M T S

The Michigan Terminal System

VOLUME 12: PIL/2 IN MTS

December 1974

The University of Michigan Computing Center
Ann Arbor, Michigan

DISCLAIMER

This manual is intended to represent the current state of the Michigan Terminal System (MTS), but because the system is constantly being developed, extended, and refined, sections of this manual may become obsolete. The user should refer to the Computing Center Newsletter, Computing Center Memos, and future updates to this manual for the latest information about changes to MTS.

December 1974

PREFACE

The software developed by the Computing Center staff for the operation of the high-speed processor computer can be described as a multi-processor supervisor that handles a number of resident, re-entrant programs. Among them is a large subsystem, called MTS (Michigan Terminal System), for command interpretation, execution control, file handling, and accounting maintenance. Most users interact with the computer's resources through MTS.

The MTS manuals, a series that will eventually consist of a dozen or more volumes, describe in detail the hardware, software, and administrative policies of the Computing Center. The volumes now in print are listed below. The date indicates the most recent edition of each volume; however, since manuals are updated frequently by means of CCMemos, users should check the Memo list, or watch for announcements in the Newsletter, to be sure that their MTS manuals are fully up to date.

Volume 1: MTS and the Computing Center, January 1973
Volume 2: Public File Descriptions, April 1971 (reprinted with updates 1-5, December 1972)
Volume 3: Subroutine and Macro Descriptions, May 1973
Volume 4: Terminals and Tapes, August 1974
Volume 5: System Services, March 1974
Volume 10: BASIC in MTS, September 1974
Volume 11: Plot Description System, April 1971
Volume 12: PIL/2 in MTS, December 1974
Volume 13: Data Concentrator User's Guide, August 1973

Other manuals are in preparation. The numerical order of the volumes does not necessarily reflect the chronological order of their appearance; however, in general, the higher the number, the more specialized the volume. Volume 1, for example, introduces the user to MTS and the Computing Center, while Volume 10 deals exclusively with BASIC. The attempt to make each manual complete in itself and reasonably independent of others in the series naturally results in a certain amount of repetition. Public file descriptions, for example, may appear in more than one volume. However, this arrangement permits the user to buy only those manuals that serve his or her immediate needs.

General Editors

December 1974

PREFACE TO VOLUME 12

PIL, the Pittsburgh Interpretive Language, was designed and implemented at the University of Pittsburgh. This manual describes the second version of PIL. The first version of PIL was documented in the second edition of Michigan Terminal System, Volume II, December 1967, and in another manual, Introduction to PIL in MTS, by Dr. Larry Flanigan, May 1968. Both of these manuals are now obsolete. PIL/2 in MTS is based on the second version of PIL, with many modifications and additions for the implementation on the Michigan Terminal System.

We wish to acknowledge Brent J. Ermlick, author of PIL/X, PITT Interpretive Language for the DEC System-10 Computer, on which this manual is based. However, this manual has been extensively revised and supplemented.

Lynn R. Leader
L. Bernard Tiffany

December 1974

Contents

Preface 3

Preface to Volume 12 4

Introduction 7

Terminal Description 10
  Editing 10
  Attention Interrupts 12

Modes of Operation 14
  Direct Mode 14
  Indirect Mode 14
    Parts and Steps 14

Variables and Constants 17
  Constants 17
  Variables 18

Expressions 21
  Arithmetic Expressions 21
  Boolean Expressions 25
  Character Expressions 27
    String Comparison 27
    String Functions 28
    String Manipulation 29

Language Statements 32
  Comments 32
  Assignment Statements 32
    SET Statements 32
    SWAP Statement 33
  Conditional Statements 33
  Simple I/O 35
    TYPE Statement 35
    DEMAND Statement 36
  Iteration Statements 38
    FOR Statements 38
    FOR Conditional Keywords . . 39
    FOR Control 40
  Transfer of Control 42
    DO Statement 42
    DO String 43
    TO Statement 43
  Deletion Statements 44
    DELETE Statement 44
    Variable Deletion 44
    Part and Step Deletion . . . 45
    Extended Delete 45
    Storage Clean-up 46
Program Stops 46
  NUMBER Statement 46
Error Reporting 47
Program Restart 50
Extended I/O List Features . . 52
  I/O FOR Lists 52
  FORMs 53
  Numerical Fields--Standard . 53
  Numerical Field--Scientific Notation 54
  Numerical Fields--Modified Standard 54
  Alphanumeric Fields 55
  Text Material Defined in the FORM 56
  Field Stop Code--Field Delimiter 56
  Special-Purpose Fields . . . 58
  Character Strings as FORMs . 59
  Literal FORMs 59
  Expressions in FORM References 59
  Formed Output 60
  Formed Input 61
  Error and Resume with Formed I/O 62
  Free Formed I/O 64
  Rules Governing Free Formed Output 65
  Rules Governing Free Formed Input 66
  TYPE FORM N 67
  TYPE ALL FORMS 68
  Form Deletion 68
Auxiliary I/O 68
  WRITE Statements 69
  READ Statements 70
  Control Operations 71
  DELETE ASSIGNMENT N 73
  TYPE ASSIGNMENT N 73
Program Management 74
  Pagination 74
  Program Saving 75
  Program Loading 76

Table 1. Arithmetic Operators and Functions 78

Table 2. Boolean Operators . . . 81

Table 3. String Operators and Functions 82

Table 4. Precedence Order . . . 83

Appendix A. Summary of PIL Statements 84

Appendix B. The Michigan Terminal System 85

Appendix C. EBCDIC Character Set 87

Index 89

December 1974

INTRODUCTION

This manual describes PIL, the Pitt Interpretive Language, as modified for use under MTS; it is based on the second version of PIL, PIL/2. Currently the University of Pittsburgh uses a third version of PIL, PIL/X; which is implemented on their PDP-10s, and thus is not compatible for use with MTS.

PIL is similar to earlier conversational languages such as JOSS[1] and TELCOMP[2] with major differences in debugging facilities, error reporting, and problem-solving capabilities. Unlike the compiler languages PL/1, FORTRAN, and ALGOL, PIL, an interpretive language, provides the user with much greater assistance through the use of terminal diagnostics, user interaction with the machine, and associated error-recovery procedures. PIL differs from compilers by providing direct man-machine interaction facilities, and from earlier conversational languages in the relaxation of restrictions that they imposed upon the user.

A major goal in the design of PIL was to allow a user to recover errors as he worked toward a solution for a particular problem. When the user needs to make corrections or improvements, PIL allows him to alter his program and to continue without starting anew.

PIL is oriented toward problem-solving, with program development and debugging facilities having highest priority. For the beginning user, PIL was designed to be clear, unambiguous, and hence, easily learned. For the experienced programmer, the language offers increased flexibility with statement structure and expanded capabilities for the solution of non-numeric problems. For the researcher, PIL reduces the amount of time and effort that must be expended in problem solving.

PIL is an interactive language and, as such, has limited application for batch users. Although it may be run in batch mode, any error condition encountered is likely to produce undesired results. In batch mode, all non-data input lines are echoed to the printer, thus providing a complete printed record of the session. Except for these differences, batch execution of PIL is essentially the same as conversational (terminal) execution.

Every statement in PIL begins with a keyword and terminates with an optional period followed by a carriage return, which transmits the statement to the computer. Periods do not always indicate the end of the statement, since they can be contained within a numerical or string constant such as 27.98 or ".". (See the section on variables, for an explanation of the

---

[1]Developed by C. Shaw at the RAND Corporation, Santa Monica, California.
[2]Developed by Bolt, Beranek, and Newman, Cambridge, Massachusetts.

terms, "numerical constant" and "string constant".) For example, the statement

```
*SET a=34.
```

is equivalent to

```
*SET a=34
```

Some statements contain more than one keyword. For example:

```
*IF a = 3, SET a = 4.
```

has two keywords, IF and SET.

In this manual, keywords will always appear in capital letters. Users should note, however, that in actual practice, PIL ignores the case of keywords. Therefore, keywords may be typed in any combination of uppercase and lowercase, i.e., SET, set, and Set are the same keyword to PIL. Keywords specify what is to be done and consequently have certain rules associated with them. The language interpreter uses three criteria to identify keywords: (1) the location of the word in the statement, (2) the first four letters of the word[3], and (3) word separation. Therefore, keywords do not have to be reserved names. Thus

```
*SET SET = 27.98.
```

is a legal statement in the language. The first SET is recognized as a keyword because it is the first word in the statement; the first word specifies the action to be taken. A variable name is expected between a keyword and an equal-sign. Since SET follows the rules for variable naming (see the section on variables for these rules), it is legal and unambiguous.

PIL uses an asterisk (*), as a prefix character to request a PIL command, and uses a question mark (?) as a prefix character when requesting data or values.

PIL executes a statement in one of two modes: direct mode (desk calculator mode) or indirect mode (stored program mode). In direct mode, the terminal behaves like a very sophisticated desk calculator. A statement is accepted by PIL, immediately executed (before more input is requested), and then lost except for its effect (i.e., to re-execute the statement, it must again be entered from the terminal). Current values given to variables are kept from statement to statement, so that one has in essence, a desk calculator with storage (memory) available and several quite sophisticated function "buttons" to push. Use of PIL in direct mode is convenient for some applications, but is obviously quite limited since one cannot re-execute statements except by typing them again. Indirect mode overcomes this problem by allowing the user to put together sequences of PIL

---

[3]All keywords may be abbreviated by their first four letters. Characters after the fourth are not checked for validity.

statements (called "parts") into a program which can be executed as a sequence as many times as the user wishes, since the part is not lost when it is executed (i.e., the part need not be retyped to use it a second time). Normal PIL usage is a blend of direct and indirect modes, using direct mode to control the execution of the various parts in indirect mode.

Please note that the examples in this manual are not continuous parts of the same terminal session; they have been chosen only to illustrate portions of the text.

December 1974

TERMINAL DESCRIPTION

The PIL user sits at a computer terminal, a device very similar to a standard typewriter except that it is connected to the computer over standard telephone lines. First, the user dials one of the telephone numbers listed below to establish connection with the computer.

Memorex 1270	763-0300
Data Concentrator	763-1500

Once a connection line has been established, the line is always open and the computer is always listening (unless, of course, the system has crashed). The user converses with the computer by typing in statements or data. For more complete information concerning the use of computer terminals and MTS, consult MTS Volume 1, MTS and the Computing Center.

PIL will type an asterisk (*) at the beginning of a line when it is ready to receive a command or an indirect statement. A question mark (?) is used as a prefix character when PIL is ready to receive data. When a prompting character is printed by PIL, the user should enter a PIL statement, PIL command, or the appropriate data immediately to the right of the character. The user must then enter the line by pressing a return key, which closes the line and sends the information to the computer.

MTS supports a diversity of computer terminals. Some are equipped with only uppercase letters; others have both uppercase and lowercase letters. Some have certain characters that are not found in other terminals, such as a cent-sign (¢) or a back slash. Each terminal has its own protocol which must be followed. MTS Volume 1, MTS and the Computing Center, and Volume 4, Terminals and Tapes, should be consulted for full details on the characteristics of various terminals which are supported by MTS. In this manual, only Teletypes and IBM 2741s will be discussed.

EDITING

Editing facilities depend on both the user's terminal and the transmission control unit. There are five basic editing facilities.

1.	End-of-line	Terminate the current line.
2.	Delete-line	Delete the whole current line.
3.	Delete-previous	Delete the previous character.

4. Literal-next — Interpret the next character as the character itself.

5. End-of-file — Terminate the input data with a logical end-of-file.

The following table shows which keys (or combinations of keys) the user must press to produce the various editing facilities according to terminal type and control unit.

	Teletype via Memorex 1270	Teletype via Data Concentrator	IBM 2741 via either control unit
End-of-line	RETURN, Control-Q, or Control-S	RETURN or Control-S	RETURN
Delete-line	Control-N	RUBOUT	Underscore
Delete-previous	Control-H	Control-H	Backspace
Literal-next	Control-P	Control-P	Exclamation point (!)
End-of-file	Control-C	Control-C	Cent sign (¢)

2741 users should note that they have to type !!, !¢, and !_ to enter the characters !, ¢, and _, respectively. Uppercase conversion can be turned off via the PIL command

```
*CONTROL 'UC=OFF' ON '*MSINK*'.
```

Then, both upper and lowercase letters may be entered.

PIL provides its own editing facilities in addition to those the terminals provide. They are:

1. If the last non-blank character in an input line is an asterisk (*), then the whole line is deleted. For example:

```
* This line is ignored*
```

2. If the last character is a minus sign (-), then PIL will assume that the next line is to be attached to the current line to form one logical line. The minus sign is not considered to be part of the logical line; the line continues with no blanks inserted. The maximum length of a logical line in PIL is 255 characters. If the user enters more than 255 characters, the line is truncated without warning. PIL uses an ampersand (&) as the prefix character for continued lines.

```
*TYPE 1+2+-
&3+4
```

is the same as

```
*TYPE 1+2+3+4
```

ATTENTION INTERRUPTS

The user may interrupt PIL at any time. The method of interrupting is dependent upon the kind of terminal and control unit.

Teletypes via Memorex 1270: press BREAK key. If using a Model 35 Teletype, then also press BRK RLS key.
Teletypes via Data Concentrator: either press the BREAK key or Control-E (ENQ) key.
IBM 2741s: press ATTN key.

If PIL is interrupted while the user is in direct mode, the message

INTERRUPTED!!

will follow. A program may be interrupted with the message:

INTERRUPTED AT STEP 3.89

Control is returned to the PIL interpreter. At this point, any direct mode statement can be made (with the exception that an active FOR or DO statement can not be deleted).

To return to MTS, the user must press the ATTN key twice. He will be met with:

PIL ATTN!!

and MTS replies with a "#" prefix character. To return to PIL, the user simply types in:

$RESTART

which causes PIL to regain control at the point at which it was interrupted.

A sample terminal session is shown below. The user has entered in all lines shown in lowercase, as well as those with the asterisk (*) prefix character.

```
(The user dials 763-1500, which connects the terminal to the computer.)
MTS :  ANN ARBOR (DC16-0086)
#signon xxxx
#ENTER USER PASSWORD.
?pill
#JOB-TYPE=TERMINAL, PRIO=NORMAL, CLASS=UNIV/GOVT
#**LAST SIGNON WAS:  09:33.05
# USER "XXXX" SIGNED ON AT 10:20.13 ON TUE AUG 27/74
#run *pil
#EXECUTION BEGINS
 PIL/2:  READY
*TYPE 125/5.
 125/5 = 25.0
*TYPE 1.32 + 12.8/32.
 1.32 + 12.8/32 = 1.72
*TYPE (sine of 12.8)**2+(cosine of 12.8)**2.
 (SINE OF 12.8)**2+(COSINE OF 12.8)**2 = 1.0
*SET a = the square root of 9.
*TYPE a, a**2.
 A = 3.0
 A**2 = 9.0
*STOP
EXECUTION TERMINATED
#signoff
#OFF AT 10:22.04    TUE AUG 27/74
#ELAPSED TIME        2.733 MIN.          $.12
#CPU TIME USED       2.394 SEC.          $.18
#CPU STOR VMI         .566 PAGE-MIN.     $.03
#WAIT STOR VMI        .698 PAGE-HR.
#DRUM READS        103
#APPROX.  COST OF THIS RUN IS     $.33
#DISK STORAGE          193 PAGE-HR.      $.03
#APPROX.  REMAINING BALANCE:   $20.53
(The terminal is disconnected.)
```

The example above shows a complete terminal session. The user with ID "xxxx" signs on. He runs *PIL, which then greets him with a short message:

PIL/2: Ready

The prompting character (*) is printed at the beginning of the next line. The user should then enter any PIL statement following the asterisk (*).

To terminate the PIL session and return to MTS, the user enters a STOP statement. Finally, he signs off. The statistics are printed, including the cost of the terminal session. Then the terminal is automatically disconnected from the computer.

MODES OF OPERATION

DIRECT MODE

As previously mentioned, in direct mode, the terminal may be used as a sophisticated desk calculator, permitting the user to evaluate arithmetic expressions, determine the value of functions, and store results for later use.

Statements in direct mode result in an immediate response by PIL. After execution of a direct mode statement, the statement is not retained, but any variables defined by it (and their values) are.

In direct mode, errors are reported immediately. After the user has corrected them, he must retype the statement. The sample terminal session in the previous section was comprised of direct mode statements.

INDIRECT MODE

Indirect statements are retained until the programmer requests that they be executed, and then are processed in a sequence defined by part and step numbers. Indirect statements comprise a stored program, as in FORTRAN or ALGOL, while the desk calculator mode is unique to conversational languages. The user may use either mode at any given time to best solve his problem.

Parts and Steps

Programs are divided into parts. A part is a collection of one or more steps (or statements) arranged in ascending order of step number. A part and step number, followed by a space, must precede every statement entered into a program.

```
*1.05 SET data = 27.98.
```

is an indirect mode statement, where .05 refers to the step number, and the integer 1, is the part number. Indirect statements may be typed in any order and will be inserted in their proper order according to part and step number by PIL.

Part numbers must lie in the range from 1 to 9999 inclusive. A step number must lie in the range .0001 to .9999. As a unit, however, a total of

only 7 digits can be specified. Therefore, 9999.9999 is an illegal part and step number. Step numbers may have an arbitrary increment between them (e.g., Step 1.1 would be followed by Step 1.95 if there were no steps between the values of 1.1 and 1.95).

At any time the user may request the typing of any part, step, all parts, or some combination. Steps are referred to by both the part and step number.

```
*TYPE step 1.5.
*TYPE part 3.
*TYPE all parts.
*TYPE part 3, step 1.5.
```

In typing a part or all parts, PIL types the most current version of the part. The steps within a part will be arranged in ascending order by step number.

Processing of a stored program is initiated by a

DO part n

where "n" is the part number. Once started, PIL will execute the steps of the designated part in order by step number. Step 2.0000 does not follow step 1.9999 in execution since they are in different parts, although it will follow step 1.9999 in program listings. Since it is possible to change a program at any time, it is advisable to allow room between steps for insertions. Additions will automatically be placed in the correct numerical sequence.

Comments may be typed and stored as part of a program. Following a part, step number, and blank, if the first character in the statement is an asterisk (*), the remainder of the statement is taken as a comment.

```
*1.64 TYPE a,b,c.
*1.65 *Output of intermediate results.
```

To change the sample program shown in the previous section to indirect mode:

```
#run *pil
#EXECUTION BEGINS
 PIL/2:  Ready
*1.05 TYPE 125/5.
*1.10 TYPE 1.32+12.8/32.
*1.20 SET a = the square root of 9.
*1.15 TYPE (sine of 12.8)**2 + (cosine of 12.8)**2.
*1.25 TYPE a,a**2.
*DO PART 1.
 125/5 = 25.0
 1.32 + 12.8/32 = 1.72
 (sine of 12.8)**2 + (cosine of 12.8)**2 = 1.0
```

```
a = 3.0
a**2 = 9.0
```

December 1974

VARIABLES AND CONSTANTS

The SET statement may be utilized when the user desires to store information for later use.

```
*SET a = 27.89.
*SET c = 10.
*SET b = 2.0+10.0.
*TYPE a, b, c.
 a = 27.98
 b = 12.0
 c = 10.0
*TYPE a+b/c.
 a+b/c = 29.18
```

In the preceding example, a, b, and c are called variables, and the values stored in them are called numerical constants. A variable is a symbolic name which has a value that may change during execution of a program. A constant (which is not a symbolic name) has a value that cannot change.

CONSTANTS

Constants may be numerical, such as 3.1415 or 7; character (or string), such as "PIL/2" or "123="; or Boolean, such as The True or The False. Numerical constants are written as numbers with or without decimal point and/or sign. Any number of digits may be used to express a numerical constant, but only the seven most significant, starting with the first non-zero digit, are retained by PIL. Because a scaling factor is used, it is possible to represent both positive and negative numbers from 1.0*10**-65 to 9.999999*10**64 inclusive. Numerical constants may also be expressed in scientific notation, e.g., 25E21 means 25*10**21, while 25E-20 means 25*10**-20. (The special character string "**" indicates exponentiation.)

Following is another example of a variable which is set to a numerical constant:

```
*SET data = 5.3e5
*TYPE data
 data = 530000.0
```

In this example, 5.3E5 is the constant. The variable "data" has associated with it the numerical value of 530,000.

A character string is any sequence of characters, including a null sequence, up to 255 characters in length. Any character may appear within the string except the hexadecimal value X'00'.

A string constant is any character string enclosed by a pair of primes (') or quotes ("), called delimiters. The constant begins in the column after the first delimiter and continues to the column before the next occurrence of the same delimiter. If a delimiter, such as an prime or a quote, is a character in the constant, it must be written as two primes or two quotes, with no intervening blanks. For example, 'it''s' is equivalent to "it's". Note that the prime in this constant is a delimiter in the former example but is not in the latter example.

In the following examples, "a", "b", "c", and "d" are set to string constants.

```
*SET a = "A string of characters".
*SET b = 'He said, "Punt".'
*SET c = "That's a good idea."
*SET d = 'it''s'.
```

There are two Boolean constants, The True and The False, to which all Boolean expressions evaluate. Boolean arithmetic is discussed in the section on expressions.

VARIABLES

Variables provide a method for retaining intermediate results for use in subsequent computation.

Variable names must conform to the following rules:

1. The first character of the name must be a letter (either uppercase or lowercase).

2. The remaining characters may be letters or numerals.

3. The total number of characters in a variable name may not exceed eight.

4. Uppercase letters <u>are</u> distinguished from lowercase.

Examples:

```
A, daTA, filename
```

but not,

```
DataName672   Too many characters
17AC          Starts with a numeral
.GB           Starts with a special character
```

If "DataName672" is used as a variable name, an error message stating "Eh? SYMBOLIC NAME TOO LONG" is printed. Using either "17AC" or ".GB" results in the error message "Eh? INVALID SEQUENCE OF OPERATIONS".

It is important for users to note that uppercase and lowercase letters are not equivalent in variable naming, as they are when indicating keywords. Thus, the variables "DATA" and "data" are distinct.

In addition to variable names with a single value associated with each name, it is possible to have many values associated with a single variable name. Single- or multiple-dimension arrays (tables) make it possible to accomplish this. A subscripted variable must be used to indicate a specific value in the array. The subscripts must be separated by commas and enclosed in parentheses. For example, DATA(1,2) represents the second entry in the first row of a table called DATA. The general rules for subscripts are:

1. Each subscript may be a constant, a variable, or a numerical expression.

2. A subscript may take any numerical value between -999,999 and +999,999, but only the integer part is used to refer to an element. Thus, array(1.5) = array(1).

3. There is no limit to the number of dimensions an array may have. When referring to an array element, one subscript for each dimension must be present.

An example illustrating Rule 1 is:

```
*SET i = 1.
*SET j = 2.
*SET a(i) = 7.
*SET DATA(a(i)+j,i,j,1) = 24.282.
```

DATA(a(i)+j,i,j,1) is the same as DATA(9,1,2,1) and may be referred to in either form so long as the values of a(i), i, and j remain unchanged. Any element of DATA must be referenced with four subscripts.

Rule 2 is illustrated by the following example:

```
*SET X(3) = 142.87.
*SET X(3.141592) = 3.141592.
*TYPE X(3.141592), X(3).
 X(3) = 3.141592
 X(3) = 3.141592
*SET X(-1.5)=28.
*TYPE X(-1.5), X(-1).
 X(-1) = 28.0
 X(-1) = 28.0
```

In the above example, X(3)'s first value of 142.87 has been replaced by 3.141592, since only the integer portion of the subscript is used.

Rule 3 makes it an error to use a different number of subscripts once an element of the array has been defined, e.g.:

```
*SET d(1,2) = 9
*SET d(3,4,5) = 100
 Eh? UNMATCHED SUBSCRIPTS
```

Elements of the same array do not all have to be of the same type. For example,

```
*SET A(1,1) = 1.
*SET A(2,2) = 'string'
```

December 1974

EXPRESSIONS

PIL provides three kinds of expressions: arithmetic, Boolean, and character string. Arithmetic expressions give numerical results, such as 1.5 or 25. Boolean expressions always evaluate to either The True or The False. Character expressions yield a character string.

ARITHMETIC EXPRESSIONS

The arithmetic operations of addition, subtraction, multiplication, division, and exponentiation are represented in PIL by +, -, *, /, and **, respectively. The multiplication operator must always be written, as the rules of variable naming prohibit adjacent positioning to imply multiplication. AB means the variable name AB and not "A times B". The phrase "A times B" is written as A*B. Variables, constants, arithmetic functions, and expressions coupled with or preceded by operators, form expressions. Thus, a, a+b, (a+b)*c, 2*((a+b)/(c-d)), and x+THE SINE OF Y are expressions.

Vertical bars (|) denote absolute value of expressions, i.e., the sign is always made positive, |-2.0| = +2.0; thus, |a| and |a+b| form expressions.

Arithmetic expressions may fall into several categories:

1. Variables or constants,
   e.g., a, temp1, 12.0, 1.56E20
2. Functions operating on an expression,
   e.g., SINE OF b, SQRT OF 25
3. Parenthesized expressions,
   e.g., (a+b-c), (1.0)
4. Expressions coupled by binary operators,
   e.g., (a+b)*(c-d)
5. Expressions preceded by unary operators,
   e.g., -(a+b), -(-1), +(a)
6. Expressions enclosed by vertical bars (|),
   e.g., |a|, |a+b|.

Grouping marks, such as parentheses, are used to delimit the scope of operators in an expression. Operators have an implied or fixed precedence order, so any expression without parentheses will always yield the same result.

The arithmetic operators and functions are listed below in order of precedence from high to low; equal precedence is shown on the same line. When an expression is evaluated, the order of evaluation is determined by the precedence of the operators; a higher precedence operation is performed before one of lower precedence.

Expression Element	Example
functions	square root of a
absolute value	| a |
exponentiation	a**b
negation (unary)	-a
multiplication, division	a*b, a/b
addition, subtraction	a+b, a-b

Whenever operators of the same precedence are encountered in an unparenthesized expression, they are executed in order from left to right. However, a function of a function, e.g., SQRT OF SQRT OF 16, is evaluated from right to left: SQRT OF SQRT OF 16 = 2.

Consider the following examples:

Expression	Equivalent
a+b/c*d	a+((b/c)*d)
a+b**c	a+(b**c)
a+b*c+d	(a+(b*c))+d
a/b/c/d	((a/b)/c)/d
-a**2	-(a**2)

Extraneous parentheses in expressions are ignored by PIL.

In addition to the arithmetic operators, many functions are available, such as "SINE OF", "COSINE OF," etc. Most functions in PIL have a short and a long form. The long and short forms of function names may be used interchangeably. The functions are listed below with a brief explanation. Like keywords, only the first four letters in any word in a function name are checked for validity, and, thus, only those need be used. Table 1 (in the back of this manual) contains a table of functions for reference.

SQUARE ROOT OF X - SQRT OF X
This function takes the square root of the argument X (>=0).

SINE OF X - SIN OF X
This function takes the sine of X (X is in radians).

COSINE OF X - COS OF X
This function takes the cosine of X (X is in radians).

LOG OF X
This function takes the logarithm, base 10, of "X".

ANTILOG OF X
This function takes the antilog of "X".

LN OF X
This function takes the natural logarithm (base e) of "X".

ARC TANGENT OF X - ATAN OF X
This function takes the inverse tangent of "X".

EXP OF X
This function takes the exponential of "X".

RANDOM NUMBER OF X - RN OF X
This function acts as a random number generator. If "X" is a single variable (or subscripted), the value of "X" changes unpredictably and should not be changed. To obtain the longest possible sequence of pseudo-random numbers, the same variable should always be used. The value of the function, e.g., "Y" in

```
*SET Y = RN OF X
```

is uniformly distributed over the interval 0 to 1.

INTEGER PART OF X - IP OF X
This function takes the integer part of a number. For example, IP of 2.25 = 2.0.

FRACTION PART OF X - FP OF X
This function takes the fractional part of a number. For example, FP of 2.25 = 0.25.

EXPONENT PART OF X - XP OF X
This function returns the scaling factor of the parameter. For example, XP of 3.5 = 0.0, XP of 101.5 = 2.0.

DIGIT PART OF X - DP OF X
This function returns the value of X/(10**exponential part of x). For example, DP of 35 = 3.5, DP of 102.6 = 1.026.

MINIMUM OF (X,Y,Z) - MIN OF (X,Y,Z)
This function may take two or more arguments. The parameter least in value is returned. This function is defined for string as well as for numerical parameters.

MAXIMUM OF (X,Y,Z) - MAX OF (X,Y,Z)
This function may take two or more arguments. The parameter greatest in value is returned. This function is defined for string as well as for numerical values.

THE MODE OF X
In addition to a value, every variable has a mode associated with it. There are five variable modes: (1) numerical, (2) Boolean, (3) character string, (4) array, and (5) undefined. The function, THE MODE OF, takes a simple or subscripted variable as an argument, and reports the variable mode of the argument as a number from one to five. These numbers correspond to the sequence of variable modes listed above.

If the statement:

*SET x = the mode of b.

Is preceded by:	The value of x would be:	Explanation
*SET b = 34.	1.0	b is defined as a numerical value.
*SET b = 2>1.	2.0	b is defined as a Boolean value.
*SET b = "PIL".	3.0	b is defined as a character string.
*SET b(1) = 3.	4.0	PIL has a defined variable called "b" where the defined variable has sub-scripts. The mode of b is an array, whereas the mode of b(1) is a numerical value.
*DELETE b.	5.0	b is not defined.

The function, THE MODE OF, refers to the variables themselves, and not to direct or indirect mode.

THE TOTAL SIZE

When a user begins a PIL session, a certain amount of space (core memory) is allocated in which to work. This space is used to store variables and their values, programs, and anything the user defines in the course of his session. This space is measured in terms of the number of PIL-words it could contain. One PIL-word is 16 bytes. A numeric variable uses one PIL-word of space, whereas a character string requires two PIL-words (if the string is less than 14 characters long). Array allocation is even more complicated. The function THE TOTAL SIZE gives a count of the number of PIL-words that are initially available to the user. The maximum size available is 65536 PIL-words.

THE SIZE

THE SIZE gives a dynamic count of the number of PIL-words available at the time of the function call.

```
*TYPE the size.
 the size = 197.0
*SET a = 1.
*SET b = 'abcdefghijkl'
*TYPE the size, the total size.
 the size = 195.0
 the total size =197.0
```

THE TIME
This function returns the time in 300ths of a second, relative to midnight (00:00). For example, 1.296000E+07 is noon.

THE DATE
This function returns the day of the year in the form YYDDD where YY refers to the year (19YY) and DDD is the day of the year. For example, THE DATE = 74236 is August 24, 1974.

THE ELAPSED TIME
This function returns the actual connect time in 300ths of a second since the user signed on.

THE CPU TIME
This function returns the CPU time the user has used in 300ths of a second since the user signed on.

THE COST
This function returns the estimated cost of the user's run thus far.

Notice that if a function takes parameters, the function name must be followed by the word "of". If the function does not take parameters, the function name must be preceded by the word "the". For example:

```
*SET A = The Time.
```

will set A to the time of day to the nearest 300ths of a second, relative to 12 midnight. It is also permissible, but not necessary, to use "the" before a function with parameters; e.g.,

```
*SET A = the square root of (2*data(i,1)+3).
```

Notice also that a function parameter may itself be an expression; keep in mind that functions have the highest precedence of all the operators, so that SQRT of 4*3 means (sqrt of 4)*3 and would evaluate to 6.0.

BOOLEAN EXPRESSIONS

Boolean expressions (sometimes called "truth-functions" or "logical" expressions) are also available in PIL. A SET statement will store the result of a Boolean expression in any specified variable and set the mode of this variable to Boolean. A Boolean expression may be:

1. A Boolean literal, i.e., The True or The False, or a variable with a Boolean value.

2. Two arithmetic or Boolean expressions coupled with a Boolean binary operator.

3. A Boolean expression preceded by a Boolean unary operator.

The following explains Boolean operators, functions, and constants.

$LT, <
This is the relational operator, less than.

$LE, <=, ¬>
This is the relational operator, less than or equal to.

$EQ, =
This is the relational operator, equal to.

$NE, ¬=
This is the relational operator, not equal to.

$GE, >=, ¬<
This is the relational operator, greater than or equal to.

$GT, >
This is the relational operator, greater than.

$AND, &
This is the logical product. If both Boolean expressions on either side of the "$AND" are true, then the value of the whole expression is true. Otherwise, it evaluates to false. For example,

```
1<2 $and 5=5 is true,
1>2 $and 5=5 is false.
```

$OR, #
This is the logical sum. If either expression or both are true, then the whole expression is true. This function evaluates to false only if both of the expressions are false. For example,

```
1<2 $or 5>7 is true,
1>2 $or 5>7 is false.
```

$NOT, ¬
This is the negation, and reverses the value of the expression.

```
$not 1>2 is true.
```

$XOR
This is the exclusive or. If one of the two expressions is true, the whole expression is true. If both are false, or both are true, the expression evaluates to false. For example,

```
5=5 $xor 1<2 is false,
5=6 $xor 1<2 is true.
```

THE TRUE
This is the constant value, THE TRUE.

THE FALSE
This is the constant value, THE FALSE.

THE BATCH
This evaluates to true if the user is in batch mode, false if the user is in conversational mode.

All Boolean operators have lower precedence than the arithmetic operators. The priority of Boolean operators is shown in the following four lines, ranked in descending order of priority. The operators shown on the same line are equal.

```
$gt (>), $lt (<), $eg (=), $ne (¬=), $ge (>=, ¬<), $le (<=, ¬>)
$not (¬)
$and (&)
$or (#), $xor
```

For example,

```
*SET x= $not a<b # b=c & e+d=f-g
```

is equivalent to:

```
*SET x= ($not(a<b))#((b=c) & ((e+d)=(f-g)))
```

The following examples show the use of Boolean expressions:

Statement	result
*SET x=The True	x=The True
*SET x=10+5<7+8	x=The False
*SET x=1=1	x=The True
*SET x=1=2 $and 1$ne 2	x=The False
*SET x=The True & (1>2#3=3)	x=The True

In the third example, the first equal sign encountered (going from left to right in the statement) is the replacement operator; the second equal sign is a relational operator.

CHARACTER EXPRESSIONS

A character string is any sequence of characters, including the null string, up to 255 characters in length. Functions are available for string manipulation.

String Comparison

Any string may be compared with any other string, using the relational Boolean operators. (See the section on Boolean expressions above.) Individual characters of the two strings are compared from left to right. If

strings are of unequal length, the shorter string is treated as though it were padded at the right with blanks. The following collating sequence is the basis for comparison of strings:

```
blank < punctuation marks < a...z < A...Z < 0...9
```

For a complete list of the collating sequence, including the punctuation marks, consult Appendix C. Here are some examples of string comparison:

```
*IF "x" < "y", TYPE "yes".
 yes
*IF "abcd" = "abcd ", TYPE "strings equal".
 strings equal
*type 'Ab' < 'bA'
 'Ab' < 'bA' = The False
```

String Functions

THE LENGTH OF - (L OF)

To determine the length of a string, THE LENGTH OF (which may be abbreviated to L OF) function is used. Its value is a count of the characters contained in a given string. It will always be an integer in the range 0 to 255.

```
*SET x = "1234567".
*TYPE the length of x.
 the length of x = 7.0
```

THE UPPER CASE OF - (UPPER OF)
THE LOWER CASE OF - (LOWER OF)

Since many applications using strings are concerned with content, it is useful to be able to ignore the case of alphabetic data. This can be done in PIL by using the following two PIL functions:

```
*SET X = the upper case of "abcde".
*SET Y = the lower case of X.
*TYPE X,Y.
 X = "ABCDE"
 Y = "abcde"
```

Special characters and numerals are unchanged by these functions.

December 1974

String Manipulation

Two strings may be concatenated, i.e., the second joined to the end of the first. The "+" (plus) operator performs this task, provided that both operands are strings. The length of the resulting concatenation is the sum of the lengths of the two operands and may not be greater than 255. To illustrate:

```
*SET x = "12345".
*SET y = "67890".
*SET z = x + y + "abc".
*TYPE z, the length of z.
 z = "1234567890abc"
 the length of z =13.0
```

It is also useful to be able to extract and examine some portion of an arbitrary string. There are three functions that allow this kind of manipulation. They are:

```
The first m characters of string1
The last n characters of string2
The substring of (string3, offset, length)
```

Here "m" and "n" may be any arithmetic expression, and "string1" and "string2" are any string expressions. "m" and "n" must be non-negative and not greater than the length of the string. With these considerations in mind, the first two functions are self-explanatory and convenient for accessing characters at the beginning or end of a string, respectively.

These two functions can be replaced by string operators which are, respectively:

```
m $fc string1
n $lc string2
```

Thus we have:

```
3 $FC 'abcdef' = "abc"
2 $LC '12345' = "45"
1 $LC ('12' + '34') = "4"
```

$FC and $LC are of the same precedence as their equivalent functions. (For precedence order of operators and functions, see the section on expressions.) Therefore, if the two operators, $FC and $LC, occur in one statement, they are executed in order from right to left. Thus, complex expressions may be written, such as:

```
3 $fc 5 $lc 'adcdefgh' = "def"
3 $lc 5 $fc 'abcdefgh' = "cde"
```

To get just the first or the last character of a string, one may use the following names:

```
The first character of string1
The last character of string2
```

or, respectively:

```
1 $fc string1
1 $lc string2
```

To access internal characters, the third function, SUBSTRING OF, is the most convenient. The three parameters must appear in the following order:

1. The string to be operated on.

2. The offset, i.e., the starting point of the desired substring. (The first character has offset 1, etc.)

3. The length of the desired substring.

The offset must be greater than 0, the length must be greater than or equal to 0, and the sum of the offset and the length must be less than or equal to the length of the string plus one.

Consider the following examples:

1.
```
*SET x = the first character of the last 2 characters of "abcd".
*TYPE x.
 x =  "c"
```

2.
```
*SET y = the substring of ("ab"+"ef", 3, 1)
*TYPE y
 y = "e"
```

3.
```
*FOR i = 1 to n: FOR x(i) = the first character of str: -
   SET str = the last (length of str - 1) characters of str.
```

The last example would put "n" successive characters of the variable "str" into x(1),...,x(n-1),x(n), respectively, and leave any remaining characters in "str". The following statement would define the array "x" as above, but would leave the variable "str" unaltered.

```
*FOR i=1 to n: SET x(i) =  the substring of (str,i,1)
```

THE VALUE OF - (VL OF)

This function takes a string as its argument and returns, as its value, the numerical value obtained by evaluating the string as a PIL expression. If the function receives an argument which is not a string, then the function value is the numerical value of the argument itself. This function is useful for converting a string containing numerical digits to a numerical value. For example:

```
*SET a = 3, b = 5, c = "a+b*2"
*TYPE the value of c.
 the value of c = 13.0
*TYPE the value of "12345".
 the value of "12345" = 12345.0
```

THE BCD VALUE - (BCD VL)

The BCD VALUE (or BCD VL) function allows conversion of all data types to string. If the operand is numerical, the decimal digits are converted to character digits. If the operand is string, the BCD VALUE is identical to the operand. If the operand is Boolean, the BCD VALUE will be either "The True" or "The False".

```
*SET a = 3.
*TYPE the bcd value of (a*a).
 the bcd value of (a*a) = " 9.0"
```

THE BCD TIME
THE BCD DATE

THE BCD TIME and THE BCD DATE return strings with the time or date in readable form.

```
*TYPE the bcd time, the bcd date.
 the bcd time = "14:29.06"
 the bcd date = "08-17-74"
```

THE USER

The last string function, THE USER, returns a four-character user ID:

```
*TYPE the user
 the user = "1B3C"
```

LANGUAGE STATEMENTS

COMMENTS

Comments may be entered at any time. If the first character of an input line is an asterisk, the entire line is considered to be a comment.

ASSIGNMENT STATEMENTS

SET Statements

The format of the simple SET statement[4] as shown in the examples used thus far, is:

```
*SET v = e
```

where "v" is a variable name and "e" is an expression.

A more general form of the SET statement, the multiple SET, allows more than one variable to be assigned a value in a single statement. The format is:

```
*SET v1 = e1, v2 = e2, v3 = e3, ...
```

where processing proceeds from left to right.

For example:

```
*SET i=1, x(i)=i, b=The True, c="Hi!"
*TYPE i, x, b, c
 i = 1.0
 x(1) = 1.0
 b = The True
 c = "Hi!"
```

To use an undefined variable, i.e., a variable without an associated value, is an error, and PIL will notify the user that an error has occurred.

---

[4]The keyword SET can be omitted. It is implied by the presence of the equal sign (=) immediately after a variable name, either simple or subscripted.

```
*SET b = i.
 Eh? i = ?
```

This SET statement is not legal, since "i" is not defined. It is now up to the user to define "i" in some manner (e.g., SET i = 1.) or continue to another task not dependent on the value of "i".

SWAP Statement

The SWAP statement interchanges the values and the mode of two variables.

```
*SWAP a, b.
```

affects a and b in the same way as (but more efficiently than) the sequence:

```
*SET temp=a, a=b, b=temp.
```

Subscripted variables are allowed in the SWAP statement:

```
*SWAP A(i), B(j,k).
```

After this statement, A(i) is set to the value that B(j,k) had, and B(j,k) has the value that A(i) had.

CONDITIONAL STATEMENTS

Conditional transfer of control is accomplished with the IF statement. In the simplest form, the PIL statement IF is followed by a conditional expression (any Boolean expression), a comma (,), and another PIL statement.

```
IF a<b, TYPE a
```

The second PIL statement is the object statement and can be any legal PIL statement. If the Boolean expression evaluates to The True, the object statement is executed; if it evaluates to The False, it is not.

Consider the following statement:

```
*1.5 IF j < 5, IF x+y+z > 57, SET j=j+1.
```

If the value of "j" is less than 5, then "x+y+z" is compared to 57. If, and only if, both conditional expressions are true, PIL will execute SET j=j+1.

The IF statement also has a provision for executing a statement when the condition is false.

```
*3.7 IF a < b, THEN DO part 1; ELSE DO part 2.
*1.1 TYPE "CONDITION WAS TRUE".
*2.1 TYPE "CONDITION WAS FALSE".
```

The words THEN and ELSE may be omitted. Their only purpose is to improve readability. The semicolon (;) separating the two object statements is necessary. Without it, PIL assumes the simpler form of the IF statement is meant. Thus, step 3.7 may be retyped as:

```
*3.7 IF a < b, DO part 1; DO part 2.
```

As mentioned earlier, the IF statement requires a Boolean expression. Thus, a Boolean variable may be used as the Boolean expression for the IF statement as in the following example:

```
*SET flag = 1 < 2.
*IF flag, TYPE "TRUE"; TYPE "FALSE".
 TRUE
```

An example of a more complex Boolean expression follows:

```
*SET a= The True, b= 1>2, c= $not b $and a
*IF (a $or b) $and ($not c), TYPE "True"; TYPE "False".
 False
```

An IF statement may have another IF statement as the object of the THEN or ELSE clause. A difficulty arises in a statement such as:

```
*IF a>b, THEN IF x=y, TYPE x; ELSE TYPE z
```

To which IF does the ELSE clause belong? In other words, is "z" to be typed: (1) when "a" is less than or equal to "b" or (2) when "a" is greater than "b", and "x" does not equal "y"? The rule in this situation is that an ELSE clause belongs to the "nearest" IF which does not have an ELSE clause. In the above example, since the ELSE clause belongs to the object statement IF, (see number 2 above), "z" is typed. In order to associate the ELSE to the outermost IF, i.e., type "z" (as described in number 1 above), the object statement IF must have its own ELSE clause. A clean way to do this is:

```
IF a>b, THEN IF x=y, TYPE x; * ; ELSE TYPE z
```

The object IF statement now has a single asterisk for its ELSE clause. The "*" is regarded as a comment (or a "no operation") and it tells PIL to go on to the next statement. It is associated with the object statement IF, causing "ELSE TYPE z" to be associated with the outermost IF statement. "z" is then typed when "a" is less than or equal to "b." If "a" is greater than "b" and "x" does not equal "y", PIL continues at the next statement after the outermost IF.

December 1974

SIMPLE I/O

Simple Input/Output (I/O) is achieved by the statements DEMAND and TYPE, followed by one or more items. Other I/O statements are described in the sections on extended I/O.

TYPE Statement

TYPE is an output statement which has been used in earlier examples. It requests output of one or more items.

```
*SET a = 3.1, b = 1<2, c = "PIL/2".
*TYPE a,b,c.
 a = 3.1
 b = The True
 c = "PIL/2"
```

The TYPE statement may also include algebraic expressions:

```
*SET a=1, b=10, c=2, d=3, e=4.
*TYPE (a+(c*d/e))/b.
 (a+(c*d/e))/b = 0.25
```

Literal strings, delimited by pairs of quotation marks (") or primes (’), may be typed:

```
*TYPE "Hello", ’Hello’+"!"
Hello
’Hello’+"!" = "Hello!"
```

Notice above that a single string constant is typed without repeating the expression in the list. This can be used to enter headings into output. In most cases, the variable is typed, followed by an equal sign and the value.

An entire array may be requested by a single reference to the name. For example:

```
*SET a(1,1)=1.0, a(15,7)=300, a(7,24)=2.0
*TYPE a.
 a(1,1) = 1.0
 a(7,24) = 2.0
 a(15,7) = 300.0
```

Only those elements of the array that are defined will be typed.

```
*TYPE a(2,1)
 Eh?  a(2,1) = ?
```

A special format is used for typing numbers whose absolute values are above 999,999 or below 0.0000001. The form of this number is SD.DDDDDDESDD, where S represents a sign (+ or -), D indicates a digit, and E represents 10**.

```
*TYPE 10**10, 1E10.
 10**10 = 1.000000E+10.
 1E10 = 1.000000E+10.
```

The TYPE statement is used for several special purposes not directly related to the execution of a program. Summarized below are some of the forms available.

1. To get a copy of a part which has been defined, specify:

   ```
   *TYPE part 5.
   ```

2. To get a copy of the entire program presently defined, specify:

   ```
   *TYPE all parts.
        or
   *TYPE all steps.
   ```

   (Note that both statements, "TYPE all parts" and "TYPE all steps" will cause the entire program to be printed. Thus, they are interchangeable.)

3. To list all defined variables and their current values, specify:

   ```
   *TYPE all values.
   ```

4. To list the entire program, all variables and their values as well as assignments and forms (these are used and described in a later section on extended I/O), specify:

   ```
   *TYPE all stuff.
   ```

Since the terminal is a relatively slow device, the user is advised to use these features sparingly. Information may be selectively obtained by:

```
*TYPE a,b,c,part 5,step 5.1,x(i).
*TYPE step 1.3,a+b,sine of x(j)+2.
*TYPE a+b/c, all parts, all values.
etc.
```

DEMAND Statement

The DEMAND statement requests the user to provide values for a list of variables. The list follows the word DEMAND.

```
*DEMAND a,b,c.
 a = ?_27.98.
 b = ?_18.46.
 c = ?_57.28.
```

The "?_" following "a =", "b =", "c =" is provided by PIL and indicates that the interpreter is ready for a response from the user, such as 27.98. The user's response to a DEMAND sequence may be any of the following:

1. Constants, e.g., 2.79828.

2. Arithmetic expressions, e.g., a+b/c, where a, b, and c are previously defined variables.

3. Functions, e.g., The size, sqrt of a.

4. Any combination of the above.

5. End-of-file to signal the end of the input.

```
*DEMAND a,b,c.
 a = ?_4.0
 b = ?_a + sqrt of a
 c = ?_a*b
*TYPE a,b,c
 a = 4.0
 b = 6.0
 c = 24.0
```

Upon demand for a subscripted variable, the value of the subscript will be given by PIL.

```
*1.1 SET i = 1.
*1.2 SET j = 2.
*1.3 DEMAND b(i,i+j,j).
*DO part 1.
 b(1,3,2) = ?_The True
*TYPE b(1,3,2)
 b(1,3,2) = The True
```

DEMAND, unlike TYPE, cannot use the variable name to imply an entire array. If "x" is an array, DEMAND x will cause the following report:

```
*SET x(1,1) = 1., x(2,4) = 0.
*DEMAND x.
 x = ?_5.
 Eh? UNMATCHED SUBSCRIPTS.
```

Notice that the error is spotted after the user supplied the first value.

ITERATION STATEMENTS

FOR Statements

The FOR statement in PIL provides the user with a convenient method for controlling the repeated execution of a particular segment of a PIL program. It is flexible enough to provide all types of loop control normally required by the user, as well as easy handling of more intricate loops.

The simplest FOR statement repeats an object statement, such as SET a(i) = i, for a given list of values.

```
*FOR i = 1,2,3,4,5,7,9,11: SET a(i) = i.
```

This statement will repeat the execution of the object statement eight times, using each of the listed values for the variable i.

The object statement may be any legal PIL statement except a TO statement. Any expression may be specified in the list. For example:

```
*SET a=11, b=10, c=5, s="This is a string"
*FOR i="A",a+b*c, a-50, s: TYPE i
 i = "A"
 i = 61.0
 i = -39.0
 i = "This is a string"
*
```

Instead of specifying each individual value the variable in a FOR loop is to take, a shorthand notation can be used. The general form is:

```
*FOR i = m TO n: DEMAND b(i).
*FOR i = m TO n BY p: DEMAND b(i).
```

where "i" is a variable, "m" and "n" may be any numerical expression, and "p" is any numerical expression which evaluates to a positive number. The default increment ("p" in the second example) is one. It is possible to use more than one TO-phrase in a single FOR statement:

```
*FOR i = 1 TO 5, 7 TO 11 BY 2: DEMAND b(i).
```

"i" takes on the values 1, 2, 3, 4, 5, 7, 9, and 11.

The BY phrase may precede the TO so that:

```
*FOR i = 1 TO 5, 7 BY 2 TO 11: DEMAND b(i).
```

is equivalent to the previous FOR statement.

Any type of expression may appear in a single FOR list, with each entity separated from the next by a comma. Thus, one can type:

```
*FOR i = a+b, x+y TO x BY w: TYPE a(i).
```

When a TO loop is encountered, if the initial value of the FOR variable is not greater than the final value, the object statement is executed. Upon return, the increment is added to the FOR variable, and then the FOR variable is compared to the final value. Whenever the value of the FOR variable becomes greater than the final value, the loop is terminated, and the process is repeated with the next item in the list. The process is repeated until the last list element (which precedes the colon) is satisfied. Note that if the initial value is greater than the final value in the TO loop, the object statement will not be executed.

One word of caution! The indexing element is interpreted on every reference. Thus, if the indexing variable ("i" in the example above) is an array element, it is looked up each time the variable is incremented. It is possible that different array elements may serve as the index during a single FOR statement. The same problem may occur if the increment or final value are specified by array elements. For example:

```
*SET i=1
*FOR a(i)=1 TO 20 : SET i=i+1
```

will result in:

```
 Eh? a(2) = ?
```

FOR Conditional Keywords

Two keywords provide loop control for FOR statements. The first is:

```
*FOR i = a BY b UNTIL r > n: TYPE x(i).
```

X(i) is typed for successive values of "i" until the Boolean expression which follows the keyword UNTIL is satisfied. The second form is:

```
*FOR i = a BY b WHILE j < n+1: SET c(i) = b(i).
```

This form will continue to repeat as long as the Boolean expression following the keyword WHILE is true. (CAUTION! If the Boolean expression has a constant value, The True, an infinite loop results.)

If the "BY" value is omitted, the index variable will NOT be incremented using WHILE or UNTIL, and the object statement is executed until the Boolean expression evaluates to The False for a WHILE clause, or to The True for an UNTIL clause. For example:

```
*SET x=1, y=4
*FOR I=1 UNTIL x>y : SET x=x+i
*TYPE x,i
 x = 5.0
 i = 1.0
```

Summarized below are some of the various FOR statements, using TYPE as the object statement.

```
*FOR i = 1 TO 3: TYPE i,i**2.
 i = 1.0
 i**2 = 1.0
 i = 2.0
 i**2 = 4.0
 i = 3.0
 i**2 = 9.0
*FOR i = 1 BY i TO 20: TYPE i.
 i = 1.0
 i = 2.0
 i = 4.0
 i = 8.0
 i = 16.0
*FOR i = 3 BY 17 WHILE i < 40: TYPE i.
 i = 3.0
 i = 20.0
 i = 37.0
*FOR i = 3 BY 18 UNTIL i > 50: TYPE i.
 i = 3.0
 i = 21.0
 i = 39.0
```

The statement:

```
*FOR i = 1 BY 1 WHILE 1 < 2: SET i = i+1.
```

will result in an infinite loop, since the Boolean expression (1 < 2) is always true.

FOR Control

Three other statements are related to the FOR statement and affect the course of the iteration. They are:

```
*NEXT i.
*LAST i.
*END i.
```

NEXT i restarts the FOR loop with the next value of the index "i" regardless of the nesting of FOR statements or parts. This allows nested FORs and parts to be automatically terminated and control to be returned to the FOR statement which has "i" as its index. For example:

```
*1.1 FOR i = 1 TO 100: DO part 2
*2.1 IF mode of a(i)=5, NEXT i.
*2.2 SET a(i)=0
```

In the above program, if a(i) is defined, its value is set to 0. The values of a(i), which were undefined, remain undefined.

The statement LAST i terminates the FOR loop whose index is "i". It is similar to NEXT i in that nested FOR statements are also terminated. The program then continues as though the FOR statement had finished normally, i.e., control is returned to the statement following the FOR loop with index "i" (or the preceding FOR if it is nested in one statement). For example:

```
*1.1 FOR i=1 TO 100: DO part 2
*2.1 IF mode of a(i)=5, LAST i
*2.2 SET a(i)=0
```

The above program sets all defined values of a(i) to 0 until the first undefined element of array "a" is found. The value of "i" is the subscript of the undefined element.

END i terminates the loop on "i" as does LAST i, but control is passed to the statement following END i. When the part containing the END i is exhausted, processing continues at the statement following the FOR on index "i" (or again, the preceding FOR, if it is nested in one step).

Neither LAST nor END change the value of the index.

An example of LAST i and END i follows:

```
*1.1 FOR i=1 TO 10:DO part 2
*1.2 TYPE 1.2,i
*2.1 IF i=3, LAST i; TYPE i
*DO part 1
 i = 1.0
 i = 2.0
 1.2 = 1.2
 i = 3.0
*2.1 IF i=4, END i; TYPE i*i
*2.2 TYPE 'DONE.'
*DO part 1
 i*i = 1.0
 DONE.
 i*i = 4.0
 DONE.
 i*i = 9.0
 DONE.
 DONE.
 1.2 = 1.2
 i = 4.0
```

TRANSFER OF CONTROL

Once execution of a program has begun, the sequence in which steps are executed is determined by the numerical order of step numbers within a part, unless a transfer-of-control statement is encountered. There are two statements that accomplish this; the DO statement and the TO statement.

DO Statement

```
*DO part 5.
```

is used to initiate part 5. If no part 5 has been defined, PIL prints an error comment. For example, consider a program that finds one real root of a quadratic equation ($ax^2 + bx + c = 0$) by:

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

A program to do this calculation might be:

```
*4.1 TYPE 'ENTER THE COEFFICIENTS OF THE EQUATION ax²+bx+c=0.'
*4.2 DEMAND a,b,c
*4.3 DO part 5.
*4.5 TYPE root.
*5.1 * to solve a quadratic equation - real roots
*5.2 SET root = (-b+SQRT OF (b**2 - 4*a*c))/(2*a)
*DO part 4.
```

In this example, part 4 is used to initiate part 5 (at step 4.3). The sequence of execution would be: 4.1, 4.2, 4.3, 5.1, 5.2, 4.5. After completing execution of part 5, control is returned to part 4 at step 4.5. A part is completed when there are no more steps in the part to be executed or when a DONE statement is encountered:

```
*5.3 DONE.
```

Besides the DO part, there is a variation that executes only one step of a part. For example:

```
*1.1 DO step 2.1
*1.2 STOP
*2.1 TYPE "2.1"
*2.2 TYPE "2.2"
*DO part 1
 2.1
 STOP AT STEP 1.2
```

Note that PIL processed step 2.1 and then returned to the normal sequence of processing, that is, to step 1.2.

DO String

The DO STRING command indicates to PIL that the string argument following the command is to be interpreted as a PIL statement and executed. This allows the user to construct statements under program control, and then to execute them. For example:

```
*SET x = "TYPE 2*2"
*DO string x
 2*2 = 4.0
*
```

Here is another example:

```
*set v="all values", x=100, y=The True
*DO string "type "+v
 v = "all values"
 x = 100.0
 y = The True
*
```

TO Statement

The second transfer-of-control statement is the TO statement.

```
*4.3 TO part 5.
```

transfers to part 5. Execution in part 5 would begin with the lowest step number (step 5.1). An important point to remember here is that control of execution is not returned to step 4.5.

The TO statement may also refer to a step number.

```
*4.3 TO step 5.2.
```

will send control to step 5.2 and thereby bypass execution of the first step in part 5. Execution begins with step 5.2 and continues through part 5.

The TO statement is meaningful only in the indirect statement mode. It is an error in direct mode.

DELETION STATEMENTS

Whenever PIL is waiting for input, the user may alter a variable, add or replace any indirect statement, or execute any direct statement. If it is necessary to replace an existing step with a new one, the old step number should be entered, followed by the new statement. The step now contains the new entry; the previous contents of the step are lost.

DELETE Statement

Storage is used by defining variables, steps, and forms (see the explanation in the section on extended I/O list features). Because there is a limit to the amount of storage each user has, it is useful to be able to remove some of these in order to free more storage space. This process is called "deletion".

Variable Deletion

```
*DELETE yooord.
```

will delete the variable "yooord" from the user dictionary and increase available storage. If "yooord" is an array or string variable, this may free a large amount of storage. After execution of the DELETE statement, the variable "yooord" is undefined.

```
*DELETE a,b,c,d(1,2),t(3).
*TYPE a
 Eh? a = ?
*DELETE a
*
```

The first DELETE causes the variables named in the list to be deleted in order from left to right. Unlike the TYPE statement, the second DELETE statement does not produce an error comment.

Like the TYPE statement, an entire array (table) may be deleted by simply mentioning its name. All variables may be deleted by:

```
*DELETE all values.
```

leaving only defined parts, steps, and forms in core.

December 1974

Part and Step Deletion

Parts and steps defined by the user may be deleted selectively by the following statements:

```
*DELETE part 5.
*DELETE step 5.6.
```

The part or step named is deleted and its storage is released. All defined parts will be deleted by:

```
*DELETE all parts
```

thus effectively destroying all programs, but leaving any defined variables and forms in storage.

Multiple steps or parts can be deleted with one command, i.e.,

```
*DELETE step 1.5, part 4
```

If a currently active DO or FOR statement could be deleted, an attempt to RESUME or GO would have unpredictable results. Hence, to delete these statements, it is necessary to deactivate all running parts and FOR statements. This is accomplished by issuing the DONE statement in direct mode. For example:

```
*1.1 DO part 2.
*2.1 STOP
*DO part 1
 STOP AT STEP 2.1
*DELETE step 1.1
 Eh? ACTIVE CONTROL STATEMENT MAY NOT BE CHANGED OR DELETED
*DONE
*DELETE step 1.1
*GO
 Eh? NO PLACE TO GO
```

Once the DONE has been issued, the program must be restarted from the beginning.

Extended Delete

As in the I/O statements (see the section on extended I/O list features) a FOR statement may operate within a DELETE statement:

```
*DELETE a, (FOR i=1 to 10:  matrix(i)),-
& (FOR k=2 by 2 to 12: b(k)).
```

This statement deletes matrix(1) through matrix(10), and the even-numbered elements of b from 2 to 12.

Storage Clean-up

To eliminate everything belonging to the user (parts, values, and forms), the statement:

```
*DELETE all stuff.
      or
*CLEAN
```

is used. After a CLEAN statement, the interpreter will respond with:

```
PIL/2: Ready
```

and the user may begin a new problem. CLEAN is a legal statement in both direct and indirect modes.

PROGRAM STOPS

In the earlier example that solved for roots of quadratic equations, if the user wishes to stop the execution of the program and check the values of a, b, and c before calculating the square-root function, the statement:

```
*5.15 STOP.
```

can be used. When this statement is executed, PIL responds with:

```
STOP AT STEP 5.15
```

PIL will then type an asterisk (*) and wait for further instructions. At this point, the user may issue any command that he would issue normally, say, after an error message. GO or RESUME would restart execution at step 5.2, the first step in part 5 with numerical value higher than 5.15.

The STOP statement is used as part of a program to allow the user to check its progress, to make a change, or to make further additions to his program.

NUMBER Statement

As the user may note, typing part and step numbers for indirect program statements (particularly if the program is long) may be a tiresome task.

Consequently, PIL provides a numbering mechanism via the NUMBER statement. The form is:

```
*NUMBER m, n.
```

This statement starts the numbering with the initial part and step number "m". PIL types the number and then waits for the user to type a statement. If the first non-blank character of the user-written statement is a dollar sign ($), the rest of the statement is taken as a direct statement. It is executed at once, and the step number is not incremented. Otherwise, the statement is stored. The next statement is incremented by "n", which must be in range of 0.0001 to 0.9999. This process continues until another NUMBER or UNNUMBER statement (both in direct mode only) is given. The following sequence illustrates the numbering mechanism:

```
*NUMBER 2.1, .05
* 2.1_SET a=1.
* 2.15_SET b=a+2.
* 2.2_TYPE a, b
* 2.25_$TYPE 1+2
 1+2 = 3.0
* 2.25_$UNNUMBER
*DO part 2.
 a = 1
 b = 3
*
```

The dollar sign in step 2.25 declares to PIL that what follows next (i.e., TYPE or UNNUMBER) is to be taken as a direct statement, and that the current step number should not be incremented. This allows the user to use direct mode, such as $TYPE 1+2, during the numbering mechanism. If both "m" and "n" are omitted in the NUMBER statement, they default to 1.0000 and 0.01, respectively.

```
*NUMBER 99
```

is the same as:

```
*NUMBER 99, .01
```

ERROR REPORTING

Errors encountered in indirect mode are reported with a reference to a step number.

```
ERROR AT STEP 1.5: j = ?
```

would be the message obtained if step 1.5 below contained a reference to undefined variable "j" when the step was executed.

```
*1.5 IF z = 0, SET x = j*y.
```

As long as z does not equal 0, the variables j and y may remain undefined, since the execution of this statement would not require these values. to the user for correction. If the user desires to restart at the beginning, he may do so by issuing another DO part statement, or restart at the point of the error by typing GO or RESUME (GO and RESUME are explained in the next section). For example, using the previous example of solving quadratic equations, if the equation is $x^2 + 4 = 0$, both roots are complex. "a", "b", and "c" are 1.0, 0.0, and 4.0, respectively, when part 5 is entered. The sequence might be:

```
*DO part 5
ERROR AT STEP 5.2: NEGATIVE ARGUMENT FOR SQUARE ROOT FUNCTION
*TYPE a,b,c.
 a = 1.0
 b = 0.0
 c = 4.0
*TYPE -4*a*c.
 -4*a*c = -16.0
*SET c = -4.
*RESUME.
*TYPE root
 root = 2.0
```

From the example, it can be seen that during the correction procedure the data have been changed (effectively changing the problem to $x^2 - 4$). When an error occurs, any direct statement may be issued, or indirect statements may be added or deleted. (The only exception is that active FOR or DO statements may not be deleted.) Steps may be replaced by typing the part and step number followed by the new statement.

There is another statement which is similar in effect to the STOP statement but which is more extreme. This is the ERROR statement:

```
*1.79 ERROR "Square root argument may not be negative.".
```

This statement permits the PIL program to report data inconsistencies (errors), and to halt itself in one program step. This halt is rather severe in that neither GO nor RESUME can restart the program. Thus, the program must be restarted. The above example would produce the following output on the terminal:

```
 ERROR AT STEP 1.79:  Square root argument may not be negative.
```

A variable may be printed in the ERROR statement to construct clearer error messages. The variable name surrounded by plus signs "+" will print the character representation of the variable in the error message. For example,

```
*1.93 ERROR 'No, Mr. "+name+". Your data produces -
&imaginary values.'.
```

PIL is designed so that a user may interrupt his program at any time. After an interrupt or a halt (e.g., a program error or a STOP statement) and before a GO or RESUME, the user is in full control, and can use any part of PIL including a DO statement. If, however, a DO is executed in indirect mode, PIL will "lose its place" in the program where it had been interrupted, and it will not be possible to resume the indirect program at the point of interruption. This problem can be solved by using a special DO statement that "remembers its place". This special DO statement (effective in direct mode only) permits the user to use a GO or RESUME statement to resume the indirect program. In this special form, the particular part or step number appears in parentheses:

```
*DO (part 5).
*DO (step 1.9).
```

Returning to the "root of a quadratic equation" example:

```
*SET a=1., b=0, c=4.0
*DO part 5
 ERROR AT STEP 5.2: NEGATIVE ARGUMENT FOR SQRT FUNCTION
*SET c=-4.0
```

The user corrects "c" and now wishes to execute only one step of the program and then resume. If he types:

```
*DO step 5.2
*RESUME
```

he obtains an error report:

```
Eh? NO PLACE TO RESUME
```

since the DO step statement actually terminated the program. To prevent such a premature termination, he should type:

```
*DO (step 5.2)
*RESUME
*TYPE root
 root = 2.0
*
```

In the "root of a quadratic equation" example, an error condition existed whenever the square root argument was negative. Thus, it is desirable to have the program check to see if the discriminant, i.e., b**2-4*a*c, is negative.

After inserting:

```
*5.15 IF (b**2-4*a*c) < 0, TO step 5.6
*5.6 TYPE "Complex Root.".
*5.5 DONE.
```

the program now looks like:

```
5.1 * to solve a quadratic equation - real roots
5.15 IF (b**2-4*a*c) < 0, TO step 5.6.
5.2 SET root = (-b+SQRT OF (b**2-4*a*c))/(2*a).
5.5 DONE.
5.6 TYPE "Complex Root.".
```

The program now checks for a negative discriminant; if it is negative, control passes to step 5.6 and a message is printed.

PROGRAM RESTART

There are two direct mode statements, GO and RESUME, which are used to restart a program that is stopped because of an indirect STOP statement, an interrupt, or an error. In case of STOP, both GO and RESUME restart processing at the statement logically following the STOP. After an interrupt or an error, there is a subtle but very important difference between GO and RESUME. GO restarts execution at the beginning of the interrupted statement, while RESUME continues from the point of interruption. The following examples show the most important instances where the two commands behave differently.

1. Multiple Set Statement

```
*1.1 SET a=1
*1.2 SET a=a+1,b=c+1
*1.3 TYPE a,b,c
*DO part 1
 ERROR AT STEP 1.2:  c = ?
*SET c=2
```

Now, if the GO command is given, the output is:

```
a = 3.0
b = 3.0
c = 2.0
```

whereas, if the RESUME command is given, the output is:

```
a = 2.0
b = 3.0
c = 2.0
```

GO causes the "a=a+1" component of step 1.2 to be executed twice, and the variable "a" then equals 3, which is incorrect. RESUME restarts where the error occurred, i.e., the "b=c+1" component of step 1.2, and thus "a" has the correct value, 2.0.

2. FOR Statement

If an error occurs in the object statement of an indirect FOR statement, correction of the error and subsequent issuing of the GO statement will cause PIL to begin the FOR statement again. This may cause another rather serious error. Consider the following example:

```
*1.1 FOR i=1 TO 10:  SET a(i)=i.
*1.2 SET SUM = 0.
*1.3 FOR i = 1 TO 11:  SET SUM = SUM+a(i).
*1.4 SET AVG = SUM/10.
*1.5 TYPE AVG.
*DO part 1.
 ERROR AT STEP 1.3:  a(11) = ?
*SET a (11) = 0.
*GO
 AVG = 11.0
```

Upon completion of this program the variable AVG should equal 5.5. However, since the accumulated sum was not reset to zero after correcting the undefined variable error, the variable AVG now equals twice the value, or 11.0. To obtain the correct result, the RESUME command should be given. In this example, RESUME restarts the FOR statement at the point where the error occurred, i.e., with i=11, and thus 5.5 is calculated for AVG.

3. Input/Output Lists

RESUME can be very useful when used with I/O lists. For example:

```
*1.1 DEMAND a,b,c,d

*DO part 1

 a = ?_1
 b = ?_2
 c = ?_"ug!
 ERROR AT STEP 1.1:  INVALID USE OF QUOTATION MARKS
```

GO results in:

```
 a = ?_
```

while RESUME results in the more desirable:

```
 c = ?_
```

so that "a" and "b" do not have to be defined again.

In addition, the output of arrays, parts, and all forms can be interrupted and restarted without danger of duplication with the RESUME command.

EXTENDED I/O LIST FEATURES

TYPE and DEMAND statements may be extended with the following features:

1. I/O FOR lists, which control the iteration within a TYPE or DEMAND statement;

2. FORMs, which declare input and output format specifications; and

3. FREE FORMs, which allow data to be entered without any restriction from format specifications.

I/O FOR Lists

A FOR statement can operate within the standard I/O lists of the DEMAND and TYPE statements. An expression must appear after the colon; therefore, it is illegal to type parts and steps using this construction.

```
*1.8 TYPE (FOR i = 1 TO 5: a(i), b(i)).
*1.9 DEMAND (FOR i = 1 TO 5: abc(i)).
```

This extension is most useful in conjunction with both FORMED and FREE FORMED I/O, as it allows specification of several items in an array without listing them individually. The standard rules for "FOR" apply, including nestings.

Consider the example:

```
*FOR i = 1 TO 5: FOR j = 1 TO 5: SET a(i,j) = i*j.
*TYPE (FOR i = 1 TO 5: (FOR j = 1 TO 5: a(i,j))).
 a(1,1) = 1.0
 a(1,2) = 2.0
   .
   .
   .
 a(5,4) = 20.0
 a(5,5) = 25.0
*
```

This example types all elements from a(1,1),...,a(5,5). The parentheses must be paired, and those around the "FOR" are required in a TYPE or DEMAND statement.

The next example will illustrate a DEMAND with an I/O FOR list:

```
*DEMAND a, (FOR y=1,2:(FOR z=3,4: W(y,z))),b
 a = ?_12.0
 W(1,3) = ?_The False
 W(1,4) = ?_a**2
```

```
 W(2,3) = ?_W(1,4)+sqrt of a
 W(2,4) = ?_-75.32486
 b = ?_"This is a string"
*
```

Notice that both simple variables and I/O FOR lists can be used in the same I/O list.

FORMs

The FORM statement is used to declare input and output format specifications, and thus allows the user to control the appearance of his output line and to extract values for variables from specific areas of his input lines. The form may provide for any number of items on a single line.

A form is declared as follows:

```
*FORM 83.
? x = __.___ Y = ___._
```

The FORM statement is used to enter a FORM DEFINITION. The operand is an integer from 1 to 9999. The first input line following the FORM statement is the FORM DEFINITION, which controls the handling of data. Notice that, for output only, the first character after the question mark (?) is taken as a carriage control character. A blank indicates single spacing, a zero indicates double spacing, and a minus sign indicates triple spacing. For a more complete list of carriage control characters, see MTS Volume 3, Subroutine and Macro Descriptions. Forms can be defined only in direct mode, although forms may be used in indirect mode.

There are several types of allowable specifications, called fields, within a FORM. A description of each type of field follows. Fields are well defined, specifying in which columns to print or demand values.

Numerical Fields--Standard

A standard numerical field is represented by a sequence of underscores (or left-pointing arrows, depending on the terminal) plus an optional period. Each underscore represents a possible digit position of a PIL number. The period represents the decimal point of the number. The user should allow a high-order digit position for a possible minus sign.

Thus, the number -0.05729 when typed in each of the following fields would appear as:

___._____	__.___	_.____	_._	__._____
-0.05729	-0.057	-.0572	-.0	-0.057290

Note that numbers are truncated (not rounded) to fit numerical fields and that trailing zeros are inserted if the number of significant digits are exceeded, as in the last example.

Likewise, the number 9287.423 when typed in each of the following fields would appear as:

```
_______._____   _____._     ___._____  ______     _____.
  9287.42300    9287.4       ********   9287      9287.
```

Note that the fourth field produced an integer. The third field was too small and did not have enough leading positions to contain the number. This is an overflow. The field is filled with asterisks, signifying an error condition, and program execution stops. More will be said about errors later.

Numerical Field--Scientific Notation

The field specifying scientific notation is represented by a series of periods. The width of the field is the number of periods specified, with a minimum of six (6) periods. If less than six periods are specified, the group of periods is regarded as a character string within the form, and is not considered to be a field.

Thus, the numbers 3.141593, 872357.1, -87.23, 0.000000000567, -.85 when typed according to the following form would appear as:

```
...........  ..........  ......  .......  ........  ...
 3.1415E+00   8.723E+05  -8E+01   5.E-10  -8.5E-01  ...
```

Notice that a plus sign is not printed for positive numbers. Also note that the series of three periods is not used as a field but as a character string, enabling the user to use constructions such as ellipsis points in the text.

Numerical Fields--Modified Standard

If the user does not wish to use scientific notation, yet anticipates that some of his results may be either too large (overflow) to fit or too small (underflow) to print any significant digits in a standard numerical field of reasonable size, he can specify a modified numerical field. The number will be printed in scientific notation upon overflow or underflow, thereby preventing an error comment.

To specify this modified numerical field, the user appends to any standard numerical field exactly four exclamation points. If no overflow or underflow occurs, the number is printed out as usual, and the exclamation

points are replaced by blanks. Whenever overflow or underflow occurs, the underscore characters are treated as periods, and the number is printed out as if the entire field were originally specified as scientific notation. The exponent is printed in the positions taken up by the exclamation points.

Therefore, the numbers 234.8961, .07, -94.232, -.000000287 would appear -2.87E+07

```
_._____!!!!   .____!!!!  ____.____!!!!  ___.__!!!!
 2.3489E+02   .0700       -94.2320      -2.870E-07
```

Note that, for scientific notation fields, room for at least six characters must be supplied. Hence, the field specification '_!!!!' is illegal.

Alphanumeric Fields

The field specifying alphanumeric information is indicated by a series of pound signs (#), each #-sign representing one character.

On output, character strings are left-justified in the field. If a string is longer than the field specification, the excess characters are truncated, while strings of a length shorter than the specified field are padded with trailing blanks. Boolean values are typed out as if they were the character strings 'The True' and 'The False'. Numeric values cannot be typed for alphanumeric fields.

Thus, when the values 'This is a STRING', The True, 5>7, "A"+'bCy' are typed in the following form:

```
##############
```

they will appear as:

```
This is a STR
The True
The False
AbCy
```

Note that the expressions 5>7 and "A"+'bCy' are evaluated and only the final values are typed out according to the alphanumeric field.

One warning, however, about attempting to read Boolean values with a character field: the second value above, The True, when appearing in the input line will be treated as the character string "The True", and NOT the Boolean value The True.

Text Material Defined in the FORM

Any characters not recognized as field specifications (with the exceptions given below) are copied directly into the output line until a legal field is found. This enables the user to include text in his form definition. Examples are:

```
*FORM 1.
? Number = _____ Quantity = ___.__
*TYPE in form 1, 12, -3.24.
 Number =   12 Quantity =  -3.24
*TYPE in form 1, 834.
 Number = 834
*
```

Note that the text material is not inserted beyond the end of the last field used; a field stop code permits text occurring after the end of the field definition to be included as part of the field.

Field Stop Code--Field Delimiter

Users frequently desire to have fields adjacent to each other. Yet some means must be available to differentiate the individual fields from each other. A FIELD STOP CODE indicates where the field ends. The field stop code in PIL is the character "|" (vertical bar) or a back slash. The FIELD STOP CODE is only specified in the form definition. It delimits adjacent fields, it does not appear in an output line or take up space in an input line. Therefore, when the field stop code is used, there may not be strict one-to-one correspondence between the appearance of the input/output line and that of the form definition.

The following example will show how the field stop code is used to separate adjacent fields:

```
*FORM 2.
?######
*1.1 DEMAND in form 2, a.
*1.2 TYPE a.
*DO part 1.
?ZEPHYR
 a = "ZEPHYR"
*FORM 3.
?#|#|###|#
*2.1 DEMAND in form 3, (FOR i=1 to 4: b(i)).
*2.2 TYPE b.
*DO part 2.
?ZEPHYR
 b(1) = "Z"
 b(2) = "E"
```

```
 b(3) = "PHY"
 b(4) = "R"
*
```

The field stop code can do more than just separate adjacent fields. Because it indicates where the field ends, the field stop code can be used to enable text following the field to be regarded as part of the field. The following examples will illustrate how trailing text is handled with and without the field stop code:

```
*FORM 4.
? X = __ Pounds Y = __ Ounces
*TYPE in form 4, 10.
 X = 10
*TYPE in form 4, 10, 32.
 X = 10 Pounds Y = 32
*
```

Notice that 'Ounces' is not typed following the '32' because it occurs after the end of the last field used. If it is desired to have the word 'Ounces' printed (and 'Pounds' in the first part of the example), field stop codes must be inserted into the form definition:

```
*FORM 5.
? X = __ Pounds| Y = __ Ounces|
*TYPE in form 5, 92.
 X = 92 Pounds
*TYPE in form 5, 92, 6.
 X = 92 Pounds Y =   6 Ounces
*
```

This is a convenient means of handling text defined in a form statement. Note that the vertical bar "|" does not appear in the output line.

The vertical bar (and also the back slash) may be used as text rather than as a field stop code. This requires the use of two adjacent vertical bars (or back slashes). The two vertical bars are treated as one single character of text, and occupy only one position in the output line. For example:

```
*FORM 6.
? ||A|| = ___ Miles
*TYPE in form 6, 256.
 |A| = 256
*
```

Note that since a vertical bar was not specified after 'Miles', the field ended at the last underscore; 'Miles' was not included as part of the field. Therefore, to cause 'Miles' to be printed out.

```
*FORM 7.
? ||A|| = ___ Miles|
*TYPE in form 7, 256.
```

```
 |A| = 256 Miles
*
```

Special-Purpose Fields

A VARIABLE-LENGTH FIELD, permitted for output only, is specified by a percent sign (%) followed immediately by a numerical or alphanumeric field indicator (# and _, respectively). This must be large enough to contain the longest string or number to be output. The %-sign is regarded as a character in the field definition. Thus, "%___" specifies 4 numerical positions. If the character string is shorter than the field specification, the field is reduced to the same size as the character string. In the example below, the comma is immediately after "Doe", even though the field width may be six characters.

```
*FORM 8.
? Mr.  %#####, you owe us %___ dollars.|
*TYPE in form 8, "Jones", 1728, "Doe", 3.
 Mr. Jones, you owe us 1728 dollars.
 Mr. Doe, you owe us 3 dollars.
*
```

WARNING: Remember, this variable-length field is to be used for output only! It is an error to attempt to use this field during input.

A FLOATING DOLLAR SIGN may also be entered in the form definition. A single dollar sign will be printed just before the first significant digit of the number. The floating $ field is:

```
$$___.__
```

The numerical field specification can be any legal numerical field with at least one digit before the decimal point. The length of the field is the total number of characters in the field definition. Therefore, the above form could be used to print out numbers to 9999.99. .05, 17.06, 1891.09, 1.24, -6.84 would appear in the output line as:

```
   $0.05
  $17.60
$1891.09
   $1.24
  $-6.84
```

The floating dollar signs are permitted for input, but the $-sign is simply ignored. A number, possibly including a minus sign, must appear immediately after the dollar sign.

Character Strings as FORMs

Character strings may be used as form definitions. This allows the user to build form definitions. For example:

```
*SET frm="##".
*FOR i=1 to 64: SET frm=frm+"|##".
*DEMAND in FORM frm, (FOR i=1 to 65: char(i)).
```

or

```
*FOR i=1 TO 3: FOR j=1 TO 3: SET a(i,j) = i*10+j.
*SET nine=" __ __ __"
*TYPE in form nine, a.
 11 12 13
 21 22 23
 31 32 33
*
```

Literal FORMs

A literal form is a form definition which is a character string defined within the I/O statement itself. The character string itself is the form definition. The following are examples of legal literal forms:

```
*TYPE in form " ___ ___ ___", 1, 2, 3.
*TYPE in form " ########", "Zephyr".
*1.2 DEMAND in form "_______ #######", w,x,y,z.
*1.3 TYPE in form " _____"+"._____", 12.4327.
```

Notice that in the last example the form is an expression composed of two literal character strings.

Expressions in FORM References

The reference to a form can be a number, a literal character string, or a variable which is defined as a character string. As shown in the last example above, the form reference can also be an expression. This expression can be any PIL expression that, after being evaluated, is either character or numerical mode. If the operand of the FORM statement is of character mode, the resulting character string is taken as the form definition; if it is numerical mode, the form definition with that number is used. Note that only integers are used (e.g., "TYPE in form 1.7, 37.5" uses form 1). A type of expression other than numerical or character string is an error.

The use of expressions in form references is illustrated by the following examples:

```
*1.84 DEMAND in form alpr+"###", String.
*38.2 TYPE in form " __"+z+".__", 25.5, 87.621
*TYPE in form 1+sqrt of 9, y+9.7+cos of omega
```

As shown in the last example, expressions can occur in the I/O list elements, in addition to the form reference for output statements. In input statements, the I/O list must be a list of single variables.

Formed Output

The TYPE IN FORM statement is used to indicate formed output. The list specification is identical to the TYPE list. The form number may be either a number or an expression, and if the form specified is undefined, an error message will be given and program execution will stop. If the following field is defined:

```
*FORM 9.
? X = __.___ Y = _._______ ##### = ___
```

the statement:

```
*TYPE in form 9, 27.4032, -.047, "Alpha", 76.
```

will type the following line:

```
 X = 27.403 Y = -.047000 Alpha =  76
```

The value of each item in the I/O list is inserted into the output line according to the corresponding field specification. The form is scanned from left to right for fields, and the values are placed into the output line one item at a time. It is an error to attempt to output numerical values in an alphanumeric field; likewise, it is an error to attempt to output Boolean values and character strings in a numeric field. If no fields are defined within a form, it is an error to use that form in FORMED I/O.

If more items are to be typed out than are defined in the form, the line is typed, then the form is rescanned from the beginning until all items in the I/O list have been printed. Thus, an N by N matrix, with a form of N fields, is typed out in row form (one row per line). This is shown in the following example:

```
*FORM 10.
? __ __ __
*FOR i=1 to 3: FOR j=1 to 3: SET a(i,j)=i*10+j.
*TYPE in form 10, (FOR i=1 to 3: (FOR j=1 to 3: a(i,j)))
 11 12 13
```

```
 21 22 23
 31 32 33
*
```

Note carefully how the I/O FOR lists are constructed. The enclosing parentheses must be paired and are required.

Formed Input

The DEMAND IN FORM statement is analogous to the TYPE IN FORM statement, and most lines created by the TYPE IN FORM can be reread by the DEMAND IN FORM. However, it must be noted that:

1. A NUMERIC FIELD merely indicates that numerical input is expected; no alignment of data within the field with respect to a decimal point is necessary. Also, no scaling of unaligned fields will be performed. No intervening blanks are permitted.

2. An ALPHANUMERIC FIELD designates a sequence of characters, which may include primes (') or quotes ("). Such a sequence should not be enclosed within delimiters as in a simple DEMAND, (DATA not "DATA".)

3. The input list must be completely satisfied. It is an error if a DEMAND IN FORM statement with four input parameters and with a form having four fields receives only three input items on the line. However, if the width of a field is longer than the variable, then the width of the field is effectively shortened to equal the number of characters in the variable.

4. If a RESUME is given after an error has occurred, the next item to be read from the input line will be located in the position occupied by the field where the error previously occurred.

5. Null lines produce an error.

6. The fields are well defined; they not only specify the length of the field, but also the columns in which the input must be found.

The following example will demonstrate FORMED INPUT:

```
*FORM 11.
? _____   ####   ..........
*9.1 DEMAND in form 11, a,b,c,d,e,f.
*DO part 9.
? -7.3   String       9
?    46   HELP
                  ***********
 ERROR AT STEP 9.1: NO DATA FOR INPUT FIELD
*RESUME
```

```
?                   -5e-9
*TYPE all values.
 a = -7.3
 b = "trin"
 c = 9.0
 d = 46.0
 e = "HELP"
 f = -5.000000E-09
```

Error and Resume with Formed I/O

As in all PIL statements, errors sometimes occur. Whenever this happens during formed I/O, information is printed as to the nature of the error and where it occurred. If an error arises during formed output, the line is printed with asterisks inserted in the field where the error occurred. If any previous items were successfully inserted into the output line, they will appear in their proper place before the asterisks, along with any text defined in the form. All processing stops whenever an error occurs, and nothing is printed in the output line after the asterisks. An example of an error during formed output, and one means of correcting the error, follows:

```
*SET hgt=19.
*FORM 12.
?Height = ___ feet| Weight = ___ pounds|
*87.4 TYPE in form 12, hgt, wgt.
*DO part 87.
 Height =  19 feet Weight = ***
 ERROR AT STEP 87.4: wgt = ?
*SET wgt=97.
*GO
 Height =  19 feet Weight =  97 pounds
```

In this case GO was used. Had RESUME been used instead, the results would have looked like:

```
*SET wgt=97.
*RESUME
                    Weight =   97 pounds
*
```

RESUME causes the output to continue at the item that produced the error. Any text associated with the field is printed out, as indicated in the example above. Any items that were printed before the error are left blank. When using RESUME, output processing continues at the field occupying the same relative position in the form as the one in which the error occurred (i.e., if the error occurred in the fourth field, output processing will then resume at the fourth field). Thus, if a field specification is the wrong type or size, the form can be altered, allowing output to continue with the corrected form. As an example:

```
*FORM 13.
? Length is __ km| __ m| __ cm|
*9.2 TYPE in form 13, 12, 347, 18.
*DO part 9.
 Length is 12 km **
 ERROR AT STEP 9.2: NUMBER EXCEEDS SPECIFIED FIELD WIDTH
*FORM 13
? Length is ___ km| ___ m| ___ cm|
*RESUME
                  347 m   18 cm
*
```

If the form is changed to one having a fewer number of fields, and the number of the field at which PIL is to resume is greater than the number of fields presently defined, field counting wraps around to the beginning of the form. The form is rescanned as many times as necessary. For example, if form 13 above is redefined to be:

```
*FORM 13.
? Length is ___
```

and a RESUME is given, the field count would restart to the beginning of the form. Thus, the first (and only) field is where the output will continue:

```
*RESUME
 Length is 347
 Length is  18
*
```

Errors are handled in a similar way for formed input. If an error occurs (after an input line has been read), an error message will be typed out. Errors can be caused by not entering data according to the field specifications (i.e., either entering data of the wrong type or not having the data properly aligned with the field). Errors can also occur from other sources, such as an undefined variable in the I/O list. An example of this type of error is:

```
*1.1 DEMAND in form "_____ _____", a, b(k(7)).
*DO part 1.
? 275 -874
      ****
 ERROR AT STEP 1.1: k(7) = ?
*SET k(7)=4.
*RESUME
?      -874
*TYPE a, b(k(7)).
 a = 275.0
 b(4) = -874.0
```

Free Formed I/O

The normal DEMAND statement is inconvenient for large amounts of data; and the use of formed I/O, which greatly speeds up terminal output, is awkward to use for terminal input. Therefore, when entering large amounts of data from the terminal, it is recommended that the input be read in FREE FORM.

There are two complementary statements in FREE FORMED I/O: TYPE IN FREE FORM and DEMAND IN FREE FORM. Any line created by a TYPE IN FREE FORM can be reread by a DEMAND IN FREE FORM. An example of free formed output is:

```
*TYPE in FREE FORM, 12, 7**2, "Hello", 1>5.
 12.0, 49.0, "Hello", The False
```

Notice that items are separated from each other by a comma and a space and that the output is compact. Also note that Boolean values are printed (and likewise read) as Boolean values, not as character strings as in formed I/O.

The following example illustrates free formed input:

```
*13.4 DEMAND in free form, w, x, y.
*13.5 TYPE all values.
*13.6 TO step 13.4.
*DO part 13.
?  12, 14.7, "1492"
 w = 12.0
 x = 14.7
 y = "1492"


? -87.426e+27, The True, "String
                         *******
 ERROR AT STEP 13.4: IMPROPER USE OF QUOTATION MARKS
*RESUME
? 'String'
 w = -8.742600E+28
 x = The True
 y = "String"
```

Acceptable items for use during free formed I/O are:

NUMBERS	Any legal type.
CHARACTER STRINGS	Enclosed by quotes (') or (").
BOOLEAN VALUES	The True and The False only.

Rules Governing Free Formed Output

1. Each item is placed into the output line and followed by a comma and a blank. Successive items are inserted until the end of the output line is reached, and the line is printed. This depends on the maximum line length. The next item begins a new line, starting in column 2. This process is repeated until all items in the I/O list are printed.

2. The last item of the I/O list is not followed by a comma. This identifies the last line printed from a single TYPE statement, since lines that are continued always end with a comma.

3. Column 1 is always blank.

4. The output line can be as long as the maximum output length (depending on terminal type) or 255, whichever is less.

Some examples are:

```
*1.1 TYPE IN FREE FORM, (FOR i=1 TO 50: i)
*1.2 TYPE IN FREE FORM, i=70, "abcdeF",1*2*3*4
*1.3 TYPE IN FREE FORM, "This is the 3rd one".
*DO part 1.
 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0,
 13.0, 14.0, 15.0, 16.0, 17.0, 18.0, 19.0, 20.0, 21.0, 22.0, 23.0,
 24.0, 25.0, 26.0, 27.0, 28.0, 29.0, 30.0, 31.0, 32.0, 33.0, 34.0,
 35.0, 36.0, 37.0, 38.0, 39.0, 40.0, 41.0, 42.0, 43.0, 44.0, 45.0,
 46.0, 47.0, 48.0, 49.0, 50.0
 The False, "abcdeF", 24.0
 "This is the 3rd one"
*TYPE i
 i = 51.0
*DELETE all values
*TYPE IN FREE FORM, 10, 10**2, 10**3, notdef.
 10.0, 100.0, 1000.0,
 Eh? notdef = ?
*
```

Note that, when an error occurs, asterisks are not printed out as in FORMED I/O, since no field is associated with the variable, and the width of the printed value is undefined.

Rules Governing Free Formed Input

1. This method provides a convenient way to read data with more than one item per line. Data are read from the input line one item at a time until the end of the line is reached. A new line is read if all variables have not yet been assigned values.

2. Items must be separated from each other by a valid delimiter, which may be a comma, one or more blanks, or a combination of both.

3. Every DEMAND starts by reading a new line; unused items left over from a previous line are not used.

4. Quotes must appear around character strings.

5. When a slash (/) is encountered which is not a part of a character string, the remainder of the line is treated as a comment, and a new line is read.

6. A blank line is ignored. A null line is an error.

7. When commas are used as delimiters and they appear adjacent to each other or with blanks between, the value of the variable read is left unchanged; superfluous commas are not ignored. This enables the user to read in new values for some list elements, and at the same time leaves other list elements set to their previous values. The first variable read is left unchanged whenever the first non-blank character in a line is a comma. An example is given on the next page.

8. An asterisk (*) which is not inside a character string and which is not the last non-blank character in the input line terminates the DEMAND. The rest of the I/O list elements are unchanged.

9. The Boolean values, The True and The False, can be in uppercase or lowercase, or any combination thereof, but must be two separate words.

The following are examples of free formed input:

```
*1.1 DEMAND IN FREE FORM, a, b, c.
*1.2 TYPE a, b, c.
*1.3 TO step 1.1.
*DO part 1.
?5, "Mary's" 10.0
 a = 5.0
 b = "Mary's"
 c = 10.0
?, 84, ,
 a = 5.0
 b = 84.0
```

```
 c = 10.0
?THE true, 106.9 /This is a comment!
? "Abcde"
 a = The True
 b = 106.9
 c = "Abcde"
?, -72.4 /Another comment.
?, /Both a and c are unchanged
 a = The True
 b = -72.4
 c = "Abcde"
? 1 * The "*" (plus something) terminates the demand
 a = 1.0
 b = -72.4
 c = "Abcde"
?1, 2, False /This will cause an error--rule 9
       *****
 ERROR AT STEP 1.1: IMPROPERLY FORMED CONSTANT
*RESUME
?THE FALSE
 a = 1.0
 b = 2.0
 c = The False
? 57.4
?
?
?18.6 12 /Note that blank lines are ignored
 a = 57.4
 b = 18.6
 c = 12.0
? <<null line--see rule 6>>
 ERROR AT STEP 1.1: NO DATA IN INPUT LINE
*
```

TYPE FORM N

The TYPE FORM statement can be used either to examine a form for errors or to print a header line entered as a form. No identification is typed with the form.

For example:

```
*FORM 14.
?  FEET    INCHES
*FORM 15
?   __      ___
*1.1 TYPE "", form 14, "".
*1.2 TYPE in form 15, (FOR i=1 TO 8: i, i*12)
*DO part 1.
```

```
 FEET    INCHES

   1       12
   2       24
   3       36
   4       48
   5       60
   6       72
   7       84
   8       96
*
```

TYPE ALL FORMS

The TYPE ALL FORMS statement types out all defined forms in ascending numerical order. For every form, the form number will be typed, followed by the form definition. This is printed in the same two-line format used for entering forms.

Form Deletion

Forms may be deleted using the DELETE statement. To delete a specific form, specify the form reference in the DELETE statement. To delete all defined forms, specify ALL FORMS in the DELETE statement. Examples illustrating these are:

```
*DELETE form 10.
*DELETE form 72, form 85, form 3.
*DELETE all forms.
```

AUXILIARY I/O

Peripheral I/O equipment (devices) are available in the system. They may be used when a user wishes to have data read from or written to a device other than his remote terminal.

To access the auxiliary I/O equipment, PIL has incorporated several statements to assign devices to a job, read from or write onto them, or perform control operations on certain devices (e.g., REWIND, END FILE, etc.). All examples in the following section use string literals for file or device names ("FDname"), and numbers for assignment numbers ("asst"). Wherever these occur, appropriate expressions may be substituted.

All I/O in PIL is done through assignments. To be used, all devices must be assigned within PIL or attached to MTS logical I/O units 0 to 19. File or device names must be string expressions. No embedded blanks are permitted, and the case of letters is not significant. The name may be as long as the user wants and may terminate with a blank. PIL assignment numbers must evaluate to an integer value in the range of 0 to 99 (the fractional part is discarded). The assignment numbers from 0 to 19 correspond to MTS logical I/O units 0 to 19, respectively. The others refer to particular files or devices not attached to MTS logical I/O units.

Devices in PIL must be associated to assignment numbers:

```
*ASSIGN "*PRINT*" TO 7.0.
```

will assign a HASP pseudo-device name "*PRINT*" to assignment number 7. This name refers to a remote line-printer. In general, the user defines the assignment number by the ASSIGN statement:

```
*ASSIGN FDname TO asst.
```

where "FDname" is a string expression representing a file or device name and "asst" is an assignment number. For example:

```
*ASSIGN 'file' TO 4.
*ASSIGN 'A' TO 5., 'B' TO 6.0.
```

Notice that multiple assignments can be made in a single statement. These assignments must be separated by a comma.

Magnetic tapes or paper tapes must be mounted by the operator. This should be done through the MTS MOUNT command, e.g.,

```
*MTS "mount pool on 9tp *tape*"
```

See the description of the MOUNT command in MTS Volume 1: MTS and the Computing Center. A brief description of the MTS statement in PIL is in Appendix B in the back of this manual.

WRITE Statements

Writing onto an assignment number is accomplished by:

```
*WRITE ONTO 4, a,b,c,d,e.
```

The statement

```
*WRITE ONTO 89, all parts.
```

will transmit to assignment 89 all currently defined parts.

Any statement that is available with TYPE is also available with WRITE. Conceptually, the TYPE may be replaced with WRITE ONTO 7 with a comma (,) following the assignment number. Thus,

```
*1.1 WRITE ONTO 7, all parts.
*1.2 WRITE ONTO 7, all forms.
*1.3 WRITE ONTO 7, in form 16, A,B,C,REG.
```

are all legal, executable statements.

It should be mentioned that the form of output to these files is exactly the same as that obtained from the TYPE command. Like TYPE, WRITE may be executed either in direct or indirect mode. Before first writing onto an assignment number, the user may remove the prior contents of the specified assignment by issuing an EMPTY statement (see "Control Operations" on the following pages).

READ Statements

As WRITE is to TYPE, so READ is to DEMAND. The two statement forms, in fact, complement one another.

```
*READ FROM 8, xsize.
```

reads one record and stores the result into the variable xsize.

The flexibility that a user has with DEMAND is also available with READ. Again, we may consider DEMAND as being replaced with READ FROM 8 with a comma (,) following the assignment number.

Thus:

```
*1.2 READ FROM 8, in free form, A, J, Ysl.
*1.3 READ FROM 8, in form 12, A, B, X.
```

are legal and meaningful statements.

PIL currently treats the end-of-file condition as an error and types:

```
Eh? END OF FILE.
```

At this point, control is returned to the user at the terminal, where he may resume calculations. Like the DEMAND statement, the READ statement can be used either in direct or indirect mode.

Control Operations

File control operations are available to allow the user to position an assignment at any chosen point.

```
*REWIND 3.
```

rewinds the file or device associated with assignment number 3 and places the user at the beginning of the file. File or device names may also be specified in place of assignment numbers such as:

```
*REWIND 'myfile'.
```

The endfile control operation, as in

```
*END FILE '*tape*'.
*END FILE 8.
```

writes a file mark on the designated assignment and is used as the logical terminator for subfiles within a file.

```
*BACK SPACE 6 FILES ON 7
```

back-spaces 6 files on assignment 7.

```
*FORWARD SPACE (n-1) RECORDS ON 9.
```

forward-spaces the assignment (n-1) records.

The END FILE operation is considered to be a WRITE operation; the BACK SPACE and FORWARD SPACE operations are considered as READ operations.

Any legal PIL expression may be used to calculate the number of files or records to be spaced forwards or backwards. But again, only the integer portion of the expression will be used. Thus, if n = 16.9879, the file would be forward-spaced or back-spaced by 16 records. Negative numbers are not allowed.

The EMPTY statement empties an MTS file. Unlike the MTS EMPTY command, no confirmation is demanded.

```
*EMPTY 'pilfile'
```

Assignment numbers may be specified in the EMPTY statements:

```
*EMPTY 2.
```

The file referred to by the assignment 2 is emptied. Devices other than files cannot be emptied.

The CONTROL statement allows execution of control operations on certain types of files and devices. It is usually written:

```
*CONTROL str ON FDname.
*CONTROL str ON asst.
```

where "str" represents a string mode control command parameter to be performed on either FDname or "asst". Examples:

```
*CONTROL 'rew' ON '*tape*'.
```

rewinds the 9-track magnetic tape named *TAPE*.

```
*CONTROL 'uc=off' ON '*sink*'.
```

forces the pseudo-device *SINK* to turn off automatic capitalization.

The CREATE statement permits the user to create an MTS file. The form is:

```
*CREATE FDname
```

where "FDname" represents a file name in the form of a string expression. The following creates an MTS line file named PILFILE with a default file size of one page. (See MTS Volume 1, MTS and the Computing Center for additional information on MTS.)

```
*CREATE "pilfile".
```

The user may wish to specify the type of a file. He may type:

```
*CREATE "pilfile", TYPE 'line'.
```

Other legal file types are 'SEQ' for sequential files and 'SEQWL' for sequential files with line numbers. To create a file which is larger than one page, the user can specify:

```
*CREATE 'pilfile', type 'line', size 25.
```

where the size refers to 4096-character pages.

For the CREATE statement, PIL will check to make sure that a file of the given name does not exist. If it does exist, a complaint will be made. Otherwise, PIL proceeds to check the user's file space allocation to determine if there is enough space remaining to allow creation of the file. It will then attempt to get the disk space for the specified size. If PIL is successful in its attempt, the user is informed of the creation of the file.

The DESTROY statement permits the user to destroy a private file or a temporary file. Unlike the MTS command DESTROY, PIL will NOT ask for confirmation. The form is:

```
*DESTROY FDname
```

where "FDname" specifies the name of the file to be destroyed. "FDname" may be a literal string (enclosed in quotes or double-quotes), or may be a character variable. An example would be:

```
*DESTROY "pilfile"
```

A complaint will be made if the file named does not exist. Otherwise, the file to be destroyed is deleted from the user's file catalog, and the disk space is released. The user is then informed of the successful destruction.

Any file accessible to the user may be renamed by the RENAME statement.

```
*RENAME "oldfile" AS "newfile".
```

Here the file name OLDFILE is changed to NEWFILE.

DELETE ASSIGNMENT N

The DELETE statement includes features to allow the user to return a device to the system.

```
*DELETE assignment 20.
```

deletes the assignment number 20. Any future references to assignment number 20 would generate an error since the assignment is now undefined. All other assignments 0 to 19, if previously specified, are retained by MTS, and may therefore be used again.

TYPE ASSIGNMENT N

The user may list all defined assignments and the devices or file names associated with them via the TYPE statement.

```
*TYPE all assignments.
```

lists the currently defined assignment numbers and their associated devices or file names. Or the user may type a specific assignment:

```
*TYPE assignment 8.
 8 = "-FILE"
```

PROGRAM MANAGEMENT

For the user who desires more flexible control over certain system functions, such as pagination of output, program saving, and program loading, PIL has several statements related to these functions, although they are not specifically related to program logic.

Pagination

If a user specifies nothing with regard to pagination, he will use every line of a page. This defaults to 66 lines per page. He may request, however, that PIL keep an accounting of lines and format pages so that they could fit into a notebook. In this case, PIL will supply certain reference information such as a page heading. To obtain pagination, the user types:

```
*PAGES YES.
```

PIL will then respond with:

```
Set paper to second line from top, hit return.
```

After the carriage return is struck, PIL will start pagination with:

```
Page   1  USERID
05-07-74
12:41.34
```

The last line of the page heading is the time of day on a 24-hour basis (00:00.00 is midnight), and USERID is the user's ID number.

After this, PIL will keep track of pages, supplying a page skip when necessary. It will also begin a new page when the word PAGE appears as either a direct or indirect statement. The page control program assumes that single spacing is used at the terminal.

Pagination may be turned off by the statement:

```
*PAGES NO.
```

If it is desired to use other than a standard page size, the PIL user can specify the number of lines to be printed per page.

```
*PAGES YES 33.
```

would set the page size to 33 lines. Thirty-three lines with a line space lever set at double spacing gives a page with same physical size as a normal single-spaced page. Pages should be set to the number of lines that can be printed on the physical page. The page size must be greater than 15 but

fewer than 99 lines per page; it includes page heading lines that are produced. The default is 66 lines for a terminal or 60 lines for a line printer.

SPACE and LINE are two statements that help the user space down the specified number of blank lines. Both can be used either directly or indirectly.

To space a single line, the statement

```
*SPACE.
```

can be used.

```
*SPACE 5.
```

spaces five lines. If the number of blank lines to be spaced would exceed the page size, a new page is begun, and the blank lines are ignored.

LINE moves the page to a desired line number of the current (or the next) page.

```
*LINE 5
```

If the currect line is 2, it skips to line 5. If the current line is 15, it skips to the fifth line after the heading of the next page. The statement LINE with no parameter:

```
*LINE.
```

spaces just one line.

If the user moves the paper himself, PIL will not detect it, and the pagination will be incorrect.

Program Saving

A program may be saved by a user for use at some later time by the SAVE statement:

```
*SAVE AS 'fname', all parts.
```

will save all currently defined parts as the file "FNAME". "FNAME" must already be created. The name is specified by a string expression and must always be followed by a comma (,). Note that a file name is forced to uppercase, so xYz is the same file as XyZ. Assignment numbers from 0 to 19 may be used instead of such file names.

The object part of the SAVE statement is an almost complete subset of the TYPE output list. Only "all assignments" or "assignment n" are not valid. Summarized below is a representative list of possible object statements:

1. a list of variables. (a,b,c,d,e,f.)
2. part n. (part 2.)
3. step n. (step 16.4)
4. all parts.
5. all values.
6. all forms.
7. all stuff.
8. part n, step 5.3, all forms, 9.

Also, any combination of the above forms separated by commas may be used. If the user wishes to define a program as all parts and forms, then the statement:

```
*SAVE AS 'Fname', all parts, all forms.
```

will save the program as FNAME. SAVE is allowed in both the direct and indirect mode. It is recommended that the user empty the specified file before using the SAVE statement.

Program Loading

A program may be loaded into the user's core by the LOAD statement. If the program is on a file referred to by the above SAVE statement:

```
*LOAD "fname"
```

will load the contents of the specified file into the user's core storage.

Many times a program will be created and saved on a terminal with lowercase capabilities and later loaded and further debugged on a terminal without lowercase capabilities. This causes difficulties if variables must be examined or altered in direct mode. To change such a program to uppercase while loading, append the MTS modifier @UC to the file PROGRM:

```
*LOAD "PROGRM@UC"
```

The LOAD statement is valid in both direct and indirect modes. Direct statements, including another LOAD, may be specified in a file that is to be loaded. No more than 31 LOAD statements can be active at the same time. If the user has parts, forms, and values defined and requests a program to be loaded, PIL will merge the two programs.

There are two basic rules to remember when merging:

1. Any value, form, or step already defined will be redefined by the new program being loaded if a value, form, or step of the new program has the same name or number.

2. If the variable names, forms, or step numbers of the program being loaded are different from those defined in the loading program, then the loading program will remain intact.

It is recommended that the user not interrupt his terminal during a LOAD sequence, as the results placed in storage are then unpredictable. The loading sequence can also be abnormally terminated if any error occurs.

December 1974

Tables

Table 1. Arithmetic Operators and Functions

OPERATOR		MEANING	EXAMPLE
Short	Long		
+		Addition	a + b
-		Subtraction	c - d
*		Multiplication	a * c
/		Division	b / d
**		Exponentiation a^b	a**b
=		Replacement	x = y
| |	Abs of	Absolute Value	|a - b|
Sqrt of	Square Root of	$\sqrt{x}$	Sqrt of x
Sin of	Sine of	Sin x	Sine of x
Cos of	Cosine of	Cos x	Cos of x
Log of		$\text{Log}_{10} x$	Log of x
Antilog of		10^x	Antilog of x
Ln of		$\text{Log}_e x$	Ln of x
Atan of	Arc Tangent of	Inverse Tan x	Atan of x
Exp of		e^x	Exp of x

OPERATOR		MEANING	EXAMPLE
Rn of	Random number of	Random number generator[1]	Rn of x (changes x)
Ip of	Integer part of	Integer part of a number	Ip of -3.5 = -3.0
Fp of	Fraction part of	Fraction part of a number	Fp of 3.5 = 0.5
Xp of	Exponent part of	Power of 10 scaling	Xp of 3.5 = 1.0
Dp of	Digit part of	Dp of x = x/(10**xp of x)	Dp of 35 = 3.5
Min of	Minimum of	Least value[2]	Min of (a,b,c)
Max of	Maximum of	Greatest value[2]	Max of (a,b,c)
The Size		Current available space[3]	
The Total Size		Total work space	
The Time		Time in 300ths of a second relative to 00:00 (midnight)	The Time = 1.296000E+07 (i.e., noontime)
The Date		Day of year in form YYDDD where YY refers to the year (19YY) and DDD is day of year.	The Date = 74236.0 (i.e., August 24, 1974)
The Elapsed Time		User's actual computer connected time in 300ths of a second from the signon time.	

OPERATOR	MEANING	EXAMPLE
The CPU Time	User's actual computer usage time in 300ths of a second from the signon time	
The Cost	User's cost in dollars from the signon time	

---

[1]When the parameter of RANDOM NUMBER OF is a single variable (or subscripted), e.g.,

```
*SET y = rn of x
```

the value of the parameter (x) changes unpredictably and should not be altered if the longest possible sequence of pseudo-random numbers is desired. The value of the function, y in the above example, is uniformly distributed over the interval 0 to 1.

[2]Min and Max may have any number of arguments and are defined for string as well as for numerical mode.

[3]The Size does not always present an accurate picture of the amount of space available because some temporary space is needed by PIL to execute a program. A few PIL cells are reserved exclusively for temporary space but still are included in the figure reported by The Size function.

Table 2. Boolean Operators

OPERATOR		MEANING	EXAMPLE
Short	Long		
<	$lt	less than	a < b
<= ¬>	$le	less than or equal to	b <= c
=	$eq	equal to	c = d
¬=	$ne	not equal to	b ¬= c
>= ¬<	$ge	greater than or equal to	c >= b
>	$gt	greater than	d > e
&	$and	logical product	a < b $and c = d
#	$or	logical sum	a > b $or c = d
¬	$not	logical negation	$not a < b
	$xor	exclusive or	a $xor b
	The True	constant true	
	The False	constant false	
	The Batch	The True if in batch, else The False	

Table 3. String Operators and Functions

OPERATOR		MEANING	EXAMPLE
Short	Long		
"		String Delimiter	"abc"
’		String Delimiter	’abc’
+		Concatenation	"a" + "b"
L of	Length of	Length of a character string	L of "ab" = 2.0
Upper of	Upper case of	Force string to uppercase	Upper case of "aBc" = "ABC"
Lower of	Lower case of	Force string to lowercase	Lower case of "Abc" = "abc"
n $fc a	The first n characters of a		2 $fc a
n $lc a	The last n characters of a		2 $lc b
Subs of (s, b, 1)	The Substring of (s, b, l)	A substring of string s, starting at character b with length l	The substring of ("abc",2,1)="b"
	THE BCD Time	Time of Day with format "hh:mm.ss"	The BCD Time = "12:00.00"
	The BCD Date	Day of Year with format "mm-dd-yy"	The BCD Date = "09-24-74"
VL of	The value of	Evaluate string as a PIL expression	
BCD VL of	The BCD value of	Convert operand to string value	
	The User	The user’s ID	

Table 4. Precedence Order

The PIL precedence rules place the various PIL elements in the following order (high to low):

Functions, $fc, $lc, |...|

**

unary -, unary +

*, /

binary +, binary -

$le (¬>, <=) , $lt (<), $ne (¬=), $eq (=), $ge (¬<, >=), $gt (>)

$not (¬)

$and (&)

$or (#), $xor

assignment =

For equal-precedence operators in a subexpression, the subexpression is evaluated from left to right. Functions (as well as $fc, $lc, and |...|) are evaluated from right to left. Enclosing parentheses may be freely used to change the implied order of expression evaluation.

Appendixes

Appendix A. Summary of PIL Statements

Below is a complete list of PIL statement keywords. Letters underscored indicate the minimum abbreviated form for these words. All other keywords, including function names, can be abbreviated only to their first four letters.

Direct or Indirect	Direct Only	Indirect Only
ASSIGN	FORM	TO
BACK SPACE	GO	
CLEAR	PAGES	
CONTROL	RESUME	
CREATE		
DELETE		
DEMAND		
DESTROY		
DO		
DONE		
EMPTY		
END		
END FILE		
ERROR		
FOR		
FORWARD SPACE		
IF		
LAST		
LINE		
LOAD		
MTS		
NEXT		
NUMBER		
PAGE		
READ		
RENAME		
REWIND		
SAVE		
SET		
SPACE		
STOP		
SWAP		
SYSTEM		
TYPE		
UNNUMBER		
WRITE		

December 1974

Appendix B. The Michigan Terminal System

There are several PIL statements that help the user to interface with MTS.

The SYSTEM statement, whether in direct or indirect mode, always terminates the PIL/2 session and causes a return to MTS (Michigan Terminal System). If the user desires to use PIL/2 again, he must rerun PIL/2. An alternate for the SYSTEM statement is a STOP statement in direct mode only. Example:

```
*SYSTEM.
*STOP.
```

There are many operations that only MTS can do; for example, mount a magnetic tape, permit a file for users, edit a line file, or copy a file to another file. Consequently, PIL provides an MTS statement. The general form is:

```
*MTS s.
```

where "s" stands for a string expression representing any legal MTS command. (Note that string expressions must be enclosed by delimiters.) For example, to list a file:

```
*MTS '$list pilfile'
#$LIST PILFILE
>
> SET a = 1
>
*
```

When the MTS command is completed, PIL regains control, unless the command given is a RUN, DEBUG, LOAD, or SIGNOFF, etc. Another example:

```
*MTS "mount c000 9tp *t* 'abc'".
```

requests an operator to mount a nine-track tape with rack name c000 and ID 'ABC'. The tape is then referred to by pseudo-device name *T*.

Another form of the MTS statement is:

```
*MTS
```

In this case, PIL returns to MTS, but PIL may be $RESTARTED to give PIL control again. An example of this is:

```
*MTS
#permit pilfile read others
#restart
*
```

Here the user permits all other users to read his own file PILFILE.

A full description of MTS commands can be found in MTS Volume 1, MTS and the Computing Center.

Appendix C. EBCDIC Character Set

The collating sequence of characters is indicated by their hexadecimal values as shown below.

Hex Value	Char-acter	Hex Value	Char-acter	Hex Value	Char-acter	Hex Value	Char-acter
40	Space	81	a	C1	A	F0	0
4A	¢	82	b	C2	B	F1	1
4B	.	83	c	C3	C	F2	2
4C	<	84	d	C4	D	F3	3
4D	(	85	e	C4	D	F4	4
4E	+	86	f	C6	F	F5	5
4F	|	87	g	C7	G	F6	6
50	&	88	h	C8	H	F7	7
5A	!	89	i	C9	I	F8	8
5B	$	91	j	D1	J	F9	9
5C	*	92	k	D2	K		
5D	)	93	l	D3	L		
5E	;	94	m	D4	M		
5F	¬	95	n	D5	N		
60	-	96	o	D6	O		
61	/	97	p	D7	P		
6B	,	98	q	D8	Q		
6C	%	99	r	D9	R		
6D	_	A2	s	E2	S		
6E	>	A3	t	E3	T		
6F	?	A4	u	E4	U		
7A	:	A5	v	E5	V		
7B	#	A6	w	E6	W		
7C	@	A7	x	E7	X		
7D	'	A8	y	E8	Y		
7E	=	A9	z	E9	Z		
7F	"						

December 1974

INDEX

<, 26
<=, 26

+, 21, 29

|, 21, 56

&, 26

#, 26

$AND, 26
$EQ, 26
$FC, 29
$GE, 26
$GT, 26
$LC, 29
$LE, 26
$LT, 26
$NE, 26
$NOT, 26
$OR, 26
$XOR, 26

*, 21
**, 21

¬, 26
¬<, 26
¬>, 26
¬=, 26

-, 21

/, 21

>, 26
>=, 26

=, 26

Absolute value, 21
Addition, 21
Alphanumeric field, 55
ANTILOG OF X, 22
ARC TANGENT OF X, 23
Arithmetic expressions, 21
Arithmetic functions, 78
Arithmetic operators, 21, 78
Arrays, 19, 35
ASSIGN, 69
Assignment numbers, 68
Asst, 68
ATAN OF X, 23
Attention interrupts, 12
Auxiliary I/O, 68

BACK SPACE, 71
BCD VL, 31
Boolean constants, 17, 18
Boolean expressions, 25
Boolean operators, 27, 81
BY, 38, 39

Carriage control character, 53
Character constants, 17
Character expressions, 27
Character strings as forms, 59
CLEAN, 46
Comments, 15, 32
Concatenation, 29
Conditional statement, 33
Constants, 17
Continuation line, 11
Control operations, 71
CONTROL statement, 71
COS OF X, 22
COSINE OF X, 22
CREATE statement, 72

Delete-line, 11
Delete-previous, 11
DELETE all parts, 45
DELETE all stuff, 46
DELETE all values, 44
DELETE ASSIGNMENT N, 73
DELETE part n, 45
DELETE statement, 44, 45
DELETE step n, 45
DEMAND IN FORM, 61

DEMAND IN FREE FORM, 66
DEMAND statement, 35, 36, 52
DESTROY statement, 72
Device name, 68
DIGIT PART OF X, 23
Direct mode, 8, 14
Direct statement, 47
Division, 21
Do (part n), 49
DO (step n), 49
DO part n, 15, 42
DO statement, 42
DO step n, 42
DO string, 43
DONE (direct), 45
DONE (indirect), 42
DP OF X, 23

EBCDIC character set, 87
Editing, 10
ELSE, 34
EMPTY N, 71
End-of-file, 11
End-of-line, 11
END FILE, 71
END i, 40, 41
Error, 47, 62
ERROR statement, 48
EXP OF X, 23
EXPONENT PART OF X, 23
Exponentiation, 21
Expressions, 21, 35
Expressions in forms, 59
Extended delete, 45

FDname, 68
Field delimiter, 56
Field stop code, 56
Fields, 53
File name, 68
Floating dollar sign, 58
FOR conditional keywords, 39
FOR control, 40
FOR statement, 38, 45, 51
Form definition, 53
Form deletion, 68
Formed input, 61
Formed output, 60
FORMs, 53
FORWARD SPACE, 71
FP OF X, 23
FRACTION PART OF X, 23
Free formed I/O, 64
Free formed input, 66
Free formed output, 65
Functions, 22

GO, 48, 49, 50, 51, 62

I/O FOR lists, 52
IF statement, 33
Indirect mode, 8, 14
Indirect mode statement, 14
Input/output lists, 51
INTEGER PART OF X, 23
Interrupt, 49
IP OF X, 23
Iteration statements, 38

Keywords, 8

L OF, 28
Language statements, 32
LAST i, 40
LINE n, 75
Literal-next, 11
Literal forms, 59
LN OF X, 22
LOAD, 76
LOG OF X, 22
Logical expressions, 25
Logical line, 11
Loop control, 38
LOWER OF, 28

MAX OF (X,Y,Z), 23
MAXIMUM OF (X,Y,Z), 23
MIN OF (X,Y,Z), 23
MINIMUM OF (X,Y,Z), 23
MTS, 69, 85
Mulitple-dimension arrays, 19
Mulitple set statement, 50
Mulitplication, 21

NEXT i, 40
NUMBER statement, 46
Numerical constants, 17
Numerical field, 53, 54

PAGES, 74
Pagination, 74
Parentheses, 21
Part deletion, 45
Part numbers, 14
Parts, 9, 14
Precedence, 21, 22, 83

arithmetic operators, 22
Boolean operators, 27
with functions, 22
Prefix character, 8, 11
Program loading, 76
Program management, 74
Program restart, 50
Program saving, 75
Program stops, 46

RANDOM NUMBER OF X, 23
READ statement, 70
RENAME, 73
Restart, 12
RESUME, 48, 49, 50, 51, 62
REWIND, 71
RN OF X, 23

SAVE, 75
Scientific notation, 17
SET statement, 17, 32
Simple I/O, 35
SIN OF X, 22
SINE OF X, 22
Single-dimension arrays, 19
SPACE, 75
Special DO statement, 49
SQRT OF X, 22
SQUARE ROOT OF X, 22
Statement, 7
Statistics, 13
Step deletion, 45
Step numbers, 14
Steps, 14
STOP (direct), 13, 85
STOP (indirect), 46
String comparison, 27
String constants, 17, 18
String functions, 28, 82
String manipulation, 29
String operators, 82
Subscripts, 19
SUBSTRING OF, 30
Subtraction, 21
SWAP statement, 33
SYSTEM, 85

Terminal description, 10
Text material in forms, 56
The Batch, 27
THE BCD DATE, 31
THE BCD TIME, 31
THE BCD VALUE, 31
THE COST, 25
THE CPU TIME, 25
THE DATE, 25
THE ELAPSED TIME, 25
The False, 25, 26
The first character of, 30
The first m characters of string, 29
The last character of, 30
The last m characters of string, 29
THE LENGTH OF, 28
THE LOWER CASE OF, 28
THE MODE OF X, 23
THE SIZE, 24
The substring of (string,offset, length), 29
THE TIME, 25
THE TOTAL SIZE, 24
The True, 25, 26
THE UPPER CASE OF, 28
THE USER, 31
THE VALUE OF, 30
THEN, 34
TO loop, 39
TO part n, 43
TO statement, 43
TO step n, 43
Transfer of control, 42
TYPE ALL FORMS, 68
TYPE all parts, 15, 36
TYPE all steps, 36
TYPE all stuff, 36
TYPE all values, 36
TYPE ASSIGNMENT N, 73
TYPE FORM N, 67
TYPE IN FORM, 60
TYPE IN FREE FORM, 65
TYPE part n, 15, 36
TYPE statement, 35, 52
TYPE step, 15

UNNUMBER, 47
UNTIL, 39
UPPER OF, 28
Uppercase conversion, 11

Variable deletion, 44
Variable length field, 58
Variable name, 18
Variables, 17, 18, 35
VL OF, 30

WHILE, 39

WRITE statement, 69

XP OF X, 23

Reader's Comment Form

PIL/2 IN MTS
Volume 12
December 1974

Errors noted in publication:

Suggestions for improvement:

Date _____

Name_____

Address_____

Your comments will be much appreciated. Please fold the completed form as shown on the reverse side, seal or staple, and drop in Campus Mail or in the Suggestion Box at the Computing Center.

fold here

Publications
Computing Center
University of Michigan
Ann Arbor, Michigan 48105
USA

---

fold here

Update Request Form

PIL/2 IN MTS
Volume 12
December 1974

Updates to this manual will be issued periodically as errors are noted or as changes are made to MTS. If you desire to have these updates mailed to you, please fold the completed form as shown on the reverse side, seal or staple, and drop in Campus Mail or in the Suggestion Box at the Computing Center. Campus mail addresses must be given for local users.

Name_____

Address_____

_____

_____

fold here

Update Subscription Service
Publications
Computing Center
University of Michigan
Ann Arbor, Michigan 48105
USA

fold here