "false" cause the condition to be false; otherwise it is true.

Examples:

```
%if %(%*+%$*-6) %then ...

%if %[day_name]=Sunday %then ...

%do %while %name1^=%name2 %; ...
```

When an %if is encountered, the macro condition immediately following is evaluated. If this string does not evaluate to either "0", "f", "false", or "no" (ignoring case), the macro-input immediately following the first %then is executed and the rest of the construct, to the matching %fi, is ignored. If the macro-condition is equal to one of the aforementioned false values, then

(a) The macro-conditions of the %elseif's are evaluated sequentially until one does not evaluate to a false string. The macro-input of its %then clause is executed and the rest of the macro-if statement is ignored.

(b) Else, if none of the %elseif's evaluates to true, then the macro-input belonging to the %else clause is executed.

(c) Else, if there is no %else clause, then no macro activity occurs as a result of this macro-if.

In short, the macro-if behaves exactly like the garden-variety programming language if...then...elseif...then...else... construct. Macro-if's may be imbedded within macro-if's.

Examples:

```
     %if %xx %then %let name=pheu%; %fi

     %if %day=Monday
        %then %include hugga;
        %else %include mugga;
     %fi

     %if %clarg(ver1)
        %then %let size=1024%;
     %elseif %clarg(ver2)
        %then %let size=2048%;
        %else %let size=4096%;
     %fi
```

Notes: There is no multiple closure of macro-if's by a single %fi.

<macro-return>::= %return

The effect of a %return within a macro is to halt processing of the macro at that point. No further activity occurs within that invocation of the macro. There is an implied %return at the %mend or %dend of a macro definition.

If the macro-return is encountered outside a macro definition, but within an include file, no further contents of the include file are seen by the compiler (in this instance of the %include). If the macro-return is encountered in loose text in the source segment, then no more of the source segment is seen by the compiler.

The macro-return is purely a control statement and produces no resultant text replacement activity.

If the %return is imbedded in another macro-construct, e.g. a macro-if or a macro-do, then the syntax of that statement must be correct, even if it is the case that the remainder can never be executed under any circumstances. As a result the following statement is in error:

```
     %if TRUE %then %return
```

```
<macro-do>::= %do<space>...<macro-input> %while<space>...
<macro-condition>%; <macro-input> %od
<macro-break>::= %break
```

The macro-input following the %do is performed. The macro-condition following the %while is tested. If it evaluates to a false string, no further processing takes place as part of this construct. Otherwise, the macro-input following the %; after the macro-condition is performed. This cycle of performance of the preceeding input, testing, and performance of the trailing input continues until the macro-condition evaluates false or a macro-break is encountered. this latter construct causes an immediate exit from the loop.

Examples:

```
%loc name=%1%;
%do %while %name^=%[active_func %gorp] %;
%let gorp = %next (%gorp) %;
%if %gorp=%then %let name=%2%; %break %fi %od

%let var_num=0%;
%do %let var_num=%(%var_num+1) %while %(%var_num<=%*) %;
ARG. NO %var_num = %{%var_num} %newline(1) %od
```

In the first example, the loop is executed until the variable "name" is equal to the value of the active function, or until the variable "gorp" is equal to the null string. In the first instance, "name" will be set to the first numbered parameter, while in the second instance, it will be set to the value of the second numbered parameter. The second example cycles through the unnamed arguments in a call to a macro printing out an identifying string and the value of the argument.

Notes: The position of the test between the two macro_inputs allows for either leading or trailing decision within one convenient syntax for looping. If the first input is null, the condition is tested before any execution of the trailing macro-input. This corresponds to WHILE condition DO statements END, whereas if the second macro-input is null, the construct corresponds to DO statements UNTIL condition END.

Nesting of macro-do's is possible, but possibly confusing to readers. Multiple closures of macro-do's is not supported.

EXPRESSION EVALUATION

<macro-arithmetic-evaluation>::= %(<macro-input>)

The effect of the arithmetic evaluator is to perform arithmetic, relational, and logical evaluation of character strings. Since macro language is string manipulative and not arithmetically orientated, the statments:

```
          %local foo=5%;
          %let foo=%foo+1%;
```

assigns to foo the string "5+1", not "6". As a result, if arithmetic evaluation is required, it must be explicitly demanded. The macro-input is expanded in a left to right scan and the resultant character string treated as though it were an arithmetic expression. The operators available are the four arithmetic operators ("+", "-", "*", and "/"), left and right parenthesis for grouping, and relational operators ("=", "^=", "<", ">", "<=", and ">="). Normal PL/I precedence rules apply with relational operators having lower precedence than "+". The relational operators return a value of "1" if the relation holds and "0" otherwise. As a result, AND'ing and OR'ing can be accomplished with "*" and "+" respectively. All arithmetic is done as fixed bin(35). It is an error if the macro-input is not capable of being evaluated due to the inclusion of non-numeric items or arithmetic overflow.

Examples:

```
     %let fugue=%(%fugue*2)%;

     %if %( (%drink=MILK_SHAKE) * (%sandwich=ROAST_BEEF) )
               %then %traf %fi

     %local x_ok=0%;
     %local var=1%;
     %do %while %var<=%$* %; %+
     %if %looks_reasonable(%${%var})
               %then %let x_ok=%(%x_ok+1)%; %fi %+
     %let var=%(%var+1)%; %od
```

In the first example the value of the macro variable fugue is doubled. In the second example, if both conditions are met, the macro variable traf is output. In the third example, each of the command line arguments, if any, is used as an argument to the macro "looks_reasonable". Each time the macro returns a non-false value, the variable x_ok is incremented. This example illustrates the use of a variable reference to a command line argument.

COMMENTS AND ERROR REPORTING

<macro-comment>::= %comment <text> %;

<comment-text>::= any string of characters not containing the <macro-terminator>

This construct has no effect on the semantics of macro processing, yields no resulting string, and allows the inclusion of comments in macro constructs.

<macro-error-statement>::= %error <space>...<macro-integer> ,<macro-input> %;

When a macro-error-statement is encountered, the macro-integer and the macro-input are evaluated. The macro-integer must be a positive integer less than 5. This construct yields no resulting string, but is used to generate a macro-error message at lex time in the compiler. The error is passed on the the compiler and is used to generate a PL/I error message with severity equal to the value of the macro-integer. The purpose of this construct is to allow the authors of macro libraries to report incorrect macro usage.

Example:

```
      %macro frob()
      %comment construct a call to the frob function and returns %;
      %if %*<=3 %then %error 2, "frob called with too few arguments" %;
       ...
       ...
      %mend
```

MACRO BUILTIN FUNCTIONS

Macro builtin fuctions provide useful functions that either can not be performed directly in macro language or can be performed much more efficiently than if constructed in macro language. They all have the syntax of macro calls, except that unlike some user defined macros, all builtins that take a fixed number(s) of arguments. These calls will result in errors if the argument count is unsatisfactory.

String Handling Builtins

The following all have the same semantics as the corresponding pl/I builtins.

%substr()	%ltrim()	%verify()
%length()	%rtrim()	%search()

%index() %reverse()

%count(<macro_input>,<macro_input>)

%count returns a macro integer and has the following semantics:

(1) If the length of the first argument = 0, %count returns 0.

(2) If the length of the second argument = 0, %count returns the length of the first argument.

(3) Otherwise the value returned by %count is the number of characters in the first argument before the first character in the first argument not present in the second.

(4) If the second argument is omitted, a single blank is assumed.

Examples:

```
     %let R=%count (<stuff1>, <stuff2>)

     is equivalent to:

     %let R=%(%verify(<stuff1>,<stuff2>)-1) %;
     %if %R=-1 %then %let R=%length(<stuff2>)%;
```

If the value of %name is the 7 character string "abc+def", then

```
          %count(%name,abcdef) = 3

          %count(%name,+*/-) = 0

          %count (xy,   wxyzab) = 2
```

Arugment Handling Builtins

%arg(<macro-input>)

When encountered in a macro, the value of the arg builtin is "1" if the <macro-input> is equal to any of the arguments in the call corresponding to numbered, i.e. optional, parameters. Otherwise, this builtin returns "0". Named parameters must be tested explicitly.

%clarg(<macro-input>)

If any of the command-line-arguments is equal to the <macro-input> this builtin returns "1"; otherwise it returns "0".

Both constructs should be used instead of explicit iteration through the argument list.

Aggregate Handling Builtins

```
%lbound(<macro-input>)                    %hbound(<macro-input>)
%empty(<macro-input>)
%delete(<macro-input>,<macro-input>)
%member(<macro-input>,<macro-input>)
```

For all the above, the first argument must expand to the name of a macro aggregate type. Scope rules apply. When a second argument appears, it may be a arbitrary character string.

%lbound() returns the lower bound of an array and the value "1" for all other aggregate types.

%hbound returns the upper bound for an array. For other aggregate types, the returned value is the number of elements currently held in that aggregate. For empty queues, stacks, and sets, this is "0", otherwise it will be a positive integer, less than or equal to the declared maximal size, if any.

%empty does not return a value. It has the effect of removing all elements from an non-array aggregate. For an array, all elements are set to the null string.

%delete does not return a value. For an array, if the second argument as a character string is equal to any element of the array, that element is set to the null string. For sets it removes the element from the set. For queues and stacks, the element is removed and the queue or stack is appropriately adjusted. It is not permitted to delete a non-existant element nor to delete an element by position rather than by value.

%member returns a "1" if the second argument is equal to any element of the aggregate type. Otherwise it returns a "0".

White Space Builtins

```
%newline(<macro-positive-integer>)
%htab(<macro-positive-integer>)
%vtab(<macro-positive-integer>)
%space(<macro-positive-integer>)
%newpage(<macro-positive-integer)
```

These builtins generate white space as a result. If invoked with an argument, that argument must evaluate to a positive integer, which is the number of such white space characters produced. If no argument is present, the number one is assumed.

Other Builtins

%unique()

%unique takes no arguments and returns a positive integer in the range 1 to 2**35-1. The first 2**17-1 such integers are guaranteed to be unique. It may be that these numbers will be produced by a random numer generator.

%number(<macro-input>)

If the expansion of the macro-input yields a character string capable of being evaluated as a macro-integer in an arithemtic expression, the value returned is "1", otherwise it is "0".

ACTIVE FUNCTION CALLING

<macro-active-function-call>::= %[<macro-input>]

The expanded macro-input is processed as as active function and the result of the active function call returned. For nested active functions, only the outermost left square bracket need be preceeded by a percent-sign. There will be a handler for active function errors, but erroneous active function calls will yield macro and pl1 lex errors. In the best of circumstances an error will be noted and the null string returned. Active functions are found according to the usual search strategy.

Examples:

```
          if reg_value = %[hexadecimal %frob]

          %let label = %[ ltrim [unique] ! ] %;

          call xx_$yy (fnp, %[my_ac_fn bac let tom], code);
```

WHITE SPACE CONTROL

<macro-null-separator>::= %.

<macro-gobbler>::= %+

The null-separator causes no output, but serves as an delimiter to separate macro lexemes from adjoining text. The gobbler absorbs all trailing white space and is usefull in formatting.

Examples:

```
            %macro hi_there
            hi there %+
            bozo %mend

            %hi_there()
```

returns the string:

```
            hi there bozo
```

More useful,perhaps is the distinction:

```
            %foo%.(x,y)
```

which is a concatenation of the value of the macro variable foo with the string "(x,y)", whereas:

```
            %foo(x,y)
```

returns the value of the macro foo called with arguments "x", and "y".

OTHER MACRO FACILITIES

Macro Protection

<macro-protected-string>::= %"<macro-protected-text>%"

The macro processor does not expand any macro constructs in the macro-protected-text. It does remove both the leading and trailing %" and converts internal double-%"'s to a single %".

Examples;

```
            %let delim = %"%;%"%;
            %let quoter = %"%"%"%"%;
```

assigns the string "%;" to delim and the string "%"" to quoter.

Literal Percent

<macro-literal-percent>::= %%

The result of the literal-percent is to produce a single percent in the output. This is used in defining macros within macros and for rescanning purposes.

Examples:

```
          %let percent = %% %;
```

assigns to the variable percent the character "%".

The following example shows nested macro definition:

```
          %macro define_mac
             %%macro %1
             %%[%2 %%{ }] %%mend
          %mend

          %define_mac(ADD,my_add)
```

will generate the following macro definition:

```
          %macro ADD
          %[my_add %{ }] %mend
```

The macro invocation

```
          %ADD(fred, sam, joe)
```

will return the string returned by the active funcion call

```
          [my_add fred sam joe]
```

Macro Rescanning

<macro-rescan>::= %rescan<macro-input>%;

The macro-input is evaluated and the resultant text is rescanned for further macro-activity. Under normal circumstances macro activity is not rescanned for subsequent macro activity. One such special case is that of include files.

Example:

```
     %macro mac1
     %rescan %%foob(%1)%; %mend

     %macro foob
     %{ ,--} %mend

     %mac1(%"hey,ho,tiddly,i,pom%")
```

will produce:

```
     hey--ho--tiddly--i--pom
```

NESTED MACRO DEFINITION

The ability to define a macro which can be used to further define other macros is often useful. There are two methods for doing this which are best demonstrated by example.

```
          %macro declare (name,attirbute) %;
          %rescan %%macro %name %"%;%"
                  declare (%%{;,})   %attribute ; %%mend %;
          %mend
```

If the macro "declare" is invoked as:

```
          %declare (FB,fixed binary)
```

the expansion of "declare" before rescanning produces:

```
          %rescan %macro FB %;
                  declare (%{;,}) fixed binary; %mend %;
```

The rescanning defines FB as a macro.

If FB is then invoked as:

```
          %FB(bat_snout,liver_wort, old_sox)
```

the resultant string is:

```
          declare (bat_snout,liver_wort,old_sox) fixed binary;
```

The alternate form of nested macro definition uses the %define...%dend construct and produces more legible output and automatically causes rescanning. With the %define statement, the macro "declare" is:

```
          %macro define (name,attribute) %;
          %define %name %;
                  declare (%%{;,})   fixed binary; %dend
          %mend
```

It is important to note that %define replace %%macro, %dend replaces %%mend. Rescanning is automatic, so it is not specified. This yields the winnage that the %; in the header of the nested macro definition need not be protected. It was required in the previous example because it would terminate the %rescan prematurely. The double percent's are still required because they refer to the numbered parameters of the defined macro, not the defining macro.

The macro "declare" can be imbedded in other macros, so for example:

```
            %macro fixed_bins %;
            %local i = 1 %;
            %do %while %i <= %* %;
               %declare(FB%{%i},fixed binary(%{%i}))
               %let i = %( %i + 1 ) %;       %od
            %mend
```

When invoked as:

```
            %fixed_bins(8,17,21,24,30,35)
```

produces the declaration macros "FB8","FB17",..."FB35". Thus the invocation:

```
            %FB8(ada,otto,madam)
```

produces the desired result:

```
            declare (ada,otto,madam)   fixed binary (8);
```

<macro-nested-definition>::= %define<space>...<macro-input> %dend

The macro input must evaluate to the form required in a macro-definition, i.e.,

(1) the name of the defined macro must be a macro-identifier;

(2) there may be an optional named-parameter-list;

(3) the macro terminator must be included in the macro-header.

Macro-nested definitions may only appear within macro-definitions.

MACRO CROSS REFERENCE

The macro processor will generate a cross reference listing of all macro variables, builtins, pseudo-ops, and macro calls used in the compiled program. This will include such items as name of the item, class of item (e.g. macro, ext stack {20}, parameter, builtin, etc), and locations of declarations and use. As in other line numbering schemes, all line numbers refer to real locations in the storage system. Use of macros with macros will be designated in a manner analagous to the present listing of include files.

Example:

MACRO ACTIVITY IN THIS COMPILATION

NO.	IDENTIFIER	TYPE	DECLARATION/USAGE
0	comm_err_	macro	15/19 9-11 M4
1	epugoms_	ext array	9-11/M4 22

LISTINGS

Macro activity in the compiler introduces two problems in the general area of listings: what should the compiler produce and how should it number its lines.

In many cases, the programmer will wish to see only the macro, e.g. in the case of macro use of named constants. In others, the expansion is more relevant, especially when debugging. What holds for macro activity also holds for include file content.To solve this problem, expansion control will be controlled by two macro builtin functions, %ilist to control include file expansion, and %mlist to control macro expansion listing.

Both builtins are called with either no arguments or one argument which may be either "source", "expand", or "both". If the argument is "source", the listing will show only the source of all macro constructs. If the argument is "expand", the listing will show only the expansion of macro constructs. If the argument is "both", both the source and the expansion is shown, with the expansion first, followed by the source, delimited by PL/I comment delimiters. The listing builtins may also be called with no argument, in which case the present value of that listing control is returned.

Line numbers always will refer to real locations in the storage system. As a result, if a macro expansion requires several lines of output, it will appear in the listing, and in the statement map as part of the line in the real storage system from which this macro activity was generated.

Examples:

Suppose lines 9-11 of the source contained:

```
          if foo = %blag
          then foo = foo + %bar;
          else foo = foo - %bar;
```

and the value of %blag were 16384, and the value of %bar 1024, then the result in the listing for mlist set to "source", "expand", and "both" respectively would be:

```
     9      if foo = %blag
    10      then foo = foo + %bar;
    11      else foo = foo - %bar;

     9      if foo = 16384
    10      then foo = foo + 1024;
    11      else foo = foo - 1024;


     9      if foo = 16384 /%blag*/
    10      then foo = foo + 1024 /*%bar*/
    11      else foo = foo - 1024 /*%bar*/
```

Further, let the macro COM_ERR be defined as:

```
%macro COM_ERR
%comment 1st param is condition, others are args of com_err_ %;
%comment defaults: param1-code^=0; param2-0%;
%loc cond=code^=0%; %loc arg1=0%;
%if %1^=%. %then %let cond=%1%; %fi
%if %2^=%. %then %let arg1=%2%; %fi
if %cond
then do;
          call com_err_ (%arg1,"mpxy", &{3:%*; ,});
          return;
     end;
%mend
```

then, if lines 20-22 of the source contained:

```
          call xxx_$yyy(name, age, soc_sec_num)
          %COM_ERR ( ,code,  "^a", soc_sec_num)
          call get_data (soc_sec_num);
```

then the result in the listing for mlist set to "source", "expand", and "both" respectively would be:

```
    20      call xxx_$yyy(name, age, soc_sec_num);
    21      %COM_ERR ( ,code,  "^a", soc_sec_num)
    22      call get_data (soc_sec_num);


    20      call xxx_$yyy(name, age, soc_sec_num);
    21      if code^=0
            then do;
                    call com_err_ (code, "mpxy","^a",soc_sec_num);
                    return;
                end;
```

```
   22      call get_data (soc_sec_num);

   20      call xxx_$yyy(name, age, soc_sec_num);
   21      if code ^= 0
           then do;
                     call com_err_ (code, "mpxy", "^a", soc_sec_num);
                     return;
           end; /* %COM_ERR ( ,code,  "^a", soc_sec_num) */
   22      call get_data (soc_sec_num);
```

For %ilist set to "source", "expand" and "both" respectively, the listing would show the %include statement, the contents of the include file, and first the %include statement and then the contents of the include file.

The default value for %ilist is "expand" and for %mlist "source".

Calls to %ilist and %mlist with no argument can be helpful in saving and restoring expansion parameters in macro library usage, macro debugging, etc.

EFFECTS UPON PROBE

The results of macro activity can depend upon the compile time environment in a manner that can not easily be duplicated at run time. As a result, it is unreasonable to expect probe (and other run time tools) to be able to recreate that environment.

It might be possible to include in the object segment some sort of tool for the reconstruction of this environment, but the costs of this could be enormous. Therefore, what probe will know about is the PL/I program as seen by the compiler after macroprocessing has occurred.

The first implication of this scheme is that probe will not be able to expand macro constructs. If it is expected that probe is possibly necessary, then a suitable listing should be obtained.

The other implication of this strategy is that all line numbers refer to real entities in the storage system. When probe is asked to display source segment statements, it will refer to the PL/I source segment and relevant include files. When probe is asked to set break points, it refers to the statement map, which always uses real storage system locations.

Therefore, should a macro on line n of the source generate m PL/I statements, probe will know these as statements 0, 1, 2,...m-1 of line n.

EFFECTS UPON FORMATTING PROGRAMS

There are no plans at this time to alter indent. Some time in the future, it would be useful to allow format_pl1 to recognize macro constructs, but it is unclear whether it would format them as well. At the present time format_pl1 only understands the %include statement.

MACRO LIBRARIES

Macro libraries will not be present in the initial implementation of the macro processor. In the future, if a macro is invoked and has not been defined in the source segment or an include file, or defined dynamially, then relevant macro libraries will be searched. These will be found using the translator search rules. Tools for library management will include listing the table of contents, adding macros, deleteing macros, updating macros, and editing macros. It may be the case that macros in libraries be kept in a semi-compiled form, as an efficiency measure. In this case, the macro library would perform syntax checking when macros were placed in the library.

CONTROL ARGUMENT ADDITIONS TO THE PL1 COMMAND

-mlist STR

> change the default value of mlist to STR, which must be either "source", "expand", or "both".

-ilist STR

> change the default value of ilist to STR, which must be either "source", "expand", or "both".

-argument
-ag

> indicates that the following strings are to be taken as command line arguments to the macro processor. If present, this must be the final control argument.

IMPLEMENTATION SCHEDULE

Before the macro processor is fully integrated into the PL/I compiler it will be available in a stand-alone form. This should occur by the first of the year, with a subset of the full language. An announcement will be made shortly. An announcement will be made shortly. shortly. Sometime in the early part of next year, the macro processor should be fully integrated into the pl1 compiler. Those who use only %include, %skip, %page, %print, and %noprint should not expect any increase in compile time.

THE STAND ALONE MACRO PROCESSOR

Until such time as the macro processor is integrated into the PL/I compiler, it can be used in a stand alone fashion with the command "macro".

Name: macro

The macro command invokes the stand alone macro processor to translate a segment with macro constructs in accordance with the defined macro language. This command can not be called as an active function.

Usage

```
     macro inpath {outpath} {-control_args}
```

where;

1. path
   is the pathname of a PL/I source segment with macro constructs. If the path does not have a suffix of pl1, then one is assumed. The use of a source segment is incompatible with the -ia control argument.

2. outpath
   is the pathname of the macro processed source segment. The suffix pl1 is assumed if not given. If outpath is omitted and there are no errors and the -pr or -ia control arguments are not used, then inpath is overwritten. If there are errors and outpath is omitted, a version of the macro processed segment is left in the process directory.

3. control_args
   can be chosen from the following list:

   -include, -inc
   removes all %include statements and replaces them by the contents of the include file, expanding that as well. The default is to leave the %include statements intact.

-no_include, -ninc
does not process %include statements, but leaves them intact. (DEFAULT)

-print, -pr
indicates that the macro processed output is printed on user_output rather than placed in a segment. This control argument is incompatible with the use of an outpath segment name, and is assumed when the -ia control argument is used.

-version, -ver
will print the version number of macro.

-no_version , -nver
will not print the version number. (DEFAULT).

-list, -ls
produces a line-numbered listing of the macro processed result with a cross reference table in a segment with suffix maclst.

-call STR
will call STR as a command after the translation is complete, if the macro processor does not discover an error.

-argument STR's, -ag STR's
indicates that the following strings are to be passed as command-line-arguments. If this control argument is present, it must be the last one and at least one argument must follow.

-interactive, -ia
indicates that the input is to come from user_input rather than a segment. This control argument is incompatible with either an input or output path name, the -call control argument, and the -list control argument.

When interactive mode is entered, macro promts for input when ready and accepts everyting until a line consisting solely of a period. At this point the input is processed and the result delivered to user_output followed by the prompt. A line consisting solely of the five characters "%quit" terminates the interactive session.

Interactive mode operates as though the -include control argument were specified.

In the stand-alone macro processor, the listing control statements %print, %noprint, %mlist, %page, %skip and %ilist are replaced verbatim, because they are intended as directions to that section of the PL/I compiler responsible for listings.

Please send comments to:

MPresser.Multics,

or

Marshall Presser
Honeywell Information Systems
575 Tech Square
Cambridge, Mass. 02139

Or call:

(617) 492-9320
HVN 261-9320

APPENDIX A

At the time of this writing, the following words are reserved and can not be used as the names of macros or variables. The list is subject to expansion, but observes the convention that all reserved words will contain only lower case letters.

comment				
do	while	break	od	
define	dend			
if	then	else	elseif	fi
error				
ext	external			
include				
int	internal			
let				
loc	local			
macro	mend			
print	noprint			
page				
rescan				
return				
skip				
quit				

At the time of this writing, the following builtin functions have been defined. It is not permissable to redefine them as either macros or variables.

arg	clarg			
hbound	lbound			
empty	delete			
member				
ilist	mlist			
ltrim	rtrim			
count				
index	verify	search		
substr				
length				
htab	vtab	space	newline	newpage
unique				