DATE: August 25, 1978 PE-T-483

TO: R & D Personnel

FROM: William M. Miller

SUBJECT: PL/P Specification

Introduction

PL/P is a variant of ANSI PL/I: although a subset in most respects, it contains several significant extensions to the standard language. Since this document describes only the differences between PL/P and ANSI PL/I, readers not familiar with the full language should consult one of the references in the bibliography before proceeding with this document. (Appendix A, "PL/P Course Syllabus," is also attached for the reader's convenience, although it is intended to be used in connection with a course wherein the teacher would expand upon its contents with fuller explanations and examples.)

ANSI Features Missing in PL/P

The following features of ANSI PL/I are not included in PL/P:

1. All forms of input and output
2. The CONDITION mechanism
3. The ALLOCATE and FREE statements
4. The STOP statement
5. The DEFAULT statement
6. All attributes except INTERNAL, EXTERNAL, STATIC, AUTOMATIC, BASED, ALIGNED, UNALIGNED, BIT, CHARACTER, VARYING, NONVARYING, ENTRY, RETURNS, LABEL, POINTER, BINARY, FIXED, LIKE, INITIAL, CONSTANT, VARIABLE, and OPTIONS and (implicit only) DIMENSION, MEMBER, PRECISION, PARAMETER, REAL, and STRUCTURE
7. All builtin functions and pseudovariables except BINARY, CHARACTER, MOD, DIVIDE, AFTER, BEFORE, COPY, DATE, INDEX, LENGTH, REVERSE, SUBSTR, TIME, TRANSLATE, VERIFY, NULL, and ADDR
8. Aggregate expressions and promotion, except promotion from scalar to array in simple assignment statements

9. Implicit declaration of user-defined names

10. Implicit conversion, except of precision or size, between VARYING and NONVARYING CHARACTER data, and between BIT and BINARY with precision <= 15

11. Condition prefixes

12. Variable extents except the "*" string size in a CHARACTER parameter description in the declaration of an external entry

13. Subscripted label constants

14. END statement closure labels

15. Scaled and imaginary arithmetic constants and the default-suppressing character P

16. The %INCLUDE statement

17. OPTIONS on the BEGIN statement

18. Unconnected array references

19. Multiple-target assignment statements

20. BY NAME assignment

21. BIT VARYING

22. The REFER option

23. The ** and / operators

In addition, the following restrictions apply:

1. No reference may contain more than one parenthesized list, except structure references in which the subscripts are distributed over the components.

2. DO indices must be either BINARY or POINTER.

3. The INITIAL attribute may only contain string or arithmetic constants, the builtin function NULL(), or "*".

4. Iteration factors in the INITIAL attribute must be integer constants.

5. Only items of storage class STATIC may be initialized.

6. The BASED attribute may not contain a pointer reference; hence, all references to BASED variables must be pointer-qualified.

7. Bitstrings are aligned so that they will not cross word boundaries, and members of bitstring arrays are word-aligned, regardless of alignment attribute.

8. Builtin functions which do not take arguments must be coded with an empty argument list.

9. The data type must be given for each parameter in an ENTRY attribute and for the return value in the RETURNS option of the PROCEDURE statement and the RETURNS attribute.

10. The declaration of the reference in the LIKE attribute must physically precede the LIKE reference in the program, and must not contain a LIKE attribute.

11. The first argument to the BIT builtin function must be either an integer constant or a BINARY variable of precision <= 15.

12. Scale factors are not allowed in the BINARY attribute.

13. The first argument to the CHARACTER builtin function must be either an integer constant or a BINARY variable.

14. The second argument to the COPY builtin function must be a constant.

15. The third argument to the DIVIDE builtin function must be present and be an integer constant and the fourth argument must be omitted.

16. The length of a SUBSTR builtin function or pseudovariable whose first argument is a bitstring must be calculatable at compile time.

17. Character string constants may not contain the newline character.

18. Unaligned bitstrings may be passed as arguments only via call-by-value.

19. For aggregate parameters and arguments, array size and dimensionality and structure shape are not considered in argument validation.

Implementation-Specific Features

1. All procedures are implicitly recursive, except these internal procedures coded with the SHORTCALL option.

2. The maximum size of character strings is 8192 characters.

3. The maximum size of bit strings is 16 bits.

4. The UNALIGNED attribute implies character alignment for CHARACTER NONVARYING data, bit alignment for non-array bitstring data, and word alignment for all other cases.

5. The ALIGNED attribute implies word alignment in all cases.

6. The maximum length of an identifier name is 32 characters.

7. The maximum length of a source line is 256 characters.

8. POINTER variables use two words (32 bits) of memory and hence may only point to word-aligned data.

9. OPTIONS(GATE) may be specified on the external PROCEDURE statement, causing ring weakening to occur for all PARAMETER, BASED, and EXTERNAL pointers upon assignment.

10. OPTIONS(SAVE(ref)) may be specified on a PROCEDURE statement to cause an RSAV into the variable "ref" to be generated before any other code in the procedure.

11. OPTIONS(SHORTCALL) may be specified on an internal PROCEDURE statement to cause it to be accessed by JSY instead of PCL; the compiler diagnoses incorrect use of this feature.

12. OPTIONS(SHORTCALL[(integer_constant)]) may be specified in a declaration with the ENTRY attribute, specifying that the external routine is to accessed by JSXB instead of PCL. See Appendix A for more details.

13. External references are resolved on the basis of the first eight characters of the name.

14. POINTER, BINARY, and BIT return values are passed in the A or L register in order to conform to the FORTRAN convention.

15. LABEL variables are stored with the first two words (i.e., the pointer to the execution address) interchanged for compatibility with FORTRAN routines expecting an alternate-return argument.

16. Arguments passed to a CHARACTER(*)NONVARYING parameter are actually passed as two arguments, the first being the string contents and the second the string length. This conforms with a popular FORTRAN calling sequence. (No additional information is passed to a CHARACTER(*)VARYING parameter, since the string is self-defining with respect to length.)

Non-Standard Extensions in PL/P

The following extensions to ANSI PL/I are available in PL/P (for more information regarding the functions of these features, see Appendix A):

1. Uppercase and lowercase are completely interchangeable, except within character string constants.

2. "$INSERT treename", if it begins in column 1, is not followed by any other text on the line, and the "I" is capitalized will be logically replaced in the compiler input by the contents of the file referenced by "treename".

3. The "$" character is legal in identifiers, except that it may not be the first character of the identifier name.

4. Argument-parameter type and number checking is disabled by declaring the entry name with no parameter list.

5. The SELECT statement

6. The LEAVE statement

7. The %NOLIST and %LIST statements

8. The %REPLACE statement

9. The UNTIL option of the DO statement

10. The CALL statement has been extended to recognize the following "builtin subroutines": INHIBIT, ENABLE, WAIT, NOTIFYB, NOTIFYE.

11. The SEARCH, LINKPTR, STACKPTR, BASEPTR, ADDREL, PTR (nonstandard definition), REL, RING, SEGNO, BASEREL, STACKBASE, CSTORE, REGFILE, ADDQT, ADDQB, REMQT, REMQB, TESTQ, and TRIM builtin functions

12. The REGFILE and REGISTERS pseudovariables

Bibliography

1. The PL/I Programming Language, Paul Abrahams--CCO-3077-151, Courant Mathematics and Computing Laboratory, New York University, March 1978

2. Multics PL/I Reference Manual--AM83 Rev. 0, Honeywell Information Systems, June 1976

Appendix A

PL/P Course Syllabus

I. Overview of differences between FORTRAN and PL/I

A. No distinction made between upper and lower case (PL/P only)

B. Freer line structure

1. No column dependencies

a. Up to 256 columns (PL/P)

b. No continuation column--statements are completely line-independent

c. No comment column--all comments begin with "/*" and must end with "*/"--may begin anywhere and run for any number of lines

d. No label field--labels are identifiers, not numbers, and may be in any column; followed by ":" to indicate label

2. Semicolon required to end statements--allows line independence

3. Spaces are required to separate identifiers (keywords, variables, labels, etc.) and numbers from themselves and each other when not separated by other non-alphanumeric characters; may be used freely anywhere except within lexical items.

4. Blank lines allowed

C. Arbitrary complexity of expressions, even in subscripts

D. Call by value instead of call by reference for constants and user-selected variables

E. Identifier names are more flexible

F. Block-structured--allows internal subroutines

* G. Multiple entry points, with same or different calling sequences

H. Storage classes available on a per-variable basis for things like -DYNM, COMMON, and runtime storage management (no parallel in FORTRAN)

I. Extended DO-loop functionality

1. Logical conditions as well as iterative counts

2. Does not execute once unconditionally

3. Able to count backwards

4. Multiple specifications for loops

5. Able to use non-integer data for index

J. IF-THEN-ELSE; nestable; usable with compound statements

K. Wide range of data types

1. More flexible arithmetic types

2. Additional types for bits, fixed- and varying-length character strings, pointers, labels, etc.

L. Able to group data items logically in storage, even if different data types

M. Able to count characters in calls for the programmer

N. Able to validate arguments in calls

O. Arrays are stored in row-major order: rightmost subscript varies most rapidly

P. Keywords are not reserved and may be used as identifiers

Q. Dynamic handling of errors and other exceptions (not in PL/P)

R. Flexible I/O (not in PL/P)

II. Identifiers

A. Naming restrictions

1. Must begin with an alphabetic character; may contain digits, "$", and "_" ("_" is good for clarity by separating words in a name)

2. Internal names may be up to 32 characters long; external names may be up to 8 characters (PL/P and SEG restrictions)

B. Except for label constants and builtin functions (see below), must be explicitly declared (PL/P restriction, but good programming practice)

C. The DECLARE statement

1. Replaces the type statements (REAL, LOGICAL, etc.) of FORTRAN; the data type of an identifier is given by keywords following it, rather than by different type

statements

2. May appear anywhere in the procecure, subject to scope rules (see below)

3. Syntax: DCL <var_spec>[, <var_spec>[, ...]];

   a. Typical syntax of <var_spec>:
   [level] variable_name [(<dimensions>)] <attributes>

      i. "level" is used only to declare structures (see below)

      ii. <dimensions>, if present, specifies that "variable_name" is an array. Syntax:
      [lbound1:] hbound1 [, [lbound2:] hbound2 [, ...]]
      If "lboundn" is omitted, default is 1. Note that arrays need not start at index=1. PL/P restriction: all bounds must be decimal integer constants.

      iii. <attributes> specify storage class, scope, initial value, and data type, size, precision, and alignment (see below). If multiple words are used for <attributes>, order is not important.

   b. <var_spec> may be "factored" by use of parentheses so that "level"s, <attributes>, and <dimensions> apply to more than identifier.

D. Attributes

   1. Storage classes

      a. AUTOMATIC

         i. Default storage class--most frequently used

         ii. Like -DYNM option in FORTRAN--storage is in stack, so recursive invocation gives a new "generation" of storage, leaving values in previous but still active invocations untouched

         iii. Does not necessarily retain value from one call to the next

         iv. Used for most variables unless there is a reason to use something else.

      b. STATIC

         i. Like SAVE in FORTRAN

"*" => planned but not yet available in PL/P

ii. All recursive invocations reference the same generation of storage

iii. Retains value across invocations

iv. Used for communication among recursive invocations of a routine or for value retention from one call to the next

v. Storage is in linkage section

c. BASED

i. No storage allocated for BASED variables at compile or load time; either references storage allocated for another variable (see ADDR builtin function below) or storage explicitly allocated by the program at runtime (see ALLOCATE statement below)

ii. Must be referenced through a pointer (i.e., "ptr -> based_variable")

iii. When referencing based storage, the pointer provides the address; the based variable provides only the data_type template to be used in accessing the storage.

iv. Used for run-time storage management (see ALLOCATE and FREE statements below), for linked lists of data, for ease of passing structures as arguments (by passing a pointer to the structure), and for accessing the same storage as different types of data (e.g., treating a word of memory as 2 ASCII characters but being able to access individual bits also)

d. PARAMETER

i. Applies only to the arguments of the current procedure

ii. PL/P supports the address type but not the keyword

e. Storage classes are mutually exclusive

2. Scopes

a. INTERNAL--variable name is known only to the block which declares it and any contained blocks (see "Scope Rules" below); this is the default scope for variables

b. EXTERNAL

i. The storage associated with the variable may be accessed by any other block which also declares the variable name as EXT

ii. Corresponds to FORTRAN named common

iii. Implies STATIC; no other storage class may be given

3. Data types, sizes, precisions

a. FIXED BIN [(precision)]

i. Corresponds to FORTRAN INTEGER

ii. 1 ≤ "precision" ≤ 31 (for PL/P)--number of bits required for the absolute value of the range of integers the variable represents--e.g., FIXED BIN(15) is the same as INTEGER*2. Must be a decimal integer constant

iii. If omitted, default for "precision" is 15--single word integer

iv. Two associated formats for constants: binary constants ("[0 | 1]+B", where "[|]" indicates choice of "0" or "1" and "+" indicates one or more instances of the preceding) or decimal integer constants

v. PL/P allows implicit conversion to and from BIT, where the corresponding bit pattern is defined as that of the absolute value of the binary item with the sign bit truncated; it is then zero-padded or truncated on the right, as necessary to match lengths

b. BIT [(size)]

i. Allows access by named variable to individual bits of memory (instead of shifts, truncates, masks, etc.)

ii. 1 ≤ "size" ≤ 16 (for PL/P)--number of bits in the data item (must be decimal integer constant in PL/P)

iii. BIT(1) may be used in the same way as FORTRAN LOGICAL in IF statements (see below for possible values--not ".TRUE." and ".FALSE."!)

iv. If omitted, default for "size" is 1

v. Constants are of the format

"[(factor)] 'char+'B[radix]"

where:

"factor" is the string replication factor; it must be a decimal integer constant and specifies that the bit constant is actually "factor" concatenated occurrences of the string given

"+" indicates one or more occurrences of "char"

"radix" is the radix factor, a decimal integer constant which is interpreted as the number of bits represented by each "char" and implies which characters are legal in the string. "B1" or omitted implies binary representation; "B2" implies quaternary; "B3" implies octal; and "B4" implies hex.

vi. PL/P allows implicit conversion to FIXED BIN (see FIXED BIN for definition of conversion result)

c. POINTER

i. A pointer contains a memory address (e.g., the result of an EAL instruction). All pointers in PL/P are of the 2-word variety--no bit offsets.

ii. There are no pointer constants; values for pointers are obtained solely through use of builtin functions (see below)

iii. No conversions of any sort are defined for pointers

d. CHAR [(size)] [(VARYING | NONVARYING)]

i. Represents a string of ASCII characters

ii. "size" (must be a decimal integer constant between 1 and 8192 for PL/P) specifies the number of characters the variable can contain. "NONVARYING" implies that character strings assigned to the variable will be blank-padded on the right, if necessary, so that the length will always be exactly "size"; VARYING implies that this blank padding will not occur and that the actual length, which may be anything from 0 to "size", will be kept with the variable (actually, in the first word of storage allocated for it).

iii. "size", if omitted, defaults to 1; if neither VAR nor NONVARYING is specified, the default is NONVARYING.

iv. Character constants are of the form

[(factor)] 'char*'

where

"factor" is the replication factor (see description of bitstring constants above)

"*" indicates zero or more occurrences of "char"

"char" is any valid ASCII character; if a single quote is to be included, it must be doubled

Character constants must not include <newline> in PL/P--i.e., they must not cross line boundaries

v. Varying and nonvarying strings may be assigned to each other; no other implicit conversions are allowed by PL/P for character data

e. LABEL

i. Label variables identify not only a particular executable statement but also a particular invocation of the containing block

ii. Label constants are defined by the label name occurring immediately before the text of the associated statement and separated from it by a colon. A single statement may have multiple label constants

iii. Label constants or variables may be used to interface with FORTRAN routines which expect an alternate-return parameter.

iv. No conversions are defined for labels

f. ENTRY [([<arg_list>])] [RETURNS (<arg_desc>)] [{CONSTANT | VARIABLE}] [OPTIONS(SHORTCALL[(size)])]

i. Defines the identifier to be the name by which a procedure or entry point is to be referenced. If the RETURNS attribute is coded, the identifier must be used only like a FORTRAN function reference (i.e., the procedure name may be used in any

circumstances where a variable of the type described in <arg_desc> could be); otherwise, the identifier name must be used only in CALL statements. If "CONSTANT" is coded, the identifier is the name of an external procedure or entry point. It must not be used of internal procedures: it may not be a member of a structure, nor may it be given a storage class or dimensions. If "VARIABLE" is specified, however, the name simply references a variable whose value is that of a procedure or entry, either internal or external, and these restrictions do not apply. The entry value may be assigned to the entry variable in any of the normal ways (the programmer is responsible for assuring consistency of parameter declarations.) If neither "CONSTANT" nor "VARIABLE" is given, the default is "CONSTANT".

ii. <arg_list> has one <arg_desc> for each argument to be passed in the call, separated by commas. An <arg_desc> is identical in format to <var_spec> (see above), even if a structure (see below), except without the variable name; in addition, attributes giving storage class, scope, and initial values must not be specified. Each <arg_desc> defines the data type expected by the called procedure in the corresponding argument position or returned by the procedure. The supplied arguments must match the specified descriptors in number and in type. An error occurs if the wrong number of arguments is supplied. If data type, precision, size, or alignment mismatch occurs, the argument is converted to the expected type, etc., if possible (an error occurs if not), the result is placed in a compiler-generated temporary, and the call is made using the temporary in place of the supplied argument (call by value).

iii. If only the parentheses are given with no <arg_list>, the corresponding function references/CALL statements must have no arguments. If the parentheses are also omitted, no checking will be performed by the compiler on argument number or type.

iv. Only in ENTRY <arg_list>s, the "size" of a CHAR <arg_desc> may be specified as "*". If <arg_desc> is VARYING, any size varying string or character constant may be supplied as an argument without causing mismatch. If NONVARYING, any size nonvarying string or character constant may be supplied without mismatch and, in addition, the generated code will have 2 arguments in this

position: first, the string, and second, the length of the string in characters. This feature of CHAR(*) NONVARYING is useful when interfacing with many FORTRAN routines which accept string-length argument pairs. Note that mismatch between VAR/NONVARYING causes call by value.

v. If coded with "OPTIONS(SHORTCALL[(size)])", the entry will be accessed via JSX5 rather than PCL, and arguments, if any, will be passed via the L-reg: if 1 arg, the L-reg will point to the argument; if >1, the L-reg will contain the address of a list of 3-word pointers to the arguments. A minimum of 8 words of space for use by the called routine will be left at SB%+20 (decimal); the size may be increased by using the optional "size" argument (which must be a constant).

4. Initial values

a. Syntax: INIT (<value_list>)

b. Causes the variable to get the given value(s) at the time of allocation-- AUTO, when the block is entered; STATIC, at load time; BASED, on explicit allocation. Illegal for PARAMETER.

c. <value_list> is a comma'ed list of values of the same type as the variable being initialized (PL/P restriction: values must be constants). If an array is being initialized, <value_list> must have exactly the same number of elements as the array; if the variable is scalar, the list must have one element. If several consecutive values are identical, they may be replaced by "(count) value", where "count" is the number of values replaced (note ambiguity with string replication factor in case of bit or character values; must specify "(count) (factor) value" to iterate these data types).

d. Must not be specified for structures (see below); individual variables within a structure may be initialized. Allowed by PL/P only for STATIC variables.

5. Alignment

a. Syntax: [(ALIGNED | UNALIGNED)]

b. "ALIGNED" optimizes ease of access; "UNAL" optimizes space used.

c. If omitted, "UNAL" is the default for BIT and CHAR NONVARYING; "ALIGNED" is the default for all else.

d. PL/P restrictions and function

i. No effect for anything other than BIT and CHAR NONVARYING; everything else is always word-aligned

ii. If unaligned, contiguous bitstrings in a structure (see below) will have no breakage between them unless necessary to avoid a bitstring crossing a word boundary; otherwise, each aligned bitstring and the following item will begin on a word boundary. In arrays of bitstrings, each element begins on a word boundary regardless of alignment.

iii. For CHAR data, each character is always byte-aligned, regardless of alignment. If unaligned, no breakage will occur between array or structure elements of type CHAR; otherwise, each aligned element and the following element will begin on a word boundary.

F. Structures

1. Used for grouping logically-related but different-typed data in storage and for linked lists.

2. Members of a structure are declared just like other variables, except that "level" is present and greater than 1.

3. Structure members may themselves be structures, in which case only alignment <attributes> may be specified and the "level" of the succeeding variable must be greater than that of the current member.

4. The top-level variable of a structure must have a "level" of 1. This variable and only this variable in the structure may have scope and storage class specified.

5. If alignment is explicitly specified for a structure, that alignment is the default for all members of the structure unless overriden by a contained structure for its own members; otherwise, alignment for members of a structure is determined as described above.

6. A structure member name must be unique within its immediately-containing structure, but it need not be unique beyond that.

7. To avoid ambiguity, structure members may be "qualified" by preceding the member name by the name of a containing

structure and separating them by a ".". As many containing structure names as are present may be specified using this syntax.

8. LIKE

   a. Syntax: LIKE structure_reference

   b. "structure_reference" is the name of a structure declared earlier in the program and known to the block containing the LIKE declaration (see "Scope Rules" below); it need not be a level-1 structure.

   c. The effect is as if the declarations of all members of "structure_reference" had been copied directly following the variable with the LIKE attribute, except that level numbers are adjusted upward or downward as necessary to be compatible with their position in the new structure.

   d. Restrictions: "structure_reference" must have been declared before its usage in the LIKE specification (PL/P only); "structure_reference" must not contain a LIKE attribute.

9. Any subscripting required in the "chain" of structure references may be specified as if the member referenced itself had all the dimensions required.

10. If a structure is BASED, a reference to a member assumes the pointer contains the address of the base, or top level, of the structure, and the offset of the member within the structure is added to the value of the pointer in evaluating the reference.

11. If a structure is external, only the top level name need match declarations in other blocks; member names are not checked.

* 12. REFER

   a. Syntax: <allocation_length> REFER reference_length

   b. Allows BASED structures to be self-defining in the amount of storage occupied by a generation of the structure; replaces a bound of an array member or the size of a character string member.

   c. <allocation_length> is an expression which is evaluated whenever the structure is allocated (see the ALLOCATE statement below); the value is used to determine the amount of storage to allocate for the structure and after allocation the value is assigned to

"reference_length" in the new generation. If the structure is simply used to overlay already-allocated storage, i.e., it is never the object of an ALLOCATE statement, <allocation_length> is ignored and may even be the constant "0".

d. "reference_length" must be an elementary (i.e., non_structure) scalar (including inherited dimensions) member of the structure; it must precede the member containing the REFER option which references it, and its "level" must be less than or equal to that of the member referencing it. Whenever the structure is referenced in the program, "reference_length" is used as the array bound or string size of the member containing the REFER option.

e. PL/P restrictions: only 1 REFER option per level-1 structure; the member containing the REFER must be the last member of the structure; "reference_length" must be "fixed bin(15)".

13. FORTRAN compatibility

a. An external structure is equivalent to a COMMON block with the name of the top level structure and with the structure members having data-type <attributes> as variables in the block.

b. A structure whose members are unaligned bit strings allows named access to individual bits, which in FORTRAN is accomplished by means of masks, shifts, and truncates.

F. Scope rules

1. Blocks are delimited by BEGIN-END and PROC-END pairs (see below); these blocks are indistinguishable in their effects on identifier scopes.

2. A internal variable is <u>known</u> to the block in which the variable is declared and in all contained blocks, <u>unless</u> the contained block or an intervening parent block has a declaration of that variable.

3. Variables may be referenced only in blocks in which they are known.

4. The scope of an internal procedure name is the same as that of an internal variable declared in the block containing the procedure.

5. The scope of an ENTRY name is the same as that of the procedure immediately containing the ENTRY statement.

III. Expressions

A. Types of expressions

1. Operators--perform an operation on argument(s), yielding the same type with possibly different precision or size; prefix and infix notation.

2. Builtin functions--like FORTRAN intrinsics, except name is not reserved. Must not be declared in PL/P.

3. Comparisons--result in BIT(1), value depending on whether the comparison was true or false; may be used anyplace a BIT(1) value is needed.

B. Arithmetic expressions

1. Operators--standard FORTRAN types: "+", "-", "*", "/", "**" (last 2 not supported by PL/P; for division, see next)

2. Builtins

   a. MOD (number, modulus) returns (fixed bin)

   b. DIVIDE (dividend, divisor, result_precision) returns (fixed bin)

3. Comparisons: "=", "^=", ">", "^>", ">=", "<", "^<", "<="

C. Character expressions

1. Operators: "||" (concatenation)

2. Builtins

   a. SEARCH (string1, string2) returns (fixed bin)--result is character position within string 1 of the first occurrence of any character in string2, or 0 if none (nonstandard)

   b. BIN (string [,result_precision]) returns (fixed bin)--result is the numeric value of the string when interpreted as the character representation of a decimal integer

   c. CHAR (integer [, result_size_in_chars]) returns (char(*))--result is the character representation of the integer as a decimal number. The size will be no less than that required to represent the most negative value an integer of the precision of the first argument may take, including the minus sign, but may be increased by means of the second argument. If not all

character positions are used in representing the number, leading spaces will be added to complete the size; no space will occur between the "-", if present, and the first digit.

d. AFTER (string1, string2) returns (char(*))--returns the remainder of string1 which follows the first occurrence of string2, or '' (the null string) if none

e. BEFORE (string1, string2) returns (char(*))--returns the portion of string1 which precedes the first occurrence of string2, or the entire string1 if string2 is not found

f. COPY (string1, count) returns (char(*))--result is "count" concatenated occurrences of string1 (PL/P restricts "count" to be a constant)

g. DATE () returns char(6)--result is current date in format YYMMDD

h. INDEX (string1, string2) returns (fixed bin)--result is character position of first occurrence of string2 in string1, or 0 if none

i. LENGTH (string) returns (fixed bin)--result is current length of string (a constant for CHAR NONVARYING)

j. REVERSE (string) returns (char(*))--value is result of interchanging first and last characters, second and next-to-last, etc.

k. SUBSTR (string, start_position [, result_length]) returns (char(*))--result is substring of "string" starting at "start_position" and running for "result_length" characters, if specified, or until the end of "string" if not

l. TIME () returns (char(9))--result is current time in format HHMMSSMMM

m. TRANSLATE (string1, output_chars [, input_chars]) returns (char(*))--result is computed according to the following algorithm:

```
for each character of string1:
  posn = index (input_chars, str1_char)
  if posn = 0
      then result_char = str1_char
  else result_char = substr (output_chars, posn, 1)
```

If "input_chars" is omitted, the ASCII collating sequence is used.

n. VERIFY (string1, string2) returns (fixed bin)--result is character position of first character of string1 which is not in string2, or 0 if none

o. TRIM (string, control_bits [, trim_char]) returns (char(*))--result is what is left of "string" after removing leading and/or trailing "trim_char"s from it. "control_bits" is a bitstring of length 2 whose first bit specifies removal of leading characters and whose second bit implies removal of trailing characters. If "trim_char" is omitted, ASCII "space" is the default. (nonstandard)

3. Comparisons: same as arithmetic; ordinal comparisons are based on ASCII collating sequence with shorter item blank-padded on the right to the length of the longer.

D. Pointer expressions

1. There are no pointer operators.

2. Builtins

a. Standard

i. NULL() returns (ptr)--result is a pointer which will generate a fault if used.

ii. ADDR (variable) returns (ptr)--result is a pointer with the value of the address of the variable.

b. Nonstandard (depend on Prime representation of pointers)

i. SEGNO (ptr1) returns (bit(12))--result is segment number of given pointer

ii. RING (ptr2) returns (bit(2))--result is ring number of given pointer

iii. REL (ptr1) returns (fixed bin(15))--result is word offset of given pointer

iv. PTR (segno, wordno [, ringno]) returns (ptr)--result is pointer whose value is given by the arguments; segno is bit(12), wordno is fixed bin(15), and ring is bit(2)

v. ADDREL (ptr1, rel_offset) returns (ptr)--result is a pointer whose segment number is that of "ptr1" and whose word number is that of "ptr1" plus "rel_offset"

vi. BASEPTR (ptr1) returns (ptr)--result is a pointer whose segment number is that of "ptr1" and whose word number is 0

vii. BASEREL (ptr1, word_offset) returns (ptr)--result is a pointer whose segment number is that of "ptr1" and whose word number is given by "word_offset"

viii. STACKPTR () returns (ptr)--result is a pointer whose value is the address of the current stack frame

ix. STACKBASE () returns (ptr)--result is a pointer whose value is the stack base

x. LINKPTR () returns (ptr)--result is a pointer whose value is LB%+256

3. Comparisons--only tests for equality ("=", "^=") are allowed

E. Bit expressions

1. Operators--'^' (complement), '&' (logical AND), '|' (logical OR), '||' (concatenation; not yet supported by PL/P)

2. Builtins

a. SUBSTR--same as for CHARACTER except first argument is a bitstring and positions and lengths refer to bits instead of characters. PL/P restriction: length must be computable at compile time.

b. BIN--same as for CHARACTER, except first argument is a bitstring and the result is derived by standard bit to binary conversion described under FIXED BIN above.

c. BIT (integer [, result_size_in_bits]) returns (bit(*))--result is derived by standard binary to bit conversion described above.

* d. SOME (bitstring) returns (bit(1))--result is '1'B if any bit in "bitstring" is '1'B, '0'B otherwise.

* e. EVERY (bitstring) returns (bit(1))--result is '1'B if every bit in "bitstring" is '1'B, '0'B otherwise.

3. Comparisons--same as character

4. Comparisons and infix operators are defined to work only on bitstrings of the same length; the shorter operand is right padded with '0'B to the length of the longer.

F. Operating system builtin functions (nonstandard)

1. CSTORE--generates "STAC" or "STLC", depending on size of operands. First arg is target, second is old value, third is new value. Result = '1'B if operation failed, '0'B otherwise.

2. REGFILE (offset) returns(fixed bin(31))--generates "LDLR offset"

3. ADDQT (queue, item) returns(bit(1))--adds "item" to the top of "queue" via "ATQ"; result = '1'B if successful, '0'B if queue is full.

4. ADDQB--same as ADDQT, with "ABQ"

5. REMQT (queue, item) returns(bit(1))--removes top item of "queue", storing it in "item", via "RTQ"; result = '1'B if successful, '0'B if queue is empty.

6. REMQB--same as REMQT with "RBQ"

7. TESTQ (queue) returns (fixed bin)--result is the number of items in "queue"

IV. Statements

A. proc_name: PROCEDURE [(<parameter_list>)] [RETURNS (<arg_desc>)] [OPTIONS (<option_list>)];

1. "proc_name" is the name used in CALL statements and ENTRY declarations.

2. <parameter_list> is the comma'ed list of variable names by which the procedure will refer to its parameters.

3. RETURNS must be coded iff the procedure is referenced as a function; <arg_desc> has the same syntax as the corresponding item in the ENTRY declaration (see above).

4. Must be the first statement in any compilation; a "MAIN" routine, in the FORTRAN sense, does not exist.

5. Each PROC statement must have a matching END statement, which terminates the procedure.

6. May be used inside another procedure for internal subroutines; see "Scope Rules" above for effect on scopes of identifiers. If normal flow of control reaches an internal PROC statement, the entire internal procedure will be skipped. An internal subroutine may not be called under any circumstances unless the immediately containing block is active, i.e., has been invoked and has not exited.

7. In PL/P, all procedures are recursible.

8. <option_list> is a comma'ed list of options.

   a. NOCOPY--may be specified for external PROCs to suppress call-by-value for constants.

   b. GATE--causes ring weakening to occur for all BASED, PARAMETER, and EXTERNAL pointer assignments.

   c. SAVE (ref)--causes an RSAV into "ref" before any other code in the procedure is executed.

   d. SHORTCALL--may be specified for internal PROCs to speed up calling and returning by using JSY instead of PCL.

* B. entry_name: ENTRY [(<parameter_list>)] [RETURNS (<arg_desc>)] [OPTIONS (<option_list>)];

1. Defines another entry point to the procedure containing the ENTRY statement besides the main one (i.e., the PROC statement).

2. Has no effect on scopes and takes no matching END statement; otherwise, function and syntax is exactly the same as a PROC statement.

3. "Invisible" to normal flow of control.

4. When an ENTRY statement is called, execution begins immediately following the ENTRY statement instead of at the beginning of the containing procedure.

C. IF <expression> THEN [<executable_statement>]; [ELSE [<executable_statement>];]

1. <expression> must be capable of being converted to BIT; if any bit in the resulting string is non-zero, the condition is true.

2. IF statements may be nested; ELSE parts match on a LIFO basis.

D. BEGIN;

1. Takes a matching END statement; the intervening statements are known as a <u>begin block</u>.

2. An entire begin block is considered as an executable statement for the purposes of THEN and ELSE (see above), allowing multiple statements to be controlled by an IF without use of GOTOs.

3. For effect on identifier scopes, see "Scope Rules" above.

E. Two flavors of "DO"

1. All DOs take a matching END statement; intervening statements are known as a <u>do group</u>.

2. Any do group may be used like a begin block as an executable statement after a THEN or ELSE.

3. Simple DO - "DO;" - like BEGIN except without effect on identifier scopes and hence cheaper

4. Complex DOs

a. All complex DOs are iterative, i.e., they loop until some test has been failed. This condition is tested at the <u>beginning</u> of the loop, so that if the test is failed upon entering the group, it will <u>not</u> be executed at all.

b. DO WHILE (<expression>);

i. <expression> is evaluated exactly like the <expression> in an IF statement.

ii. The statements in the do group are repetitively executed until the condition is false at the beginning of an iteration.

c. DO index = <spec_list>;

i. For PL/P, "index" is restricted to be either a fixed bin(15) or ptr.

ii. <spec_list> is a comma'ed list of <spec>s; if more than one <spec> is supplied, the group is executed under control of the first <spec> until the test in that <spec> fails; the group is then executed under control of the second <spec> until that test fails; etc.

iii. <spec> syntax: init_value [{[TO final_value] [BY increment] | REPEAT <expression1>}] [WHILE (<expression2>)]

iv. "init_value" is assigned to "index" before executing the loop the first time. The TO/BY option may only be used if "index" is a fixed bin(15); "increment" may be negative. If this option is chosen and TO is specified, the loop is executed, incrementing the value of "index" by "increment" after each loop, until the value of

(final_value - index) * increment is less than 0. If the TO option is omitted but the BY option is specified, the check at the beginning of each loop is omitted, but incrementation continues as described. If the BY option and "increment" are not specified, the default is 1. If TO and BY are both omitted, the group is executed at most once, subject to the WHILE clause (see below).

vi. If the REPEAT option is specified, at the end of each loop <expression1> is evaluated and the value assigned to "index" for the next iteration. This is particularly useful for traversing linked lists.

vii. If the WHILE option is coded, the loop will be terminated if <expression2> yields a false value when the conditions are being evaluated at the beginning of each iteration. If coded with the TO/BY option, the WHILE clause can only shorten, not extend, the loop duration. NOTE: the WHILE option used in this way applies only to the current <spec>.

d. UNTIL (nonstandard) may be used in place of or in conjunction with WHILE. The syntax is the same, but the sense of the test is inverted and is performed at the end of the loop instead of the beginning.

F. END; - terminates current block/group/procedure. Must have exact number to terminate all outstanding blocks/groups/procedures.

G. CALL name [<arg_list>];

1. <arg_list> must match the parameter list in the ENTRY declaration for "name" in number, data type, alignment, etc. (see ENTRY attribute above).

2. Call by value can be selected for any argument(s) by enclosing the variable name in parentheses (constants and expressions are automatically done as call by value). In addition, this technique can be use to suppress compiler warning messages about argument/parameter mismatch.

3. The following undeclared names ("builtin subroutines") may be CALLed, but instead of invoking a procedure they generate a particular machine instruction. (nonstandard feature)

a. INHIBIT()--generates "INH"

b. ENABLE()--generates "ENB"

c. WAIT(semaphore)--generates "WAIT"

d. NOTIFYB(semaphore)--generates "NFYB"

e. NOTIFYE(semaphore)--generates "NFYE"

H. GO TO label;

1. "label" may be either a label constant or variable; if the label is not in the current block (procedure or begin block), an implicit RETURN is done to the invocation of the block containing the label--i.e., all intervening stack frames are popped by a non-local GOTO.

2. May be abbreviated "GOTO" (no space).

I. RETURN [(<expression>)];

1. <expression> must be provided iff the associated PROC or ENTRY statement was coded with the RETURNS option (see above); this mechanism replaces the FORTRAN practice of assigning the return value to the function name.

2. The RETURN statement with no expression is optional; if the flow of control encounters the END statement of a procedure, an implicit return is done.

J. Assignment statement

1. Syntax: {<variable_reference> | <pseudovariable>) = <expression>;

2. <variable_reference> may be structure or pointer qualified or subscripted.

3. A pseudovariable is like a builtin function except that it receives a value.

a. SUBSTR--the portion of the first argument specified by the second and optional third argument is replaced by the expression on the right hand side of the "=". Valid for BIT and CHAR. Note-- the SUBSTR pseudovariable does <u>not</u> change the length of a CHAR VAR argument.

b. REGISTERS()--no arguments; causes an "RRST" from the value of the right hand side. (nonstandard)

c. REGFILE (offset)--causes "STLR offset" using the value of the right hand side. (nonstandard)

4. Any PL/I data type may be assigned.

* K. ALLOCATE based_name SET (ptr_reference);

1. Causes storage to be allocated from a system-managed pool of free storage, performs value initialization as required by INIT and REFER clauses in the declaration of "based_name", and sets the value of a pointer to the address of the beginning of the storage allocated.

2. The amount of storage allocated is calculated by the compiler from the declaration of "based_name" rather than being specified in the statement.

3. "based_name" is the name of a variable, simple or aggregate, which was declared with the BASED attribute and is known to the block containing the ALLOCATE statement; it must not be pointer-qualified.

4. "ptr_reference" receives the address of the allocated storage.

* L. FREE <based_reference>;

1. Returns to the system-managed pool of free storage a single generation of storage which has been explicitly allocated (i.e., no storage may be freed which has not been the object of an ALLOCATE statement).

2. The amount of storage freed is determined by the compiler from the declaration of the variable in <based_reference> rather than being specified in the statement.

3. The generation of storage to be freed is determined by the address value of a pointer; therefore, <based_reference> must be explicitly pointer-qualified. with a pointer.

* M. SELECT [<select_expression>];

1. A "CASE" statement (not in standard PL/I); terminated by an END statement; may be used as the <executable_statement> of a THEN or ELSE clause.

2. The cases are specified as

    WHEN <expression_list> <executable_statement>;

    where <expression_list> is a comma'ed list of <expression>s; the last case may be

    OTHERWISE <executable_statement>;

3. If <select_expression> is specified, the <executable_statement> of the first WHEN clause, any of whose <expression>s compares equal to <select_expression>.

is executed; if <select_expression> is omitted, WHEN <expression>s are evaluated like the <expression> in an IF statement, and the <executable_statement> of the first WHEN clause, any of whose conditions is true, is executed. If no WHEN clause is selected, the <executable_statement> of the OTHERWISE clause, if any, is executed. After the selected <executable_statement>, if any, is executed, control is transferred to the END statement associated with the SELECT; i.e., under no circumstances will more than one case be selected.

4. The <executable_statement>s may be DO or SELECT groups or BEGIN blocks, as well as simple statements.

5. FORTRAN compatibility: the SELECT statement can be used to simulate a computed-GOTO and will generate equally good code when used this way.

N. $INSERT filename--just like FORTRAN (nonstandard)

O. LEAVE [label]; (nonstandard)

1. Causes control to be transferred to the statement following the END of the selected DO-group--the innermost, if "label" is omitted, or the one whose "DO" is labelled with "label", if specified.

2. "label" must be a label constant; control cannot leave a BEGIN or PROC block.

P. %NOLIST; and %LIST; (nonstandard)

1. %NOLIST turns off listing generation for subsequent lines.

2. %LIST restores the listing, if one is being generated. As many %LIST statements must be coded as needed to match the unterminated %NOLIST statements for listing to resume.

Q. %REPLACE id BY lexeme [, id BY lexeme [, ...]]; (nonstandard)

1. Subsequent references to "id" are treated as if "lexeme" had appeared in the source at that spot instead.

2. "lexeme" is a single lexical item (e.g., identifier, string constant, decimal constant, etc.).

3. "id" may not be replaced in a subsequent %REPLACE statement.