ESD-TR-70-441

TRI FILE COPY

ESD ACCESSION LIST
TRI Call No. 73069
Copy No. 1 of 3 cys.

PRELIMINARY PPL USER'S MANUAL

E. A. Taft

ESD RECORD COPY
RETURN TO
SCIENTIFIC & TECHNICAL INFORMATION DIVISION

October 1970

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
HQ ELECTRONIC SYSTEMS DIVISION (AFSC)
L. G. Hanscom Field, Bedford, Massachusetts 01730

This document has been approved for public release and sale; its distribution is unlimited.

(Prepared under Contract No. F19628-68-C-0101 by Harvard University, Cambridge, Massachusetts.)

AD723643

LEGAL NOTICE

When U. S. Government drawings, specifications or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

OTHER NOTICES

Do not return this copy. Retain or destroy.

ESD-TR-70-441

PRELIMINARY PPL USER'S MANUAL

E. A. Taft

October 1970

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
HQ ELECTRONIC SYSTEMS DIVISION (AFSC)
L. G. Hanscom Field, Bedford, Massachusetts 01730

This document has been approved for public release and sale; its distribution is unlimited.

(Prepared under Contract No. F19628-68-C-0101 by Harvard University, Cambridge, Massachusetts.)

FOREWORD

This report was prepared in support of Project 2801, Task 280102 by Harvard University, Cambridge, Massachusetts under Contract F19628-68-C-0101, monitored by Dr. John B. Goodenough, ESD/MCDS, and was submitted 30 November 1970.

This technical report has been reviewed and is approved.

Sylvia R. Mayer

SYLVIA R. MAYER
Project Officer

EDMUND P. GAINES, JR., Colonel, USAF
Director, Systems Design & Development
Deputy for Command & Management Systems

ABSTRACT

PPL is an interactive, extensible programming language incorporating data definition facilities and operator definition facilities. This manual introduces the various features of the language and serves as a practical reference document for its use. The manual is indexed and several appendices detail parts of the language in tabular form.

PPL runs under the standard 10/50 monitor series of the Digital Equipment Corporation's PDP-10 and also under the TENEX monitor system of the Bolt, Beranek and Newman Corporation. A library of PPL extensions is available containing extensions for doing matrix, rational, formula, polynomial, complex, and vector arithmetics and, in addition, for manipulating lists, strings, and trees. The language is useful for manipulating data in a wide variety of application areas. Being interpretive and conversational, however, it is more well adapted toward personal and educational uses of computers using interactive terminals than it is for doing large institutional computing applications requiring long running times or voluminous input-output.

TABLE OF CONTENTS

Abstract		iii
Table of Contents		v
Section I.	Introduction	1
Section II.	Getting Started with PPL	2
Section III.	Using PPL as a Desk Calculator	3
Section IV.	Variables	6
Section V.	Other Built-In Data Types	7
Section VI.	Function Calls	9
Section VII.	User-Defined Functions	11
Section VIII.	Errors and User Suspension	15
Section IX.	Text Editing	18
Section X.	More on Numbers	24
Section XI.	Definition of Operators	27
Section XII.	Data Definitions	29
Section XIII.	Miscellaneous Important Information	40
	A. Copying and Sharing	40
	B. $Identifiers	42
	C. Erasing Definitions	42
	D. Read and Write	43
Section XIV.	Advanced Debugging Facilities	44
	Appendix A - Table of Built-In Functions	46
	Appendix B - Automatic Type Conversion	49
	Appendix C - Summary of Editing Commands	51
	Appendix D - Summary of Data Definitions	52

Appendix E - ASCII Character Codes..... 53

Appendix F - System Errors..... 54

Historical Note and References..... 56

Implementation Note..... 57

Index to PPL User's Manual 59

SECTION I

Introduction

PPL is a conversational, extensible language oriented toward personal rather than institutional uses of computers. For example, in PPL one can write a program to differentiate formulas in twenty minutes, but a PPL program to do large reactor core calculations or to compute payrolls would be impractical even though it is possible. PPL has been designed to promote rapid composition, modification and debugging of programs over a wide variety of application areas. To achieve this property, the mechanical efficiency of the language has been consciously sacrificed, in certain cases, in favor of flexibility and generality.

The version of PPL described in this manual represents the first of several phases of development. In a nutshell, this first version consists of a typeless conversational language similar to Iverson's APL in which the Iversonian mechanics of arrays have been subtracted out, and to which powerful data definition facilities as well as operator definition facilities have been added. PPL's conversational mechanics include the abilities to trace running programs, to interrupt running programs, to set and remove breakpoints, to write programs that converse with users, and to edit the text of programs. Running time is proportional to the demand for computation, and, in particular, trivial requests are satisfied rapidly. Using the data definition facilities, it is literally possible within an hour to write programs that perform useful operations on data such as: matrices, complex numbers, vectors, formulas, rational numbers or polynomials, or on strings, lists or trees. Defined operations may be associated with unary and binary operators of the user's own choosing, including redefining the meaning of operators given in the initial state of the language.

Additional development of PPL is contemplated in three areas: (1) control structure definition facilities [including concurrent processes, continuously evaluating expressions, the creation of indivisible processes, and coroutines, among other things], (2) syntax extension facilities [including syntax macros and tree mappings], and (3) a fully compatible compiler [the present version of PPL being interpretive].

SECTION II

Getting Started with PPL

Assuming one has somehow gained access to the Harvard PDP-10, PPL is started simply by typing, in response to the monitor's period, the command

.R PPL

PPL will respond with a title line identifying the version number and date, and will then move out eight spaces from the left margin and stop. This is PPL's signal that it is expecting input from the user.

A command may now be typed to PPL. The command is not interpreted or executed until a carriage return is typed. PPL contains text editing mechanisms that allow the user to correct errors in the command being typed. Editing is accomplished by the use of several control keys, two of which are as follows.

RUBOUT: Pressing the rubout key (sometimes labelled 'DEL') causes the most recently typed character to be erased. The erased character is typed back or 'echoed' by PPL. Further rubouts will erase preceding characters on the line.

Control-U: To erase the entire line being typed in, press down the control key (labelled 'CONT' or 'CTRL') and simultaneously type U. PPL will respond by typing '↑U' and moving eight spaces from the left margin on the next line. Another command may now be begun.

SECTION III

Using PPL as a Desk Calculator

One of PPL's many features is that it is conversational and interactive. Any PPL command may be typed in for immediate execution.

The value of a numerical expression may be obtained by simply typing it in and pressing return. For example:

```
        3+2
5
        25*(46+2)
1200
        3.14159*(2.5↑2)
19.63494
```

It may be seen above that user typein always begins eight spaces from the left margin. PPL typeout in response to a user command, however, always begins at the left margin.

PPL recognizes the following arithmetic operations:

+	addition
-	subtraction
*	multiplication
/	division
↑	raising to a power

In addition, PPL provides two unary arithmetic operations:

+	unary plus
-	unary minus

Left and right parentheses may be used freely to indicate precedence of operations. Expressions within parentheses are evaluated by PPL before expressions outside. It is recommended that beginners in PPL make liberal use of parentheses in constructing expressions. This is because PPL's default assumptions, used in the absence of explicit grouping information, are somewhat different from both normal algebraic conventions and most other programming languages.

PPL's precedence rule is very simple: every operator takes as its right operand the value of the expression up to the next right parenthesis, right bracket, comma, or end of line. (This scheme is used in APL and is called 'Iversonian precedence' after its designer). A few examples will suffice to demonstrate PPL's precedence.

```
        3*3*3+5*5*5                 ... example 1
1152
        3*(3*(3+(5*5*5)))
1152
        (3*3*3)+(5*5*5)
152

        10.0/3+2                    ... example 2
2.
        10.0/(3+2)
2.
        (10.0/3)+2
5.333333
```

In each example above, the first expression typed in contains no parentheses. The second indicates the assumptions made by PPL in evaluating such an expression. The third shows the most common assumptions made in other programming languages.

Numbers entered in PPL may be integer, real, or double-precision real (hereafter referred to as INT, REAL, and DBL). An INT is a sequence of decimal digits not containing (or immediately preceded or followed by) a decimal point (period). A REAL is a sequence of decimal digits either (a) containing, immediately preceded by, or immediately followed by a decimal point, or (b) immediately followed by the letter E, an optional sign, and a one or two digit decimal exponent. The magnitude must contain eight or less significant digits; otherwise it is a DBL, which may also be explicitly indicated by using D instead of E to precede the decimal exponent.

Examples of INTs:

31 0 987654321 000130

Examples of REALs:

3.14159 .52 0. 1.347E28 52E-7

Examples of DBLs:

1.4D0 9.327863415 83.5216D27 1D-10

PPL arithmetic operations are an example of PPL's 'polymorphic' capabilities. REALs, INTs, and DBLs may be freely mixed in arithmetic expressions; the necessary type conversions are all handled automatically. When confronted with an arithmetic operation involving operands of different type, the 'weaker' operand is converted to the type of the 'stronger', with strength defined on the scale INT<REAL<DBL. This policy requires little attention on the part of the user except in a

single case: that of division of an INT by another INT. In this case it should be remembered that the result is also an INT and will therefore be truncated.

The following are a few examples of polymorphic arithmetic operations:

```
        3/2
1
        3/2.
1.5
        3./2
1.5
        3+4-.1
6.9
```

SECTION IV

Variables

Values in PPL may be assinged to variables. A variable is represented by an 'identifier', which is a string of letters, digits, and periods, beginning with a letter, not ending with a period, and containing no two adjacent periods. The following are legal identifiers:

A W3 TH1.3V SUM.OF.9.DIGITS

Assignment is performed by the '←' operator (left-arrow on most terminals, but underline on some). The left operand must be a variable name (identifier), and the right operand a value of some sort. Once so defined, variables may be used in PPL expressions exactly like numbers or other expressions.

```
      N←5.4
      N
5.4
      Q3←(N↑3)+54
      Q3/7
30.20914
      N←N-2.1
      N
3.3
```

As the last example above indicates, it is quite legal to assign a new value to an already-defined variable. The new value need not be of the same type as the old. In PPL's polymorphic handling of data, 'type' is an attribute, not of variables, but rather of values. A variable is capable of holding any type of quantity; what might be considered to be the variable's type is actually the type of value stored in it at any given time. Thus, variable type declarations, required in many programming languages, are completely unnecessary in PPL. Further examples of assignment:

```
      N←3
      (N+1)/2
2
      N←3.35E+8
      (N+1)/2
167500000.
      PI←3.1415927
      PI*N↑2
3.525652E+17
```

SECTION V

Other Built-In Data Types

Besides the types INT, REAL, and DBL already presented, PPL has three other built-in data types, as follows:

A Boolean datum (BOOL) has as its value either TRUE or FALSE. BOOL-type values are generated by logical and comparison operations (see below). For convenience, the identifiers TRUE and FALSE are initially defined as variables containing these values.

A character (CHAR) is a datum consisting of a single ASCII character. A constant of type CHAR is typed in as a single quote followed by the desired character. For example:

```
        C← 'Q
        C
Q
```

A STRING is an arbitrary-length sequence of ASCII characters. To type in a string, surround it with double quotes. If it is desired to include a double quote within a string, it should be entered by typing two adjacent double quotes.

```
        "I AM A CHARACTER STRING"
I AM A CHARACTER STRING
        QQ←"HE SAID ""FOO"".  "
        QQ
HE SAID "FOO".
```

PPL has a number of built-in operators for generating and manipulating data of type BOOL. These may be divided into two groups: comparison operators and logical operators.

Comparison operators operate on two arithmetic (INT, REAL, or DBL) quantities and produce a result which is a BOOL. In the table below, the result is TRUE if the condition holds, otherwise it is FALSE.

A = B	A equal to B
A < B	A less than B
A <= B	A less than or equal to B
A > B	A greater than B
A >= B	A greater than or equal to B
A # B	A not equal to B

Logical operators operate on values of type BOOL and produce a result of the same type. The result is TRUE if the condition below holds, otherwise FALSE:

A & B	the 'AND' operation - A and B both TRUE.
A ! B	the 'OR' operation - either A or B (or both) TRUE.
- A	the 'NOT' operation - A not TRUE.

Note that the meaning of the '-' operator depends on its operand. When it operates on a datum of type INT, REAL, or DBL, it indicates negation, whereas when it operates on a BOOL it indicates logical complementation.

SECTION VI

Function Calls

All data operations examined thus far have been expressed through the use of binary and unary operators such as '+' and '-'. Each operator invokes a process that operates on one or two operands or arguments, and returns a single result which may itself be used as an operand within a larger expression. PPL also has a more general function calling mechanism capable of invoking processes that take any number of arguments (or none at all); indeed, it turns out that unary and binary operations are handled by PPL as special cases of function calls.

An explicit function call is made by giving its name (an identifier) followed by a list of arguments contained within a pair of parentheses. If there is more than one argument, the arguments are separated by commas. If the function takes no arguments, the parentheses may be omitted except in certain cases where a pair of parentheses with nothing between them is necessary to relieve ambiguity between the name of a function and the value obtained by calling it.

As an example, consider the system function 'ADD', which is implicitly invoked by the binary operator '+' but which may also be called explicitly:

```
        ADD(53, 27)
80
        3.4+(ADD(ADD(1, 5), 7)-8.5)
7.9
```

All the built-in operations so far presented are similarly associated with internal functions accessible by using the proper identifier, according to the following table:

A + B	ADD(A, B)
A - B	SUB(A, B)
A * B	MUL(A, B)
A / B	DIV(A, B)
+ A	PLUS(A)
- A	MINUS(A)
A = B	EQ(A, B)
A < B	LESS(A, B)
A <= B	LESSEQ(A, B)
A > B	GR(A, B)
A >= B	GREQ(A, B)
A # B	NOTEQ(A, B)
A & B	AND(A, B)
A ! B	OR(A, B)
A ← B	ASSIGN(A, B)

Note that the assignment operation is performed by a system function; that is, the statement 'ASSIGN(Q, 3)' is exactly the same as 'Q←3'. Though the primary operation of this function (the assignment) is done internally, the function does in fact return a value which is the value assigned. Thus, assignments may be embedded within expressions.

```
        INT(3.1+(I←9.7)/2)
7
        I
9.7
        A←B←C←1.0
        B
1.
```

The user might wonder at this point what prevents the immediate printout of the value returned by the system function 'ASSIGN' in a normal assignment expression such as 'X← 3'. PPL treats this case specially, according to the following rule: if the final operation performed in a statement is an assignment, printout is suppressed. The same occurs if the final operation is a call to a non-value-returning function (several such functions exist and will be presented later).

In addition, there are a number of built-in functions not associated with operators. These must be explicitly called when they are needed. As an example, two such built-in functions are:

INT(A): For A a REAL or DBL, has as its value the result of converting A to an INT.

REAL(A): Converts the value of A to a REAL and returns it.

A list of all PPL's built-in functions is given in Appendix A.

SECTION VII

User-Defined Functions

PPL allows the user to define his own functions in such a way that they may be called in exactly the same manner as the system functions described in the previous section. Function definition is the means by which a set of PPL statements may be stored for later execution.

A user-defined function consists of a function header and a set of numbered statements comprising the body (executable part) of the function. As these statements are typed in, PPL stores them away rather than executing them immediately.

The following example will be used to explain the mechanics of function definition. The defined function, called FACT, takes a single argument N and computes N factorial.

```
      $FACT(N); I
[1]   FACT←1.0
[2]   I←N
[3]   (I<=0)-->%0
[4]   FACT←FACT*I
[5]   I←I-1
[6]   -->%3
[7]   $
```

The first line of the definition begins with a dollar sign. This is a signal to PPL that what is being typed is a definition rather than an executable statement. After the dollar sign is the name of the function (FACT), which may be any legal identifier that has not previously been defined. Within the body of the function, this name (referred to as the 'procedure identifier') is used as a simple variable which represents the value of the function; during execution, the last value assigned to the procedure identifier (by a statement of the form 'FACT←expression') will be the value returned to the caller.

Following the procedure identifier appears the dummy argument list enclosed in parentheses. Each dummy argument is an identifier. If more than one dummy argument is listed, the arguments are separated by commas. If the function takes no arguments, the parentheses are omitted. Dummy arguments are used within the body of the function to refer to the values of the corresponding arguments specified when the function is called.

The last part of the function header is a list of local variables. These are separated from the rest of the header by a semicolon; if more than one local exist they are separated by commas or semicolons. These variables are called 'local' because their values are accessible

only within the body of the function in which they are defined. Thus, within the sample function FACT, the variable I is a local. It is entirely separate from any meaning the identifier I may have either within other functions or in the global environment (in which control resides when PPL is started). Thus, the use of locals is an effective means of avoiding name conflicts between identically-named variables in different functions.

The dummy arguments (sometimes known as formal parameters) may also be treated as locals, in the sense that they are accessible only within the function that uses them. Formal parameters and locals differ in one respect, however. Whereas formals are initialized with the values of the calling arguments when the user function is entered, locals initially have no value. This is true on each function entry no matter how many times the same function is called. The procedure identifier is likewise unassigned on entry.

The function header line is terminated normally, by pressing the return key. PPL now responds by typing '[1]', indicating that it is prepared to accept the first line of the function. PPL statements may be typed in this manner on successive lines, with the line numbers supplied automatically, until a line containing only a dollar sign is typed. This indicates the end of the definition, and PPL responds by skipping a line and moving eight spaces from the left margin.

Now that the function has been defined, it may be called:

```
        FACT(3)
6.
        FACT(10)
362880.
        25+FACT(FACT(4))
6.204484E+23
```

In the first example above, the call of the function FACT takes place as follows. The argument (3) is evaluated and assigned as the value of the formal parameter N within the function being called. Control is then passed to statement [1] of FACT.

Unless otherwise directed, PPL executes statements within the function in numerical order. Two methods of altering this flow are demonstrated in the FACT function, both of which use the --> operator. In statement [6], where it is unary, this operator means 'unconditionally goto', and its execution passes control, in this case, to line 3. Note that the line number is preceded by a percent sign. This means "relocatable line 3", and, as we shall see, comes in very handy when we go to edit the function (discussed in the next section). However, statement [6] could have just as well been written as

```
[6]  -->3.
```

The other transfer mechanism is the binary operator -->, which means 'conditionally goto'. The left operand, which must be an expression with a result of type BOOL, is evaluated, and the 'goto' operation takes place if its value is TRUE. Otherwise, control is not altered and execution continues with the next statement. This operator allows user-defined functions to make decisions about what to do next based on current values of certain variables.

By convention, execution of a user-defined function is terminated whenever an attempt is made to execute a nonexistent statement. This can occur either by PPL's sequencing past the last line of the function or by making an explicit transfer (goto or conditional goto) out of the function. For example, the FACT function exits when, during the execution of statement [3], the value of the boolean expression 'I<=0' is TRUE and control is transferred to statement zero.

When a function terminates or 'returns', the currently-assigned value of the procedure identifier (FACT) is taken as the value of the function and substituted in the calling expression. (If the procedure identifier is never assigned a value during execution, the function is considered to be 'non-value-returning').

Functions may in turn call other functions by the same mechanism just outlined. A function called within another function is considered to be 'nested': execution of the called function is nested within the evaluation of a single expression in one statement of the caller.

In PPL, it is perfectly legal for a function to call itself. This technique is known as 'recursion' and is extremely useful in many applications. In writing recursive functions, it should be borne in mind that local variables belonging to nested calls of the same function are completely independent. Locals within the called function are not visible to the caller or vice versa.

We will now give two examples of recursive functions. The first demonstrates an alternate method for computing N factorial.

```
      $RFACT(N)
[1]   RFACT←1.0
[2]   (N=0)-->%0
[3]   RFACT←N*RFACT(N-1)
[4]   $

      RFACT(4)
24.
      RFACT(17)
3.556874E+14
```

This function computes the factorial by using the fact that N factorial is equal to N times N-1 factorial, where zero factorial is defined as one. Thus, in computing RFACT(4), the RFACT function is actually called (recursively) four times. Each successive call is nested within a single expression (in statement [2]) of the caller. RFACT(4) is computed by finding RFACT(3) and using it in an expression; in turn, RFACT(3) is computed using RFACT(2), and so on. It should be emphasized once again that the values of the procedure identifier RFACT and of the parameter N are completely independent at each level.

For the second example, we will compute Ackerman's function of two arguments. Ackerman's function is a mathematical curiosity whose primary note of interest is that small values of the arguments cause very large amounts of recursion.

```
      §ACK(X, Y)
[1]   (X=0)-->%5
[2]   (Y=0)-->%7
[3]   ACK←ACK(X-1,ACK(X, Y-1))
[4]   -->%0
[5]   ACK← Y+1
[6]   -->%0
[7]   ACK←ACK(X-1,1)
[8]   §

      ACK(3,2)
29
      ACK(1,2)
4
      ACK(2,4)
11
```

The user is warned that attempts to compute Ackerman's function for larger arguments (such as ACK(4,1)) will fail on the Harvard PDP-10 for lack of sufficient memory in which to keep track of all the recursions.

A final note: if a parameterless procedure calls itself recursively, the call must be made by appending an empty pair of parentheses to the name. Otherwise, PPL will assume that what is wanted is the current value of the procedure identifier rather than the value obtained by calling the function again.

SECTION VIII

Errors and User Suspension

When PPL detects one of any number of error conditions, it stops whatever it is doing and prints an error message. These errors belong to one of two classes: syntax errors and execution errors.

A syntax error is detected by PPL immediately after the user has typed in a statement. The error may be some sort of illegal format in an editing command, or it may be an error such as unbalanced parentheses in a statement. In all cases, PPL prints out a message that attempts to pinpoint the source of the error. Whenever a syntax error is detected by PPL, none of the given statement will have been executed. If a line of a function was being typed, PPL will repeat the line number and await a corrected version.

Note that after an error, as well as at any other time, the most recently typed line may be reopened for editing simply by pressing the alt-mode key. The mechanisms for text editing are presented in the next section.

The second class of errors occurs during actual execution of a statement. Such errors as attempts to use unassigned variables in expressions can be detected by PPL only when the statement containing the expression is actually executed. Messages indicating execution errors are preceded either by the words 'EXECUTION ERROR' or by an indication of what system function was being executed when the error was detected. The next line typed indicates where execution was stopped. The message 'STOPPED IN DIRECT STATEMENT' means that the error was detected within a line the user had typed in for immediate execution. However, if the message is 'STOPPED IN LINE' followed by a function name and line number, the error was detected during the execution of a user function. In this case, we say that execution has been 'suspended' within the function in question.

While in this suspended state, the user may issue statements to be immediately executed, just as he could while in the global (top level) environment in which PPL was started. However, such statements are now interpreted in the environment of the suspended function; that is, the procedure identifier, the formals, the locals, and the labels are accessible as if the immediate statement were a line of the function itself.

The user is now allowed to change the values of any variables whose names are accessible in the current environment, and to resume execution at any statement in the function by means of the --> operator. (However, the current implementation of PPL does not allow the user to edit any functions if he is currently suspended within a function. In

order to do editing, the user must first call the RESET function, which is a built-in parameterless procedure, to return control to the global environment.)

Occasionally it is necessary for execution of a program to be prematurely terminated (because it has gone into an infinite loop, for example). To do so, the user should type control-C twice to return to the PDP-10 timesharing monitor. Then the REENTER command should be typed. PPL will now suspend execution at the beginning of the next statement, and will print out the name of the current function and the number of the line it was about to execute. Execution may be resumed if desired by use of a --> operator to that statement.

The following example illustrates some possible errors and an example of user suspension.

```
        §PRIME(N); I      ...FUNCTION TO DETERMINE WHETHER
                                N IS PRIME
  [1]   PRIME←FALSE
  [2]   (N<4)-->%7
  [3]   (N=2*N/2)-->%0
  [4]   I←3
  [5]   (N=I*N/I)-->%0
  [6]   ((I↑2)<=N)-->%5
  [7]   PRIME←TRUE
  [8]   §

        PRIME(5)
TRUE
        PRIME(7)                          [Left off right parenthesis.]
ILLEGAL PHRASE OF LENGTH THREE            [Syntax error comment by PPL.]
        PRIME(7)
TRUE
        PRIME(13)
↑C                                        [Program goes into a loop. ↑C
↑C                                         used to escape to the monitor.]

.REENTER                                  [Return control to PPL.]

PRIME [6]                                 [Suspended at line [6] of PRIME.]

        N                                 [Examining local variables in
13                                         environment of suspension.]
        I
3
        -->%6                             [Resuming execution.]
↑C                                        [Function resumes but is still in
                                           loop.]
↑C                                        [User must suspend it again.]
```

```
.REENTER

PRIME [5]
        I                          [Value hasn't changed - there is a
3                                   bug in the program!]
        RESET                      [Reset to global environment.]
        $PRIME[5.5]                [Edit the function, using text editing
 [5.5]  I←I+2                       facilities explained below.]
 [5.6]  $

        PRIME(13)
TRUE
        PRIME(14)                  [Operation now correct.]
FALSE
```

There is a third case which we might consider a 'suspension', though it really is not. A running program will request input from the user when it encounters a question mark (called the 'demand symbol') in a statement. The expression now typed in by the user will be evaluated and substituted for the demand symbol, and execution will continue.

This feature is useful for talkative programs that require input during their execution. A statement containing a demand symbol should usually be preceded by a statement that prints a message explaining what is being requested from the user. The following is a trivial example of the use of the demand symbol:

```
        $SQUARE
 [1]    ''HI. I FIND SQUARES.''
 [2]    ''GIVE ME A NUMBER, PLEASE.''
 [3]    N← ?
 [4]    ''THE SQUARE OF THAT NUMBER IS''
 [5]    N↑2
 [6]    $

        SQUARE
HI. I FIND SQUARES.
GIVE ME A NUMBER, PLEASE.
        29
THE SQUARE OF THAT NUMBER IS
841
```

There is a final class of error, one that should never be encountered. This is the 'system error'. System errors cause control to return to the PDP-10 time-sharing monitor. It is usually possible to recover from such an error by giving the monitor REENTER command. For a more detailed discussion of system errors, see appendix F.

SECTION IX

Text Editing

One of the most important of PPL's many features is that it is conversational and interactive in all phases of its operation. Once a function has been defined to PPL, there must be a convenient means for editing that function in order to make changes and remove errors.

PPL has an extensive text editing mechanism that may be used any time PPL is requesting input from the user. Basically, what we can do is open an existing function for editing, make changes via insertion, deletion, or updating, and close the function again.

Opening a function for editing consists of typing a dollar sign, the function name, and a line number enclosed in square brackets. PPL responds by typing the line number at the left margin. A new statement may now be typed to replace the old statement on that line.

Pressing carriage return causes the newly-typed line to replace the old, unless the new line contained no characters (in that case, the old line remains). PPL responds by typing the line number of the next line in sequence. This statement is in turn open for replacement if desired.

To delete the current line without replacing it with a new line, control-G should be typed. PPL responds by typing '↑G' and sequencing to the next line.

To edit a line which is not the next in sequence, a line should be typed containing only the number of the line to be edited enclosed in square brackets. This line is interpreted as an editing command and does not replace the current line. PPL responds by typing the number of the new line at the left margin.

A line is inserted between two others by specifying a 'Dewey-decimal' line number containing a decimal point and a one or two digit fractional part. Thus, to insert a new line between lines [2] and [3], one might input a line [2.5]. When return is typed after this line, PPL sequences to line [2.6] automatically. Note that this sequencing is Dewey-decimal, so that [2.9] sequences to [2.10].

The function is closed by typing a line containing only a dollar sign. This again is interpreted only as an editing command,and the dollar sign does not replace the current line.

These operations become clearer by the use of an example. We first define a function MAX, which takes two arguments and returns the maximum value. The function entered below contains several errors

and produces incorrect results:

```
        $MAX(A,B)
[1]     (A<B)-->%3
[2]     MAX←B
[3]     MAX←A
[4]     $

        MAX(3,5)
3
```

We now want to insert a statement between [2] and [3] to return after the assignment MAX←B. We also want to reverse the comparison in statement [1]. The required editing is as follows:

```
        $MAX[2.5]                [Open MAX at statement 2.5]
[2.5]   -->%0                    [Insert a function return]
[2.6]   [1]                      [Go edit statement 1]
  [1]   (A>B)-->%3
  [2]   $                        [Close function]

        MAX(3,5)
5                                [Results are now correct]
        MAX(6,2)
6
```

To find out exactly what our editing has done to the MAX function, we now display it. This is done by typing a question mark followed by the name of the function.

```
        ?MAX
        $MAX(A,B)
[1]     (A>B)-->%4
[2]     MAX←B
[3]     -->%0
[4]     MAX←A
```

The first thing that may be noticed is that the lines have been renumbered using sequential line numbers. This operation is performed automatically upon closing the function. The second item of note is that, in statement [1], the -->%3 has been changed to -->%4, thus continuing to refer to the statement it previously referred to. Whenever PPL renumbers lines, it performs this conversion on all 'relocatable line numbers'. A relocatable line number is a decimal number of one to three digits preceded by a percent sign. During execution, a relocatable line number behaves essentially the same as an integer in all calculations that use it; its primary use, however, is in the range of a --> operator.

It should be noted at this point that all editing operations presented thus far are also available while typing in a function for the first time.

The following example demonstrates some of the possibilities encountered while creating a function MIN to compute the minimum of three arguments.

```
        $MIN(X, Y, Z)
   [1]  (X<Y)-->%4
   [2]  (Z>Y)-->%6
   [3]  MIN←Y
   [4]  (Z<X)-->%7
   [5]  MIN←X
   [6]  [3.5]                    [Go insert a line 3.5]
 [3.5]  -->%0
 [3.6]  [5.5]                    [Go insert a line 5.5]
 [5.5]  -->%0
 [5.6]  [6]                      [Now continue original sequence]
   [6]  (X<Z)-->%5
   [7]  MIN←Z
   [8]  [2]                      [Go correct error in line 2]
   [2]  (Z<Y)-->%6
   [3]  $                        [Close function]

        MIN(3, 5, 7)
3
        MIN(7, 3, 5)
3
        MIN(7, 5, 3)
3

        ?MIN
        $MIN(X, Y, Z)
   [1]  (X<Y)-->%5
   [2]  (Z<Y)-->%8
   [3]  MIN←Y
   [4]  -->%0
   [5]  (Z<X)-->%9
   [6]  MIN←X
   [7]  -->%0
   [8]  (X<Z)-->%6
   [9]  MIN←Z
```

A few more points should be noted before we proceed further with text editing. First, comments may be appended to lines by preceding them with a string of three periods. When PPL sees three periods it considers the rest of the statement to be a comment and ignores it. The comment remains in the text, however, and is printed out when the function is displayed using the question-mark command. The user is warned that the first period should not be adjacent to a digit; otherwise, the period may be construed as a decimal point and the comment as part of the statement. A separation of a single space or tab is sufficient.

Second, a statement may be continued on more than one line. To continue to a new line, the line feed key is used; PPL spaces down one line, moves eight spaces from the left margin, but does not print a line number. There are only two restrictions on the use of continuation lines: first, a single lexeme (number, identifier, multi-character operator, etc.) may not be split between lines. Second, the entire statement may not be more than 500 characters in length. Note that a comment may be continued in the same way. Continuing a line within a string causes a return and a line feed to be included within the string:

```
        "THIS IS A              [Line-feed for continuation]
        SPLIT STRING"
THIS IS A
SPLIT STRING
```

Besides the line-by-line editing already presented, PPL also offers extensive character-by-character editing. The character editing mode is entered by pressing the alt-mode key (sometimes labelled ESC). Depending on when alt-mode is typed, PPL will now take one of three actions. If no function is open and alt-mode is the first character typed on a line, PPL will open the most recently typed line for character editing. If it is not the first character on the line, the current line is terminated and opened for editing. If a function is open at the time, and alt-mode is the first character typed after PPL types the line number, then the previous text of that line is opened for editing if it exists; otherwise, the line most recently typed in by the user is opened.

The selected text is now retyped by PPL on the next line. (If the statement contains continuation lines, only the first line is typed). PPL then positions the typehead below the first character on the line.

The user now has several options. If a printing character, a space, or a tab is typed, it replaces the character immediately above the typehead. If a rubout is typed, it deletes the character above the typehead; a back-slash (\) is echoed.

Characters in the existing line may be passed over without change by the use of any of the following control characters:

Control-N: passes by and echoes the Next character on the line.

Control-E: passes over and echoes every character until the End of the current line (leaving the typehead at the end).

Control-T: passes over and echoes Ten characters.

Control-S: passes over and echoes characters until a Separator character (something other than a letter or digit) is reached.

The following control characters may be used to perform the indicated actions:

Line feed: moves to and retypes the next continuation line if one exists. This line may now be edited.

Alt-mode: retypes the first line of the statement and reopens it for editing.

Control-R: retypes the current line of the statement and reopens it for editing.

Return: closes the entire statement, with changes, and enters it just as if the user had typed it directly. If a function is open, the statement is stored; if not, the statement is immediately executed.

Control-A: This character is used for insertion. PPL responds by spacing down one line and printing an upward arrow. This points to the character on the current line before which characters are to be inserted. All printing characters, as well as space or tab, may now be typed for insertion. In this mode, the rubout key will delete and echo the most recently-inserted character (if all inserted characters have been rubbed out, it will start deleting characters before the insertion in the current line). Insertion mode is terminated by typing control-A or control-R, which will retype the current line. Line feed, alt-mode, return, and control-E retain their normal functions.

One useful rule to remember with regard to control characters is that all characters recognized by PPL cause some visible action. If an illegal control character is typed, it is simply ignored.

Because of the difficulty of representing the use of non-printing control characters on the printed page, no examples will be presented of character-by-character editing. It is hoped that the preceding description has been clear enough to enable the user to become familiar with the control characters on his own.

Some further notes: It is possible to edit the header of the currently open function. This is done by making a request to edit line zero. It is illegal, however, to change the name of the function by this means.

PPL has a provision for labelled statements. A label is an identifier immediately followed by a colon, and must appear first in a statement. The label name is considered to be a local and has all the same properties. However, when the function is called, the label

variable is initialized with the integer line number of the statement in which it was defined. This allows an alternate method of ensuring that --> statements refer to the proper lines even after repeated editing and renumbering of the function. The label variable may even be reassigned during execution of a function; however, this affects only the call of the function and does not change the position of the label within the function definition. The following is an example of the use of labels:

```
      §SSUM(N,P)
[1]   SSUM←0
[2]   AGAIN: SSUM←SSUM+N↑P
[3]   ((N←N-1)>0)-->AGAIN
```

If no function is currently open, the text of a single line may still be displayed. This is done using the question-mark command; the line number desired should be given in square brackets after the name. For example:

```
      ?MIN[5]
[5]   (Z<X)-->%9
```

SECTION X

More on Numbers

A more formal presentation of the text representation of numbers will now be presented. In particular, it is necessary to become familiar with the lexical contexts within which numeric constants are recognized.

In scanning from left to right in a statement, PPL will recognize a number (INT, REAL, or DBL) when it sees one of the following character sequences:

a) Any decimal digit which is not part of an identifier. This will be the first digit of an INT, REAL, or DBL.

b) A period followed by a digit. This indicates the beginning of a REAL or DBL.

c) A number sign (#) followed by a digit. This indicates the start of an octal (base 8) number, which is simply another lexical representation of an INT. There may be one to twelve octal digits (0 through 7) following the '#'. The digits 8 and 9 are not allowed.

d) A percent sign (%) followed by a digit. This indicates the start of a relocatable integer, which has the property of being changed whenever a line to which it refers is renumbered. There may be one to three decimal digits after the percent sign.

The user should note carefully that certain text characters may have different meanings when used in other contexts. For example, the character '#' is the relational operator 'not equal to' if it does not precede a digit; on the other hand, it signals the following digit string to be an octal integer if it is followed by a digit. Thus, a statement such as:

```
[3]    (X#3)-->%5
```

will not perform as intended; the '#' takes on the latter meaning. In order to make it mean the 'not equal to' operation, it is sufficient to place a space or tab between the '#' and the digit, thus:

```
[3]    (X# 3)--> %5
```

The arithmetic characteristics of numbers in PPL are as follows:

An INT is capable of representing any integer number from -34359738368 to 34359738367.

A REAL can represent, to approximately eight significant digits' precision, a real number whose magnitude is either zero or between approximately 1.4695E-39 and 1.7014E+38.

A DBL can represent, to approximately sixteen digits of precision, any real number within the limits given for REALs.

When numbers are printed out by PPL, either by the 'implied print' feature (simply typing an expression on a line for evaluation) or by the PRINT system function (see section 12), certain conventions are followed. They are as follows:

An INT, being inherently precise, is printed to its full precision (up to 11 digits) with leading zeroes suppressed. No decimal point is printed.

A REAL or DBL suffers from the problems of conversion to and from the computer's internal floating-point binary notation; thus, few decimal numbers can be represented 'exactly'. Thus, though a REAL has nearly eight digits of precision, and a DBL nearly sixteen, the last digit in either case is not particularly reliable.

By convention, PPL rounds a REAL to seven significant digits before printing, and a DBL to fifteen. It then suppresses leading and trailing zeroes. A decimal point is always printed. Exponential notation is used if it is more compact than the standard decimal notation.

A system function, called FORMAT, is provided to allow user control over numeric output formatting. This is necessary, for example, when attempting to format output into columns. FORMAT takes any number of arguments, which it processes from left to right.

If the argument is of type STRING, it is decoded and used as a 'format specification' for future output by FORMAT. This specification is composed of the following elements:

'E' or 'D': print in exponential format (with 'E' or 'D' and a two-digit exponent). If this part of the specification is omitted, standard (non-exponential) decimal format is assumed.

'nZ': where n is an integer less than 64, indicating the number of zero-suppressed digit positions to be allowed to the left of the decimal point. If this specification is omitted, zero is assumed.

'nD': the number of non-zero-suppressed digit positions to the left of the decimal point.

'.' (period): indicates a decimal point is to be printed (otherwise it is omitted, and the next two specifications may not be included).

'nD': the number of non-zero-suppressed digit positions to the right of the decimal point.

'nZ': the number of zero-suppressed digit positions to the right of the decimal point.

A FORMAT argument of type INT, REAL, or DBL is printed according to the most recently-specified format specification. Blanks are substituted for leading and trailing zeroes if they are in zero-suppressed positions; otherwise, the zeroes are printed out. If the number is negative, the minus sign is printed in the last leading zero-suppressed position (if there is one). If the 'E' or 'D' specification has been included, the number is adjusted by powers of ten so as to exactly fill the leading non-zero-suppressed places.

If the magnitude of a number is too large to fit in all the leading positions, an execution error occurs, with the message 'SIGNIFICANT HIGH-ORDER DIGITS LOST'.

The following examples illustrate some uses of FORMAT. Note that FORMAT does not print a carriage return after printing a number; this must be done explicitly by the user.

```
      $FTEST
[1]   FORMAT(''3Z.4D'', 7.41)
[2]   '' ''                          ... go to next line
[3]   FORMAT(''3D.4Z'', 7.41)
[4]   '' ''
[5]   FORMAT(''E1Z2D.4D'', 7.41)
[6]   '' ''
[7]   $

      FTEST
  7.4100
007.41
 74.1000E-01
```

In addition, a format specification of ''FF'' or ''FF.'' indicates printing in free format (the latter specifies a decimal point to be printed unconditionally). Free-format printing is the same as PPL's conventional print format except rounding is done to one greater digit of precision, i.e. 8 digits for a REAL and 16 for a DBL.

SECTION XI

Definition of Operators

We now turn to a new area of PPL's capabilities: that of extensibility. PPL contains the mechanisms for extension in several directions; the first we shall consider is the definition and redefinition of symbolic operators.

As has already been mentioned in passing, the execution of unary and binary operators is considered to be a special case of system function calling; thus, the expression 'A+B' is evaluated as 'ADD(A, B)'. This is done through a predefined association between the operator '+' and the built-in function 'ADD'. However, the user is allowed to change this association and to define new operators.

An operator in PPL is a sequence of one to four characters taken from the following set:

```
! #$%&*+-./<=>@\↑←:
```

To associate an operator with a function taking one argument (thus defining a unary operator), we call the system function UNARY. This function takes two arguments, the first of which is a datum of type STRING defining the operator, and the second of which is the name of an already-defined function (system or user) taking one argument. We can similarly give an operator a binary meaning by associating it with a two-argument function by executing the system function BINARY.

In the following example, we shall create a function REMAIN, which takes two REAL arguments and returns the remainder resulting from dividing the first by the second. We will then associate this function with the binary operator '\'. Having done this, we may then use the '\' operator freely within expressions for computing the remainder.

```
      $REMAIN(A,B)
[1]   REMAIN←A-B*INT(A/B)
[2]   $

      REMAIN(5.0,3.0)
2.
      BINARY(''\'',REMAIN)
      5.0\3.0
2.
      (34.2+547.1)\37.0
26.3
```

Redefinition of existing operators may be done in exactly the same

manner. A word of caution is in order. When writing a function designed to take over the meaning of a commonly-used operator (such as '+'), the user must take care not to use that operator within the definition of the function. Any such use of the operator will result in a recursive call of the function; this is not usually what is intended! Also note that, since the built-in operations are still accessible by name (the ADD function, for example), it is possible to restore the original meaning of an operator that has been redefined.

For example:

```
BINARY("+", ADD)
```

SECTION XII

Data Definitions

Of the six data types already presented, five (INT, REAL, DBL, BOOL, and CHAR) are 'atomic', meaning that they are the basic forms predefined within PPL; a number of built-in operations may be performed on these types.

Being an extensible language, PPL contains the mechanisms required for defining and manipulating new types of data. Once the user has defined a data type, he may write functions for manipulating instances of that type, and he may associate these functions with symbolic operators. The result is that complicated programming projects involving lists, trees, and other such structures are greatly simplified; through the use of data definitions, PPL may be extended to such a degree that simple expressions become extremely powerful in their actions.

In PPL we may define three different classes of data types. First, we may have structures. A structure is a datum consisting of any number of possibly heterogeneous, named parts. We will make use of the following example to explain the use of structure definitions:

```
$COMPLEX = [RP:REAL, IP:REAL]
```

This is read: "Define a COMPLEX to be a structure containing an RP of type REAL and an IP of type REAL". Issuing this definition causes the following actions to be taken:

First, a 'constructor' function COMPLEX is created by PPL. This is a function which takes as many arguments as there are elements in the data definition (i. e. two for COMPLEX), and creates a datum of type COMPLEX. For example,

```
X←COMPLEX(3. 7, 59. 0)
```

assigns X a value obtained by constructing a COMPLEX whose RP is 3. 7 and whose IP is 59. 0. The types of the arguments must agree with the types of the corresponding components in the data definition.

Second, each identifier to the left of a colon is defined to be a 'selector' name. A selector then becomes a function that operates on a constructed datum and extracts one of its parts. In the definition of COMPLEX above, RP and IP are defined as selectors. An example of the selection operation is as follows:

```
        R←RP(X)
        R
3.7
        13+IP(X)
72.
```

The selection operation may alternatively be denoted in subscript notation by placing the selector name within square brackets immediately following the value from which the selection is being made, thus:

```
        X[RP]
3.7
        13+X[IP]
72.
```

Third and finally, the name of the data definition (COMPLEX) is also defined as a 'predicate'. A predicate may be thought of as a type name, and there exists an operator for testing whether or not a given datum is an instance of a particular type. This is the '==' operator. It takes as its left operand a value, which it checks against the right operand which must be a predicate. If the value is an instance of the predicate, the == operator returns the value TRUE; otherwise it is FALSE. For example, having constructed a COMPLEX and assigned it to the variable X, we may execute the following statements:

```
        X==COMPLEX
TRUE
        3.427==COMPLEX
FALSE
```

At this point we shall note that predicates also exist for the five atomic data types. These predicates, as one might expect, are named INT, REAL, DBL, BOOL, and CHAR. The following example will illustrate their use:

```
        3==INT
TRUE
        4.529==INT
FALSE
        V←93+(45.2/6)*3.5
        V==REAL
TRUE
```

The second type of data definition we may make is a sequence definition. A sequence is a datum consisting of any number of homogeneous, numbered parts. The indexing is done by some continguous subset of the integers.

As an example of a sequence definition, we will define:

```
        $ROW = [1: ] REAL
```

which is read: "Define a ROW to be a sequence of REALS, indexed by integers starting at 1 and extending to an indefinite upper limit".

As is the case for structures, issuing a sequence definition causes a constructor function to be created by PPL. The constructor function ROW will take any number of arguments of type REAL and will construct a ROW out of them. For example,

```
X←ROW(3.56, -28.59/2, 3*55.21/4)
```

assigns to X a ROW consisting of three elements which are the values of the three arguments.

Given a datum of type ROW, we may select any one of its elements by means of indexed selection or 'subscripting'. When the value of X was constructed, the first element of the ROW was associated with the index 1, the second with the index 2, and the third with the index 3. We select a component of X by placing the desired index in square brackets after the name X, thus:

```
      X[1]
3.56
      I←3
      Q←X[I-1]
      Q
-14.295
      (Q+X[I])/2
13.55625
```

Note that the indices start at the number specified on the left side of the colon in the sequence definition. Thus, if we had defined ROW as:

```
      $ROW = [-10: ] REAL
```

then the three elements of X would be X[-10], X[-9], and X[-8]. Attempts to select nonexistent elements of a sequence will result in an execution error.

Making a sequence definition also causes a predicate to be created, as in the case of a structure definition. We may test for occurrences of sequences in exactly the same way as for structures:

```
      X==ROW
TRUE
      X==COMPLEX
FALSE
```

We may also make sequence definitions with fixed upper bounds; for example:

```
$ROW4 = [1:4] INT
```

defines ROW4 to be a sequence of four components of type INT, numbered 1, 2, 3, and 4.

The third form of data definition we may make is an alternate definition, which defines a new predicate to be the union of any number of other predicates. For example:

```
$ARITH = INT ! REAL ! DBL
```

defines an ARITH to be either an INT, a REAL, or a DBL. The names on the right side of the definition may be built-in predicates (e.g. INT), user-defined predicates (e.g. COMPLEX), or predicates defined in other alternate definitions.

Predicates defined in alternate definitions are useful in a variety of ways. The first is in explicit type checking:

```
        (3.276*54)==ARITH
TRUE
        X==ARITH                    ...Recall that X contains a ROW of REALs
FALSE
        (X[3]/X[2])==ARITH
TRUE
```

However, alternate types are much more powerful than this. Consider the following definition:

```
$VECTOR = [1: ] ARITH
```

Given such a definition, it is possible to construct VECTORs consisting of any set of components which are defined as being of type ARITH (i.e. INT, REAL, or DBL). The following example illustrates the behavior of this kind of construction:

```
        X←VECTOR(3, 7.9, 1.54/7)
        X[1]
3
        X[1]==INT
TRUE
        X[1]==REAL
FALSE
        X[1]==ARITH
TRUE
        X[3]==REAL
TRUE
        X[3]==ARITH
TRUE
```

There exist five built-in predicates with the following properties:

ATOMIC - the union of all atoms (INT, REAL, DBL, BOOL, and CHAR).

STRUCTURE - the union of all structures.

SEQUENCE - the union of all fixed sequences (i.e. those with upper bound declarations).

V. SEQUENCE - the union of all variadic sequences (i.e. those with no upper bound declarations).

GENERAL - the union of all data types in the entire system.

There also exists the following built-in data definition:

```
$TUPLE = [1: ] GENERAL
```

The usual data operations may be performed using this definition; for example:

```
X←TUPLE(3, 'Q, TUPLE(4.1, ''ABCDE''))
```

However, a more concise syntax is available for constructing TUPLEs. A TUPLE is indicated by a list of zero or more values enclosed in square brackets. For example:

```
X←[3, 'Q, [4.1, ''ABCDE'']]  ... a tuple of length 3
Y←[TRUE]                     ... a tuple of length 1
Z←[]                         ... a tuple of length 0
```

When PPL is requested to print out the value of a user-defined datum, it does so in the following form:

```
        X←VECTOR(3, 7.9, 1.54/7)
        X
[3,7.9,.22]
        COMPLEX(5.7, -32.4)
[RP:5.7,IP:-32.4]
```

In this representation, the square brackets delimit the extent of the datum in question. This is done because user-defined structures may be nested. For example, a two-dimensional array might be defined, constructed, and displayed as follows:

```
$ARRAY = [1: ] VECTOR
```

```
        A←ARRAY(X, VECTOR(3, 2, 1))
        A
[[3, 7.9, .22], [3, 2, 1]]
        Q←1
        A[Q+1]
[3, 2, 1]
```

Selection is a much more general operation than the examples so far have shown. First, a selection expression may appear on the left side of an assignment statement as well as on the right. Such an operation causes the value of the right-hand expression to be assigned as the selected component of the named variable (of course, this is legal only if the right-hand expression is of the correct type). A few examples will illustrate this behavior.

```
        X
[3, 7.9, .22]
        X[2]←88.54E-27
        X
[3, 8.854E-26, .22]
        X[2]
8.854E-26
        A
[[3, 7.9, .22], [3, 2, 1]]
        A[1]←A[2]
        A
[[3, 2, 1], [3, 2, 1]]
```

Second, we may perform 'multiple selection' on compound structures whose elements are in turn compound structures themselves. The following example illustrates two possible notations for doing this in the case of user-defined sequences:

```
        X←ARRAY(VECTOR(11, 12, 13), VECTOR(21, 22, 23), VECTOR(31, 32, 33))
        X
[[11, 12, 13], [21, 22, 23], [31, 32, 33]]
        (X[1])[3]                    ...Select the third element of X[1]
13
        X[1][3]                      ...Exactly equivalent to above
13
        X[3][2]
32
        X[3, 2]                      ...A more compact representation
32
```

For compound data types consisting of structures, we have even more possibilities. Let us define a NUM as either an ARITH (that is, a REAL, INT, or DBL) or a COMPLEX. Then we shall define a NUMSEQ as an arbitrary sequence of these types.

```
        §NUM = ARITH ! COMPLEX

        §NUMSEQ = [1: ] NUM

        X←NUMSEQ(3.42,COMPLEX(8.1,-33.0),-5)
        X
[3.42,[RP:8.1,IP:-33.],-5]
        X[2][IP]
-33.
        X[2,IP]
-33.
        RP(X[2])
8.1
```

It should be noted at this point that a STRING is actually a compound data structure, for which a data definition is predefined in PPL. This definition is:

```
        §STRING = [1: ] CHAR
```

Thus, it is possible to access individual characters of a string by selection.

```
        ''STRING CONSTANT''[6]
G
        X← ''STRING''
        X[4]← 'U
        X
STRUNG
        X==STRING
TRUE
        X[4]==STRING
FALSE
        X[4]==CHAR
TRUE
```

Note that a STRING is an exception to PPL's usual printing convention. Since a STRING is a compound datum, it would normally be printed thus:

```
        ''ABCDEFGH''
[A,B,C,D,E,F,G,H]
```

However, PPL omits the compound structure notation for STRINGs.

As a demonstration of the power of the data definition facility so far presented, we shall now give an example of formula manipulation in PPL. First, the following synonyms must be established (note: these are not PPL statements).

FORM	means Formula,
BF	means Binary Formula,
UF	means Unary Formula,
LO	means Left Operand,
RO	means Right Operand, and
OP	means Operator.

These definitions are made to PPL as follows:

```
$FORM = UF ! BF ! ATOM

$BF = [LO:FORM, OP:CHAR, RO:FORM]

$UF = [OP:CHAR, RO:FORM]

$ATOM = STRING ! CHAR ! REAL ! INT ! DBL
```

Having made these definitions, we are now in a position to represent formulas as follows: a variable is represented as a CHAR, as is an operator. A numeric constant is represented in the usual way. Some examples of legal formulas are:

```
      F←BF('X, '+, 3)
      G←UF('-, BF('X, '*, F))
      F                          ...F contains formula 'x+3'
[LO:X,OP:+,RO:3]
      G                          ...G contains formula '-(x*(x+3))'
[OP:-,RO:[LO:X,OP:*,RO,[LO:X,OP:+,RO:3]]]
```

We shall now present a 22-statement function that performs differentiation on any formula containing the operators '+', '-', '*', and '/'. The DERIV function takes two arguments; the first is the formula to be differentiated, and the second is the variable of differentiation (represented by a CHAR).

```
      $DERIV(F,X)
[1]   (F==ATOM)-->A
[2]   (F==BF)-->B
[3]   (F==UF)-->U
[4]   A:  (F=X)-->%7
[5]   DERIV←0
[6]   -->%0
[7]   DERIV←1
[8]   -->%0
[9]   B:  (F[OP]='+)-->PL
[10]  (F[OP]='-)-->MI
[11]  (F[OP]='*)-->TI
[12]  (F[OP]='/)-->DI
[13]  PL: DERIV←BF(DERIV(F[LO],X),'+,DERIV(F[RO],X))
[14]  -->%0
```

```
[15]    MI:  DERIV←BF(DERIV(F[LO], X), '-, DERIV(F[RO], X))
[16]    -->%0
[17]    TI:  DERIV←BF(BF(F[LO], '*, DERIV(F[RO], X)), '+,
                 BF(DERIV(F[LO], X), '*, F[RO]))
[18]    -->%0
[19]    DI:  DERIV←BF(BF(BF(DERIV(F[LO], X), '*, F[RO]), '-,
                 BF(F[LO], '*, DERIV(F[RO], X))), '/, BF(F[RO], '*, F[RO]))
[20]    -->%0
[21]    U:   DERIV←UF(F[OP], DERIV(F[RO], X))
[22]    $
```

The DERIV function computes derivatives by recursive applications of the following well-known identities:

```
(d/dx) const  =  0
(d/dx) x  =      1
(d/dx) (a+b)  =  da/dx + db/dx
(d/dx) (a-b)  =  da/dx - db/dx
(d/dx) (a*b)  =  a*db/dx + b*da/dx
(d/dx) (a/b)  =  (a*db/dx - b*da/dx)/(b*b)
```

We shall now use this procedure to find the derivatives of the formulas F and G, which have already been constructed.

```
      DERIV(F, 'X)
[LO:1, OP:+, RO:0]
      DERIV(G, 'X)
[OP:-, RO:[LO:[LO:X, OP:*, RO:[LO:1, OP:+, RO:0]], OP:+, RO:[LO:1, OP:*,
RO:[LO:X, OP:+, RO:3]]]]
```

As is apparent by the second result, the output is quite unreadable when printed directly by PPL. What we need now is some way of printing a formula in a more readable format. We shall do this by defining a function PF, which takes as its argument a datum of type FORM and prints its value.

We introduce at this point a system function PRINT, which offers somewhat more flexibility in formatting. PRINT takes any number of arguments and prints out their values, all on one line. This differs from the implicit print command executed by PPL when the user types an expression for immediate evaluation; in the latter case, PPL spaces down one line after printing the value. An example of the use of PRINT is:

```
      PRINT(3, ''STRING'', 93.27E+4, 'Q)
3STRING932700.Q
```

The following is the required definition for PF, and an example of its use.

```
      $PRINTF(F)
[1]   (F==UF)-->%5
[2]   (F==BF)-->%7
[3]   PRINT(F)
[4]   -->%0
[5]   PRINT('(')
[6]   -->%9
[7]   PRINT('(')
[8]   PRINTF(F[LO])
[9]   PRINT(F [OP])
[10]  PRINTF(F[RO])
[11]  PRINT('))
[12]  $

      $PF(F)
[1]   PRINTF(F)                 ... print the formula
[2]   '' ''                     ... go to next line
[3]   $

      PF(DERIV(F, 'X))
(1+0)
      PF(G)
(-(X*(X+3)))
      PF(DERIV(G, 'X))
(-((X*(1+0))+(1*(X+3))))
```

The result is immediately much more readable. A further improvement could be gained by a straightforward simplification of the expressions, which will not be presented here.

Given the ability to define new data types, it is desirable to be able to redefine common operators to perform operations on them. Some general guidelines should be followed when doing so.

Usually what one wants to do is to extend an operator to manipulate user-defined types, yet to have the standard meaning retained in the case of atomic types. The required technique in writing a function to perform this kind of operation is to make type tests on the operands. If they do not match the user-defined types in question, then the equivalent system function should be explicitly called on the same arguments.

In the following example, we extend the addition operator to handle instances of type COMPLEX.

```
      $CADD(A,B)
[1]   (A==COMPLEX)-->%7
[2]   (B==COMPLEX)-->%5
[3]   CADD←ADD(A,B)
[4]   -->%0
[5]   CADD←COMPLEX(ADD(A,B[RP]),B[IP])
```

```
[6]    -->%0
[7]    (B==COMPLEX)-->%10
[8]    CADD←COMPLEX(ADD(A[RP],B),A[IP])
[9]    -->%0
[10]   CADD←COMPLEX(ADD(A[RP],B[RP]),ADD(A[IP],B[IP]))
[11]   $


       BINARY("+",CADD)        ... Redefine meaning of '+'.
       X←COMPLEX(8.37,-2.1)
       Y←COMPLEX(-1.14E+7,-32.85)
       X+Y
[RP:-1.1399992E7,IP:-3.495E1]
       X+3
[RP:1.137E1,IP:-2.1]
       X+Y+1.47
[RP:-1.139999E7,IP:-3.495E1]
       42+7
49
```

SECTION XIII

Miscellaneous Important Information

This section covers some information necessary for certain advanced kinds of programming in PPL. Some system functions not previously mentioned are also covered. The user is urged to consult Appendix A for a complete list of built-in functions.

A. Copying and Sharing

In writing programs using many varieties of data one discovers that sometimes one wants to copy substructures of structures and, on other occasions, one wants to share the use of certain substructures among several structures or variables. For example, in matrix manipulation (and indeed in almost all kinds of arithmetic such as formula or polynomial arithmetic) one wants to copy substructures. Thus, if one extracts the row vector V that is the third row of the matrix M, the value of V should be a copy of the third row of M. If V were to share the same row vector as the third row of M, then any assignment of new components to V would retroactively change components of M[3], an undesirable behavior. On the other hand, sharing is just the relationship that permits changes in components to be transmitted to all owners of a shared substructure. If for example, the title, author and subject cards in a library card catalog could share common information about a book in the form of a shared substructure, then a change in information about the book could be made once instead of three times. This arrangement saves time and space.

In PPL, a measure of control exists over the choice between copying and sharing of structures at the time of assignment. In particular, expressions of the form V←E cause the value of the expression E to be copied before being assigned to be the value of V, whereas expressions of the form V←←E cause the value of the expression E to be assigned as the value of V without first copying it. This latter form of assignment is called Non-Copy Assignment. It is used to set up sharing relationships.

Example of copying:

V←M[3]

Example of sharing:

V←←M[3]

Non-copy assignment can also be used to set up data structures containing cycles and to model pointers. For example:

```
§X=[1:3]GENERAL
Y←X(1,2,3)
Y[2]←←Y                 ... Y[2] shares value of Y.
                        ... It does this by a pointer since that is
                        ... the only way to model an infinite regress
                        ... in a computer.
```

Pointers can be modeled by use of sharing as follows (note: the use of '§X' as a dummy argument is explained below).

```
      §PNTR=[VAL:GENERAL]

      §MKREF(§X)
[1]   MKREF←PNTR(0)     ... makes pointer with dummy 0 as value
[2]   MKREF[VAL]←←X     ... shares referent X as value of pointer
[3]   §
```

Having made these definitions, R←MKREF(X) creates a pointer to a datum X, and VAL(R) acts on the reference R to recover the value X. Thus, a PNTR is an object, usable as a component of another object, having a value, VAL, which designates a second object X by sharing.

The issue of copying versus sharing also arises in parameter passing when making a function call. PPL allows the user a choice of whether or not arguments are to be copied before being passed to a subroutine.

The normal convention is known as 'call by value', which means that the callee is provided only with the value of the argument; no other information about the corresponding actual parameter is passed. As explained in section 7, the dummy arguments or 'formal parameters' are simply initialized with the values of the calling parameters; from then on they are treated as locals. It follows that PPL's normal convention with regard to parameter passing is one of copying each argument before making the call.

The other system is known as 'call by reference'. A parameter passed by reference is not copied and retains the information necessary to directly access the actual parameter.

The user indicates, at function definition time, which arguments are to be passed by reference. This is done by preceding each such dummy argument by a dollar sign in the function header line. All other arguments will be passed by value as usual.

The following examples illustrate the behavior of call by reference as opposed to call by value.

```
        $CBV(A,B,C)                 ... Example of call by value
  [1]   A←3.4
  [2]   B[3]←5
  [3]   C←C+1
  [4]   $

        X←1.0
        Y←[1,2,3,4]
        Z←5
        CBV(X,Y,Z)
        X
1.                                  ... Note unchanged calling parameters
        Y
[1,2,3,4]
        Z
5

        $CBR($A,$B,$C)              ... Example of call by reference
  [1]   A←3.4
  [2]   B[3]←5
  [3]   C←C+1
  [4]   $

        CBR(X,Y,Z)
        X
3.4                                 ... Note effect of call by reference
        Y
[1,2,5,4]
        Z
6
```

B. $Identifiers

There are several functions that act on the names of other functions given as arguments. For example, ERASE(F) erases the current definition of the text of the function F. If F is a function of one or more arguments, it is easy to distinguish a call of F with arguments from the use of F to designate its own name since all calls are accompanied by argument lists. However, for parameterless procedures, the call of F and the use of F to denote its own name coincide. In this case, by convention, the use of F denotes the call of the parameterless procedure (which may deliver a value), and the special notation $F must be used to denote the name of the function F. Thus, to erase a parameterless procedure one would use an expression of the form ERASE($F).

C. Erasing Definitions

The definitions of functions and data may be erased from an environment by means of the ERASE command. This is a variadic

function which given the names of functions or data definitions as arguments will remove the definition from the environment as if it had never been made. In addition, if ERASE is given the name of a variable as an argument it will remove the value of that variable, as if the variable had never been assigned, and if ERASE is called with no arguments all variables and definitions are erased, as if PPL had been started over. Caution must be exercised not to erase a data definition before all instances of data created according to that definition have been erased. Failure to observe this precaution may lead to run time errors. As mentioned above, to erase a parameterless procedure, an unevaluated identifier is needed of the form ERASE($F).

D. READ and WRITE

PPL has some rudimentary file I/0 mechanisms for saving and restoring PPL programs. These work as follows:

WRITE(S), where S is a string, causes all currently-defined functions, data definitions, and operator definitions to be written out on the file described by S. The string S should contain standard PDP-10 file specifications; if not otherwise specified, the default output device is DSK: and the default extension is .PPL. For example:

```
WRITE(''FOO'')
WRITE(''DTA3:ZOT.XYZ'')
```

The file is written in ASCII format which may be read by PIP, TECO, etc.

READ(S), where S is a string file name specification as explained above, causes the data in the named file to be read in and treated exactly as if it were teletype input. Programs written with WRITE may be read using READ. The user is cautioned that READ(S) should be given only as a direct command. Also, it is the user's responsibility to ensure that definitions in the current environment do not conflict with those in the data file. If such an error occurs on input, the READ operation is terminated. The statement that caused the error may be displayed by pressing ALT-MODE. Example:

```
READ(''DERIV[61,101]'')
```

Note: It is recognized that these READ and WRITE procedures do not constitute a complete, flexible I/0 mechanism. Two improvements are contemplated in future versions of PPL: (1) facilities for reading and writing files of data, and (2) a mechanism for saving and restoring entire working environments. For the present, the user is reminded that WRITE saves only function, operator, and data definitions; specifically, values of globals and locals are not saved.

SECTION XIV

Advanced Debugging Facilities

To supplement PPL's extensive editing facilities and conversational mechanics, a set of debugging features have been included; collectively they are known as 'Stop and Trace'.

To activate and deactivate the stop and trace features, there exist four system functions: STOP, UNSTOP, TRACE, and UNTRACE. Each of these functions takes a function name as its first argument, and, optionally, some additional integer arguments which designate the numbers of statements to be affected. If no integer arguments are furnished, all steps of the function designated by the first argument are affected. For example:

STOP(F, 1, 3, 4, 7)	... Marks steps 1, 3, 4, and 7 ... of the function F to stop.
STOP(G)	... Marks all steps of the ... function G to stop.
UNSTOP(G, 4, 5)	... Removes the stop codes ... from steps 4 and 5 of G.
TRACE(G)	... Marks all steps of G for ... tracing, and
UNTRACE(G, 3)	... Removes the trace code ... from statement 3 of G.

If step N of function F is marked for tracing, then, after each execution of that statement, PPL will print 'F[N]' followed by the value of the expression on line N of F. Normally, all expressions in PPL have printable values except for non-value-returning procedure calls and conditional go to's in which the condition evaluates to FALSE. In these latter two cases, the value is empty and no printing is performed. In the case of assignment statements the value of the right hand side is printed, and in the case of go to's the number of the target statement is printed.

When step N of the function F has its stop code set, an attempt to execute step F[N] will cause the function to be suspended, and control will be passed to the user in the environment of suspension after PPL prints 'F[N]'. The suspension occurs before the execution of statement F[N]. At this point, values of variables may be examined and reset and expressions may be computed in the environment of suspension (see section 8). The function F may be resumed at any statement (including N) by executing the expression '-->N'. In this case, step N will be executed even though its stop code is set.

The following precautions should be heeded while using stop and trace:

If the function F is a parameterless procedure, then '$F' must be specified in place of 'F' in all calls to STOP, UNSTOP, TRACE, and UNTRACE. This is because, in the absence of a dollar sign, 'F' denotes a call of the parameterless procedure, which is not what is desired. The dollar sign is required to indicate reference only to the name of the function, rather than to the value returned by calling that function. For example:

```
STOP($F, 5, 6)
```

If STOP, UNSTOP, TRACE, or UNTRACE is called with improper second and succeeding arguments (e.g. statement numbers out of range, or non-integers), an error will result and one cannot be assured that the designated operation has taken place properly.

After editing a function, the stop and trace codes of every statement of that function are reinitialized to the 'off' condition. This is because statements may have been altered, deleted, or renumbered.

In the following example, we use TRACE to trace the flow of control through the factorial function defined in section 7.

```
      ?FACT
      $FACT(N); I
[1]   FACT←1.0
[2]   I←N
[3]   (I<=0)-->%0
[4]   FACT←FACT*I
[5]   I←I-1
[6]   -->%3

      TRACE(FACT)        ... Set trace codes on every statement of FACT
      FACT(3)
FACT  [1]  1.
FACT  [2]  3
FACT  [3]                ... False conditional go to's have no value
FACT  [4]  3.
FACT  [5]  2             ... I became 2 after execution of I←I-1
FACT  [6]  3             ... Target of go to is value of go to statement
FACT  [3]
FACT  [4]  6.
FACT  [5]  1
FACT  [6]  3
FACT  [3]
FACT  [4]  6.
FACT  [5]  0
FACT  [6]  3
FACT  [3]  0             ... Value of a TRUE conditional go to is the
6.                       ...   value of the right hand side.
```

APPENDIX A

Table of Built-in Functions

This table is intended only as a handy guide. See Appendix B for details on polymorphic type conversions.

Op	Name	Description
A + B	ADD(A, B)	Addition.
A - B	SUB(A, B)	Subtraction.
A * B	MUL(A, B)	Multiplication.
A / B	DIV(A, B)	Division.
A ↑ B	POWER(A, B)	Raising to a power.
+ A	PLUS(A)	Unary plus (fairly useless).
- A	MINUS(A)	Unary negation or logical complementation.
A < B	LESS(A, B)	Less than.
A <= B	LESSEQ(A, B)	Less than or equal.
A > B	GR(A, B)	Greater than.
A >= B	GREQ(A, B)	Greater than or equal.
A = B	EQ(A, B)	Equal.
A # B	NOTEQ(A, B)	Not equal.
A & B	AND(A, B)	And (logical operation).
A ! B	OR(A, B)	Or.
A ← B	ASSIGN(A, B)	Assign the value B to be the value of A.
A ←← B	NONCOPY(A, B)	Make A share the value of B.
A == B	INSTANCE(A, B)	Is A an instance of type B?
--> B	GOTO(B)	Transfer control to statement B.
A --> B	CGOTO(A, B)	If A is TRUE, go to B.

Name	Description
RESET:	Erase all nests of function calls. This function is called in order to escape from environment of suspension, usually to do editing.
PRINT(A, B, . . .):	Print the values of all arguments, with no intervening spaces or returns.
FORMAT(A, B, . . .):	Perform formatted numeric output, using STRING arguments as format specifications and numbers (INT, REAL, DBL) as values to be printed.
UNARY(A, B):	Associate the operator A (a STRING of not more than four characters taken from the set ! #$%&*+-./ <=>@\↑←:) with the system or user function B, in such a way that use of the operator in a unary context will cause the calling of the named function.
BINARY(A, B):	Same as UNARY, but with regard to operators used in a binary context.
INT(A):	Convert the value of A to an integer, where A may be any atomic value (INT, REAL, DBL, CHAR, or BOOL). A CHAR is converted to its equivalent ASCII character code (see Appendix E). A BOOL is converted to 1 for TRUE and 0 for FALSE.
REAL(A):	Convert the value of A to a REAL. See INT above for conventions on BOOL and CHAR arguments.
DBL(A):	Convert the value of A to a DBL.
BOOL(A):	Convert the value of the atomic argument A to BOOL. A REAL, INT, or DBL is converted to TRUE if nonzero and to FALSE if zero. A CHAR is converted to TRUE if and only if it is the character 'T.
CHAR(A):	Convert the value of the atomic argument A to a CHAR. For INT, REAL, or DBL, the value is truncated and used as a character code (See Appendix E). For BOOL, a TRUE is converted to the character 'T and a FALSE to 'F.
WRITE(S):	Write all operator definitions, data definitions, and function definitions onto the file described by the string S. If S is omitted, type the information on the teletype.
READ(S):	Accept input from the file described by the string S as if it were coming from the teletype. Example of a legal string: READ(''DTA3:FOO.PPL'').
LENGTH(A):	Returns the number of components of A. If A is atomic, LENGTH(A) is defined to be one.
L.BOUND(A):	Returns the lower bound of A if it is a sequence (fixed or variadic). If A is a structure, L.BOUND(A) is defined to be one.

Name	Description
MAKE(T, N, E):	Constructs a variadic sequence of type T containing N elements, each of whose initial values is E.
CONCAT(A, B):	Returns the result of concatenating the variadic sequences A and B (which must be of the same type), with A on the left and B on the right.
TYPE(X):	Returns the data type of X as a STRING (usually, simply to be printed out).
NTYPE(X):	Returns the data type of X as an INT (making it easier to compare the types of two objects). At any particular time, the numbers for all different data types will be different; however, the number for any particular type is not guaranteed to be the same from session to session.
ERASE(A, B, C, . . .):	Causes the meanings of the identifiers A, B, and C to be erased from the system. If ERASE is called with no arguments, causes everything to be erased and PPL restarted.
SIN(A):	Sine of A, where A is a REAL representing an angle expressed in radians. A may also be an INT or DBL, but is converted to a REAL internally during calculation (thus the full precision of a DBL is not utilized). The result is always REAL.
COS(A):	Cosine of A (A in radians).
ATAN(A):	Arctangent of A (result in radians).
SQRT(A):	Square root of A.
LN(A):	Natural logarithm of A.
EXP(A):	Exponential of A.
STOP(F, N1, N2, . . .):	Set stop codes. The first argument, F is the name of the function to be affected (must be expressed as $F if parameterless). If there are no further arguments, all statements of F are affected. Otherwise, integers N1, N2, etc. designate individual statements to be affected.
UNSTOP(F, N1, N2, . . .):	Remove stop codes.
TRACE(F, N1, N2, . . .):	Set trace codes.
UNTRACE(F, N1, N2, . . .):	Remove trace codes.
NOT(A):	If A is a BOOL, NOT(A) is the logical denial of A. If A is an INT, NOT(A) is the bitwise complement of the octal representation of A.

APPENDIX B

Automatic Type Conversion

In all PPL's built-in functions called by the execution of operators, a certain amount of automatic type conversion takes place for operands that are not suitable. The circumstances under which this will take place are explained below; all other combinations will result in the execution error message 'Illegal argument type'.

Arithmetic operators (+-*/) convert their operands as follows: A CHAR is converted to an INT by taking its ASCII character code (see Appendix E). A BOOL is converted to the integer value 1 if it is TRUE and 0 if FALSE. If the two operands are now not of the same type, the 'weaker' operand is converted to the type of the 'stronger', with strength defined on the scale INT < REAL < DBL.

The unary operator '-' allows an operand of type INT, REAL, DBL, or BOOL. The result is always of the same type as the operand. For INT, REAL, or DBL, the result is the arithmetic negation of the operand, whereas for BOOL it is the logical complement.

Four of the relational operators (<, <=, >, and >=) allow only operands of type INT, REAL, and DBL. If the two operand types are not the same, conversion is performed as for arithmetic operations.

Two of the relational operators (= and #) allow operands of type BOOL or CHAR if both operands are of the same type. No conversion is necessary in these cases. Conversion for arithmetic types is the same as for the other four relational operators.

Boolean operations (& and !) carry a somewhat more involved convention. If both operands are of type BOOL, then no conversion is necessary. Otherwise, at least one operand must be of type INT or an error will result. A logical operation between non-BOOLs consists of the appropriate operation performed bitwise between the operands, using the 36-bit PDP-10 hardware representations for INT, REAL, and DBL. If a BOOL is one of the operands, it is represented as a word containing all zeroes (for FALSE) or all ones (for TRUE). The result of the operation has the type of the 'stronger' of its operands, with strength defined on the scale BOOL < CHAR < INT < REAL < DBL. For completeness, we present the format of the internal representation for the five atomic types:

INT	A 36-bit 2's complement integer (see PDP-10 System Reference manual).
REAL	A 36-bit hardware floating point number.
DBL	A 72-bit hardware floating point number.
CHAR	A right-justified integer 7-bit ASCII character code.

BOOL A word containing the integer 0 (for FALSE) or 1 (for TRUE).

The --> operator requires a right operand which is an INT, REAL, or DBL. Note that a relocatable line number, which is what is usually used, is really an INT with special properties (which come into play only during text editing). The left operand, if used, must have type BOOL.

The == operator allows any value as its left operand. Its right operand must be a valid predicate; that is, an atomic type name (INT, REAL, DBL, BOOL, or CHAR), a user-defined type name, or a name defined in an alternate definition (e.g. ARITH).

APPENDIX C

Summary of Editing Commands

To display the text of a function F:
?F

To display line 5 of the function F:
?F[5]

To open function F for editing, starting at line 3:
$F[3]

While within a function, move to line 2 for editing:
[2]

To insert a statement between 4 and 5:
[4.5]

To close the function (causing renumbering and translation):
$

Control characters:

Return:	Terminate current statement, sequence to next.
Line feed:	Sequence to next continuation line.
Control-G:	Delete current statement, sequence to next.
Control-U:	Abort current typein and try again.
Alt-mode:	Open (or re-open) current text for character-by-character editing.

Control characters used only in character-by-character editing:

Control-N:	Step past Next character in old line.
Control-E:	Move to End of old line.
Control-T:	Move past Ten characters in old line.
Control-S:	Move to next Separator character.
Control-R:	Retype current line of statement.
Control-A:	Enter insertion mode, for insertion BEFORE the character over the typehead when ↑A struck.

APPENDIX D

Summary of Data Definitions

Structure definition:

§COMPLEX = [RP:REAL, IP:REAL]

Where COMPLEX is the new type name,
RP, IP are selector names for the components, and
REAL is the element type for both components.

Sequence definition (variadic, i.e. indefinite upper bound):

§VECTOR = [1:] ARITH

Where VECTOR is the new type name,
1 is the lower bound for indexing,
the upper bound is unspecified, and
ARITH is the element type.

Sequence definition (fixed):

§ROW4 = [1:4] INT

Where ROW4 is the new type name,
1 is the lower bound,
4 is the upper bound, and
INT is the element type.

Alternate definition:

§ARITH = INT ! REAL ! DBL

Where ARITH is the new predicate, and
INT, REAL, and DBL are the types that are to be considered instances of an ARITH.

APPENDIX E

ASCII Character Codes

Capital letters			Small letters			Other		
	Decimal	Octal		Decimal	Octal		Decimal	Octal
A	65	101	a	97	141	space	32	40
B	66	102	b	98	142	!	33	41
C	67	103	c	99	143	"	34	42
D	68	104	d	100	144	#	35	43
E	69	105	e	101	145	$	36	44
F	70	106	f	102	146	%	37	45
G	71	107	g	103	147	&	38	46
H	72	110	h	104	150	'	39	47
I	73	111	i	105	151	(	40	50
J	74	112	j	106	152	)	41	51
K	75	113	k	107	153	*	42	52
L	76	114	l	108	154	+	43	53
M	77	115	m	109	155	,	44	54
N	78	116	n	110	156	-	45	55
O	79	117	o	111	157	.	46	56
P	80	120	p	112	160	?	47	57
Q	81	121	q	113	161	0	48	60
R	82	122	r	114	162	1	49	61
S	83	123	s	115	163	2	50	62
T	84	124	t	116	164	3	51	63
U	85	125	u	117	165	4	52	64
V	86	126	v	118	166	5	53	65
W	87	127	w	119	167	6	54	66
X	88	130	x	120	170	7	55	67
Y	89	131	y	121	171	8	56	70
Z	90	132	z	122	172	9	57	71
						:	58	72
[	91	133				;	59	73
\	92	134				<	60	74
]	93	135				=	61	75
↑	94	136				>	62	76
←	95	137				?	63	77
bell	7	7						
tab	9	11						
l'feed	10	12						
return	13	15						
altmode	125	175						

APPENDIX F

System Errors

PPL is still under development and will remain so indefinitely. Thus, it is always possible that an internal error will occur. Furthermore, PPL is not fully 'user-proof' in several areas; that is, the user can do something so bad as to cause an internal error.

When a system error occurs, a message is printed and control returns to the PDP-10 time-sharing monitor (signified by printout of a period at the left margin). Nearly all possible errors are detected internally, and PPL exits in such a state that recovery is generally possible by typing the REENTER monitor command. 'Recovery' in this sense means control returning to PPL (which does an implicit RESET in the meantime) with functions, data definitions, operator definitions, and global data intact.

Upon successful recovery, it is wise at this point to save the current programs and definitions (using the WRITE procedure). This is because internal data might have been affected by the error in such a way that a non-recoverable error could occur later.

There are a number of ways in which a user may bring about a system error. A few will be mentioned here to enable the user to know what to avoid.

Erasing a data definition (using the ERASE procedure) is generally not a good idea. If it must be done, the user should take great care that he has first erased all instances of data constructed according to that definition.

Using the '←←' operator (NONCOPY assignment), it is possible to create structures that are 'cyclic' in nature. For example:

```
X←[2,0,3]
X[2]←←X
```

There is nothing illegal about this. However, PPL is not, at present, able to intelligently print out such a structure; if an attempt is made to do so, a system pushdown-list overflow error will occur (it is always possible to recover in this case). For example:

```
      X
[2,[2,[2,[2,[2,[2,[2,[2, ...
      ... SYSTEM ERROR - PDL OV
```

Under certain circumstances it is possible for a data structure to become so complex as to cause PPL to run out of space in certain key

internal tables. It is not possible to state the circumstances because, at the time of writing this manual, this error has not been observed.

Finally, PPL makes heavy demands on PDP-10 core memory resources. In the event PPL runs out of working space and cannot obtain more from the time-sharing monitor, it is forced to give up and exit to the monitor. The message 'DATA ZONE EXHAUSTED' or 'LAPS EXHAUSTED' is printed in this case. Recovery is not possible unless more memory can be obtained.

The designers of PPL would appreciate reports of system errors; only in this way can attempts be made to remove them. However, such reports, to be useful, must be as complete as possible. Examples should be attached.

HISTORICAL NOTE

A first design of PPL was undertaken in Mexico City at the National University of Mexico in the summer of 1968 (see [3]). This design evolved over the next year into the version of PPL presented at a Working Conference on Extensible Languages held at Carnegie-Mellon University on 2-4 Dec. 1968 and at the Extensible Language Symposium held in Boston on 13 May 1969 (see [4], [5]). Implementation was begun at Harvard in July 1970 by T. A. Standish and E. A. Taft and after proceeding for six months, has resulted in the present version of PPL. E. A. Taft designed the editing features of PPL, and, recently, R. M. Stallman has assisted in writing the numerical format routines and certain data routines.

REFERENCES

[1] Iverson, K. E., A Programming Language, Wiley, New York, 1962.

[2] Iverson, K. E., Falkoff, A. D., et al., APL/360 Manual of Operation, IBM, Nov. 1967.

[3] Standish, T. A., A Preliminary Sketch of a Polymorphic Programming Language, Centro de Calculo Electronico, Universidad Nacional Autonoma de Mexico, July 1968.

[4] Standish, T. A., Some Features of PPL -- A Polymorphic Programming Language, Proc. of Extensible Language Symposium, Christensen and Shaw (eds.), SIGPLAN Notices, Vol. 4, Aug. 1969.

[5] Standish, T. A., Some Compiler-Compiler Techniques for Use in Extensible Languages, Proc. of Extensible Language Symposium, Christensen and Shaw (eds.), SIGPLAN Notices, Vol. 4, Aug. 1969.

IMPLEMENTATION NOTE

Several versions of PPL have been produced to conform to software conventions of different PDP-10 systems and to take advantage of special features.

In order for the control-U editing character to operate as described in this manual, a special monitor modification is required. Versions of PPL running on systems without this change accept control-Z as a substitute for control-U. At the time of this writing, control-Z must be used on all systems except Harvard and Tenex (i.e. Telcomp, TSC, Yale, etc. all require control-Z).

The following built-in functions are available on systems with ARDS display terminals (Harvard, TSC at present):

ARDMODE: Enter graphical output mode.

SETPOINT(X, Y): Set the beam to the given coordinates, where (0, 0) is in the center, and useful ranges are in the order of plus or minus 600. SETPOINT does not intensify the beam. To draw a dot, print a period at the given position.

SOLIDVEC(DX, DY): Draw a solid vector through the given displacement, starting at the current beam position.

DOTTEDVEC(DX, DY): Draw a dotted vector through the given displacement.

TTYMODE: Leave graphical output mode.

The following system function is for the Sylvania tablet and stylus on the Harvard PDP-10 only.

READSTYLUS: Causes the Sylvania tablet to be interrogated and the X, Y, and Z coordinates of the pen to be stored, as values of type INT, in variables called XSTYLUS, YSTYLUS, and ZSTYLUS. The X and Y coordinates are similar to those for ARDS output; the Z coordinate indicates height of the pen above the tablet, where 5 to 7 indicates pen off surface, 4 indicates pen touching the tablet, and 0 indicates pen pressed down.

INDEX TO PPL MANUAL

A

Ackerman's Function 14
ADD 9, 46, 49
Addition Operator 9, 49
  for Complex Numbers 38
Alternate Definitions 32
ALTMODE 15, 22, 43, 51
APL 1, 3, 56
AND 8, 9, 46, 49
And Operator 9, 46, 49
Arc Tangent 48
ARDMODE 57
ARDS Display 57
Arguments to Functions 9, 10
  see also parameters
ARRAY 33, 34
Array Notation
  see Subscript Notation
ASCII 7, 53
ASSIGN 9, 10, 46
Assignment Operator 6, 9, 10, 46
  with Selection Expressions
    on the left side 34
  in Tracing 44
ATAN 48
Atomic Data Types 29, 30
Automatic Type Conversion 49

B

BINARY 27, 47
BOOL 7, 29, 30
  in Conditional Go To's 13
  conversion function 47
Boolean 7
Breakpoints (see Stop Codes) 44
Built-In Operators 9, 46
  Predicates 33
  Functions 46

C

Call by Reference 41
Call by Value 41
Changing Function Name 22
  see ERASE
CHAR 7, 29, 30
  conversion function 47
  data type 7, 29, 30
Character 7
  use in definition
    of Strings 35
Character by Character
  Editing 21, 22
Comments in PPL 20
Comparison Operators 7
Complement (see NOT, MINUS)
COMPLEX 29
  addition function for 38, 39
CONCAT 48
Concatenation 48
Conditional Go To 13, 44, 46, 50
Conflict of Names 11
Conflict of Definitions 43
Constructors 29, 31
Continued Lines 21
Control A 22, 51
Control C 16
Control E 21, 51
Control G 18, 51
Control N 21, 51
Control S 21, 51
Control T 21, 51
Control U 2, 51
  use of Control Z in place
    of Control U in certain
    installations 57
Control Z 57
Conversation 17
Conversion of Numbers 25, 49
  Loss of Precision During 25
Conversion of Type 49
Copying 40
COS 48
Cosine 48

D

D 4, 25
Data Definitions 29
  for Structures 29
  for Sequences 30
  for Alternates 32
Data Type 7, 29
DBL 4, 24, 25
  conversion function 47
  in Atomic Data type 29, 30

Debugging 44
Declarations in PPL 6
Definitions in PPL
of Functions 11
of Operators 27, 28
of Data 29
Erasure of 42
Demand Symbol 17
Deletion 18
with Control G 18, 51
DERIV 36
Desk Calculator 3
Dewey Decimal 18
Displaying Functions 19
Displaying Lines 23
DIV 9, 46, 49
Division Operator 9, 46, 49
integer division 5
Dollar Sign 11, 12, 51, 52
to define a Function 11
to Close Function Def. 12, 18
to Open a Function for Editing 18
in Data Definitions 29
in Call by Reference Parameters 41
Dollar Identifiers 42
DOTTEDVEC 57
Double Precision Numbers
see DBL
Dummy Arguments 12
Dummy Argument Lists 4, 12

E

E 4, 25
Editing 18
see Text Editing
Editing while Suspended 15
End of Definition of Functions 12
End of Execution of Functions 13
Environment
Global 15
of Suspension 15, 44
EQUAL 9, 46, 49
Equality Operator 7, 9, 46, 49
ERASE 42, 48
Erasing Definitions 42, 48
Errors in PPL 15, 54, 55
Syntax Errors 15
Execution Errors 15
System Errors 17, 54, 55
Error Messages 15
Syntax Error Messages 15
Execution Error Messages 15
System Error Messages 17, 54, 55
Execution Errors 15
EXP 48
Exponentiation 46, 48
see also Power

F

Factorial Function 11, 13, 14
FF 26
FILES, reading and writing 43
Fixed Upper Bounds 31
Floating Point 4, 24, 25
FORM 36
Formals
see Formal Parameters
Formal Parameters 12
in Environment of Suspension 15
FORMAT 25, 47
Formatted Printing 25, 47
Formula Manipulation 35
Free Format (or FF) 26
Function Body 11
Function Calls 9, 12
Parameterless 14
Function Definitions 11
Function Header 11, 12
Function Identifier 11, 12
see also Procedure Identifier
Function Name 11, 12
Function Return 13
Function Termination 13
Function Value 11, 13
Functions, Parameterless 14

G

Global Environment 12
GOTO 12, 46, 50
Go To Operator 12, 46, 50
Unconditional Go To 12, 46, 50
Conditional Go To 13, 46, 50
used to Resume Execution 16, 44
in Tracing 44
GR 9, 46, 49
Greater Than Operator 7, 9, 46, 49
GREQ 9, 46, 49
Greater Than or Equal to Operator 7, 9, 46, 49

H

Harvard Version of PPL 57
Historical Note 56

I

Identifiers 6
  used as Formal Parameters 11
  used as Procedure Identifier 11
  Local Identifiers 11
Implementation Note 57
Insertion 18
  see also Control A 22
Instance Operator 30, 46, 55
INT 4, 24, 29, 30
  the type INT 4, 24, 29, 30
  the conversion function 10, 47
Integer (see INT) 4
Integer Division 5
Iverson 1, 3, 56
Iversonian Precedence 3

L

Labels in PPL 22
Labelled Statements 22
L. BOUND 47
LENGTH 47
LESS 9, 46, 49
LESSEQ 9, 46, 49
Less Than Operator 7, 9, 46, 49
Less Than or Equal
  to Operator 7, 9, 46, 49
Line Continuation 21
Line Editing 15
  see also Text Editing and
  Character by Character
    Editing
LINE FEED 21, 22, 51
Line Numbers 12, 18
Line Renumbering 19
LN 48
Locals (see Local Variables)
Local Variables 11, 12, 13
  in Environment of Suspension 15
Logarithm 48
Logical Operators 7
  see also NOT 48

M

Make 48
Marking a Function for
  Stop and Trace 44
MINUS 9, 46, 49
Minus Operator 9, 46, 49

N

Name Conflicts 12
Natural Logarithm 48
Nested Function Execution 13
Non-Copy Assignment 40, 46
NONCOPY 46
Non-Value Returning
  Function 13, 43
NOTEQ 7, 9, 24, 46, 49
Not Equal Operator 7, 9, 24, 46, 49
NOT Operator 8, 48
  see also Minus
NTYPE 48
Numbers in PPL 4, 24
  Integers 4, 24
  Reals 4, 24, 25
  Doubles 4, 24, 25
  Formatted Printing 25, 26
  Conversion 4
  Octal Numbers 24

O

Octal Numbers 24
Opening a Function 18
  for Editing
Operator Definitions 27
Operator Precedence in PPL 3, 4
OR 8, 9, 46, 49
Or Operator 8, 9, 46, 49

P

Parameters
  Formal Parameters 11, 12
  Actual Parameters 9
  Call by Value 41
  Call by Reference 41
Parameterless Procedures
  In Recursion 14
  Preventing Calls with s 42
Parentheses in PPL 3, 4, 9
  In Expressions 3, 4
  In Parameter Lists 11
  In Recursive Parameterless
    Procedure Calls 14
  Unbalanced 15
Per Cent Sign
  In Relocatable Line
    Numbers 12, 19, 23
PLUS 9, 46

Plus Operator 9, 46
Pointers 41
POWER 46
Precedence in PPL 3, 4
Predicates 30
  for Atomic Types 30
  for Structures 30
  for Sequences 31
  from Alternate Definitions 32
  Built-In Predicates 33
PRINT 37, 47
Printing in PPL
  After Assignment 10
  After Non-Value Returning Function Call 44
  Conventions for Numbers 25
  Conventions for Structured Data 33
  Use of Print Function 37
  Use of Format Function 25
Procedure Identifier 12, 13, 14
  In Environment of Suspension 15
  Used to Call Parameterless Procedures 14, 42
Procedures
  see Functions
Pure Procedure
  see Non-Value Returning Procedures

Q

Question Mark
  As Demand Symbol 17
  To Display a Function 19, 51
  To Display a Line of a Fn. 23, 51

R

.R PPL 2
READ 43, 47
  Reading and Writing Files 43, 47
READSTYLUS 57
REAL 4, 24, 29, 30
  the type REAL 4, 24, 29, 30
  the conversion function 10, 47
Real 4
Recursion in PPL 13, 14
  implicit Recursion through Operator Definitions 28
Redefinition of Operators 28
REENTER 17
References 56
  see also Call by Reference
  see also Pointers
Relocatable Line Numbers 12, 19, 23
REMAIN 27
Remainder Function 27
Reopening a Line 15
Replacement of Text 18
  see Text Editing
RESET 15, 47
Restoring Original Meanings of Operators 28
Result of a Function 13
Resuming Execution after Suspension 15, 16, 44
Rounding 25
  see also Integer Division
ROW 30
RUBOUT 2

S

Selectors 29, 31
  on Left Hand Side of Assignments 34
  Notation 29, 34
  Compound Selection 34
Sequence Definitions 30
Sequencing to a New Line during Editing 18
SETPOINT 57
Sharing 40
Sharp Sign 24
  see also Not Equal Operator
Sine 48
SIN 48
SOLIDVEC 57
Stallman, R. M. 56
Standish, T. A. 56
Starting PPL 4
Statements in PPL 12
  Labels for Statements 22
STOP 44, 48
Stop Codes 44
Stopping
  Execution of A Function 17
  Setting Stop Codes 44
Strings 7
  Data Definition of 35
Structures 29
Structure Definitions 29
SUB 9, 46, 49

Subtraction Operator 9, 46, 49
Subscript Notation 29, 34
Summary
of Editing Commands 51
of Data Definitions 52
of ASCII Character Codes 53
of System Errors 54
Suspension 15, 44
Sylvania Tablet 57
Syntax
for TUPLES 33
for STRINGS 7
Syntax Errors 15
System Errors 17, 54, 55

T

Table
of Operators 46
of Built-In Functions 46, 47
of ASCII Character Codes 53
Taft, E. A. 56
Talkative Programs 17
Telcomp Version of PPL 57
Tenex Version of PPL 57
Termination of Function
Definition 12
Execution 13, 16
with Control C 16
Testing Data Types 29
see Instance Operator
Text Editing 18
While Typing In 19
Insertion of Lines 18
Deletion of Lines 18
Opening a Function 18
Displaying a Function 19
Character by Character 21, 22
of Function Header 22
Effect on Stop and Trace 45
TRACE 44, 48
Tracing 44
Trig Functions 48
Truncation in Integer Division 5
TSC Version of PPL 57
TTYMODE 57
Tuples 33
TYPE 48
see also Data Type
and Type Conversion

U

UNARY 27, 47
Unconditional Go to
see Go To
UNSTOP 44, 48
UNTRACE 44, 48
Updating
see Text Editing
User Defined Functions 11
User Suspension 15
use of Control C
to Suspend a Fn. 16

V

Value of a Function 11, 13
Variables 6
Local 12
Unassigned 15
VECTOR 32
Versions of PPL 57

W

WRITE 43, 47
writing and reading files 43, 47

Y

Yale Version of PPL 57

Z

Z 25, 57
Control Z 57
Use of Z in Formats 26
Zero Suppression 26

s

s see Dollar Sign

?

? see Question Mark

#

see Sharp Sign

%

% see Per Cent Sign

Unclassified
Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)* Harvard University Cambridge, Massachusetts	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED 2b. GROUP N/A

3. REPORT TITLE

PRELIMINARY PPL USER'S MANUAL

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*

None

5. AUTHOR(S) *(First name, middle initial, last name)*

E. A. Taft

6. REPORT DATE October 1970	7a. TOTAL NO. OF PAGES 68	7b. NO. OF REFS 5
8a. CONTRACT OR GRANT NO. F19628-68-C-0101 b. PROJECT NO. c. d.	9a. ORIGINATOR'S REPORT NUMBER(S) ESD-TR-70-441 9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*	

10. DISTRIBUTION STATEMENT

This document has been approved for public release and sale; its distribution is unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY Deputy for Command and Management Systems Hq Electronic Systems Division (AFSC) L G Hanscom Field, Bedford, Mass. 01730

13. ABSTRACT

PPL is an interactive, extensible programming language incorporating data definition facilities and operator definition facilities. This manual introduces the various features of the language and serves as a practical reference document for its use. The manual is indexed and several appendices detail parts of the language in tabular form.

PPL runs under the standard 10/50 monitor series of the Digital Equipment Corporation's PDP-10 and also under the TENEX monitor system of the Bolt, Beranek and Newman Corporation. A library of PPL extensions is available containing extensions for doing matrix, rational, formula, polynomial, complex, and vector arithmetics and, in addition, for manipulating lists, strings, and trees. The language is useful for manipulating data in a wide variety of application areas. Being interpretive and conversational, however, it is more well adapted toward personal and educational uses of computers using interactive terminals than it is for doing large institutional computing applications requiring long running times or voluminous input-output.

DD FORM 1 NOV 65 1473

Unclassified
Security Classification

Unclassified
Security Classification

14. KEY WORDS	LINK A ROLE	LINK A WT	LINK B ROLE	LINK B WT	LINK C ROLE	LINK C WT
extensible language						
programming language						
data definition facilities						
data structures						

Unclassified
Security Classification