TO: MULTICS Programmers.

FROM: J. H. Saltzer.

SUBJ: PL/I Documentation.

DATE: July 10, 1965.

Enclosed are three documents describing the PL/I language as it will be available on the MULTICS system. Together with the I.B.M. PL/I Reference Manual, they provide both a tutorial summary and introduction to the PL/I language, and a complete reference description of MULTICS PL/I (including early PL/I.) The four documents required for the complete description are:

1. A Brief Summary of the PL/I Language. (Saltzer)

2. Early NPL Subset. (Morris, McIlroy)

3. Data Layouts in ENPL for the GE636. (McIlroy)

4. PL/I Language Specifications, IBM form C28-6571-0.

The reader should realize that this documentation is temporary and detailed specifications are subject to change. As a rough guide, the following assumptions may be made:

1. There will be minor changes in the Early PL/I subset, but a translator for the subset will be available shortly.

2. Data layouts will remain the same for the complete PL/I translator.

3. The complete translator will probably not be available until mid-1966, and it may not correspond precisely with the complete PL/I reference manual.

4. More adequate documentation will someday be available.

A Brief Summary of the PL/I Language

J.H. Saltzer
June 28, 1965

Introduction

These notes have been written in an attempt to provide quickly some readable documentation on a reasonably useful subset of the PL/I language. They should do little more than provide an anchor point from which a programmer may build up detailed knowledge with the help of published complete reference manuals. (e.g., IBM manual S360-29, form C28-6571-0.) Although every effort to insure correctness in this summary has been made, the reader should bear in mind that the author has not had an opportunity to write, translate, and run a PL/I program; his only contact with the language has been with published manuals and technical reports. It should also be kept in mind that the precise implementation of this language on the GE 636 has not yet been completely established. Preliminary specifications have been taken into account in this summary.

The PL/I language requires many pages to describe in all its glory. The programmer may specify any of dozens of options at any point in his program. Fortunately, the concept of "default" specification has permeated the language. Through default, if the programmer fails to specify an option, he is given automatically (by "default") the option he will most likely need. This summary will, in most cases, describe the language as though the default options were the only ones available; the programmer should realize however, that the language provides the ability to specify virtually any meaningful combination of facilities.

Substitution Statements and Expressions

Substitution statements are statements of the form

```
A = X+3
```

or, more generally,

```
variable = expression.
```

In fact, in PL/I, the most general substitution statement has the form

```
data structure = expression
```

This last form is illustrated by the statement

```
A = B+C
```

in which A, B, and C are all arrays of the same dimension; each element of A is set equal to the sum of the corresponding elements of B and C.

Identifiers are used to name variables, arrays, etc. They contain 31 or fewer characters and must start with an alphabetic character. Examples:

```
ALPHA
LØØP1
INITIALIZATION
```

Variables are named by an identifier. A variable may be declared to represent data of one of at least four types: FLØAT, FIXED, CHARACTER, and BIT. Variables starting with I,J,K,L,M,N are assumed FIXED, all others are assumed floating unless declared otherwise.
Integer arithmetic is done with FIXED variables, on numbers of magnitude less than 2**17.
Normal arithmetic is done with FLOAT variables, with range

$$10^{38} \geq |n| > 10^{-38}$$

A CHARACTER variable or a BIT variable represents a string of characters or bits (often the string is of length one).

Constants of all four data types are provided as follows:

FIXED: a decimal integer. Example: 12, 7942

FLOAT: a decimal number with a decimal point and a <u>required</u> exponent of the form E±n where "n" represents a multiplying power of ten.
example: 12.5E1, .041E0

CHARACTER: Any ASCII characters except single quote between single quote marks
example: 'ABC' , '$100.00'

BIT: A string of zeros and ones between quote marks, followed by the letter B.
example: '1100101'B

Any meaningful combination (mix) of data types in an arithmetic expression is allowed. For example, if Y is a BIT string variable, Z is FIXED, and X is FLOAT

```
X = Y+Z
```

would cause the bit string Y to be converted to a binary integer, and the resulting integer sum is converted to floating. Note: in the early PL/I translator, CHARACTER

mode variables cannot be mixed with other types.

Allowed arithmetic operations are:

**	exponentiation
*	multiplication
/	division
+	addition
-	subtraction

evaluated with the usual rules of precedence. Parenthetical expressions may be used to insure correct precedence.

Although expressions such as X < Y are allowed, there is no "Boolean" data type. Instead such expressions are given BIT values of 0 or 1, and are manipulated in BIT mode. For example, if Q is a BIT variable.

```
Q = X < Y;
```

will give Q the BIT value '1'B if X is less then Y. Mixtures of such BIT data with other data can be used for special effects. For example, in the statement

```
X = Y*(A < B);
```

A and B are compared, producing a BIT value of 0 or 1, which is then converted to the decimal integer 0 or 1, and used to multiply Y. If A is not less than B, X is assigned the value 0.

Allowed relational operations are:

>	greater than
<	less than
<=	less than or equal
>=	greater than or equal
=	equal
¬ =	not equal

The following operations may be used on pieces of BIT data:

&	logical 'and'
|	logical 'or'
¬	logical 'not'

if A = '1'B and B = '0'B

A & B	has value '0'B
A | B	has value '1'B
¬ A	has value '0'B

Program Structure

Statement syntax is free format. The usual statement has the form

label: statement ;

Where "label" is an optional identifier naming the statement. The colon appears only if the label is used. The semicolon is always required at the end of the statement. Tabs and carriage returns are considered as blanks by the translator. Note that in general, blanks are not ignored, but are considered delimiters between syntax elements. Thus, ABCD and AB CD may have different meanings.

A subroutine is a set of statements bracketed by the pair of statements:

```
label:  PRØCEDURE (ARG1, ARG2,...) ;
        ———————
        ———————
        ———————
        ———————
        END label;
```

The label is required, and gives a name to the subroutine. (The matching label after the END statement is optional, but good practice.) ARG1, ARG2, etc., are formal parameters of the subroutine, to be replaced by actual arguments when the subroutine is called. The words PROCEDURE and END are keywords of the language, but they are not reserved.

Control Statements

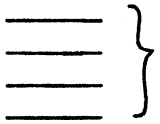
GØ TØ label;
: causes control to be transferred to the statement having the label "label".

CALL subname (ARG1, ARG2, ...);
: invokes the subroutine "subname", with arguments ARG1, ARG2, etc., used in the place of the corresponding formal parameters in the PROCEDURE declaration in the subroutine. The number of arguments in the calling sequence must be the same as the number of parameters in the parameter list of the subroutine.

RETURN;
: used within a subroutine to return control to the caller. When the RETURN statement is executed all subroutine variable storage is released; special declarations are required if some variables are to

be remembered from call to call.

DØ iteration list;



END;

The set of statements up to the matching END statement is executed repeatedly under control of the iteration list. The set of statements including DØ and END is known as a group. The iteration list may be absent, in which case the group is executed once; this degenerate form is used to control the extent of the IF statement (see below.) The general form of the iteration list is

DØ variable = expr1 BY expr2 TØ expr3, WHILE expr4;

part I part II

Both parts of the iteration list are optional, providing the ability to do simple counting of loops, tests for complex termination conditions, or both simultaneously. Both tests are done before the loop is entered the first time.

EXAMPLES: to add up ten numbers in an array:

```
SUM = 0;
DØ  J = 1 BY 1 TØ 10;
      SUM = SUM + A(J);
END;
```

Same example, but stopping if SUM exceeds 100.

```
SUM = 0;
DØ J = 1 BY 1 TØ 10 WHILE SUM < 100;
     SUM = SUM + A(I);
END;
```

In the last example, when control leaves the DO-group, J will point to one place after the array element that caused the sum to overflow.

IF expression THEN statement1; ELSE statement2;

If the expression has non-zero bit-value, statement 1 is executed; otherwise statement 2 is executed. The suffix "ELSE statement2;" is optional. A group of statements delimited by DØ and END may be used in place of either single statement.

Declarations

All declarations describing identifiers are made with a single multipurpose statement, DECLARE. The general method of attaching attributes to names is to type an identifier followed by all of its attributes separated by blanks. A comma allows a second identifier to be declared in the same statement.

For example,

```
DECLARE ALPHA FIXED, INPUT CHARACTER(1), SWITCH BIT(1)
```

would make the variable "ALPHA" of "FIXED" type, the variable "INPUT" a one-character string, etc.

Arithmetic data may be given any one of the attributes:

```
FLOAT
FIXED
CHARACTER(n)
BIT(n)
```

where n is a required integer giving the number of characters or bits in the string.

Arithmetic data may be preset by the INITIAL attribute, as in

```
DECLARE  LIMIT  FIXED  INITIAL (100);
```

The FIXED variable LIMIT is preset to the value 100.

An array may be declared by putting subscript ranges in parentheses after the array identifier in a DECLARE statement. Any number of subscripts may be used. The subscript range is specified as (lower bound:upper bound). If lower bound is missing, it is assumed 1. An array transmitted to a subroutine is declared in the subroutine by an asterisk in the place of the subscript range.

Example:

```
DECLARE  LINE(100)  CHARACTER(1), BOX(-1:10,15),  INPUT(*)
```

would declare the array LINE to contain 100 character strings (each of length 1). The first subscript on the array BOX varies from minus one to ten, the second from 1 to 15. Array INPUT, a formal argument of this subroutine, has its subscript range declared in the calling program.

Structures

The PL/I language includes the ability to arrange data in arbitrary groupings, called structures. These structures resemble an outline in form, and permit the programmer to talk about subgroupings of his data. A complete description of data structures is very lengthy, and better left to the complete reference manual.

Programming Style

In PL/I, as in all languages, it is possible to write obscure programs. For the programmer who prides himself on his clearly-written, easy-to-understand programs, the following suggestions may be of some help:

1. Liberal use of tabulate characters will make a listing easier to read. An appropriate technique would be to type an initial tab on all statements not having a label, with an extra tab for each level of nesting within DO END blocks. Statement labels are then inserted immediately before the first tab character.

2. The language syntax permits blank lines to be used for vertical punctuation between logical pieces of coding.

3. Although the 636 PL/I translator will map upper and lower case alphabetic letters of program text into the same internal code, it would appear to be wise not to utilize this facility to identify a variable with "AbC" in one place and then "aBc" elsewhere in the program. The readability of a program can be greatly increased if all identifiers are typed in upper case, while all PL/I key words (do, procedure, etc.) are typed in lower case.

4. Comments of the form

   /* comment */

   may be placed in the program anywhere that a blank may appear. This freedom should not, however, be considered as a license to strew comments indiscriminately about in the middle of arithmetic expressions.

5. The usual warnings about not making a program depend on the special tricks of a particular language implementation apply in force. A reasonable rule of thumb might be: "Would this program produce correct results on a machine with, say, 32 bits per word and a translator written by

a different programmer?" It should be clear that the question of whether the program would be efficient on a different machine is somewhat irrelevant; the issue is whether or not the program's results can be duplicated precisely.

Sample Program

On the next page is a PL/I subroutine that might be used by the RUNOFF command. The subroutine right-justifies a line of characters by inserting blanks between words. The calling sequence is

```
CALL ADJUST (LINE,NCHARS,SIZE,RMARG)
```

where LINE(1)...LINE(NCHARS) contain the characters of the line. Since not all characters move the carriage forward, SIZE is the physical length of the line; RMARG is the desired physical line length.

```
/*  Routine to right-justify a line.  J. H. Saltzer, May, 1965.  */

ADJUST:   procedure(LINE, NCHARS, SIZE, RMARG);
               declare LINE(*) character(1), (SIZE, RMARG, TMARG,
                    POINTER, DEL, BEGIN, END) fixed;
               if (SIZE  >= RMARG) then return;


/*  Make sure that there are some blanks after the first character.  */

               do I = 1 to NCHARS while LINE(I) = ' '; end;
               ISTART = I;
               do I = I+1 to NCHARS while LINE(I) ¬= ' '; end;
               if (I  >= NCHARS) then return;

/*  Scan line backwards, inserting spaces where there are spaces.  */

               TMARG = NCHARS + RMARG - SIZE;
BACK:          POINTER = TMARG;
               DEL = -1;
               BEGIN = NCHARS;
               END = ISTART;
SCAN:          LNOW = NCHARS;
               do J = BEGIN by DEL to END;
                    LINE(POINTER) = LINE(J);
                    POINTER = POINTER + DEL;
                    if (LINE(J) = ' ') then
                    do;
                         LINE(POINTER) = ' ';
                         POINTER = POINTER + DEL;
                         NCHARS = NCHARS + 1;
                         SIZE = SIZE + 1;
                         if (SIZE  >= RMARG) then return;
                    end;
               end;
               if (LNOW = NCHARS) then return;
               if (DEL >  0) then go to BACK;

/*  Scan line forwards this time.  */

FORWARD:       DEL = +1;
               BEGIN = POINTER + 1;
               POINTER = ISTART;
               END = TMARG;
               go to SCAN;

          end ADJUST;
```