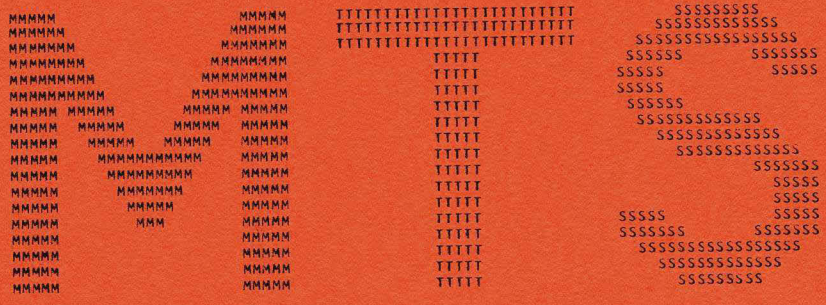




# VOLUME II

## second edition
## December 1, 1967

MTS-500-0

12-1-67

VOLUME II

This section comprises the second volume of the MTS manual. It contains writeups concerned with the content of the languages available, whereas the first volume contains writeups concerning usage of the language processors within MTS.

COMPLETE TABLE OF CONTENTS

# LANGUAGE PROCESSOR DESCRIPTIONS

This major section contains writeups for the "Language Processors" in MTS. Included are two assemblers: Assembler F and 8ASS (PDP-8 assembler), two FORTRAN compilers: FORTRAN G and WATFOR, two string processors: SNOBOL4 and UMIST, an interactive calculator language - PIL, and an I/O format interpreter - IOH/360.

This section contains details pertinent to usage of these processors. Each processor (except IOH/360) is in a library file in MTS, and details of access to the processor will be found in the pertinent library file descriptions in section MTS-280.

MTS-510-0

12-1-67

# A S S E M B L E R

# F

## F-LEVEL ASSEMBLER

The assembler in MTS is IBM's Operating System F-level Assembler for the 360.

The language processed by the F-level Assembler is described in the IBM publication form number C28-6514.

The translator exists in the library file *ASMBLR. Details for running it and for specifying parameters are given in the library file description for *ASMBLR in section MTS-280/21445.

The following are descriptions of the assembly listing produced and a list of the error comments that may be produced.

## ASSEMBLER LISTING

The assembler listing (Figure 1) consists of five sections, ordered as follow: external symbol dictionary items, the source and object program statements, relocation dictionary items, symbol cross reference table, and diagnostic messages. In addition, three statistical messages may appear in the listing:

1. After the diagnostics, a statements flagged message indicates the total number of statements in error. It appears as follows: nnn STATEMENTS FLAGGED IN THIS ASSEMBLY.

2. After the statements-flagged message, the assembler prints the highest severity code encountered (if non-zero). This is equal to the assembler return code. The message appears as follows: nn WAS HIGHEST SEVERITY CODE.

3. After the severity code, the assembler prints a count of lines printed, which appears as follows: nnn PRINTED LINES. This is a count of the actual number of 121-byte records generated by the assembler; it may be less than the total number of printed and blank lines appearing on the listing if the SPACE n assembler instruction is used. For a SPACE n that does not cause an eject, the assembler inserts n blank lines in the listing by generating n/3 blank 121-byte records--rounded to the next lower integer if a fraction results; e.g., for a SPACE 2, no blank records are generated. The assembler does not generate a blank record to force a page eject.

In addition to the above items, the assembler prints the deck identification and current date on every page of the listing and the time of day to the left of the date on page 1 of the ESD listing. This is the time when

printing starts, rather than the start of the assembly, and is intended only to provide unique identification for assemblies made on the same day. The format of the time is hh:mm where hh is the hour of the day (midnight beginning at 00), and mm is the number of minutes past the hour.

## External Symbol Dictionary (ESD)

This section of the listing contains the external symbol dictionary information passed to the loader in the object module. The entries describe the control sections, external reference, and entry points in the assembler program. There are six types of entries, show in Table 1 along with their associated fields. The numbers refer to the corresponding heading in the sample listing (Figure 1). The X's indicate entries accompanying each type designation.

Table 1. Types of ESD Entries

| 1<br>SYMBOL | 2<br>TYPE | 3<br>ID | 4<br>ADDR | 5<br>LENGTH | 6<br>LD ID |
|---|---|---|---|---|---|
| X | SD | X | X | X | - |
| X | LD | - | X | - | X |
| X | ER | X | - | - | - |
| - | PC | X | X | X | - |
| X | CM | X | X | X | - |
| X | XD | X | X | X | - |

1. This column contains the name of every external dummy section, control section, entry point, common section, and external symbol.

2. This column contains the type designator for the entry, as show in the table. The type designators are defined as:

   SD--Names section definition. The symbol appeared in the name field of a CSECT or START statement.
   LD--The symbol appeared as the operand of the ENTRY statement.
   ER--External reference. The symbol appeared as the operand of an EXTRN statement, or was defined as a V-type address constant.
   PC--Unnamed control section definition.
   CM--Common control section definition.
   XD--External dummy section.

3. This column contains the external symbol dictionary identification number (ESDID). The number is a unique two digit hexadecimal number identifying the entry. It is used by the LD entry of the ESD and by the relocation dictionary for cross-referencing the ESD.

4. This column contains the address of the symbol (hexadecimal notation) for SD-and LD-type entries, and zeros for ER-type entries. For PC- and CM-type entries, it indicates the beginning address of the control section. for XD-type entries, it indicates the alignment by printing a number one less than the number of bytes in the unit of alignment, e.g., 7 indicates double word alignment.

5. This column contains the assembled length, in bytes, of the control section (hexadecimal notation).

6. This column contains, for LD-type entries, the identification (ID) number assigned to the ESD entry that identifies the control section in which the symbol was defined.

Source and Object Program

This section of the listing documents the source statements and the resulting object program.

7. This is the four-character deck identification. It is the symbol that appears in the name field of the first TITLE statement. The assembler prints the deck identification and date (item 16) on every page of the listing.

8. This is the information taken from the operand field of a TITLE statement.

9. Listing page number. Each section of the listing starts with page 1.

10. This column contains the assembled address (hexadecimal notation) of the object code.

11. This column contains the object code produced by the source statement. The entries are always left-justified. The notation is hexadecimal. Entries are machine instructions or assembled constants. Machine instructions are printed in full with a blank inserted after every four digits (two bytes). Constants may be only partially printed (see the PRINT assembler instruction in the Assembler Language publication).

12. These two columns contain effective addresses (the result of adding together a base register value and displacement value):

    a. The column headed ADDR1 contains the effective address for the first operand of an SS instruction.

b. The column headed ADDR2 contains the effective address of the second operand of any instruction referencing storage.

Both address fields contain six digits; however, if the high-order digit is a zero, it is not printed.

13. This column contains the statement number. A plus sign (+) to the right of the number indicates that the statement was generated as the result of macro instruction processing.

14. This column contains the source program statement. The following items apply to this section of the listing:

a. Source statements are listed, including those brought into the program by the COPY assembler instruction, and including macro definitions submitted with the main program for assembly. Listing control instructions are not printed, except for the following case: PRINT is listed when PRINT ON is in effect and a PRINT statement is encountered.

b. Macro definitions obtained from a macro library are not listed.

c. The statements generated as the result of a macro instruction follow the macro instruction in the listing.

d. Assembler or machine instructions in the source program that contain variable symbols are listed twice: as they appear in the source input, and with values substituted for the variable symbols.

e. Diagnostic messages are not listed in-line in the source and object program section. An error indicator, ***ERROR***, follows the statement in error. The message appears in the diagnostic section of the listing.

f. MNOTE messages are listed in-line in the source and object program section. An MNOTE indicator appears in the diagnostic section of the listing for MNOTE statements other than MNOTE *. The MNOTE message format is severity code, message text.

g. The MNOTE * form of the MNOTE statement results in an in-line message only. An MNOTE indicator does not appear in the diagnostic section of the listing.

h. When an error is found in a programmer macro definition, it is treated the same as any other assembly error: the error indication appears after the statement in error, and a diagnostic is placed in the list of diagnostics. However, when an error is encountered during the expansion of a macro instruction (system- or programmer-defined), the error indication appears in place of the erroneous statement, which is not listed. The error indication follows the last statement listed before the

erroneous statement was encountered, and the associated diagnostic message is placed in the list of diagnostics.

i. Literals that have not been assigned locations by an LTORG statement appear in the listing following the END statement. Literals are identified by the equal (=) sign preceding them.

j. If the END statement contains an operand, the transfer address appears in the location column (LOC).

k. In the case of COM, CSECT and DSECT statements, the location field contains the beginning address of these control sections, i.e., the first occurrence.

l. In the case of EXTRN, ENTRY, and DXD instructions, the location field and object code field are blank.

m. For a USING statement, the location field contains the value of the first operand.

n. For LTORG and ORG statements, the location field contains the location assigned to the literal pool or the value of the ORG operand.

o. For an EQU statement, the location field contains the value assigned.

p. Generated statements always print in normal statement format. Because of this, it is possible for a generated statement to occupy three or more continuation lines on the listing. This is unlike source statements, which are restricted to two continuation lines.

15. This column contains the identifier of the assembler (F) and the date when this version was released by Systems Development Division to DPD Program Information Department.

16. Current date (date run is made).

17. Identification-sequence field from the source statement.

## Relocation Dictionary

This section of the listing contains the relocation dictionary information passed to the loader in the object module. The entries describe the address constants in the assembled program that are affected by relocation.

18. This column contains the external symbol dictionary ID number assigned to the ESD entry that describes the control section in which the address constant is used as an operand.

19. This column contains the external symbol dictionary ID number assigned to the ESD entry that describes the control section in which the referenced symbol is defined.

20. The two-digit hexadecimal number in this column is interpreted as follows:

    First Digit. A zero indicates that the entry describes an A-type or Q-type address constant. A one indicates that the entry describes a V-type address constant. A three describes a CXD entry.
    Second Digit. The first three bits of the digit indicate the length of the constant and whether the base should be added or subtracted:

| Bits 0 and 1 | Bit 2 |
|---|---|
| 00 = 1 byte | 0 = + |
| 01 = 2 bytes | 1 = - |
| 10 = 3 bytes | |
| 11 = 4 bytes | |

21. This column contains the assembled address of the field where the address constant is stored.

## Cross Reference

This section of the listing information concerns symbols which are defined and used in the program.

22. This column contains the symbols.

23. This column states the length (decimal notation), in bytes, of the field occupied by the symbol value.

24. This column contains either the address the symbol represents, or a value to which the symbol is equated.

25. This column contains the statement number of the statement in which the symbol was defined.

26. This column contains the statement numbers of statements in which the symbol appears as an operand. In the case of a duplicate symbol, the assembler fills this column with the message:

    ****DUPLICATE****

The following notes apply to the cross-reference section:

- Symbols appearing in V-type address constants do not appear in the cross-reference listing.

- A PRINT OFF listing control instruction does not affect the production of the cross-reference section of the listing.

- In the case of an undefined symbol, the assembler fills columns 23,24, and 25 with the message:

****UNDEFINED****.

## Diagnostics

This section contains the diagnostic messages issued as a result of error conditions encountered in the program. The text, severity code, and explanatory notes for each message are contained below.

27. This column contains the number of the statement in error.

29. This column contains the message, and, in most cases, an operand column pointer that indicates the vicinity of the error. In the following example, the approximate location of the addressability error occurred in the 9th column of the operand field:

Example:

```
STMT  ERROR CODE  MESSAGE

 21   IEU035      NEAR OPERAND COLUMN 9--ADDRESSABILITY ERROR
```

The following notes apply to the diagnostic section:

- An MNOTE indicator of the form MNOTE STATEMENT appears in the diagnostic section if an MNOTE statement other than MNOTE * is issued by a macro instruction. The MNOTE statement itself is in-line in the source and object program section of the listing. The operand field of an MNOTE * is printed as a comment, but does not appear in the diagnostic section.

- A message identifier consists of six characters and is of the form:

IEUxxx

IEU identifies the issuing agent as Assembler F, and xxx is a unique number assigned to the message.

```
                                EXTERNAL SYMBOL DICTIONARY

                                                                                    Page    1
                                                                                  00.16  4/11/66
EXAM
SYMBOL     TYPE  ID   ADDR     LENGTH   LD   ID
(1)
SAMPLR     SD    01   000000   000388


(7)  EXAM      (8)  SAMPLE PROGRAM                                                  (9)  Page  3
(10)  LOC   (11)  OBJECT CODE   (12) ADDR1  ADDR2  (13) STMT  (14) SOURCE STATEMENT   (15) F 14FEB66  (16) 4/11/66

000000  47F0 F00A              0000A      59+BEGIN  B       10(0,15)  BRANCH AROUND ID
000004  05                                60+       DC      AL1(5)
000005  C2C5C7C9D5                        61+       DC      CL5'BEGIN'  IDENTIFIER
00000A  90EC D00C              0000C      62+       STM     14,12,12(13)  SAVE REGISTERS                    (17)
00000E  05C0                              63        BALR    R12,0         ESTABLISH ADDRESSABILITY OF PROGRAM  SAMPL057
000010                                    64        USING *,R12           AND TELL THE ASSEMBLER WHAT BASE TO USE  SAMPL058

(7)  EXAM                          RELOCATION DICTIONARY                            (9)  Page  1
(18) POS.ID  (19) REL.ID  (20) FLAGS  (21) ADDRESS                                  (16) 4/11/66

   01         01          0C         0001FC
   01         01          0C         00020C
   01         01          0C         00021C
   01         01          0C         0002D4
   01         01          0C         000334

(7)  EXAM                          CROSS-REFERENCE                                  (9)  Page  1
(22) SYMBOL  (23) LEN  (24) VALUE  (25) DEFN  (26) REFERENCES                       (16) 4/11/66

BEGIN     00004  000000  0059  0156  0158  0174  0184  0186  0220
EXIT      00004  00007E  0096  0111
HIGHER    00002  0000F4  0130  0125
IHB0005   00001  00007B  0093  0090
IHB0005A  00002  00007C  0094  0089

(7)  EXAM                          DIAGNOSTICS                                      (9)  Page  1
(27) STMT  (28) ERROR CODE  (29) MESSAGE                                            (16) 4/11/66

  19    IEU025      NEAR OPERAND COLUMN  7--RELOCATABILITY ERROR
  21    IEU035      NEAR OPERAND COLUMN  9--ADDRESSABILITY ERROR

   2  STATEMENTS FLAGGED IN THIS ASSEMBLY
   8  WAS HIGHEST SEVERITY CODE
 261  PRINTED LINES
```

Figure 1. Assembler Listing

## DIAGNOSTIC MESSAGES

| Code | Message | Explanation | Severity Code |
|---|---|---|---|
| IEU001 | DUPLICATION FACTOR ERROR | A duplication factor is not an absolute expression, or is zero in a literal; * in duplication factor expression; invalid syntax in expression. | 12 |
| IEU002 | RELOCATABLE DUPLICATION FACTOR | A relocatable expression has been used to specify the duplication factor. | 12 |
| IEU003 | LENGTH ERROR | The length specification is out of permissible range or specified invalidly; * in length expression; invalid syntax in expression; no left-parenthesis delimiter for expression. | 12 |
| IEU004 | RELOCATABLE LENGTH | A relocatable expression has been used to specify length | 12 |
| IEU005 | S-TYPE CONSTANT IN LITERAL | Self-explanatory. | 8 |
| IEU006 | INVALID ORIGIN | The location counter has been reset to a value less than the starting address of the control section; ORG operand is not a simply relocatable expression or specifies an address outside the control section. | 12 |
| IEU007 | LOCATION COUNTER ERROR | The location counter has exceeded 19777215, or passed out of the control section in negative direction (3 byte arithmetic). | 12 |
| IEU008 | INVALID DISPLACEMENT | The displacement in an explicit address is not an absolute value within the range of 0 to 4095. | 8 |
| IEU009 | MISSING OPERAND | Self-explanatory. | 12 |

| Code | Message | Explanation | Severity Code |
|---|---|---|---|
| IEU010 | INCORRECT REGISTER SPECIFICATION | The value specifying the register is not an absolute value within the range 0-15, an odd register is specified where an even register is required, or a register was used where none can be specified. | 8 |
| IEU011 | SCALE MODIFIER ERROR | The scale modifier is not an absolute expression or is too large, negative scale modifier for floating point, * in scale modifier expression; invalid syntax or illegally specified scale modifier. | 8 |
| IEU012 | RELOCATABLE SCALE MODIFIER | A relocatable expression has been used to specify the scale modifier. | 8 |
| IEU013 | EXPONENT MODIFIER ERROR | The exponent is not specified as an absolute expression or is out of range; * in exponent modifier expression; invalid syntax; illegally specified exponent modifier. | 8 |
| IEU014 | RELOCATABLE EXPONENT MODIFIER | A relocatable expression has been used to specify the exponent modifier. | 8 |
| IEU015 | INVALID LITERAL USAGE | A valid literal is used illegally, it specifies a receiving field or a register, or it is a Q-type constant. | 8 |
| IEU016 | INVALID NAME | A name entry is incorrectly specified, e.g., it contains more than 8 character, it does not begin with a letter, or has a special character imbedded. | 8 |

| Code | Message | Explanation | Severity Code |
|---|---|---|---|
| IEU017 | DATA ITEM TOO LARGE | The constant is too large for the data type or for the explicit length; operand field for packed DC exceeds 32 characters and for zoned DC exceeds 16 characters (excluding decimal points). | 8 |
| IEU018 | INVALID SYMBOL | The symbol is specified invalidly, e.g., it is longer than 8 characters. | 8 |
| IEU019 | EXTERNAL NAME ERROR | A CSECT and DSECT statement have the same name, or a symbol is used more than once in an EXTRN or the name field of DXD statements. | 8 |
| IEU020 | INVALID IMMEDIATE FIELD | The value of the immediate operand exceeds 255, or the operand requires more than one byte of storage. | 8 |
| IEU021 | SYMBOL NOT PREVIOUSLY DEFINED | Self-explanatory. | 8 |
| IEU022 | ESD TABLE OVERFLOW | The combined number of control sections and dummy sections plus the number of unique symbols in EXTRN statements and V-type constants exceeds 255. (A DSECT which appears as XD makes two entries). | 12 |
| IEU023 | PREVIOUSLY DEFINED NAME | The symbol which appears in the name field has appeared in the name field of a previous statement. | 8 |
| IEU024 | UNDEFINED SYMBOL | A symbol being referenced has not been defined in the program. | 8 |
| IEU025 | RELOCATABILITY ERROR | A relocatable or complex relocatable expression is specified where an absolute expression is required, an absolute expression or complex relocatable expression is specified where a relocatable expression is required, or a relocatable term is involved in multiplication or division. | 8 |

| Code | Message | Explanation | Severity Code |
|---|---|---|---|
| IEU026 | TOO MANY LEVELS OF PARENTHESES | An expression specifies more than 5 levels of parentheses. | 12 |
| IEU027 | TOO MANY TERMS | More than 16 terms are specified in an expression. | 12 |
| IEU028 | REGISTER NOT USED | A register specified in a DROP statement is not currently in use. | 4 |
| IEU029 | CCW ERROR | Bits 37-39. of the CCW are set to non-zero. | 8 |
| IEU030 | INVALID CNOP | An invalid combination of operands is specified. | 12 |
| IEU031 | UNKNOWN TYPE | Incorrect type designation is specified in a DC, DS, or literal. | 8 |
| IEU032 | OP-CODE NOT ALLOWED TO BE GENERATED | Self-explanatory. | 8 |
| IEU033 | ALIGNMENT ERROR | Referenced address is not aligned to the proper boundary for this instruction, e.g., START operand not a multiple of 8. | 4 |
| IEU034 | INVALID OP-CODE | Syntax error, e.g., more than 8 characters in operation field, not followed by blank on first card, missing. | 8 |
| IEU035 | ADDRESSABILITY ERROR | The referenced address does not fall within the range of a USING instruction. | 8 |
| IEU036 | (No message is assigned to this number) | | |
| IEU037 | MNOTE STATEMENT | This indicates that an MNOTE statement has been generated from a macro definition. The text and severity code of the MNOTE statement will be found in line in the listing. | Variable |

| Code | Message | Explanation | Severity Code |
|---|---|---|---|
| IEU038 | ENTRY ERROR | A symbol in the operand of an ENTRY statement appears in more than one ENTRY statement, it is undefined, it is defined in a dummy section or in common, or it is equated to a symbol defined by an EXTRN statement. | 8 |
| IEU039 | INVALID DELIMITER | This message can be caused by any syntax error, e.g., missing delimiter, special character used which is not a valid delimiter, delimiter used illegally, operand missing, i.e., nothing between delimiters, unpaired parentheses, imbedded blank in expression. | 12 |
| IEU040 | GENERATED RECORD TOO LONG | There are more than 236 characters in a generated statement. | 12 |
| IEU041 | UNDECLARED VARIABLE SYMBOL | Variable symbol is not declared in a defined SET symbol statement or in a macro prototype. | 8 |
| IEU042 | SINGLE TERM LOGICAL EXPRESSION IS NOT A SETB SYMBOL | This single term logical expression has not been declared as a SETB symbol. | 8 |
| IEU043 | SET SYMBOL PREVIOUSLY DEFINED | Self-explanatory. | 8 |
| IEU044 | SET SYMBOL USAGE INCONSISTENT WITH DECLARATION | A SET symbol has been declared as undimensioned, but is subscripted, or has been declared dimensioned, but is unsubscripted. | 8 |
| IEU045 | ILLEGAL SYMBOLIC PARAMETER | An attribute has been requested for a variable symbols which is not a legal symbolic parameter. | 8 |

| Code | Message | Explanation | Severity Code |
|---|---|---|---|
| IEU046 | AT LEAST ONE RELOC-ATABLE Y TYPE CONSTANT IN ASSEMBLY | Self-explanatory. | 4 |
| IEU047 | SEQUENCE SYMBOL PREVIOUSLY DEFINED | Self-explanatory. | 12 |
| IEU048 | SYMBOLIC PARAMETER PREVIOUSLY DEFINED OR SYSTEM VARIABLE SYMBOL DECLARED AS SYMBOLIC PARAMETER | Self-explanatory. | 12 |
| IEU049 | VARIABLE SYMBOL MATACHES A PARAMETER | Self-explanatory. | 12 |
| IEU050 | INCONSISTENT GLOBAL DECLARATIONS | A global SET variable symbol, defined in more than one macro definition or defined in a macro definition and in the source program, is inconsistent in SET type or dimension. | |
| IEU051 | MACRO DEFINITION PREVIOUSLY DEFINED | Prototype operation field is the same as a machine or assembler instruction or a previous prototype. | 12 |
| IEU052 | NAME FIELD CONTAINS ILLEGAL SET SYMBOL | SET symbol in name field does not correspond to SET statement type. | 8 |
| IEU053 | GLOBAL DICTIONARY FULL | The global dictionary is full, assembly terminated. | 12 |
| IEU054 | LOCAL DICTIONARY FULL | The local dictionary is full, current macro aborted. If in open code, assembly terminated. | 12 |
| IEU055 | INVALID ASSEMBLER OPTION(S) IN THE PARAMETER LIST | Self-explanatory. | 8 |

| Code | Message | Explanation | Severity Code |
|---|---|---|---|
| IEU056 | ARITHMETIC OVERFLOW | The intermediate or final result of an expression is not within the range of $-2^{31}$ to $2^{31}-1$. | 8 |
| IEU057 | SUBSCRIPT EXCEEDS MAXIMUM DIMENSION | &SYSLIST or symbolic parameter subscript exceeds 200, or is negative, or zero, or SET symbol subscript exceeds dimension. | 8 |
| IEU058 | RE-ENTRANT CHECK FAILED | An instruction has been detected, which, when executed, might store data into a control section or a common area. This message is generated only when requested via control cards and merely indicates a possible re-entrant error. | 4 |
| IEU059 | UNDEFINED SEQUENCE SYMBOL | Self-explanatory. | 12 |
| IEU060 | ILLEGAL ATTRIBUTE NOTATION | L', S', or I' requested for a parameter whose type attribute does not allow these attributes to be requested. | 8 |
| IEU061 | ACTR COUNTER EXCEEDED | Self-explanatory. | 12 |
| IEU062 | GENERATED STRING GREATER THAN 255 CHARACTERS | Self-explanatory. | ' |
| IEU063 | EXPRESSION 1 OF SUBSTRING IS ZERO OR MINUS | Self-explanatory. | 8 |
| IEU065 | INVALID OR ILLEGAL TERM IN ARITHMETIC EXPRESSION | The value of a SETC symbol used in the arithmetic expression is not composed of decimal digits, or the parameter is not a self-defining | 8 |

| Code | Message | Explanation | Severity Code |
|---|---|---|---|
| IEU066 | UNDEFINED OR DUPLICATE KEYWORD OPERAND OR EXCESSIVE POSITIONAL OPERANDS | The same keyword operand occurs more than once in the macro instruction; a keyword is not defined in a prototype statement; in a mixed mode macro instruction, more positional operands are specified than are specified in the prototype. | 12 |
| IEU067 | EXPRESSION 1 OF SUBSTRING GREATER THAN LENGTH OF CHARACTER EXPRESSION | Self-explanatory. | 8 |
| IEU068 | GENERATION TIME DICTIONARY AREA OVERFLOWED | Self-explanatory. | 12 |
| IEU069 | VALUE OF EXPRESSION 2 OF SUBSTRING GREATER THAN 8 | Self-explanatory. | 8 |
| IEU070 | FLOATING POINT CHARACTERISTIC OUT OF RANGE | Self-explanatory. | 12 |
| IEU071 | ILLEGAL OCCURRENCE OF LCL, GBL, OR ACTR STATEMENT | LCL, GBL, or ACTR statement is not in proper place in the program. | 8 |
| IEU072 | ILLEGAL RANGE ON ISEQ STATEMENT | Self-explanatory. | 4 |
| IEU073 | ILLEGAL NAME FIELD | Either a statement which requires a name has been written without a name, or a statement has a name which is not allowed to have a name. | 8 |
| IEU074 | ILLEGAL STATEMENT IN COPY CODE OR SYSTEM MACRO | Self-explanatory. | 8 |

| Code | Message | Explanation | Severity Code |
|---|---|---|---|
| IEU075 | ILLEGAL STATEMENT OUTSIDE OF A MACRO DEFINTION | Self-explanatory. | 8 |
| IEU076 | SEQUENCE ERROR | Self-explanatory. | 12 |
| IEU077 | ILLEGAL CONTINUATION CARD | Either there are too many continuation cards, or there are non-blanks between the begin and continue columns on the continuation card. | 8 |
| IEU078 | (No message is assigned to this number) | | |
| IEU079 | ILLEGAL STATEMENT IN MACRO DEFINITION | This operation is not allowed within a macro definition. | 8 |
| IEU080 | ILLEGAL START CARD | Statements affecting or depending upon the location counter have been encountered before a START statement. | 8 |
| IEU081 | ILLEGAL FORMAT IN GBL OR LCL STATEMENTS | An operand is not a variable symbol. | 8 |
| IEU082 | ILLEGAL DIMENSION SPECIFICATION IN GBL OR LCL STATEMENT | Dimension is other than 1 to 255. | 8 |
| IEU083 | SET STATEMENT NAME FIELD NOT A VARIABLE SYMBOL | Self-explanatory. | 8 |
| IEU084 | ILLEGAL OPERAND FIELD FORMAT | Syntax invalid, e.g., AIF statement operand does not start with a left parenthesis; operand of AGO is not a sequence symbol; operand of PUNCH, TITLE, MNOTE not enclosed in quotes. | 8 |

| Code | Message | Explanation | Severity Code |
|---|---|---|---|
| IEU085 | INVALID SYNTAX IN EXPRESSION | Invalid delimiter, too many terms in an expression, too many levels of parentheses, two operators in succession, two terms in succession, or illegal character. | 8 |
| IEU086 | ILLEGAL USAGE OF SYSTEM VARIABLE SYMBOL | A system variable symbol appears in the name field of a SET statement, is used in a mixed mode or keyword macro definition, is declared in a GBL or LCL statement, or is an unsubscripted &SYSLIST in a context other than N'&SYSLIST. | 8 |
| IEU087 | NO ENDING APOSTROPHE | There is an unpaired apostrophe or ampersand in the statement. | 8 |
| IEU088 | UNDEFINED CODE | Self-explanatory. | 12 |
| IEU089 | INVALID ATTRIBUTE NOTATION | Syntax error inside a macro definition, e.g., the argument of the attribute reference is not a symbolic parameter. | 8 |
| IEU090 | INVALID SUBSCRIPT | Syntax error, e.g., double subscript where single subscript is required or vice versa; not right parenthesis after subscript. | 8 |
| IEU091 | INVALID SELF-DEFINING TERM | Value is too large or is inconsistent with the data type,e.g., severity code greater than 255. | 8 |
| IEU092 | INVALID FORMAT FOR VARIABLE SYMBOL | This first character after the ampersand is not alphabetic, or the variable symbol contains more than 8 characters. | 8 |

| Code | Message | Explanation | Severity Code |
|---|---|---|---|
| IEU093 | UNBALANCED PARENTHESIS OR EXCESSIVE LEFT PARENTHESES | Self-explanatory. | 8 |
| IEU094 | INVALID OR ILLEGAL NAME OR OPERATION IN PROTOTYPE STATEMENT | Self-explanatory. | 12 |
| IEU095 | ENTRY TABLE OVERFLOW | Number of ENTRY symbols, i.e., ENTRY instruction operands, exceeds 100. | 8 |
| IEU096 | MACRO INSTRUCTION OR PROTOTYPE OPERAND EXCEEDS 255 CHARACTERS IN LENGTH | Self-explanatory. | 12 |
| IEU097 | INVALID FORMAT IN MACRO INSTRUCTION OPERAND OR PROTOTYPE PARAMETER | This message can be caused by:<br>1. Illegal "=".<br>2. A single "&" appears somewhere in the standard value assigned to a prototype keyword parameter.<br>3. First character of a prototype parameter is not "&".<br>4. Prototype parameter is a subscripted variable symbol.<br>5. Invalid use of alternate format in prototype statement, e.g.,<br>10 16 72<br>PROTO &A,&B,<br>or<br>PROTO &A,&B, X<br>&C<br>6. Unintelligible prototype parameter, e.g., "&A*" or "&A&&."<br>7. Illegal (non-assembler) character appears in prototype parameter or macro-instruction operand. | 12 |

| Code | Message | Explanation | Severity Code |
|---|---|---|---|
| IEU098 | EXCESSIVE NUMBER OF OPERANDS OR PARAMETERS | Either the prototype has more than 200 parameters, or the macro instruction has more than 200 operands. | 12 |
| IEU099 | POSITIONAL MACRO INSTRUCTION OPERAND, PPOTOTYPE PARAMETER OR EXTRA COMMA FOLLOWS KEYWORD | Self-explanatory. | 12 |
| IEU100 | STATEMENT COMPLEXITY EXCEEDED | More than 32 operands in a DC, DS, DXD, or literal DC, or more than 50 terms in a statement. | 8 |
| IEU101 | EOD ON SYSIN | EOD before END card. | 12 |
| IEU102 | INVALID OR ILLEGAL ICTL | The operands of the ICTL are out of range, or the ICTL is not the first statement in the source program. | 16 |
| IEU103 | ILLEGAL NAME IN OPERAND FIELD OF COPY CARD | Syntax error, e.g., symbol has an illegal character. | 12 |
| IEU104 | COPY CODE NOT FOUND | The operand of a COPY statement specified COPY text which cannot be found. | 12 |
| IEU105 | EOD ON SYSTEM MACRO LIBRARY | EOD before MEND card. | 12 |
| IEU106 | NOT NAME OF DSECT OR DXD | Self-explanatory. | 8 |
| IEU107 | INVALID OPERAND | Invalid syntax in DC operand, e.g., invalid hexadecimal character in hexadecimal DC; operand string too long for X, B, C, DC's; operand unrecognizable, contains invalid value, or incorrectly specified. | 4 |
| IEU108 | PREMATURE EOD | Indicates an internal assembler error; should not occur. | 16 |
| IEU109 | PRECISION LOST | Self-explanatory. | 8 |

| Code | Message | Explanation | Severity Code |
|---|---|---|---|
| IEU109 | PRECISION LOST | Self-explanatory | 8 |
| IEU110 | EXPRESSION VALUE TOO LARGE | Value of expression greater than -16777216 to +167772159<br><br>Expressions in EQU and ORG statements are flagged if (1) they include terms previously defined as negative values, or (2) positive terms give a result of more than three bytes in magnitude. The error indication may be erroneous due to (1) the treatment of negative values as three-byte positive values, or (2) the effect of large positive values on the location counter if a control section begins with a START statement having an operand greater than zero, or a control section is divided into subsections. | 8 |

# F O R T R A N

## G

## FORTRAN G

The G-level FORTRAN compiler from IBM's Operating System for the 360 is available in MTS in either *FORTRAN or *NEWFORT. Consult the library file descriptions of these in section MTS-280 to see which should be used, and for details of I/O unit and parameter specification.

The following is a listing of compiler diagnostics followed by a "Fortran Users Guide".

## SOURCE MODULE ERROR/WARNING MESSAGES

The error/warning messages produced by the compiler occur in the source listing immediately following the source statement to which they refer. A maximum of four error messages are printed per line. The following example illustrates the format of these messages as they appear in the source listing.

```
      XX = A+B+-C/(X**3-A**-75)
                  $           $
n) y message, n) y message
```

Where: n is an integer noting the positional occurrence of the error on each line.

y is a 1-to-3 digit message number of the form IEYxxxI.

$ is the symbol used by the compiler for flagging the particular error in the statement (this symbol is always noted on the line following the source statement and underneath the error).

message is the actual message printed.

The error and warning messages are distinguished by the resulting completion codes. Serious error messages have a code of eight, while warning messages may produce either a code of four or zero.

IEY001I ILLEGAL TYPE

Explanation: The variable in an assigned go to statement is not an integer variable; or the variable in an assignment statement on the left of the equal sign is of logical type and the expression on the right side does not correspond. CC=8, i.e., The completion code is 8.

IEY002I LABEL

Explanation: The statement in question is unlabeled and follows a transfer of control; the statement therefore cannot be executed. CC=0.

IEY003I NAME LENGTH

Explanation: The name of a variable, common block, name list or subprogram exceeds six characters in length; or two variable names appear in an expression without a separating operation symbol. CC=4.

IEY004I COMMA

Explanation: The comma required in the statement has been omitted. CC=0.

IEY005I ILLEGAL LABEL

Explanation: Illegal usage of a statement label; for example, an attempt is made to branch to the label of a format statement. CC=8.

IEY006I DUPLICATE LABEL

Explanation: The label appearing in the label field of a statement has previously been defined for statement. CC=8.

IEY007I ID CONFLICT

Explanation: The name of a variable or subprogram has been used in conflict with the type that was defined for the variable or subprogram in a previous statement. CC=8.

IEY008I ALLOCATION

Explanation: The storage allocation specified by a source module statement cannot be performed because of an inconsistency between the present usage of a variable name and some prior usage of that name. CC=8.

IEY009I ORDER

Explanation: The statements contained in the source module are used in an improper sequence. CC=8.

IEY010I SIZE

Explanation: A number used in the source module does not conform to the legal values for its use. CC=8.

IEY011I UNDIMENSIONED

Explanation: A variable name is used as an array and the variable has not been dimensioned. CC=8.

IEY012I SUBSCRIPT

Explanation: The number of subscripts used in an array reference is either too large or too small for the array.

IEY013I SYNTAX

Explanation: The statement or part of a statement to which this message refers does not conform to the FORTRAN IV syntax. CC=8.

IEY014I CONVERT

Explanation: The mode of the constant used in a DATA or in an Explicit Specification statement is different from the mode of the variable with which it is associated. The constant is then converted in the correct mode. CC=0.

IEY015I NO END CARD

Explanation: The source module does not contain an end statement. CC=0.

IEY016I ILLEGAL STA.

Explanation: The context in which the statement in question has been used is illegal. CC=8.

IEY017I ILLEGAL STA. WRN.

Explanation: A RETURN i statement appears in a function subprogram. CC=0.

IEY018I NUMBER ARG

Explanation: The reference to a library subprogram specifies an incorrect number of arguments. CC=4.

IEY019I FUNCTION ENTRIES UNDEFINED

Explanation: If the program being compiled is a function subprogram and there is no scalar with the same name as the function nor is there a definition for each entry, the message appears on SPRINT. A list of the names in error is printed following the message.

IEY020I COMMON BLOCK name ERRORS

Explanation: This message pertains to errors that exist in the definitions of equivalence sets which refer to the common area. The message is produced when there is a contradiction in the allocation specified, a designation to extend the beginning of the common area, or if the assignment of common storage attempts to allocate a variable to a location which does not fall on the appropriate boundary; "name" is the name of the common block in error.

IEY021I UNCLOSED DO LOOPS

Explanation: The message is produced if DO loops are initiated in the source module, but their terminal statements do not exist. A list of the labels which appeared in the DO statements but were not defined follows the printing of the message.

IEY022I UNDEFINED LABELS

Explanation: If any labels are used in the source module but are not defined, this message is produced. A list of the undefined labels appears on the lines following the message. However, if there are no undefined labels, the word NONE appears on the same line as the message.

IEY023I EQUIVALENCE ALLOCATION ERRORS

Explanation: The message is produced when there is a conflict between two equivalence sets, or if there is an incompatible boundary alignment in the equivalence set. The message is followed by a list of the variables which could not be allocated according to the source module specifications.

IEY024I EQUIVALENCE DEFINITION ERRORS

Explanation: This message denotes an error in an equivalence set when an array element is outside the array.

IEY025I DUMMY DIMENSION ERROR

Explanation: If variables specified as dummy array dimensions are not in common and are not global dummy variables, the above error message is produced. A list of the dummy variables which are found in error is printed on the lines following the message.

IEY026I BLOCK DATA PROGRAM ERRORS

Explanation: This message is produced if variables in the source module have been assigned to a program block but have not been defined previously as common. A list of these variables is printed on the lines following the message.

IEY032I NULL PROGRAM

Explanation: This message is produced when an end of file mark precedes any true FORTRAN statements in the source module.

## Files and Data Set Reference Numbers

As used in this write-up, a direct access file is a data set defined by a FORTRAN IV DEFINE FILE statement; all other data sets are termed sequential files regardless of the storage media involved. Input/output operations on direct access files are always performed with the indexed modifier on, whereas it is always off for sequential files.

The legal data set reference numbers (DSRN's) in MTS FORTRAN are 0 through 9 for sequential files or previously created direct access files and 10 through 19 for temporary direct access files. Logical devices 0 through 9 should be assigned when the $RUN command is issued for the FORTRAN object module. Reference numbers 10 through 19 will be assigned at execution time when the associated DEFINE FILE statement(s) are encountered, i.e., temporary files named -DAF..0 through -DAF..9 will be created as required.

## Tape Support Statements

The REWIND, BACKSPACE and END FILE statements are ignored for all direct access files; however, they may be applied to any sequential file with varying degrees of success. It is strongly recommended that these statements only refer to DSRN's actually assigned to tape units. If these statements are applied to a read-only device, e.g., a card reader, a diagnostic comment will be forthcoming. If they are applied to a non-tape device capable of output (e.g., a file) the comments REW, BSR and WEF will be written.

## Sequential Files

The first reference to a DSRN via a READ/WRITE statement causes the specified data set to be opened for either reading or writing. The READ, PRINT and PUNCH statements described in Appendix B of the IBM System/360 FORTRAN Language SRL (C28-6515-4) are connected to logical devices SCARDS, SPRINT and SPUNCH, respectively. In IBM systems the READ, PRINT and PUNCH statements usually default to data set reference numbers 5,6 and 7, respectively. Reference to a direct access file with the sequential form of the READ/WRITE statements, or reference to a sequential file with the direct access form of these statements results in the salutation IHC231I.

## Record Format for Sequential Files

The length of an I/O record of a FORTRAN data set is defined either by a FORMAT statement or the list in an I/O statement. By definition, a record written or read under FORMAT control is termed a FORTRAN record, while one constructed solely on the basis of an I/O list is termed a logical record. The type of record being used together with the default record length determine the course of I/O operations as described in the following paragraphs.

When writing FORTRAN records, the length of the written record is determined by the FORMAT statement; however, an error (IHC212I) is recognized if an attempt is made to output a record longer than the data set record length. If a FORTRAN record shorter than the data set record length is read, it will be extended to the default record length by appending trailing blanks. If a FORTRAN record longer than the data set record length is read, the number of characters read will be retained.

Logical records being written may not exceed

254*(data set record length - 4) bytes.

The factor of -4 in the above formula is due to the fact that logical records are written as a sequence of partial records, each containing a FORTRAN control word popularly known as the green word. The partial record length does not exceed the data set record length. The green word in the last partial record contains a flag indicating that it is the last partial record. Aside from this adjustment for logical record length, the green word is transparent to the FORTRAN programmer. When reading logical records, an error (IHC213I) is recognized if an attempt is made to read beyond the last partial record, i.e., the number of bytes in the logical record are not sufficient to satisfy the I/O list.

## Default Record Length for Sequential Files

| DSRN | RECORD LENGTH | MAX. LOGICAL RECORD |
|---|---|---|
| 0 to 5 | 80 bytes | 29,304 bytes |
| 6 to 9 | 132 bytes | 32,512 bytes |

## Record Format for Direct Access Files

The length of an I/O record for a FORTRAN direct access file is specified in the DEFINE FILE statement. For L and E type files the record length (the second parameter) is specified in bytes, while for U type files the record length is specified in words. The maximum record length is 32,768 bytes, while any record length less than 16 will be increased to

this minimum of 16. The length of all records is as specified in the define file statement. FORTRAN records shorter than the defined record length are padded with blanks. A logical record can exceed the record length specified in the DEFINE FILE statement, but if it is shorter than the record length, it will be extended with trailing hexadecimal zeroes. Note that if the logical record length exceeds the specified record length then multiple direct access records will be written or read.

A single buffer, allocated at execution time, is used for all direct access files. The length of this buffer is the maximum of the record lengths of the currently defined direct access files. If buffer space is not available when requested an error (IHC900I) is recognized. If a record longer than the current buffer is read an error (IHC901I) is recognized. This may occur if a file is written with one record length and later read with a shorter record length. Note that if the reverse situation occurs, the records read will be automatically extended with trailing blanks or hexadecimal zeroes.

A direct access file recognizes records 1,2... up to the maximum record number given in the DEFINE FILE statement. Any attempt to read or write a record other than these will be greeted by an IHC232I. With respect to the FORTRAN program, once the define file statement has been executed the entire file automatically exists, i.e., unwritten records are all blank or all hex zeroes. Note that the $LIST command is not equivalent to reading the file and listing it via a FORTRAN program.

It is recommended that before attempting to use the direct access facility, that the user scrutinize the appropriate sections of the FORTRAN language SRL. Specific points to be considered:

(1) the DEFINE FILE statement does not initialize the associated variable.

(2) within MTS, the FIND statement is useless except to set the associated variable, since retrieval is automatically concurrent with (somebody's) computation.

(3) the DEFINE FILE statement must <u>logically</u> be the first occurrence of the DSRN. Because of the buffer allocation technique it is advisable to issue a DEFINE FILE statement for the file with the largest record length as soon as possible.

(4) the REWIND, BACKSPACE and END FILE statements are ignored. Rewinding can be accomplished by simply setting the associated variable to 1, while backspacing by n records is accomplished by decrementing the associated variable by n.

(5) the associated variable may be either a full or half word integer.

## The STOP Statement

The execution time implementation of the STOP statement has been altered. The message produced by a STOP statement is

IHC002I STOP XXXXX ***** RESTART AT LOCATION YYYYYY

and will be written only on SERCOM. If the STOP number is zero, i.e., either STOP or STOP 0, a normal OS return to MTS will occur. If the STOP number is non-zero a return to the MTS subroutine ERROR will occur. In both cases XXXXX represents the five digit stop number and YYYYYY represents the address of the first executable instruction after the STOP statement. In batch mode this means a non-zero STOP number will cause termination with a core dump. In conversational mode, it means that control will pass to MTS in such a fashion that later issuance of a $RESTART or $RESTART AT YYYYYY command will cause execution to be resumed with the first statement following the STOP.

## The PAUSE Statement

The PAUSE statement causes a message of the form

```
                    { n
IHC001A PAUSE       {'text'
                    { 0
```

to be written both to the logical device SERCOM and the operator's console. The operator is then solicited for a reply. When a reply has been received, it will be relayed to the logical device SERCOM and execution will be resumed. If the PAUSE source statement contains text, only the first 86 characters will be transmitted. CAUTION: Each PAUSE statement requires an operator reply; hence, extended messages should be communicated directly.

## Execution Error Messages

Execution time errors cause a message of the form

IHCdddI 'optional-message'

to be written on the logical device SERCOM. After the message has been written, control passes to the MTS subroutine ERROR. The program module name which originates the error is included in the following text descriptions of the error codes. The following list of module names represents the subroutine support for FORTRAN IV input/output operations.

IHCFCOMH - Input/output supervisory routine. This module also handles program interrupts and other execution time errors.

IHCFCVTH - Performs conversion of data for I/O operations.

IHCFIOSH - Performs read/write operations on all sequential files.

IHCDIOSE - Performs read/write operations on all direct access files and does execution time implementation of the DEFINE FILE statement.

IHCUATBL - Unit assignment table governing all I/O operations performed by IHCFIOSH and IHCDIOSE.

IHC211I An invalid character has been detected in a format. IHCFCOMH

IHC212I A FORTRAN record exceeds the data set record length. IHCFCOMH

IHC213I A list in a non-formatted read statement requires more bytes of data than the logical record being read contains. IHCFCOMH

IHC214I An attempt has been made to write more than 254 partial records in one logical record. The list in a non-formatted write statement is too long. IHCFCOMH

IHC215I An invalid character exists in a field being read according to a D,E,F,G or I format specification. IHCFCVTH

IHC216I Illegal sense light number detected in a call to IHCFSLIT(SLITE) or IHCFSLIT(SLITET)

IHC217I End of data set sensed during read operation and no "END=" parameter supplied. IHCFIOSH

IHC218I A device error condition exists and the ERR= parameter was not given in the READ/WRITE statement.

IHC220I Illegal data set reference number. IHCFIOSH and IHCDIOSE

IHC221I An input variable name exceeds eight characters in a NAMELIST I/O operation. IHCNAMEL

IHC222I An input variable name is not in the NAMELIST dictionary, or an array is specified with an insufficient amount of data. IHCNAMEL

IHC223I An input variable name or a subscript has no delimiter. IHCNAMEL

IHC224I A subscript is encountered after an undimensioned input name.

IHC225I An illegal character exists in a field being read according to a Z format specification. IHCFCVTH

IHC230I SOURCE ERROR AT ISN XXXX - EXECTUION FAILED AT SUBROUTINE-NAME. During load module execution, a source statement error has been encountered. Source statement errors which do not force termination of the compilation , occasion compilation of a call to the

library module IHCIBERH. If at execution time this subroutine is called, it produces the above comment.

IHC231I Direct access form of READ/WRITE statement refers to sequential file (IHCDIOSE), or vice versa (IHCFIOSH).

IHC232I Record number in direct access I/O statement is less than or equal to zero or exceeds the maximum given in the DEFINE FILE statement for this DSRN. IHCDIOSE

IHC233I Record length in a DEFINE FILE statement exceeds the maximum of 32,768 bytes. IHCDIOSE

IHC235I A DSRN already opened as a sequential file has been encountered in a DEFINE FILE statement. IHCDIOSE

IHC236I A DSRN which has not been opened as a sequential or direct access file has been encountered in a direct access I/O statement. IHCDIOSE

IHC241I For an exponentiation operation (I**J) in the subprogram IHCFIXPI (FIXPI#) where I and J represent integer variables or integer constants, I is equal to zero and J is less than or equal to zero.

IHC242I For an exponentiation operation (R**J) in the subprogram IHCFRXPI (FRXPI#), where R represents a real*4 variable or real*4 constant, and J represents an integer variable or integer constant, R is equal to zero and J is less than or equal to zero.

IEC243I For an exponentiation operation (D**J) in the subprogram IHCFDXPI (FDXPI#), where D represents a real*8 variable or real*8 constant and J represents an integer variable or integer constant, D is equal to zero and J is less than or equal to zero.

IHC244I For an exponentiation operation (R**S) in the subprogram IHCFRXPR (FRXPR#), where R and S represent real*4 variables or real*4 constants, R is equal to zero and S is less than or equal to zero.

IHC245I For an exponentiation operation (D**P) in the subprogram IHCFDXPD(FDXPD#), where D and P represent real*8 variables or real*8 constants, D is equal to zero and P is less than or equal to zero.

IHC246I For an exponentiation operation (Z**J) in the subprogram IHCFCXPI(FXCPI#), where Z represents a complex*8 variable or complex*8 constant and J represents an integer variable or integer constant, Z is equal to zero and J is less than or equal to zero.

IHC247I For an exponentiation operation (Z**J) in the subprogram IHCFCDXI(FCDXI#), where Z represents a complex*16 variable or

complex*16 constant and J represents an integer variable or integer constant, Z is equal to zero and J is less than or equal to zero.

IHC251I In the subprogram IHCSSQRT(SQRT), the argument is less than 0.

IHC252I In the subprogram IHCSEXP(EXP), the argument is greater than 174.673.

IHC253I In the subprogram IHCSLOG(ALOG and ALOG10), the argument is less than or equal to zero. Because this subprogram is called by an exponential subprogram, this message also indicates that an attempt has been made to raise a negative base to a real power.

IHC254I In the subprogram IHCSSCN(SIN and COS), the absolute value of an argument is greater than or equal to $2^{18} \cdot \pi$($2^{18} \cdot \pi$=.823549664062499 96D+06).

IHC259I In the subprogram IHCSTNCT (TAN or COTAN), the argument value is too close to one of the singularities ($\pi \pm \frac{\pi}{2}, \pm \frac{3\pi}{2}$,...for the tangent or $\pm\pi, \pm 2\pi$,... for the cotangent).

IHC261I In the subprogram IHCLSQRT(DSQRT), the argument is less than 0.

IHC262I In the subprogram IHCLEXP(DEXP), the argument is greater than 174.673.

IHC263I In the subprogram IHCLLOG(DLOG and DLOG10), the argument is less than or equal to zero. Because the subprogram is called by an exponential subprogram, this message also indicates that an attempt has been made to raise a negative base to a real power.

IHC264I In the subprogram IHCLSCN(DSIN and DCOS), the absolute value of the argument is greater than or equal to $2^{50} \cdot \pi$($2^{50} \cdot \pi$=.353711887378022 39D+16).

IHC265I In subprogram IHCLATN2, when entry name DATAN2 is used, both arguments are equal to zero.

IHC266I In the subprogram IHCLSCNH (DSINH or DCOSH), the absolute value of the argument is greater than or equal to 174.673.

IHC267I In the subprogram IHCLASCN (DARSIN or DARCOS), the absolute value of the argument is greater than 1.

ICH268I In the subprogram IHCLTNCNT (DTAN or DCOTAN), the absolute value of the argument is greater than or equal to $2^{50} \cdot \pi$($2^{50} \cdot \pi$=.353711887378022 39D+16).

IHC269I In the subprogram IHCLTNCT (DTAN or DCOTAN), the argument value is too close to one of the singularities ($\pm \frac{\pi}{2}, \pm \frac{3\pi}{2}$,...for the tangent; $\pm\pi, \pm 2\pi$,... for the cotangent).

IHC271I In the subprogram IHCCSEXP (CEXP), the value of the real part of the argument is greater than 174.673.

IHC272I In the subprogram IHCCEXP (CEXP), the absolute value of the imaginary part of the argument is greater than or equal to $2^{18} \cdot \pi$ ($2^{18} \cdot \pi$=.82354966406249996D+06).

IHC273I In the subprogram IHCCSLOG (CLOG), the real and imaginary parts of the argument are equal to zero.

IHC274I In the subprogram IHCCSSCN (CSIN or CCOS), the absolute value of the real part of the argument is greater than or equal to $2^{18} \cdot \pi$ ($2^{18} \cdot \pi$=.82354966406249996D+06).

IHC275I In the subprogram IHCCSSCN (CSIN or CCOS), the absolute value of the imaginary part of the argument is greater than 174.673.

IHC281I In the subprogram IHCCLEXP (CDEXP), the value of the real part of the argument is greater than 174.673.

IHC282I In the subprogram IHCCLEXP (CDEXP), the absolute value of the imaginary part of the argument is greater than or equal to $2^{50} \cdot \pi$ ($2^{50} \cdot \pi$= .35371188737802239D+16).

IHC283I In the subprogram IHCCLLOG (CDLOG), the real and imaginary parts of the argument are equal to zero.

IHC284I In the subprogram IHCCLSCN (CDSIN or CDCOS), the absolute value of the real part of the argument is greater than or equal to $2^{50} \cdot \pi$ ($2^{50} \cdot \pi$=.35371188737802239D+16).

IHC285I In the subprogram IHCCLSCN (CDSIN or CDCOS), the absolute value of the imaginary part of the argument is greater than 174.673.

IHC290I In the subprogram IHCSGAMA (GAMMA), the value of the argument is outside the valid range ($2^{-252} < x < 57.5744$).

IHC291I In the subprogram IHCSGAMA (ALGAMA), the value of the argument is outside the valid range ($0 < x < 4.2937 \times 10^{73}$).

IHC300I In the subprogram IHCLGAMA (DGAMMA), the value of the argument is outside the valid range ($2^{-252} < x < 57.5744$).

IHC301I In the subprogram IHCLGAMA (DLGAMA), the value of the argument is outside the valid range ($0 < x < 4.2937 \times 10^{73}$).

IHC900I Attempts to obtain core space for a direct access I/O buffer have failed. A restart at location 7E (HEX) of module IHCDIOSE will again attempt to allocate this buffer.

IHC901I A direct access read statement has resulted in buffer overflow. This is probably caused by an incorrect record length specification in a DEFINE FILE statement. IHCDIOSE

## Program Interrupt Messages

All program interrupts will be signalled by the message

IHC210I PROGRAM INTERRUPT--OLD PSW* IS XXXXIIIIXXAAAAAA

on the logical device SERCOM. After the printing of this message, program execution is resumed if the interrupt was one of the eight arithmetic interrupts. The non-arithmetic interrupts result in control being passed to the MTS subroutine ERROR. Caution: Any attempt to usurp program interrupt control from the FORTRAN-provided processor will be viewed as a violation of the First Law of FORTRAN.

## Non-arithmetic Program Interrupts

Operation Exception - Interrupt Code 0001
An attempt has been made to use an unassigned operation code. Probably due to incorrect boundary alignment or an attempt to execute data. ILC=1,2,3**

Privileged-Operation Exception - Interrupt Code 0002
Certain System/360 instructions are privileged and may be executed only when the computer is in the supervisor state. FORTRAN object modules are always run with the computer in problem state. An attempt to execute one of these privileged instructions will thus be greeted by an interrupt. The operation is suppressed. Probable cause is incorrect boundary alignment or an attempt to execute data. ILC=1,2

Execute Exception - Interrupt Code 0003
When the subject instruction of an EXECUTE is another EXECUTE, an exception is recognized. The operation is suppressed. ILC=2

Protection Exception - Interrupt Code 0004
Contained in the program status word (PSW) is a four bit protection key. Similarly, associated with every 2048 byte block of storage there is a protection key. When a mismatch between the PSW key and the core storage key is detected, a protection exception is recognized. The operation is suppressed on a store violation, except in

---

*The program status word (PSW) is fully explained in The System/360 Principles of Operation A22-6821-5. In a nutsnell, two fields of this double word are significant: the interrupt code field denoted IIII and the address field denoted AAAAAA. This address usually points to the instruction immediately following the one that caused the error.

**Instruction length code.

the case of variable length operations, which are terminated. Except for EXECUTE, which is suppressed, the operation is terminated on a fetch violation. Probably incorrect argument in a subroutine call, or a wild subscript or transfer. ILC=0,2,3

Addressing Exception - Interrupt Code 0005
When an address specifies any part of data, an instruction, or a control word outside the available storage for the particular installation, an addressing exception is recognized. Probable causes are the same as for the protection exception. ILC may be anything.

Specification Exception - Interrupt Code 0006
A specification exception is recognized when:

1. A data, instruction, or control-word address does not specify an integral boundary for the unit of information.

2. Improper register designation.

There are a number of other possible causes of this exception; however, the first one listed above should be the most popular with FORTRAN users. Probable cause is an incorrect argument to a subroutine, i.e., a short operand where a long operand should be or a half-word integer instead of a full-word integer. ILC=1,2,3

Data Exception - Interrupt Code 0007
A data exception is recognized when a decimal operand is incorrectly specified. The operation is terminated. ILC=2,3

Arithmetic Program Interrupts

Fixed-Point-Overflow Exception - Interrupt Code 0008
When a high-order carry occurs or high-order significant bits are lost in fixed-point add, subtract, shift, or sign control operations, a fixed-point-overflow exception is recognized. The operation is completed by ignoring the information placed outside the register. The interrupt may be masked by PSW bit 36, but is not. ILC=1,2

Fixed-Point-Divide Exception - Interrupt Code 0009
A fixed-point-divide exception is recognized when a quotient exceeds the register size in fixed-point division, including division by zero, or the result of CONVERT TO BINARY exceeds 31 bits. Division is suppressed. Conversion is completed by ignoring the information placed outside the register. ILC=1,2

Decimal-Overflow Exception - Interrupt Code 000A
When the destination field is too small to contain the result field in a decimal operation, a decimal-overflow exception is recognized. The operation is completed by ignoring the overflow information. The

interruption may be masked by PSW bit 37, but is not. Since FORTRA does not use the decimal operations, this exception should be rare ILC=3

Decimal-Divide Exception - Interrupt Code 000B

When a quotient exceeds the specified data field size, a decimal divide exception is recognized. The operation is suppressed. Sinc FORTRAN does not use the decimal operation, this exception should b rare. ILC=3

Exponent-Overflow Exception - Interrupt Code 000C

The result exponent of a floating-point addition, subtraction multiplication or division overflows, i.e., exceeds 63, and the resul fraction is not zero. The operation is completed by replacing th result with a true zero. The condition code is set to 3 for additio and subtraction and remains unchanged for multiplication and division ILC=1,2

Exponent-Underflow Exception - Interrupt Code 000D

The result exponent of a floating-point addition, subtraction, multiplication or division underflows, i.e., is less than -64, and th result fraction is not zero. The operation is completed by replacin the result with a true zero. The interruption may be masked by PS bit 38, but is not. The condition code is set to zero for additio and subtraction and remains unchanged for multiplication and division. ILC=1,2

Significance Exception - Interrupt Code 000E

The result fraction of an addition or subtraction is zero. A progra interrupt occurs if the significance mask bit (PSW bit 39) is one, which it is not. Because this interrupt is not enabled, no interrup occurs and the operation is completed by replacing the result with a true zero. Any attempt to enable this interrupt by changing the program mask in the PSW will be regarded as a blatant violation of the Second Law of FORTRAN.

Floating-Point-Divide Exception - Interrupt Code 000F

The divisor in a floating-point divide operation has a zero fraction. The division is suppressed. Because the operation is suppressed and because FORTRAN resumes execution, division by zero is equivalent to division by 1, i.e., the quotient equals the numerator. ILC=1,2

# I O H / 3 6 0

# IOH/360 - I/O WITH CONVERSION

This section describes the usage and structure of the I/O routines whic provide conversion via a format for MAD/I and assembly language users (no for FORTRAN users). This writeup assumes that the user is acquainted wit the ASSEMBLER LANGUAGE for the IBM/360 or with MAD/I. In this writeup, th phrase "I/O Conversion Subroutine" is synonymous with "IOH/360".

## I. PRELIMINARY DEFINITIONS

These definitions are complete in themselves, the meaning and usag of H through P (below) will be explained in the course of th writeup.

### A. CHARACTER

A character is one of the 256 combinations of two hexadecimal digit (00 to FF [base 16]) in the IBM/360. When printed on an IBM 140 printer, they print as 63 distinct characters (blank is considered a character; "!" and "¢" do not print on the normal print chains).

### B. DIGIT

A digit is one of the following characters:

0 1 2 3 4 5 6 7 8 9

### C. LETTER

A letter is one of the following characters:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z

### D. SPECIAL CHARACTER

A special character is one of the following:

= ' ( $ * " ) / + - # < > | ; : % ? _ @ ¢ . , & ¬ !

E. NUMBER

A number is a string of one or more digits in the range -32768 to +32767.

F. LINE_IMAGE

The word line-image in this writeup refers to :

A) The image of the output line that will be printed (usually 132 columns wide).

OR

B) The image of the card that has been read (usually 80 columns wide).

OR

C) The image of the card that will be punched (usually 80 columns wide).

OR

D) Any other character string of length ≤ 32,767 bytes in a core-to-core movement.

G. LINE-POINTER

The line-pointer indicates the column in the line-image where the next character will be put. It is analogous to the position of a typewriter carriage. The value of the line pointer, at any point, is the number of the column to which it is pointing.

H. I/O CALL

An I/O (INPUT-OUTPUT) call is one to a subroutine the purpose o which is to effect a movement of information --- input (from th outside world to the computer or from one area in the computer t another area in the computer) or output (from the computer to th outside world or from one area in the computer to another area in th computer). Information given to the subroutine for formatted I/ CALLS consists of (1) the location of a format, (2) a set of lists and optionally (3) the location of a symbol table and logical uni number. In this writeup only formatted I/O calls will be discussed and the term I/O CALL will mean formatted I/O CALL.

I. LANGUAGE CALLS

a) An ASSEMBLER LANGUAGE call is a call on one of the followin subroutines using the ASSEMBLER LANGUAGE:

IOHIN
IOHOUT
IOHETC
ONE@ATIM

IOHETC and ONE@ATIM must have been preceded by calls to IOHIN o IOHOUT. See the description in section 255 of macros fo generating these calls.

b). A MAD/I call uses one of the following statement types:

'PRINT FORMAT'
'PUNCH FORMAT'
'READ FORMAT'
'LOOK AT'

J. LIST

A list is a sequence of adcons. A list is headed by a full-wor adcon pointing to a half-word integer which contains the number o elements on the list (i.e. the number of adcons on the lis following the header adcon). The half-word also acts as a switch i telling whether the following list elements are A-type or S-typ adcons. The head is followed by an adcon pointing to the firs character of the format. Following the format-adcon are a variabl number of sublists.

K. SUBLIST

A sublist is a sequence of adcons. The adcon at the head of a sublist points to a half-word integer which contains a number which is treated as the number of elements in the sublist following the header; the half-word also acts as a switch in telling how the following adcons in the sublist are to be interpreted.

L. FORMAT

A format is a string of characters which tells the form and positioning of the input or output image.

M. FORMAT-BREAK CHARACTER

A format-break character separates or delimits format terms. There are two types: explicit (written) and implicit (not written).

1) Explicit
   a) the left-delimiting format-break characters are:
      i) "," - comma; the general break character
      ii) "(" - left parenthesis
      iii) beginning of format
   b) the right-delimiting format-break character is ")" - right parenthesis

2) Implicit - the "*" has an implicit format-break character to its left. The "#", "/", "|", "?", and literal fields have implicit format-break characters both to their right and left. That an implicit format-break character exists in a given position in a format means that a format-break character does not have to be explicitly written in that place; e.g. it is not necessary to put a comma before or after a literal term to separate it from the surrounding terms.

N. FORMAT-TERM

A format-term is the string of characters between two format-brea characters. It does not include the format-break characters. It ma be an empty string (i.e. consecutive format-break characters ar allowed).

O. FORMAT TERMINATOR

The format terminator is a character which signifies the logical en of a format. This is an asterisk, "*". (NOTE: the question mark "?", is also a kind of format terminator - see USAGE).

P. CONTEXTS

The meaning and effect a given character has in a format depends o the context in which it appears. The five contexts (each to b elaborated) are:

1. Normal
2. Literal
3. Format Variable
4. Format-Off
5. Default-Scan

Q. CONTROL CHARACTER

A control character (or specification character) is either a lette or a special character. It is the character which designates wha general type of conversion will take place. Each format term has exactly one control character; e.g. the control character C desig- nates character conversion.

R. MODIFYING CHARACTER

A modifying character is either a letter or a special character. It cannot stand alone in a format term (with one exception -- an F may modify an immediately preceding F, as in FF2.3). It must always appear with a control character. Its presence changes or modifies the action of the control character; e.g. on input the modifier R with the control character C causes characters read to be right-justified instead of left-justified. A modifying character which does not require a count may appear anywhere in the format term; modifiers which require counts (i.e. @, G, and A ) must come after the specification control character and the fields associated expressly with the control character and must be separated from the preceding fields by a colon, ":". In addition, if there are two modifying fields requiring counts, they must be separated by a colon, ":".

S. DATA-TRANSMISSION

A data-transmission format term or control character is one which requires reference to the parameter list.

T. NON-DATA-TRANSMISSION

A non-data-transmission format term or control character is one which does not require reference to the parameter list.

U. EXTERNAL FIELD WIDTH

An external field width is a set of contiguous columns on an input or output image; the maximum width is 256. This is not the restriction to the image length. If the external field width is explicitly zero in a format term, then the following will occur. For input, a zero will be read into the associated parameter list item if the control character specifies numeric conversion; if the control character specifies character conversion, then the associated parameter list item will be filled with the present input fill character (which is initially a blank). For output the number will be put into zero columns; i.e. the list item will be skipped.

V. FIELD WIDTH

The field-width is the number of columns, bytes, or digits in the specified field.

## II. I/O SUBROUTINE CALLING SEQUENCES

In this section, the means of calling the I/O subroutines from MAD/I and the ASSEMBLER LANGUAGE are given. In the case of MAD/I only the statement type is given. Details may be found in the MAD/I MANUAL. In the case of the ASSEMBLER LANGUAGE, only one of the numerous ways to write the call is given. In general assembly language users should use the macros provided for generating these calls (see section 255).

### A. GENERAL STRUCTURE OF THE CALLING SEQUENCES

There are usually three parts of a call to the I/O conversion subroutine. The first part is the executable code; the other parts are the parameter lists. The executable code points (through general register 1) to the beginning of the first parameter list when the actual call to the I/O conversion subroutine is made. The last parameter in the first parameter list contains a pointer to the beginning of the second parameter list. The first parameter list is made up of four full-word adcons in the following form:

```
PLIST1  DC A(OPEN)     pointer to "OPEN ROUTINE"
        DC A(CLOSE)    pointer to "CLOSE ROUTINE"
        DC A(PLIST3)   pointer to third parameter list
        DC A(PLIST2)
```

The adcons pointing to the generic names OPEN and CLOSE will be explained shortly. PLIST3 is the address of the third (optional) parameter list. The third adcon may be a full-word zero; this implies that the third parameter list is not present. This parameter list will be described shortly. The fourth adcon points to the head of the second parameter list. The second parameter list is made up of a variable number of adcons. This list has at its head a full-word adcon pointing to an integer half-word location which contains the total number of adcons following the head. If this integer half-word is positive, then the rest of the list is made up of full-word (type-A) adcons; if it is negative, then the list is made up of half-word (type-S) adcons. The first adcon following the head of the second parameter list points to the beginning of the format to be used by the I/O conversion subroutine. The format must be present in calls to IOHIN or IOHOUT. It may be null in calls to IOHETC in which case the adcon should be a zero. If the format adcon is non-zero in a call to IOHETC, then the contents of this adcon will be taken as a pointer to a new format and all "remembrance" of the previous format is erased. The adcons following the format adcon constitute a variable number of sublists. Each sublist also has a header which points to an integer half-word location which contains the total number of adcons in the sublist following the sublist header. These adcons (the ones following the sublist header) point

to the locations from which or to which I/O conversion is to take place. The half-word integer to which the header of a sublist points may be positive or negative. If positive, each adcon in the sublist is used for only one conversion. If negative, then the (negative) number must be even. The adcons in this kind of sublist will be taken as pairs for block addresses. The first adcon of a pair contains the beginning address of the block; the second adcon of a pair contains the end address of the block. I/O processing is terminated whenever one of the adcons in a sublist following a header is zero. A count of zero in the half-word integer constant to which the sublist header points causes the I/O subroutine to interpret the next adcon in the list as a pointer to a new sublist header.

The third parameter list is of variable length. The first word of this list points to a halfword which contains the number of arguments in this list. The second adcon in this list points to a double-word aligned location which is the beginning of the user's symbol table. If this adcon is zero, then no symbol table is present. The third adcon (if present) points to a full-word location containing either a logical unit number or a FDUB pointer. If this adcon is zero then input (or output) will be done on SCARDS (or SPRINT or SPUNCH). This parameter list may be extended in the future. Note that if the UNIT address is given but not the symbol table address, the count must be 2 and SYMTBL must be a full-word zero.

<PLIST1>::=<pointer to OPEN> <pointer to CLOSE> <optional pointer to PLIST3> <pointer to PLIST2>

<PLIST2>::=<pointer to total number args in PLIST2> <pointer to format> <list>

<PLIST3>::=<pointer to total number of args in PLIST3>
[<pointer to SYMBTBL>[<pointer to unit number>]]

<list>::=<sublist> | <list> <sublist>

<sublist>::=<pointer to number args in sublist> <slist>

<slist>::= | <slist> <argument>

<argument>::=legal OS expression

example of the second parameter list:

```
PLIST2  DC A(FULCNT) full list count
        DC A(FORMAT) format location
        DC A(H1) sublist count (a two in this case)
        DC A(ARG) single argument
        DC A(ARGS) single argument
        DC A(S2) sublist count (a minus 4 in this case)
```

```
                DC A(ARRAY) beginning address in block
                DC A(ARRAY+100) end address of block
                DC A(BARRAY+200) beginning address of block
                DC A(BARRAY+1000) end address of block
                DC A(S3) sublist count (a one in this case)
                DC A(0) and of I/O conversion
                ...
                ...

FULCNT  DC H'9'
H1      DC H'2'
S2      DC H'-4'
```

B. DESCRIPTION OF THE OPEN AND CLOSE ROUTINES

The names OPEN and CLOSE to be used henceforth are generic names: they stand only for the concepts embodied in the following paragraphs. The user may supply his own specific OPEN and CLOSE routines for obtaining and releasing LOGICAL images. If the user is forming his own lists rather than using the macros defined to do so (i.e. RDFMT, PRFMT, etc.), then he must supply the adcons pointing to the OPEN and CLOSE routines whether these routines are provided by the system or by the user (see MACRO CALLS TO IOH/360 writeup in section 255 for Macro Type calls). The OPEN and CLOSE routines are called by the I/O conversion subroutine to close-out a line image or get a new line image.

C. THE FUNCTION OF THE OPEN AND CLOSE ROUTINES

INPUT - On input, the OPEN routine returns to IOH/360 with general register 1 pointing to a two-word adcon area. The first adcon contains the address of the beginning of the input image. The second adcon points to a half-word location containing the length of the input image. When IOH/360 calls the OPEN routine, register 1 points to a four word adcon area the fourth adcon of which points to a FDUB pointer or logical unit number. If this adcon is zero, then standard logical unit is assumed. (SCARDS for input, SERCOM,SPRINT or SPUNCH for output). The OPEN routine should take this into account. A call to the CLOSE routine may cause (it need not do so) the present image to be accepted so that a subsequent call to the OPEN routine will transmit information about a new image with which IOH/360 is to work. When IOH/360 calls the CLOSE routine, general register 1 points to a four-word adcon area. The first adcon contains the address of the beginning of the image. The second adcon points to a half-word location containing the length of the image. The third adcon points to a half-word location containing the greatest excursion of the line pointer during conversion in the image. The fourth-adcon points to the full-word logical unit number or FDUB pointer. If this adcon is zero, then the standard logical unit is assumed (see above). The CLOSE routine may or may not wish to use this information. Successive calls to OPEN without intervening CLOSEs should present IOH/360

with the same input image. The return code must be given by the OPEN and CLOSE routines: 0-successful, 4-EOF.

OUTPUT - On output, the OPEN routine returns to IOH/360 with general register 1 pointing to a two-word adcon area. The first adcon points to the beginning of the output image. The second adcon points to a half-word location containing the length of the image. Note that IOH/360 only inserts conversion terms; it does not blank out the image before processing --- it is the responsibility of the OPEN routine to do this (if it wishes to do so, which is the usual procedure). The OPEN routines provided by the system will blank out the line image, however. On a call to the CLOSE routine general register 1 points to a four-word adcon area. The adcons point respectively to the image, the half-word containing the length of the image, the half-word containing the greatest excursion of the line pointer, and a full-word containing a logical unit number of a FDUB pointer. If the fourth adcon is zero, then the standard logical unit is assumed (see above). Again, use of this information is at the discretion of the user. The return code must be given by the OPEN and CLOSE routines: 0-successful, 4-EOF (whatever an EOF might mean on output).

examples:

a) return from an OPEN routine to IOH/360

```
            LA 1,PLIST pickup address of return list
            ...
            SR 15,15 return code
            BR 14 return
            ...
            ...
            ...
PLIST       DC A(IMAGE)
            DC A(AH)
            ...
            ...
            ...
AH          DC H'256'length of image region
IMAGE       DS 256C image region
```

b) call from IOH/360 to a CLOSE routine:

```
            LA 1,LIST point to parameter list
            L 15,=V(CLOSE) address of CLOSE routine
            BASR 14,15
            ...
            ...
            ...
LIST        DC A(IMAGE)  point to IMAGE region
            DC A(COUNT) length of IMAGE region
            DC A(LASTCOL) amount used by IOH/360
```

```
            DC A(RUNIT) logical unit address
            ...
IMAGE       DS 256C image region
COUNT       DS H half-word count;image length
LASTCOL     DS H highest excursion of line pointer
RUNIT       DS F word containing logical unit number
                 or a FDUB pointer
```

SPECIFICATION CHARACTERS

Usage - Normal Context

Since the meaning of a character in a format depends on the context in which it appears, a separate listing of character meanings is provided for each context. For each of the contexts other than Normal Context, a statement of its purpose and the way entry to and exit from this context is specified are given.

## NORMAL CONTEXT

The characters are listed according to their hexadecimal equivalents (given within brackets). Thus, there are 256 possible characters. Note that in all cases where the multiplicity applies, omitting the multiplicity is equivalent to giving a multiplicity of 1.

Note: IOH/360 does not actually "see" the characters in the format. Each character is used as an index to a translate table (through the use of the TRT instruction). The indexed byte contains a code representing what the format character means. The following list of characters is what each one means to IOH/360 when it is loaded. The user may cause such meanings to be changed (i.e. codes are changed in the table itself). For such manipulations, one should see the section entitled DEFAULT-SCAN CONTEXT. Also, in some of the following descriptions a DEFAULT CASE is given. The user may also change the DEFAULT CASE for any such conversion which has a DEFAULT CASE. See the section entitled DEFAULT-SCAN CONTEXT.

[00] to [3F]

These are illegal characters.

••••

[40] BLANK

A blank is ignored.

••••

[41] to [49]

These are illegal characters.

••••

¢ [4A] CENT-SIGN

The CENT-SIGN is an illegal character.

••••

. [4B] PUNCTUATION: PERIOD

The period is used in C, E, F, P, and X specifications. It tells the format scanner that the number accumulated so far (in the format) is to be considered as some kind of field width, and that a new number is to begin accumulating.

••••

< [4C] CHANGE-OF-CONTEXT: "LESS-THAN" SIGN

The "less-than" sign is used to signify a change from Normal Context to Format-Variable Context. Format-Variable Context remains in effect until the occurrence of a ">" ("greater-than" sign). (This is not yet implemented)

••••

( [4D] PUNCTUATION: LEFT PARENTHESIS

A group of format terms may be repeated by enclosing the group in parentheses and preceding the left parenthesis with a multiplicity (the multiplicity may be omitted in which case it is taken to be 1). Thus:

```
3E1.5.15,2(I2,3F2.5.10),2C8*
```

is equivalent to the following:

```
E1.5.15,E1.5.15,E1.5.15,I2,F2.5.10,F2.5.10,F2.5.10,I2,
F2.5.10,F2.5.10,F2.5.10,C8,C8*
```

Nested parentheses are allowed; there is no limit to the nesting depth. If the multiplicity in front of a left parentheses is zero, this means "Do what is inside zero times", which means "Do not do it at all." This causes a change from Normal Context to Format-Off context. Normal Context is resumed when the right parenthesis is found which pairs with the left parenthesis in front of which the zero multiplicity was found.

••••

\+ [4E] PLUS-SIGN

A Plus-Sign is ignored (it is treated as if it were a blank).

••••

| [4F] STROKE

The STROKE causes termination of an image without any reset. Thus, the appearance of a stroke causes a call to be made to the CLOSE routine WITHOUT a succeeding call to the OPEN routine; the image area and the line-pointer will not be reset; i.e. the image status remains as it was before the call to CLOSE.

••••

& [50] AMPERSAND

The AMPERSAND (when used in DEFAULT-SCAN CONTEXT) indicates a change from normal to keyword mode in this context. In Normal Context, the AMPERSAND is an illegal character.

••••

[51] to [59]

These are illegal characters.

••••

! [5A] EXCLAMATION-MARK

The EXCLAMATION-MARK is an illegal character.

$ [5B] DOLLAR-SIGN : FLOATING DOLLAR-SIGN MODIFIER

When this modifier is used in E, F, I, O, or P OUTPUT format terms, a dollar-sign, "$", is inserted in the output field immediately to the left of the first digit (or to the left of the sign if it appears). NOTE: The Dollar Sign is illegal as a character in the input image for any numeric field.

••••

* [5C] ASTERISK

The ASTERISK is the format terminator. It is the last logical character in the format. It causes a call to the CLOSE routine. A test is then made to determine whether or not the next conversion address is a zero (a zero adcon). If the zero adcon is not found, then resumption of the format scan continues at the last zero level left parenthesis, or if this does not exist, at the beginning of the present format. The OPEN routine will be called before the format scan is resumed. If the zero adcon has been found, then control is returned to the user at the statement after the last call to the I/O-conversion subroutine.

••••

) [5D] RIGHT-PARENTHESIS

See LEFT PARENTHESIS [4D] for use of parentheses.

••••

; [5E] SEMI-COLON

The SEMI-COLON is an illegal character.

••••

¬ [5F] "NOT" SYMBOL

The NOT SYMBOL is used on input in P fields and specifies that the low order byte of the packed-decimal number is to contain two decimal digits rather than the normal sign and digit.

••••

- [60] MINUS-SIGN

The MINUS-SIGN causes the sign of the number being accumulated by the format scanner to be inverted (i.e. --40 is equivalent to 40).

••••

/ [61] SLASH

The SLASH causes a call to the CLOSE routine followed by a call to the OPEN routine. Thus it causes the release of the current image and a request for a new image. The line pointer is reset to column 1.

••••

[62] to [6A]

These are illegal characters.

••••

[6B] PUNCTUATION: COMMA

The COMMA separates format terms. Successive commas are redundant.

••••

% [6C] PERCENT SIGN

The PERCENT SIGN tells the format scanner to use the current image length in place of the PERCENT SIGN. Thus the user may specify in his format:

T%,S-1

to place the line pointer at the second character from the end of the current image.

••••

_ [6D] UNDERLINE

The UNDERLINE is an illegal character.

••••

> [6E] "GREATER-THAN" SIGN

The "GREATER-THAN SIGN" is used to terminate Format-Variable Context. See description of "<" [4C].

••••

? [6F] QUESTION MARK

The QUESTION-MARK is used to tell the I/O-conversion subroutine to immediately return to the user and to ignore the rest of the present parameter list. The QUESTION MARK is to be used in conjunction with the secondary entry point IOHETC. The user may

return to the I/O-conversion subroutines via IOHETC and specify in his parameter list a new format address. This will be taken as the present format address and any references to a previous format address will no longer exist. In particular, placement of parentheses previously found and their associated multiplicities will be "forgotten". If the user specifies a null-format address on entry to IOHETC (i.e. an actual 0 in the format adcon), then the format scan is resumed at the character immediately following the QUESTION MARK.

••••

[70] to [79]

These are illegal characters.

••••

: [7A] COLON

The COLON separates the specification part of a format term from the optional modifiers which require counts (i.e. @, G, and A). It also separates these modifiers from each other if there is more than one.

examples:

I5:G8 or F4.6.20:A50:@+5

••••

# [7B] POUND-SIGN

The POUND-SIGN is used to change from Normal Context to Default-Scan Context and vice-versa.

••••

@ [7C] AT-SIGN

The AT-SIGN is used only in E or F output fields and is used to specify a scaling factor to be applied to the number. The AT-SIGN modifier is followed by a number specifying the scaling factor. The appearance of the AT-SIGN must come after all field widths have been specified for the accompanying the format term and must be preceded by a colon, ":".

Let P be the number found after the AT-SIGN. Then the scale factor is applied to an F-type number according to the formula:

external number = internal number * (10**P)

The scaling factor is essentially applied after the conversion is done and causes the decimal point to be moved and/or zeros to be supplied in front of or after the converted number.

EXAMPLE: Let the format be 2F5.5.11:@1,F5.5.11*. Then, the three numbers which would print

.56789 .98765 .12345

according to 3F5.5.11 would now print

5.67890 9.87650 .12345

Note that in F-type terms, application of the scale factor actually changes the number by some factor of ten by movement of the decimal point.

In E-type formats, only the exponent is changed; the scale factor is added to the exponent.
Thus, a number which would print as:

.9321E-03

according to a format E0.4.10 would print as:

.9321E-01

according to the format E0.4.10:@2

••••

[7D] PRIME

The PRIME indicates a change from Normal Context to Literal Context. See the section LITERAL CONTEXT.

••••

= [7E] EQUAL SIGN

The EQUAL SIGN is used in conjunction with vector-type format variables. The count preceding the equal sign (the "current count") must be greater than zero or omitted (in which case the count is taken as equal to 1). The count is then used as an index to a user-defined 'format-variable vector' which consists of

half-word integers. The indexed entry from the vector "replaces" the equal sign and its multiplicity (i.e. it becomes the new "current count"). See ENTRY POINTS TO IOH: IOHSFVAR.

EXAMPLE:
Assume that the third and fourth entries in the vector contain a 4 and a 2 respectively. Then the format term:

3=F4=.4=.20

is equivalent to the format term:

4F2.2.20

••••

" [7F] QUOTATION MARK

The QUOTATION MARK is an illegal character.

••••

[80]
This is an illegal character.

••••

a [81] Lower Case "A"

See A [C1].

••••

b [82] Lower Case "B"

See B [C2].

••••

c [83] Lower Case "C"

See C [C3].

••••

d [84] Lower Case "D"

See D [C4].

••••

e [84] Lower Case "E"

See E [C5].

••••

f [86] Lower Case "F"

See F [C6].

••••

g [87] Lower Case "G"

See G [C7].

••••

h [88] Lower Case "H"

See H [C8]

••••

i [89] Lower Case "I"

See I [C9].

••••

[8A] to [90]

These are illegal characters.

••••

j [91] Lower Case "J"

See J [D1].

••••

k [92] Lower Case "K"

See K [D2].

••••

l [93] Lower Case "L"

See L [D3].

••••

m [94] Lower Case "M"

See M [D4].

••••

n [95] Lower Case "N"

See N [D5].

••••

o [96] Lower Case "O"

See O [D6].

••••

p [97] Lower Case "P"

See P [D7].

••••

q [98] Lower Case "Q"

See Q [D8].

••••

r [99] Lower Case "R"

See R [D9].

••••

[9A] to [A1]

These are illegal characters.

••••

s [A2] Lower Case "S"

See S [E2].

••••

t [A3] Lower Case "T"

See T [E3].

••••

u [A4] Lower Case "U"

See U [E4].

••••

v [A5] Lower Case "V"

See V [E5].

••••

w [A6] Lower Case "W"

See W [E6].

••••

x [A7] Lower Case "X"

See X [E7].

••••

y [A8] Lower Case "Y"

See Y [E8].

••••

z [A9] Lower Case "Z"

See Z [E9].

••••

[AA] to [C0]

These are illegal characters.

••••

A [C1] CENTERING CONTROL

The CENTERING CONTROL character is used to tell the I/O-conversion subroutine at what column in the output image the converted field is to be centered. Thus with the format:

F1.4.10:A14

if the converted number was

bbb2.10000

(where b stands for a blank), then the decimal digit "2" in the converted number would appear in column 13 with three blanks before it and .10000 after it.

Let K represent the number found after the A in the format and let N represent the field width in the image (the total field width). Then the beginning of a field with field width N and centering at column K will start at column K-[N/2]+1 where the brackets denote integer division.

••••

B [C2] BYTE-SIZE FLAG

The appearance of the BYTE-SIZE FLAG anywhere in a format term indicates that conversion is to take place from or into an internal field one-byte long. See also BLOCK ADDRESSING SECTION for an implicit use. When using the B modfier on integer conversions the number is assumed positive on output and must be positive on input.

example: BI2, BC1, BI3

••••

## C [C3] CHARACTER FIELD

The normal form of a character field format term is:

C m.n

where m is a count referring to the width in the external image, and n is a count referring to the width of the internal character string.

OUTPUT:

N characters are left-justified (right-justified if R specified) in a field m columns wide with trailing (preceding, if R specified) padded output fill characters (which are initially blanks) if necessary. If m is omitted, then its default case is taken. If n is omitted, then n is set to the value of m. If both m and n are omitted but a decimal point appears in the format term, then the default case for m is taken and then n is set to the value of m. If standard format is used, the default for n is taken and the field is placed in the line image with literal-break characters surrounding it.

examples:

Let the argument be a seven byte character string, "ABCDEF" Then:

| | | |
|---|---|---|
| C7.1 | gives | Abbbbbb |
| C7.3 | gives | ABCbbbb |
| C7.7 | gives | ABCDEFG |
| C7 | gives | ABCDEFG |
| C3 | gives | ABC |
| C | gives | 'A' |

INPUT:

The first n of m characters are taken from the input image and are placed in n bytes. If R is specified, then the last n of m characters are used. If m is omitted, then its default case is taken. If n is omitted, then n is set to the value of m. If both m and n are omitted and no decimal point appears in the format term, then standard-format input is assumed (see STANDARD FORMAT INPUT SECTION).

examples:

Let columns 1 thru 7 contain the character string 'ABCDEFG'; then with the following format terms, the associated results will occur:

| | | |
|---|---|---|
| C1.7 | gives | Abbbbbb |
| C6.7 | gives | ABCDEFb |
| C7.10 | gives | ABCDEFGbbb |

APPLICABLE MODIFIERS:

R, L, D, DD, W, H, B - for input or output
see BLOCK-ADDRESSING SECTION for use of D, W, H, B

DEFAULT CASE for m and n is 1.

••••

D [C4] DOUBLE-WORD SIZE FLAG

The appearance of this flag anywhere in a format term indicates that conversion is to take place from or into an internal field eight bytes long which begins on a double-word boundary . However, if two D's appear in a format term, then conversion is to take place from or into an internal field sixteen bytes long which begins on a double-word boundary. In the case of E and F conversions, when two D's appear, then double-precision floating point numbers are assumed. See also BLOCK-ADDRESSING SECTION for an implicit use.

examples: DE, DD1.30.40, DE0.15.22

••••

E [C5] EXPONENTIAL FLOATING POINT

The normal form of an exponential floating point format term is :

E m.n.q

where:
m is the number of digits to the left of the decimal point
n is the number of digits to the right of the decimal point
q is the total external field width

OUTPUT:
If m, n, and q are absent and no decimal points appear in the format term, then the default cases for m and n for E-type formats are used. Q is taken as m+n+6 in this case (the "6" includes the sign, the decimal point, and the four characters in the exponent). If either m or n are absent (but either a decimal point is present or q is present [implying that two decimal points are present]) then the default cases are taken for the missing fields m and n (one or the other or both may be defaulted). Then if q is absent, it is set to m+n+6 .

examples:
The number 123.456 would print as the following with the indicated formats:

123.456E+00 E3.3.11

```
12.346E+01 E2.3.10
 1.235E+02 E1.3.9
 .124E+03  E0.3.8
 1.235E+02 E.3.9
 1.235E+02 E
```

INPUT:
If neither any widths nor any decimal points appear in the format term, then standard format input is assumed (see STANDARD FORMAT INPUT SECTION). Otherwise, if m or n or both are omitted, then the default cases are taken for them. If q is absent, then it is set to m+n+6 . If there is no decimal point in the input field, then the input number is scaled by 10**-n (i.e. a decimal point is assumed to be to the left of the nth digit counting from the right). The appearance of a decimal point in the input image overrides any specification for m or n.

examples:
The following numbers are all converted to an internal floating-point number equal to 123.456 with respect to the formats given:

```
123.456E+00  E
12.3456E+01  E1.4
 123456E+01  E2.4
```

APPLICABLE MODIFIERS:
DD, D, W - for input or output
@, J, $, Y, R, L, U, - for output only

DEFAULT CASE for m is 1 and n is 3. The mode default is D.

An exponential type floating point number is represented by:

[sign] [digit string] [decimal point] [digit string] [E|+|-|D] [digit string]

Note that a string such as 1+5 is equivalent to the string 1E+05.

All floating point numbers in E-format are output in the following form:

[sign] [m digits] . [n digits] E [±] [exponent]

••••

F [C6] NON-EXPONENTIAL TYPE FLOATING POINT

The normal format term representing this type of conversion is:

F m.n.q

where:
m is the number of digits to the left of the decimal point
n is the number of digits to the right of the decimal point
q is the total external field width

OUTPUT:
If m or n are absent, then they are filled in by their respective default cases. If q is absent and n is non-zero, then q is taken as m+n+2 (the "2" includes the sign and decimal point); if q is absent and n is zero, then q is taken as m+n+1 . If n is zero, then no decimal point will appear in the converted number. If all three widths are omitted and there are no decimal points present in the format term, then the number to be converted is examined with respect to the default case for m: if the number has an absolute value less than 10**-1 or greater than or equal to 10**(m-1) then the E-type standard format is used. Otherwise, m and n are set to their default cases and q is set to m+n+2 (if n is non-zero) or to m+n+1 (if n is zero).

NOTE: if the F is immediately followed by another F (i.e. FF), and if the number of digits in the converted number exceeds q, then the q high-order characters (including the sign if present) of the converted number are used (the last digit is not rounded). This corresponds to the G conversion on the 7090.

EXAMPLES:
The number 123.456 would print as the following with the indicated formats:

```
  123.456 F3.3
   123.46 F3.2
      123 F3.0
123.4560 F
 123.456 FF3.4.7
   123.4 FF3.3.5
```

INPUT:
F-type input is exactly like E type input. See above.

APPLICABLE MODIFIERS:
DD, D, W - for input or output
@, J, $, Y, R, L, U, - for output only

DEFAULT CASE for m is 5 and for n is 4. The mode default is D.

••••

G [C7] BASE MODIFIER

The BASE MODIFIER appears only in I-type format terms. It must appear after the specification of the field width and must be separated from main part of the format term by a colon, ":". The number after the "G" is taken as the conversion base. Conversion bases from 2 thru 36 inclusive may be used. The letters A thru Z represent the "digits" ten through thirty-five.

Example: I10:G16 converts an integer field using base-16 arithmetic.

••••

H [C8] HALF-WORD SIZE FLAG

The half-word flag indicates that conversion is to take place from or into a half-word . See BLOCK ADDRESSING SECTION for an implicit use.

EXAMPLES: IH5, HC4

••••

I [C9] INTEGER CONVERSION

The normal form of the format term for integer conversion is:

Im

where m is the total external field width.

OUTPUT:
If m is absent, then the default case for m for integer fields is taken. A field of size m is used into which an integer conversion takes place.

INPUT:
If m is absent, then standard format input is assumed (see STANDARD FORMAT INPUT SECTION). If m is present, then m columns on the input image are scanned for an integer number and are so converted.

APPLICABLE MODIFIERS:
B, H, W, D, G - for input or output
Y, $, J, U, - for output only

DEFAULT CASE for m is 8. The default mode is W.
Note: see B [C3] for restriction.

••••

[CA] to [D0]

These are illegal characters.

••••

J [D1] IGNORE FIELD-WIDTH MODIFIER

If a J appears in an E, F, I, O, or P format term, then the external field width is taken to be the least number of characters which can accomodate the converted field without any fill characters being used. Note that with the specification of J, the R, L, and Z modifiers have no effect.

••••

K [D2]

This is an illegal character.

••••

L [D3] LEFT-JUSTIFICATION MODIFIER

The appearance of the L modifier in a C, E, F, I, O, P, or X output format term causes the field to be left-justified with trailing output fill characters.

EXAMPLES:
In the following, let "b" stand for a blank. On output, if a number was converted as

b+1.234

according to the format term FY1.3.7, then it will be converted as

+1.234b

with the format term LFY1.3.7.

••••

M [D5]

This is an illegal character.

••••

N [D5] NULLS MODIFIER

The NULLS MODIFIER is used only in C fields on input. It specifies that the fill character (if needed) is to be a hexadecimal zero, [00].

••••

## O [D6] "OWN" CONVERSION

The normal form of this format term is:

O m.n.q

The user may specify that he wants to do his own conversion. He may access the m, n, and q fields and switches which tell whether these fields are blank (if the widths happen to be zero). He may use the modifiers R, L, DD, D, W, H, and B and may set them or reset them at his discretion. See USEFUL ENTRY POINTS TO IOH/360: OWN CONVERSION on how to set up a program to use this type of conversion.

••••

## P [D7] PACKED DECIMAL

The normal form of this format term is:

P m.n

where:
m is the external field width
n is the internal field width (the number of internal packed-decimal bytes to be used)

OUTPUT:
If m and n are present, then n bytes are unpacked and placed in m output columns --- the "¬" modifier [5F] may be used to include the low-order digit of the last byte as a decimal digit and not as a sign. If standard-format is specified, a scan takes place of the internal representation until either a legal sign appears as a low order digit or an illegal digit occurs in a byte (only numerics may appear in the high-order portion of a byte). If the "¬" modifier [5F] was specified the scan stops at the last byte containing legal digits (after which there is a byte containing illegal digits). Absence of the =¬" modifier implies that a sign must appear. A field width equivalent to one plus the number of possible digits is used for the output image field width. If n is absent, then the same internal scan as that used for standard-format is used, and n is determined accordingly. If m is absent, then it is given its default.

INPUT:
m digits are packed into n bytes with the sign in the low-order byte (see "¬" [5F] if the sign is not wanted). If standard-format is specified then n is set to the minimum number of bytes into which the external field can be packed -leading zeros count as digits in this case. If standard-format is not specified and

if m is absent, m is set to the minimum number of bytes int which the digits can be packed.

APPLICABLE MODIFIERS
D, W, H, B - for block-addressing
¬ - for input only
$, J, L, U, Y - for output only

DEFAULT CASE for m is 8.

Note: Decimal points may appear in input P-field (but only on per field). However, they are ignored. Thus, the user must kno the scaling factor implied by the placement of the decimal point if it appears.

••••

Q [D8] QUIT IF LIST EMPTY

The appearance of a "Q" anywhere (except within a literal field causes a switch to be tested; this switch is set only if the nex parameter address is zero (i.e. if the next list element signifie the termination of format conversion). If the switch has been set then the image is CLOSEd and control returns to the user. If th switch is not set, then format scan continues after the "Q".

EXAMPLE:
With the format (on output)

```
Q' NUMBER=' I5
```

the literal ' NUMBER=' will be output only if there is a conversion argument for the I5 format term following. If th "Q" were not in the format, there would be a portion of th output line with the literal 'NUMBER=' with nothing on th line after the literal.

••••

R [D9] RIGHT JUSTIFICATION

The appearance of an "R" anywhere in an E, F, I, O, P, C, or L output field means that the field is to be right justified wit leading output fill characters if needed (the output fill characte is initially a blank). This is the normal case for numeric conversions. On input, the R modifier pertains only to C fields and causes the rightmost (instead of leftmost) characters to be moved from the line image.

••••

[DA] to [E1]

These are illegal characters.

••••

S [E2] SKIP: COLUMN MANIPULATOR

The normal form is:

S n

where n is a number which tells how many columns to skip. N may be either positive or negative. If n is omitted, it is assumed to be 1.

EXAMPLE:
If the line pointer is at column 18, then S5 moves the line pointer to column 23, whereas S-5 moves the line pointer to column 13.

••••

T [E3] TRANSFER: COLUMN MANIPULATOR

The normal form is:

T n

where n is the column number to which the line pointer is to moved.
EXAMPLE: T50 moves the line pointer to column 50.

••••

U [E4] FILL IF ZERO

This modifier is for output only. If the argument to a P, I, E, or F field is zero, then the external field will be filled with output fill characters (the output fill character is initially a blank).

••••

V [E5] LIST-TYPE FORMAT VARIABLE

Whenever a "V" is encountered in a format term, the I/O-conversion subroutine assumes that the next item on the list points to a half-word integer; the contents of this half word is substituted for the appearance of the V.

EXAMPLE: let the next two list elements point to half-words containing respectively 5 and 8. then the format term:

VFV.2.16

is equivalent to

5F8.2.16

••••

W [E6] FULL-WORD FLAG

The appearance of a W anywhere in an F, E, or I field means tha the conversion is to take place into or from a full-word. Se BLOCK-ADDRESSING SECTION for an implicit use.

EXAMPLES: WI6, WE1.3.10

••••

X [E7] HEXADECIMAL CONVERSION

The normal form for this format term is:

X m.n

where:
m is the external field width (the number of hexadecimal digit: to be packed or unpacked)
n is the number of bytes into which or from which conversion i: to take place.

INPUT:
M digits in the input image are packed and right justified in a field n bytes wide. If both m and n are missing and no decimal point appears in the format, then standard format is assumed and n is set to the minimum number of bytes into which the field ca: be packed. Otherwise, if m is omitted, it is set to its default case. If n is omitted, then it is set to [(m+1)/2], where the brackets denote integer division. Note: leading zeros are not ignored on input.

OUTPUT:
N internal bytes are unpacked and placed in m image columns. If m is omitted, then it is set to its default case. If n is omitted, then it is set to [(m+1)/2], where the brackets denote integer division.

Note: Field width is never exceeded in X-type output conversions. Thus if, internally (in two bytes) one had 0123 and if one specified X3

(which is equivalent to X3.2), then the digit string 012 would be moved to the output field.

DEFAULT CASE for m is 8.

••••

Y [E8] FORCED PLUS-SIGN MODIFIER

This modifier is for output only. If the argument to an E, F, I, or P field is positive or zero, then a plus-sign is forced.
EXAMPLE:
With the conversion term I6 one might get 12345; with YI6, one would then get +12345.

••••

Z [E9] ZEROS MODIFIER

This modifier can be used on input for C fields only. It specifies that the input fill character is to be a character zero, [F0], for this conversion. On output, it can be used in an I, E, F, O, P, X, or C field to specify that the output fill character is to be a character zero, [F0], for this conversion. If the output is numeric and right-justified (the default case), then the zeros will be placed between the prefix characters ($,+,-) if they appear and the actual number. On non-numeric output and numeric output without prefix characters, zeros will fill on the right if L was specified or on the left if R was specified (or L was not specified). Note O-type conversion is considered as numeric output.

••••

[EA] to [EF]

These are illegal characters

••••

0 [F0] NUMERIC

The digit "zero".

••••

1 [F1] NUMERIC

The digit "one".

••••

2 [F2] NUMERIC

The digit "two".

••••

3 [F3] NUMERIC

The digit "three".

••••

4 [F4] NUMERIC

The digit "four".

••••

5 [F5] NUMERIC

The digit "five".

••••

6 [F6] NUMERIC

The digit "six".

••••

7 [F7] NUMERIC

The digit "seven".

••••

8 [F8] NUMERIC

The digit "eight".

••••

9 [F9] NUMERIC

The digit "nine".

••••

[FA] to [FF]

These are illegal characters.

Literal Context

LITERAL CONTEXT

Literal Context is entered when any literal break character (normally prime, i.e. "'"), is encountered in Normal Context. The purpose of tl Literal Context is to provide characters in the format which can be pu into the line-image on output or be replaced by characters from tl line-image on input. This is to be used for titles, labels, and othe constant information.

Literal Context remains in effect until the next occurrence of a litera break character after which there is no immediate occurrence of anothe literal break character. The appearance of two successive literal brea characters after the initial literal break character allows one to includ the character which is the first of the two successive literal brea characters as part of the literal. Thus such pairs of literal brea characters are taken as one appearance of such a literal within the litera field. The format:

```
'THIS IS A LITERAL ''' *
```

would print as

```
THIS IS A LITERAL '
```

whereas

```
'THIS IS NOT A LITERAL '' *
```

is illegal.

On input the appearance of two successive literal break characters withi the literal field is ignored; neither terminates Literal Context.

NOTE: the limit to the number of characters between any occurrence of literal break character (whether it initiates, terminates, or exists i literal context) is 256. This does not restrict the total length of literal field, however.

Format-Off Context

A multiplicity of zero in front of a format term or left parenthesis means " do it zero times", i.e. do not do it. Therefore 0F5.5.11 will do nothing and

```
S10,0(S10,F5.5.11,'TRA '/),I3*
```

will skip 10 columns and print out a 3 column integer. Nothing inside the parentheses will be done. (This finds most use where there is some kind of format-variable, rather than an explicit zero multiplicity in front of the left parenthesis). A zero multiplicity in front of a left parenthesis causes a change from Normal Context to Format-Off Context. When scanning in this context, the only things recognized are left and right parentheses. The format terminator is not recognized in Format-Off Context. The context changes back to Normal Context when the right parenthesis is found which matches the left parenthesis which had the zero multiplicity.

## Default-Scan Context

In Default-Scan Context the user may change the default cases which are initially set by IOH/360 and may change the interpretation that IOH/360 is to make on any of the characters in a format term --- thus, among other things, the user may change the literal break character to any character he likes.

Default-Scan Context is initiated by a #-sign in Normal Context and is terminated by the occurrence of a #-sign in Default-Scan Context. There are two modes of operation within Default-Scan Context. One mode looks very much like Normal Context. The other mode is keyword mode.

### KEYWORD MODE

Keyword mode is entered when the keyword initiator is found (which is initially an ampersand, "&"). The keywords are not translated by IOH/360 (except to be translated so that all characters are in upper case); thus the keywords must always appear as in this writeup. The keyword is almost always terminated by an equal sign "=" (the exception is PUSH and POP) --- the terminator may not be changed as it is considered actually a part of the keyword. Following the equal sign may be a variable length operand (depending on the keyword used).

### DESCRIPTION OF KEYWORDS

Note: the following characters can never have their interpretations changed, but other characters may assume the interpretations of the three. They are: # [7B], [00], and [FF]. It should also be noted that although the upper and lower case letters have equivalent interpretations initially, there is nothing preventing, for example, a to have a different interpretation than A has.

••••

OUTFILL - change the current output fill character to the character immediately following the equal sign. Thus after interpretation of the following format, the output fill character will be "@".

```
#&OUTFILL=@#
```

••••

INFILL - change the current input fill character to the character immediately following the equal sign. Thus after interpretation of the following format, the input fill character will be a character zero.

```
#&INFILL=0#
```

••••

OVCHR - the overflow fill character is set to the character immediately following the equal sign. The overflow fill character is initially an asterisk. The overflow fill character fills a whole field on output when field width has been exceeded and when the bit is set (done on a call to SETIOHER) which allows such filling to occur. Thus after interpretation of the following format, the overflow fill character will be a dollar-sign

```
#&OVCHR=$#
```

••••

BREAK - interpret the character immediately following the equal sign as a literal break character from now on. The user may have any number of literal break characters. Thus, after interpretation of the following, the double-quote will be a literal break character.

```
#&BREAK="#
```

••••

EQUIV - the first character immediately following the equal sign is given the interpretation of the second character. Thus after interpretation of the following format, the appearance of an A in a format has the same interpretation as B has at the time that this format is interpreted.

```
#&EQUIV=AB#
```

••••

EXCHANGE - the two characters immediately following the equal sign have their interpretations exchanged. Thus after the interpretation of the following format, D stands for character conversion and C stands for double word flag (assuming that their interpretations have not been changed previously).

```
#&EXCHANGE=DC#
```

••••

REPLACE - the first character immediately following the equal sign is given the interpretation of the second character and the interpretation of the second character is then made null (i.e. illegal). Thus after the interpretation of the following format, C will stand for type-E format and E will be an illegal character (assuming that their interpretations have not been changed previously).

```
#&REPLACE=CE#
```

Note: to make a character illegal, one should "replace" the character with itself --- i.e. #&REPLACE=aa# makes "a" an illegal character.

••••

MODE - the characters immediately after the equal sign specify another keyword which may be either BCD or EBCDIC. Initially MODE is set to EBCDIC. The MODE setting determines whether plus-signs when put into the line image in a numeric conversion will be either in BCD mode (a 12 punch) or EBCDIC mode (a 12-6-8 punch). IOH/360 accepts either the BCD or EBCDIC representations of the plus-sign on input regardless of the mode setting. Thus after the interpretation of the following format, all plus-signs produced on a numeric conversion by IOH/360 will be in BCD mode.

```
#&MODE=BCD#
```

••••

PUSH and POP - these keywords allow the user to go from one default level to another. PUSH is used to go to a new default level and after this PUSH, the user has at this level all default cases as they are initially given. POP is used to return to a previous default level. After a POP the information in the level from which we POPped is lost. Thus the user may have one meaning for A at one level and a totally different interpretation at another level. Thus:

```
#&BREAK="&REPLACE=''&PUSH&BREAK=?#
```

makes the double quote the only break character at the first level while quote-mark and question mark are legal break characters at the second level.

•••

POPALL - returns the user to level 0 with respect to PUSH and POP.

••••

STATUS - the next item in the parameter list is assumed to point to a half-word into which the present PUSH-POP level number is placed. The initial level number is 0.

••••

RESET - the RESET keyword is used to reset certain defaults at the present level. The amount that may be to be reset (to initial conditions) is specified in a second keyword immediately following the equal sign. The second keywords are as follows.

ALL - all characters will have their initial interpretations and all defaults for all data transmission conversions will have their initial values.

TABLE - only the interpretations of the format characters will be reset.

DEFAULT - only the default cases for data transmission terms will be reset to their initial values.

BREAK - the literal break character is reset (to a quote-mark) and all

other literal break characters are reset to their initial interpretations.

OUTFILL - the output fill character is reset to its initial value (a blank).

INFILL - the input fill-character is reset to its initial value (a blank).

OVCHR - the over flow fill character is reset to its initial value (an asterisk).

MODE - the mode is reset to EBCDIC

If none of the above keywords follows the equal sign, then the character immediately following the equal sign is given its initial interpretation.

••••

## NORMAL-TYPE CONTEXT

The user may specify that he wishes to change or reset the default cases for certain data transmission format terms. This is done in the following manner. If a length modifier appears in an E, F, or I format term, then the normal mode of that conversion will be set to the mode specified by the length modifier. If neither any widths nor decimal points appear in the format term, then the default cases for that type of conversion are set to the initial conditions (this resets only the defaults for the widths). If any fields are explicitly present, then only the default cases for such fields are set to reflect what is in the format term.

EXAMPLES: E1 resets the m default for E-type conversions to 1. DE resets the mode default for E-type conversions to D and resets m and n defaults to their initial values. WF.3 resets the mode default for F-type conversion to W and resets the n default to 3.

••••

## Format-Variable Context

Use of FORMAT-VARIABLE CONTEXT allows substituting the value of a variable or expression in the program making the I/O call into a format anywhere where a number would otherwise be placed. This substitution takes place at the time the Format-Variable Context is encountered during the scan of the format. The appearance of a "<" (less-than sign) initiates Format-Variable Context; the appearance of a ">" (greater-than sign) terminates Format-Variable Context. Note there are two other types of format variables also. See "V" [E5] and "=" [7E] for their use. The format-variables described here are somewhat more powerful than the other two types as one can allow any expression which does not include function calls to act as a format-variable. The variables in a format-variable may be of any mode --- the resulting value of the format-variable must be an integer in the range -32768 to +32767. The use of this type of format-variable requires that the symbol table address be given as the third adcon of the first parameter list.

Warning: when using a format-variable as a multiplicity, remember that varying the multiplicity does not vary the number of items on the list. If it is necessary to skip items, use data-transmission format terms with zero external field-widths to do it.

examples:

```
<A+5*C> F 2.<Q+Z/2>
```

Note: As yet the format of the Symbol Table has not been defined. Thus "<" is an illegal character until further notice.

USEFUL ENTRY POINTS TO IOH/360

SETFRVAR - register 1 points to the beginning of a format-variable vector. The beginning of the vector must be aligned on a half-word boundary. See = [7E] on how to use this vector in formats.

SETIOHER - general register 1 points to a 4-byte area (need not be aligned) which contains codes to tell IOH/360 what to do in special circumstances. The codes for the first byte are:

Bit 1 - ERROR RETURN allowed if set ON
Bit 2 - EOF RETURN allowed if set ON
Bit 3 - attempt at full recovery after an error. Thus after error processing, I/O processing will attempt to continue as if the error had not occurred.
Bit 4 - this bit is set if the user does not wish to have any error comments outputted through SERCOM
Bit 5 - not used at present
Bit 6 - if this bit is set and a field-width-exceeded error occurs on output, then the output field will be filled with the overflow fill character and I/O processing will continue.

At the moment none of the other bits or bytes mean anything.

DROPIOER - general register 1 points to a 4-byte area (need not be aligned) which contains codes which tell IOH/360 what bits are to be reset (of those that might have been turned on by a call to SETIOHER). See above.

OWNCONVR - With the "O" format term, the user may use his own conversion routine for either input or output. See "O" [D6]. Previous to using such a conversion, the user must specify the entry-point address of the user's routine that will attempt such a conversion. The call:

```
LA      1,MYCONV
L       15,=V(OWNCONVR)
BASR    14,15
```

will accomplish such if 'MYCONV' is the entry point address. The routine used need save only register 14 (so it can return). If and when the return is made, register 15 should be zero if the conversion was successful, four if an EOF condition was found upon a call to the OPEN routine (see later), or eight if the conversion was unsuccessful. All other codes are illegal and will be treated as an error. When IOH/360 calls such a routine, general register 1 points to the beginning of an area which has the following structure:

```
OWNW1     DS   F
OWNW2     DS   F
OWNW3     DS   F
OWNARG    DS   F
OWNOPEN   DS   F
OWNCLOSE  DS   F
```

```
OWNCUR    DS   F
OWNLAS    DS   F
OWNPUT    DS   F
OWNFLAGS  DS   XL2
```

where:

OWNW1 - first field width (e.g. the m in E m.n.q)
OWNW2 - second field width (e.g. the n in E m.n.q)
OWNW3 - third field width (e.g. the n in E m.n.q)
OWNARG - address of conversion argument (i.e. from where or to where the conversion is to take place).
OWNOPEN - The address of the routine the user's conversion routine should call if he wishes to get a new input or output image.
OWNCLOSE - the address of the routine the user's conversion routine should call if he wishes to close out an input or output image.
OWNCUR - pointer to current byte in the input or output image
OWNLAS - pointer to the last byte in the input or output image.
OWNPUT - the address of a routine the user may call to put something into an output image.
OWNFLAGS - two bytes of flags.

byte 1: bit 0 - on if 'R' was specified
1 - on if 'L' was specified
2 - on if 'N' was specified
3 - on if 'DD' was specified
4 - on if 'D' was specified
5 - on if 'W' was specified
6 - on if 'H' was specified
7 - on if 'B' was specified

Only one of bits 3,4,5,6,7 may be on.

byte 2: bit 0 - on if standard format specified (i.e. no 0's or field widths)
1 - on if width 1 was blank
2 - on if width 2 was blank
3 - on if width 3 was blank
4 - on if width 3 was found
5 - on if any . was found in the format term
6 - on for output; off for input
7 - on if this is first time through for this format term

Note:

1) on input the user will get the same input image as previous if he does not call OWNCLOSE before calling OWNOPEN --- thus the user may rescan image (starting at column 1 by calling OWNOPEN and not previously calling OWNCLOSE for the previous image.

2) user may change any of the fields specified.

On input:

User does all of his own conversion into the address specified in OWNARG; on return OWNCUR should contain the address of the byte where the line pointer is to be placed in the input image.

On output:

User has two options a) he moves his conversion into the output image starting at the address in OWNCUR and not going past the address in OWNLAS for each image. He may use successive images --- each time a new image is obtained OWNCUR and OWNLAS are subject to change. b) the user may call OWNPUT to place his output into the output image and thus take advantage of the R, L, Z, $, Y modifiers and the default fill character. When OWNPUT is called register 1 points to a list of 3 adcons; the first adcon points to the beginning of the user's conversion output area; the second word points to the full-word number of bytes in that area; the third word points to the full-word conversion width. Parameter three must be greater than or equal to parameter two or an error will result.

## Block-Addressing Section

When using block-addresses, the I/O-conversion subroutine must have some means to know in what manner the block addresses are to be incremented. Block-addresses may either run forwards or backwards in core. Whenever a length modifier is explicitly given, the length implied by that modifier will be used in computing the next block address (i.e. a length modifier of 'D' will cause a change of 8 while an 'H' will cause a change of only 2). Thus a length modifier may be applied to even a C conversion even though it actually does not affect the conversion at all. If there is no length modifier in a C P, or X conversion, then the change in the block address will be the amount specified by the internal width field of the C P, or X conversion. Thus C5.3 will cause an increment of 3. With respect to the above, one should realize not to ever give a zero internal field to a format term whose corresponding conversion address may be part of a block. Thus C0.0 would increment the block address by zero - an intolerable situation which will be marked as an error if it occurs when a block address is being computed.

## Standard-Format Input Section

STANDARD FORMAT on numeric input means that the I/O-conversion subroutine will scan the input image until the next occurrence of a character which is not an input fill-character which its initially blank (the first such character may be called the initiator character). After finding this character, it scans until the occurrence of the next comma or input fill-character or the end of the input image (any of which may be called the terminator character). All characters between the initiator character (including it) and the terminator character (excluding it) are taken as the input field . If the scan reaches the end of the input image before finding the first character which is not an input fill-character, then then a new image is requested (i.e. a call to CLOSE followed by a call to OPEN).

On character input, the input image is scanned until the next occurrence of any non-blank character excepting a comma (the warning is given here not to use the blank or comma as a literal break character). If no such character is found in the present input image, then IOH/360 calls the OPEN routine for a new image. When such a non-blank character occurs, it is checked to see if the character is a literal break character; if it is not, then an error condition arises. If it is, then the scan continues looking for the next occurrence of a literal break character. Two successive literal break characters in the scan stand for one such character (the first of the pair is used as input). Eventually a literal break character must occur immediately after which there is no occurrence of another

literal break character. All characters between the initial literal break character and the terminal break character are input starting at the location specified by the argument address.

MTS-550-0

12-1-67

P I L

# PIL - - PITT INTERPRETIVE LANGUAGE

The following is a description of a remote terminal language in use at the University of Pittsburgh, the Pitt Interpretive Language (PIL). This language processor was installed in MTS as received from the University of Pittsburgh, with minor modifications due to different system interfaces. The following writeup is an almost exact copy of the University of Pittsburgh's writeup. The most noticable difference is that the read-prefix-character is an equal sign, since the greater-than which Pitt uses is already used in MTS. PIL is accessed in MTS by means of a $RUN *PIL command - see the description of the library file *PIL in section MTS-280 for further details.

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. PIL, unlike the compiler languages MAD, FORTRAN, and ALGOL, provides the user with much greater assistance through the use of console diagnostics, user interaction with the machine, and associated error recovery procedures. PIL differs from the compilers by providing direct man-machine interaction facilities, and from earlier conversational languages in the relaxation of restrictions imposed upon the user by the earlier languages.

A major goal of this language was that errors be recoverable. To a user, this means that it is possible to sit down at a console with a problem and work toward a solution. As he becomes aware of the need to make corrections or improvements, PIL allows him to alter his program and to continue without requiring a new start. PIL gives up machine efficiency in hope of gaining increased human efficiency. PIL is thus designed for personal use by researchers who feel their own time is valuable.

In this writeup you will find that certain words (keywords) always appear in capital letters. These words specify what is to be done and consequently have certain rules associated with them. The combination of the location of the word in a statement, its exact spelling, and its separation from the next word by a space (blank), are necessary information for the language interpreter. Therefore, these words do not have to be reserved by the interpreter. Thus,

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

---

[1]Developed by C. Shaw and at the RAND Corporation, Santa Monica, California.

[2]Developed by Bolt, Beranek, and Newman, Cambridge, Massachusetts.

is a legal statement in the language. (Note: The following convention is used in this manual. All typing by a user is preceded by a "=". All typing by PIL is not.)

In the example, the first SET is recognized as a key word because it is the first word in the statement; the first word specifies the action to be taken. The second SET is in a place where a variable name is expected to be encountered (i.e., between the word SET and an equals sign). Since the variable name SET follows the rules for variable naming, it is legal and unambiguous.

Every statement in PIL begins with a key word and terminates with a period. Some statements contain more than one key word. For example,

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

A summary of the various PIL statements appear in Appendix A.

PIL is oriented toward problem solving, with program development and debugging facilities having the 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.

## DESK CALCULATOR MODE

In the simpler mode of operation, the console may be used as a sophisticated desk calculator. This mode of operation allows the user to evaluate arithmetic expressions, determine the value of transcendental functions, and store intermediate results for later inspection.

```
=TYPE 125/5.
 125/5 = 25.0
=TYPE 1.32 + 12.8/32.
 1.32 + 12.8/32 = 1.72
=TYPE (THE SINE of 12.8)**2+(THE COSINE OF 12.8)**2.
 (THE SINE of 12.8)**2+(THE 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
```

Statements in the desk calculator mode result in an immediate response by PIL, and are referred to throughout this writeup as DIRECT MODE statements. After execution of a direct mode statement, the statement is not retained so the user must retype any expression that is needed again.

In the direct mode, errors are reported immediately. After the user has corrected them, he must retype the statement because the statement was not retained.

## VARIABLES AND CONSTANTS

Information may be stored for later use by the SET statement (and others).

```
=SET a = 27.98.
=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, the variables are a, b, and c and the values stored into them are known as constants.

### Constants

Constants may be numeric, such as 3.1415 or 7.8, or they may be alphabetic, such as "PIL/L" OR "123=". Alphabetic constants (strings) may be up to 255 characters in length; they are explained in Section 12. There are only two Boolean constants - THE TRUE and THE FALSE.

```
=SET data = 5.3.
```

In this statement, 5.3 is the constant. The variable name, data, has associated with it the numeric value of 5.3.

### Variables

Storage of variables provides a convenient method for retaining intermediate calculations and allows examination of the contents by the TYPE statement.

Variables are given names by the user according to the following rules:

1. The first character of the name must be a letter (either upper or lower case).

2. The remaining characters may be letters or numerals. Upper case letters are distinguished from lower case.

3. The total number of characters in a variable name may not exceed eight. This restriction may later be relaxed so that any number of characters may be used.

   Examples:

   A,A12, daTA, filename

   But not:

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

In addition to variable names with a single value associated with each name, it is possible to have many values associated to a single name. Single or multiple dimensional arrays (tables) provide a convenient means to accomplish this. A subscripted variable is represented by a variable name, followed by a list of subscripts, separated by commas and enclosed in parentheses. For example, DATA(1,2) could represent the second entry in the first row of a table called DATA.

The general rules for subscripts are summarized below:

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

2. An array may have up to twelve subscripts.

3. A subscript may take any integral value between -9,999,999 and +9,999,999, but only the integer part is used to reference an element.

```
=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 referenced in either form so long as the values of a(i), i, and j remain unchanged.

```
=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.

In certain cases, the entire array may be referenced by mentioning only the name.

```
=SET A(1,1) = 1.
=SET A(1,2) = 2.
=SET A(2,2) = 3.
=TYPE A.
 A(1,1) = 1.0
 A(1,2) = 2.0
 A(2,2) = 3.0
=TYPE A(2,1).
 Eh? A(2,1) = ?
```

In this example, we see that the element A(2,1) was not defined by the program, thus the reference to it in the TYPE statement generated an error report to the user.

The variable dictionary in PIL is dynamic, using space only for currently defined variables. Therefore, sparse arrays are kept in a greatly reduced space. This dictionary is kept in an alphabetical order (lower case before upper case) with subentries for subscripted variables.

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

```
=SET b = data+i.
 Eh? i = ?
```

is an error if i were 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. After the definition was given, it would be necessary to retype the statement involving i since the statement was given in the direct mode.

In addition to a value, any variable has a mode associated with it. It may be referred to as follow:

```
=SET x = THE MODE OF b.
```

| Preceded By: | Would Result In: | |
|---|---|---|
| =SET b = 34. | x = 1 | b is defined with a numeric value. |
| =SET b = 1 < 2. | x = 2 | b is defined and Boolean (TRUE or FALSE). |
| =SET b = "PIL". | x = 3 | b is defined and a character string. |
| =SET b(1) = 3. | x = 4 | The dictionary contains a variable by the same name as that appearing in the mode statement, but the two have a different number of sub- |

scripts.

```
=DELETE b.          x = 5    b is not defined.
```

Variables which are defined occupy space in the user's memory. For this reason, it is useful to be able to eliminate variable names (dictionary entries) when they are no longer needed. To accomplish this, there is the DELETE statement. DELETE may refer to variables or arrays.

```
=DELETE a.
=DELETE B,C,D(1).
```

When an entry is deleted, the space it occupied is returned to the user as free space which can be used again later.

When a user starts working, he is allocated a certain amount of space (virtual memory) in which to work. This space is best measured by how many dictionary entries it could contain.

THE TOTAL SIZE is a measure of how much space the user started with, or could have if he deleted everything. THE SIZE is a dynamic measure of the core space available at any given time.

```
=TYPE THE SIZE.
 THE SIZE = 197.0
=SET a = 1.
=SET b = 2.
=TYPE THE SIZE, THE TOTAL SIZE.
 THE SIZE = 195.0
 THE TOTAL SIZE = 197.0
```

## ALGEBRAIC EXPRESSIONS

The arithmetic operators 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 term "A times B" is written as A*B.

Operators are used in conjunction with variables, constants, and functions to form expressions. More generally, variables, constants, arithmetic functions and expressions coupled 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.

In summary, expressions are:

1. variables or constants a, 12.0

| | | |
|---|---|---|
| 2. | functions operating on an expression | SINE OF b |
| 3. | parenthesized expressions | (a+b) |
| 4. | expressions coupled by binary operators | (a+b)*(c-d) |
| 5. | expressions preceded by unary operators | -(a+b) |

The operators which may occur in an arithmetic expression are +, -, *, /, ** and | (vertical bar). Grouping marks, such as parentheses, are used to delimit the scope of operators in an expression. Parentheses are necessary to distinguish (a+b)*(c-d) from a a+b*c-d. However, expressions such as the latter are not ambiguous since there is an implied parenthesizing (which is understood to yield a+(b*c)+d for the second expression).

The precedence rules of the arithmetic operators are shown in the following list with the top of the list having highest precedence and equal precedence shown on the same line.

| | |
|---|---|
| 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.

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 |

These expressions may be used in arithmetic expressions interchangeably with the same numerical result. Redundant parentheses are ignored by PIL.

In addition to the arithmetic operators, many functions are available, such as SINE OF, COSINE OF, etc. table 1 provides a complete listing of arithmetic functions and operators available to the user.

## Boolean Expressions

Boolean expressions are also available in PIL. A SET statement will store the result of a Boolean expression in any desired variable and set the mode of this variable to Boolean. A Boolean expression may be:

1. A Boolean constant.
2. Two arithmetic or Boolean expressions coupled by a Boolean binary operator.
3. A Boolean expression preceded by a Boolean unary operator.

The difference from an arithmetic expression is that the expression results in a Boolean value (i.e., TRUE or FALSE). TABLE 2 gives a complete list of the Boolean operators available.

The following examples show the use of Boolean expressions:

| Statement | Result |
|---|---|
| =SET X = 1 < 2. | X = TRUE |
| =SET X = (10+5) < (7+8). | X = FALSE |
| =SET X = 1 = 1. | X = TRUE |
| =SET X =$NOT 1 > 2. | X = TRUE |
| =SET X = 1 = 2 $AND 1 $NE 2. | X = FALSE |

For the third example, the first equals sign encountered going from left to right in the statement is the replacement operator; the other is a relational operator.

## Interchange

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

```
=SWAP a,b.
```

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

```
=SET temp = a.
=SET a = b.
=SET b = temp.
```

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 | | Log x (base 10) | Log of x |
| Antilog of | | $10^x$ | Antilog of x |
| Ln of | | Log x (base e) | Ln of x |
| Atan of | Arc Tangent of | $Tan^{-1}$ x | Atan of x |
| Exp of | | $e^x$ | Exp of x |
| Rn of | Random number of | Random number generator | Rn of x (changes x) |
| Ip of | Integer part of | Integral part of a PIL number | Ip of 3.5 = 3.0 |

| | | | |
|---|---|---|---|
| Fp of | Fraction part of | Fractional part of a PIL 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*(10 Xp of x )=x | Dp of 35 = 3.5 |
| Min of | Minimum of | Least value | Min of (a,b,c) |
| Max of | Maximum of | Greatest value | Max of (a,b,c) |
| The Size | | Current avail. space | |
| The Total Size | | Total work space | |
| The Time | | Time in 300's of a second relative to 00:00 (midnight) | |
| The Date | | Day of year in form YYDDD where YY is 19YY and DDD is day of year. | |
| The Elapsed Time | | User's actual computer usage time (cpu time) in 300's of a second. | |

TABLE 2

BOOLEAN OPERATORS

| OPERATOR | | MEANING | EXAMPLE |
|---|---|---|---|
| Short | Long | | |
| < | $lt | is less than | a < b |
| | $le | is less than or equal | b $le c |
| = | $eq | is equal | c = d |
| | $ne | is not equal | b $ne c |
| | $ge | is greater than or equal | c $ge b |
| > | $gt | is 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 | |

## STORED PROGRAM MODE

All key words in PIL are acceptable with any combinations of upper and lower case letters: e.g., Set, set, SET are all legal so long as their spelling is correct. In variable naming, however, this is not true. DATA is not the same as the variable data. Thus, it is possible to have variables with the same spelling but different case combinations.

There are two statement modes in PIL. The first is the direct statement mode, which has already been discussed. The second is the stored program mode or indirect statement mode. Recall that direct statements are executed immediately and that the text is not retained. Indirect statements are executed under program control in a sequence defined by part and step numbers. Indirect statements comprise a stored program, while the desk calculator mode is unique to languages in which the user is in conversation with the computer. The user may go back and forth between these modes at will, using them as best facilitates solution of the problem at hand.

```
=SET data = 27.98.
```

was an example of the direct mode statement.

### Parts and Steps

The indirect statement is always associated with a sequence of instructions. To define the proper sequence, a part number, followed by a decimal point, followed by a step number is used. Each may be a four digit number. Indirect statements may be typed in any order and will be inserted in their proper places by the computer. A part is a collection of one or more steps with the same part number, and arranged in order of ascending step number. These numbers, followed by a space, must precede the statement itself.

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

is our direct mode example as it might be given in the indirect mode.

To allow comments to be typed and stored as part of the program listing, PIL has the following convention. If, after a step number and blank, the first character in a 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 VALUES.
=1.66 SET A = a+the SQRT of (b/c).
```

Any statement ending with an "*" followed immediately by a RETURN without a period is completely ignored by the interpreter. In this way it is possible to place comments as you type, but not save them in storage. This is the same mechanism used to delete an incorrectly typed line.

Step 2.0000 does not follow 1.9999 in execution since they are in different parts, although it does follow it in program listings (TYPE all parts). Since it is possible to change a program at any time, it is desirable to leave room in the numbering in order to make insertions. Additions will automatically be placed in the correct numerical sequence.

At any time the user may request the typing of any part, step, or all parts.

```
=TYPE step 1.5.
=TYPE part 3.
=TYPE all parts.
```

In typing a part or all parts, PIL will type the current program including all modifications. The typed result will be in order by parts and steps.

## Indirect Error Reporting

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

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

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

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

As long as $a \neq 0$, the variables j and y may remain undefined, for the execution of this statement does not require these values.

Some final notes on the indirect mode are:

1. Errors are reported when encountered in execution.
2. A step may be replaced by typing the corrected statement with the same step number, thereby replacing the old step.
3. A part or step may be removed by deleting it.

```
=DELETE STEP 1.5.
=DELETE part 4.
```

will eliminate step 1.5 and part 4 from the program.

## RUNNING A STORED PROGRAM

Using the statements encountered up to this point, the user is capable of writing small programs. Consider a program consisting of a series of calculations involving the SET statement and the printing of a final value via the TYPE statement.

A good example of this is the solution of one real root of a quadratic equation $ax^2 + bx + c = 0$ by:

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

The program to do such calculation may be:

```
=5.1 *Program to calculate one root of a quadratic equation.
=5.2 *Coefficients are in x(1), x(2), and x(3).
=5.3 SET a = x(1).
=5.4 SET b = x(2).
=5.5 SET c = x(3).
=5.6 SET root = (-b+SQRT OF (b**2-4*a*c))/(2*a).
```

If a part, such as PART 5 above, is entered, it may be executed, (either partially or completely) starting with its lowest step number, by the direct DO statement. The user simply types:

```
=DO part 5.
```

The program will then proceed as directed, or until an error occurs, and upon completion, PIL will type "=" and then wait for further instructions.

Once the program is started, errors may be detected by the interpreter, reported to the user, and corrected by the user. If the user desires to restart he may restart at the beginning by issuing another DO, or resume at the point of the error by typing GO. For example assume the equation is, $x^2 + 4 = 0$, of which both roots are complex.

```
ERROR AT STEP 5.6: NEGATIVE IN SQUARE ROOT ARGUMENT
=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.
=GO.
```

From the example, it can be seen that during the correction procedure the data has been changed (effectively changing the problem to $x^2 - 4$). The program may also have been changed, if desired, to determine whether the term (b**2-4*a*c) is negative. During this time, any direct statement may be used, or indirect statements added or deleted.

When an error is detected in the execution of a statement it is reported at that point. A GO issued after correction of the error will begin that statement over. This is particularly important in relation to the FOR statement as discussed in a later section.

A DO statement may be given in the direct or indirect mode. Indirect DO statements are used to execute other parts or steps within a program and regain control at the statement after the DO statement.

## Program Stops

If the user of our example wished to stop the execution of his program and check the values of a, b, and c before calculating the square root of the factors:

```
=5.55 STOP.
```

will result in the response from PIL:

```
STOP AT STEP 5.55.
```

The computer will then wait for further instructions. The user may do anything at this point that he was able to do after an error report. GO at this point would resume execution at step 5.6.

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

PIL is designed so that a user may interrupt his program at any time. This is done by causing an attention interrupt. How to do this depends on there terminal: ATTN button on 2741, BREAK on teletypes. See the user's guides for the terminals in Section MTS -150 and 170 for details.

After an interrupt, and before a GO, the user is in full control, and can use any part of PIL including a DO statement. If, however, a normal DO is executed directly, resumption conditions are destroyed and GO has no meaning. A special DO statement allows these to be preserved. In this special form, the particular part or step appears in parentheses.

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

The major resumption condition is the step to be executed when GO occurs.

## TRANSFER OF CONTROL

Once execution of a program has begun, the sequence in which steps ar executed is determined by the sorted order of step numbers within a part unless a transfer of control statement is encountered. There are tw statement types which accomplish this. One, which we already hav encountered, is the DO statement. It will be remembered that thi statement is acceptable in either the direct or indirect mode.

### DO Statement

```
=DO part 5.
```

was used to initiate the computation of one root of a quadratic equation ir an earlier example. In this example, it was assumed that the coefficients of the quadratic equation were in x(1), x(2), and x(3). Let us build ontc this example by assuming that x(1), x(2), and x(3) were calculated in some manner in part 4.

```
=4.1 SET x(1) = SIN OF y(1).
=4.2 SET x(2) = SIN OF y(2).
=4.3 SET x(3) = (SIN OF y(3)/COS OF y(3)).
=4.4 DO part 5.
=4.5 SET result = root/2.
=4.6 TYPE result.
=DO PART 4.
```

In this example, part 4 is now used to initiate part 5 (at step 4.4). The sequence of execution would be; 4.1, 4.2, 4.3, 4.4, 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 4.5, 4.6. Here, step 4.4 is used to transfer control to part 5. After completely doing all 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:

```
=5.6 DONE.
```

will inform the interpreter that execution of this part is complete.

### TO Statement

The second type of statement is the TO statement.

```
=4.4 TO part 5.
```

If step 4.4 were replaced with the above statement, after execution of steps 4.1, 4.2, and 4.3, step 4.4 would transfer control 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, but rather terminates after the execution of step 5.6. The TO statement may also reference a step number:

```
=4.4 TO step 5.4.
```

will send control to step 5.4 and thereby bypass execution of the first three steps.

The TO statement is only meaningful in the indirect mode and may only be used as an indirect mode statement. It may not be the object of a FOR.

## IF Statement

Conditional transfer of control is accomplished with the IF statement. The word IF followed by a space starts the conditional statement. This is followed by the conditional expression (any Boolean expression) and then a comma (,). After the comma any other statement may occur. Returning to our root of a quadratic equation example, it will be remembered that an error condition existed whenever the square root argument (discriminant) was negative. To determine if the discriminant is negative, it is desirable to have the program check for this error.

```
=5.55  IF (b**2-4*a*c) < 0, TO step 5.9.
=5.9   TYPE "Complex Root.".
=5.8   DONE.
=TYPE part 5.

 5.1   Program to calculate one root of a quadratic equation
 5.2   *Coefficients are in x(1), x(2), and x(3).
 5.3   SET a = x(1).
 5.4   SET b = x(2).
 5.5   SET c = x(3).
 5.55  IF (b**2-4*a*c) < 0, TO step 5.9.
 5.6   SET root = (-b+SQRT OF (b**2-4*a*c))/(2*a).
 5.7   TYPE root.
 5.8   DONE.
 5.9   TYPE "Complex Root.".
```

With the newly edited part 5 before us, it may be seen that the program now checks for a negative discriminant; if it is negative, control passes to step 5.9 and a message is printed.

Consider the following statement:

```
=1.5 IF j < 5, SET j=j+1.
```

j < 5 is the Boolean expression and its value is either true or false. If

true, the statement SET j=j+1 is executed and control is passed to the nex statement. If false, the next statement is executed immediately.

```
=1.6 IF j < 5, IF X+Y+Z > 57, SET j=j+1.
```

j is compared to 5. If j's value is less than 5, then X+Y+Z is compared t 57, etc. If, and only if, both conditional expressions are true will SE j=j+1 be executed.

The IF statement has a provision for executing a statement when 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 ad clarity. The semi-colon(;) separating the two object statements i necessary, as it serves to distinguish this from the more simple form o the IF statement. Thus, step 3.7 may be retyped as:

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

As has been mentioned earlier, the IF statement requires a Boolea expression. A Boolean variable may be used as the Boolean expression fo the If statement:

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

## SIMPLE CONSOLE I/O

Console Input/Output (I/O) is achieved by the statements TYPE and DEMAN followed by one or more items (a list). I/O statements described here are designed for use at a remote terminal. Other statements are described later

### Output

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

```
=SET a = 3.1.
=SET b = 1 < 2.
```

```
=SET c = "PIL/L".
=TYPE a,b,c.
 a = 3.1
 b = true
 c = "PIL/L"
```

The seven most significant figures of a number are kept internal to PIL. Because a scaling factor is used, it is possible to represent zero, and both positive and negative numbers between $9.999999*10^{48}$ and $1.0*10^{-51}$ in magnitude.

A special format is used for typing numbers above 9,999,999 or below $10^{-7}$. The form of this number is SD.DDDDDDESDD, where D indicates a digits, S represents a sign (+ or -), and E represents 10**.

```
=TYPE 10**10, 1E10.

 10**10 = 1.000000E+10.

 1E10 = 1.000000E+10.
```

The TYPE statement may involve algebraic expressions.

```
=SET a = 1.

=SET b = 10.

=SET c = 2.

=SET d = 3.

=SET e = 4.

=TYPE (a+(c*d/e))/b.
 (a+(c*d/e))/b = 0.25
```

The TYPE statement may also include alphabetic messages, contained with quotation marks (").

```
=SET a = SQRT OF 25.
=TYPE "SQUARE ROOT OF", a
 SQUARE ROOT OF
 a = 5.0
```

In a TYPE statement, an entire array may be requested by a single reference to its name.

```
=SET a(1,1) = 1.0
=SET a(15,7) = 2.0
=SET a (7,24) = 3000.
=TYPE a.
 a(1,1) = 1.0
```

```
a(15,7) = 2.0
a(7,24) = 3000.0
```

Only elements of the array that are defined by the user will be typed

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

1. To get a copy of a part, sorted in order, and as presently entered,
   =TYPE part 5.

2. To get a copy of the entire program, sorted in order and a presently entered,
   =TYPE all parts.

3. To list all defined variables and their current values.
   = TYPE all values.

4. To list the entire program and variables in storage.
   = TYPE all stuff.

Other specialized and/or modified TYPE statements are available and ar described in other sections related to I/O.

Since the typewriter is a relatively slow device, the user will find i practical to use these features sparingly. Information may be obtaine selectively by:

```
=TYPE a,b,c.
=TYPE step 1.3.
=TYPE a+b/c.
 etc.
```

Input

The DEMAND statement requests the user to provide values for a list of variables. The list of variables requested is the same as that followin the word DEMAND.

```
=2.87 DEMAND a,b,c.
=DO step 2.87.
  a=
= 27.98
  b=
= 18.46
  c=
= 57.28
```

The user's response to a DEMAND sequence may be in terms of anything the interpreter knows about. The response may be in terms of:

1. constants (e.g., 2.79828)

2. arithmetic expressions (e.g., a+b/c) in terms of previously defined variables.

3. functions (e.g., The SIZE, SQRT OF a.)

4. any combination of the above.

```
=1.1 DEMAND a,b,c.
=1.2 TYPE a,b,c.
=DO PART 1.
  a=
= 4.0
  b=
= a+SQRT of a
  c=
= a*b
```

On a 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) =
```

DEMAND, unlike TYPE, cannot use the variable name to imply an entire array. If x had been an array, DEMAND x would cause and error report to be printed.

```
=1.1 SET x(1,1) = 1.
=1.2 SET x(2,4) = 0.
=1.3 DEMAND x.
=DO part 1.
 x=
=5
 ERROR AT STEP 1.3: x UNMATCHED SUBSCRIPTS.
```

Notice that the error was spotted when the first value was supplied by the user.

More complex I/O (Input/Output) is described later.

PROGRAM CHANGES

Whenever PIL is waiting for a user to type a command, a direct or indirect statement may be entered. If a new step is typed using an old step number the step will reflect only the new entry and the old step will be lost. At the time PIL is waiting for a line, the user may alter a variable value, add or replace any indirect statement and execute any direct statement.

Deletion

Storage is used by defining variables, entering parts or steps, and by creating FORMS (explained later). Quite often, it is useful to be able to remove them from the user's core (delete them).

VARIABLE DELETION

```
=DELETE data.
=1.3 DELETE data.
```

will delete the variable "data" from the user dictionary and thereby increase the capacity of his storage. After execution, data is undefined, since it no longer has an associated value, and any reference to it as long as it is undefined will generate an error report.

```
=DELETE a,b,c,d(1,2), t(3).
```

causes the variables named in the list to be deleted from the user's storage.

A variable may be deleted by the user at any time, with the exception of currently running iteration indices. This exception is fully explained later in the section in iteration.

As with the TYPE statement, and entire array (table) may be deleted by simply mentioning its name. All values may be deleted by:

```
=DELETE all values.
```

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

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 named part or step is deleted, returning the storage it used for later reuse. All defined parts may be deleted by:

```
=DELETE all parts.
```

thus effectively destroying the program but leaving and defined variables and forms in storage.

FORM DELETION

Form deletion will delete a particular FORM or all FORMS by:

```
=DELETE FORM 10.
=DELETE ALL FORMS.
```

Storage Clean-up

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

```
=CANCEL.
```

is used. This statement is direct mode only, whereas the DELETE statement may be given in the direct or indirect mode. After a CANCEL, the interpreter will respond with:

```
READY:
```

and the user may begin a new problem.

ITERATION 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, plus facilities to handle easily 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, including another FOR, but excluding the TO statement. Any expression may be specified in the list.

## Implied Loops

Another form is the simple TO loop. It enables the programmer to provide a series of values without listing them explicitly. The general form is:

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

By combining the two types given above, it is possible to type:

```
=FOR i = 1 TO 5,7 TO 11 by 2: DEMAND B(I).
```

with i taking on the values 1,2,3,4,5,7,9,11.

Notice from this example that any form of the FOR statement may be repeated in the FOR list, with each separate entity, or list, separated by commas. Thus, one could type:

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

In the TO loop, if the initial value is not greater than the final value, the object statement is executed. Upon return, the increment (or 1, if none is specified) is added to the FOR variable, and then it 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 next item in the list is used. The process is repeated until the last list element is satisfied (the one preceding the colon).

## Explicit Loops

Two additional FOR list forms exist. The first is:

```
=FOR i = a BY b UNTIL r > n: DELETE x(i)
```

This form will repeat for successive values of i until the Boolean expression, following the word 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 word WHILE is true. In each of these forms, if no increment

is specified, the initial expression is repeated. (CAUTION: If the Boolean expression has a constant value, an infinite loop may result).

As has been mentioned earlier, a user may delete a variable at any time, with the exception of a FOR index (iteration variable).

```
=FOR i = 1 to 10: DELETE i.
 Eh? ATTEMPT TO DELETE FOR INDEX
```

FOR indices may be deleted after the execution of the object statement is complete (i.e., the Boolean condition, implied or explicit, becomes true).

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.

## Restart

The FOR statement is legal in both direct and indirect mode. Should an error occur in the object statement of an indirect FOR statement, correction of the error and the word GO will cause PIL to begin that statement over. This may cause a rather serious error in further computation.

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.
 ERROR AT STEP 1.3: a(11) = ?
=SET a(11) = 0.
=GO.
 AVG = 11.0
```

For this example the variable AVG should equal 5.5. However, since the accumulated sum was not reset to zero after correcting the undefined variable error, the accumulated sum became twice the value it should have acquired. Currently, it is the user's responsibility to be aware of such circumstances and to correct for them when necessary.

For Control

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

```
=NEXT i.
=LAST i.
=END i.
```

The purpose of NEXT i is to get the next value of i in the FOR loop whenever the statement is encountered, regardless of FOR nestings. This allows intermediate FOR's to be automatically terminated and control to be returned to the FOR statement with i as its index.

```
=1.8 IF mode of a(i) = 5, NEXT i.
```

effectively gets the next index on i if the current a(i) is undefined.

The statement LAST i is similar in that looping on i terminates intermediate FOR's and control is returned to the statement following the FOR on i (or the preceding FOR if it is nested).

```
=1.9 FOR i = 1 TO n: IF a(i) = 10, LAST i.
```

will search the array a for the value 10, and will exit with either i being equal to n+1, if one is not found, or with i being the subscript of the entry with the value 10.

END i terminates the loop on i as does LAST i, but control is passed to the statement following END i.

## CHARACTER STRINGS

One of the more powerful features in PIL is the handling of character strings. A character is any sequence of characters (including a null sequence). A character string constant is any string enclosed in double quotation marks.

```
=SET a = "A string of characters".
```

Any legitimate input characters may be included in a string, except for ("), which may act only as a delimiter in a character string constant and is not a part of the string. An upper limit of 255 characters in a single string is imposed.

### String Comparison

Any string may be compared with any other string, using any of the defined relational Boolean operators. Strings are compared left to right. If strings are of unequal length, the shorter string is treated as though it were padded at the right end with blanks for comparison. The following collating sequence is the bases for comparison of strings:

(blank) < punctuation < a...z < A...Z < 0...9

```
=IF "x" < "y", TYPE "yes".
 yes
=IF "abcd" = "abcd  ", TYPE "blanks ignored".
 blanks ignored
```

It is assumed that the blank is the lowest character that a string will con

It is assumed that the blank is the lowest character that a string will contain. Table 3 presents a complete list of all string related operators and functions.

### String Functions

To determine the length of a string, THE LENGTH 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
```

Since many applications using strings are concerned with content, and not specific forms, 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 in case shift operations.

All of these facilities allow the user to create, copy, and compare strings of fixed length. In many applications, it is also useful to break down and combine strings. PIL has three functions related to these applications.

## String Operations

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 of string mode. The length of the concatenation result is the sum of the lengths of the two operands. 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
```

String subtraction is not well defined, and is therefore not allowed. It is useful, however, to either remove or examine some portion of a long string. There are two functions that allow this kind of manipulation. They are:

```
THE FIRST n CHARACTERS OF STRING 1.
THE LAST  m CHARACTERS OF STRING 2.
```

Here n and m may be replaced by any arithmetic expression. Each function is self-explanatory. The number specified must be non-negative, and not greater than the length of the string to be operated on. Combinations of THE FIRST and THE LAST allow examination at any point within a string.

```
=SET x = THE FIRST 1 CHARACTER OF THE LAST 2
         CHARACTERS OF "abcd".
=TYPE x.
 x = "c".
```

(i.e., x is equal to the third character of the string.)

Consider the next example as if it were typed on one line.

```
=FOR i = 1 to n: FOR a(i) = THE FIRST 1 CHARACTER OF
 string: SET string = THE LAST (LENGTH OF string-1)
 CHARACTERS OF string.
```

This would put n successive characters of string into a(1),...,a(n), and leave any remaining characters in the variable string.

There are some additional string functions unique to PIL. The most unusual is THE VALUE function. It is defined as follows: if the mode of the operand is string, this string is evaluated as a PIL expression. If the mode is not string, the result is the same as the operand. One use for this function is converting a string containing numeric digits to internal notation.

Examples:

```
=SET a =3.
=SET b = 5.
=SET 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 function allows conversion in the other direction. If any operand is numeric, the result will be a string of digits identical to the way the number would look if typed out with a length of 14. If the operand is string, the BCD VALUE is identical to the operand. If the operand is Boolean, the BCD VALUE will be either "TRUE" or "FALSE".

```
=SET a = 3.
=TYPE THE BCD VALUE OF (a*a).
 THE BCD VALUE OF (a*a) = "9.0  "
```

The format of the result of THE BCD VALUE OF is always the same as that generated by TYPE.

Two additional functions, THE BCD TIME and the BCD DATE give strings with the time or data in readable form.

```
=TYPE THE BCD TIME, THE BCD DATE.
 THE BCD TIME = "14:29.06      "
 THE BCD DATE = "02-17-66      "
```

TABLE 3

STRING OPERATORS AND FUNCTIONS

| OPERATOR | | MEANING | EXAMPLES |
|---|---|---|---|
| Short | Long | | |
| + | | Concatenation | "a" + "b" |
| " | | String Delimiter | "abc" |
| L of | Length of | Length of a character string | L of "ab" = 2.0 |
| Upper of | Upper case of | Force all to Upper case | Upper case of |
| Lower of | Lower case of | Force all to Lower case | Lower case of "AB" |
| n $fc a | The first n characters of a | Deconcatenation | 2 $fc a<br><br>The first 2 characters of a |
| n $lc b | The last n characters of a | Deconcatenation | o $1c b |
| 1 $fc a | The first character of b | | |
| 1 $lc b | The last character of b | | |
| | The BCD Time | Time of Day in readable form | |
| | The BCD Date | Day of year in readable form | |
| | The value of | Evaluate contents as a PIL expression | |
| | The BCD value of | Convert radix of all operands to string | |

EXTENDED CONSOLE I/O

There is a method by which the user may control the format of an output or input line, allowing specification of any number of items on a single line, up to the physical limitation of the device (80 columns on a card, 120 characters on a printer). The FORM statement is used to declare the output or input specifications. The declaration is as follows:

```
=FORM 1.
=X = __.___Y = ___._
```

The first input line (not ended with an *) following the FORM statement specifies the form which will control the typeout of data. There are several types of allowable specifications within a FORM. Note that a terminal period is not required for a FORM.

A. Numeric Information

A standard numeric field is represented by a series of underline characters and an optional decimal point. Each underline indicates a possible digit position, limited by the number of allowable significant digits in a PIL number. At least one high order position should be specified in order to accommodate a possible minus sign. Thus, the number 1.2376 typed in the following form:

```
__.___
```

would be typed as follow:

```
1.237
```

Notice that additional high order positions are left blank, while the number is truncated (not rounded) after the number of allowable digits to the right of the decimal point.

Scientific notation may be requested in a special FORM specification. The form:

```
.........
```

will type the number-.123456 as follows:

```
-1.234E - 01
```

At least seven (7) consecutive periods are required for one significant digit of output (using scientific notation), each additional period allowing one more digit of significance, up to the limitations of PIL numbers. Any number can by typed out successfully in this form.

Numbers which are too large for standard numeric forms will generate a diagnostic message. Such numbers may be either typed in scientific notation, or typed in the following special form.

```
__.___!!!!
```

The form is a standard numeric form followed by four exclamation points. The form will type out standard numeric when the number is within range of the specification, but will switch to scaled notation when the number is too large or too small. As with scientific notation, all numbers can be typed in this form. Consider the following form:

```
__.___!!!!___.___!!!!
```

It will type the numbers 3.14159 and 13276.5 as follows:

```
3.141   132.765E+02
```

B. Alphabetic Information

The character # indicates one position of alphabetic information. Operands in double quotes, string variables, or Boolean variables will type in this form.

```
=FORM 10.
=## __ ####
=TYPE IN FORM 10, "x", 10, "string".
 x 10 stri
```

C. Other Characters

The character "|" (vertical bar) is used as a field separator, i.e., it is used as a delimiter, but does not take up any space in the output line.

All other characters not recognized as field specifications are copied one-for-one to the output device, which allows labeling, etc.

Note: The difference between the forms in Part B and C is that in B the characters are provided by the list, while in Part C the characters are supplied in the FORM.

## Form Statement

The FORM statement is used to enter a form. The operand is an integer from one to four digits long. The line following the FORM statement is the actual FORM definition. FORM definitions are given only in the direct mode.

Type In Form n, List

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 arithmetic expression. If the form specified is undefined, a diagnostic message is given.

```
      Given:

      x = 3.14

      y = 70.3

      i = 10

      j = 510
 Form 1.
 =X__.___ Y__.___ I___ J___
```

The statement:

```
 =TYPE IN FORM 1, x, y, i, j.
```

will type the following line:

```
 X 3.140 Y70.300  I 10  J510
```

Should the list contain more items than the form allows, the form will be rescanned from the beginning until all items in the list have been typed. Thus, an n x n matrix, typed in a form with n positions, would be typed out in row form (one row per line).

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

TYPE FORM n

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

Example:

```
 =FORM 17.
 =AGE HEIGHT WEIGHT
```

```
=TYPE FORM 17
 AGE HEIGHT WEIGHT
```

TYPE ALL FORMS

This statement will type out the form number, followed by the form, in the same two line format as the form was entered.

Form Deletion

To DELETE a specific form:

```
=DELETE FORM 10.
```

or to DELETE all defined FORMS:

```
=DELETE ALL FORMS.
```

User Directed Input

The normal DEMAND sequence is somewhat inconvenient for large amounts of data. The DEMAND IN FREE FORM statement speed up console input and allows card reader input. The form is:

```
=DEMAND IN FREE FORM, a,b,c.
```

The list is any standard DEMAND list. The statement has three differences from the normal DEMAND statement.

1. The names of the input parameter are not typed out for each input item.
2. More than one item may be requested in a single input line.
3. Only numeric information and character string constants are acceptable.

This provides a convenient method for reading in cards with more than one value on them. Each input item must be separated by a valid delimiter, i.e., one or more blanks, a comma, or both of these. Input is demanded until the list is completed unless otherwise terminated. Items are processed from a line until either the end or the character "/" is encountered, at which time another input line is demanded if necessary. An "*" which is not the last non-blank character in the input line terminates the demand, regardless of position in the list. (Backspaces, etc, are allowed from the typewriter.) Since identifiers are not typed out, it is recommended that the user precede a DEMAND IN FREE FORM statement with a

TYPE statement, warning him that the equal sign coming up means "start typing in data."

A DEMAND or a DEMAND IN FREE FORM always requires a new input line at the beginning of the list. That is, it does not use anything that may be unused from the last input request.

Another alternate input statement is DEMAND IN FORM n, list. This statement is analogous to TYPE IN FORM, and most lines created by TYPE IN FORM can be re-read by DEMAND IN FORM. However, it must be noted that:

1. Numeric FORM fields merely indicate that numeric input is expected and no alignment of input to decimal points is necessary, and no scaling is performed.

2. The character "(double quote) is not an acceptable input character in an alphabetic form field.

3. The FORM fields driving the input line must be completely satisfied. It is an error if a DEMAND IN FORM statement with four input parameters, coupled with a FORM with four fields, receives only three input items on the line.

## Extended I/O List Features

There is a way by which a FOR can operate within both DEMAND and TYPE statements in the standard I/O lists. The form is:

```
=1.8 TYPE (for i = 1 to 5: a(i), b(i)).

=1.9 DEMAND (for i = i to 5: a(i)).
```

This extension is most useful in conjunction with user directed Input and Output, as it allows specification of several items in an array without listing them individually. The standard rules for FOR apply, including nesting (see Section 11).

Consider the next 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(i,1) = 1
 a(1,2) = 2
    .
    .
    .
 a(5,4) = 20
 a(5,5) = 25
```

results in all elements from a(1,1),...,a(5,5) being typed. The parentheses must be evenly matched, and those around the FOR are required.

## Literal Forms

TYPE IN FORM or DEMAND IN FORM allow the use of a literal constant (e.g., "__.___!!!!") or the use of an alphabetic variable as the form specification.

Thus:

```
=1.1 DEMAND IN FORM"___ ___ ___", x(1), x(2), x(3).

=1.2 SET a = "__.___ __.___ __.___".

=1.3 TYPE IN FORM a, x(1), x(2), x(3).
```

are legal statements to the interpreter and would be executed in the same manner as though a form number were referenced. Form numbers may be determined by an arithmetic valued expression. These features provide a means to construct forms via program control and calculate references to directly defined forms.

# PROGRAM MANAGEMENT

The user may desire to have more flexible control over certain system functions, such as pagination of output, program saving and loading, and storage acquisition. PIL has several statements related to these functions, but not specifically related to program logic.

## Pagination

If a user specifies nothing with regard to pagination, he will use every line of a 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:

MTS-550-0

12-1-67

```
=PAGES YES.

 Set paper to 2nd line from top, type carriage return.
=
```

---

```
Page   1  USERID
03-07-67
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).

After this, PIL will keep track of pages, supplying a new one when necessary. It will also supply 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 console.

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 double spacing gives a page with same physical size a normal single spaced page.

To space a single line, the statement:

```
=LINE.
```

is used, either directly or indirectly.

```
=1.7 FOR i = 1 to 5: LINE.
```

will space 5 lines.

If the user moves paper himself, the computer will not detect it, and the pagination will not be correct.

## Storage Acquisition

The message:

Eh? I NEED MORE SPACE.

is used to tell the user that his program needs more storage. The user may increase his storage capacity by the ACQUIRE statement.

=ACQUIRE 1 block.

will add enough additonal storage to be to store 512 variables (This is 2 pages of storage). The message:

Eh? STORAGE NOT AVAILABLE.

indicates that the user has exhausted all the storage available to him. His program must now be able to operate in this amount of storage.

## PILMANSHIP

For simple computing tasks, PIL typically generates answers as fast as the user at the console can assimilate them. As more complex tasks are programmed, a point may be reach where the user regards the performance of PIL as less than ideal:

| | |
|---|---|
| psychologically | response time is too slow, |
| economically | too much computer time is being used, or |
| sociologically | response time of users at other consoles is being degraded |

It should be recognized that PIL is not the appropriate vehicle for large "production" jobs. It may be of value, however, to code such a problem initially in PIL. The debugging facilities of PIL can be used to check out the logic of the procedures. Then the code should be transcribed into a compiler language (e.g., MAD, ALGOL, FORTRAN) for more rapid execution.

This section is concerned with problems of lesser magnitude, where it may pay to "fight before switching"--try to utilize some insight into the internal structure of PIL to write more efficient code.

In order to make PIL easy to use and debug from a console, it was organized quite differently from the common compilers. Some of these differences will now be discussed.

1. Running a program written in a compiler language takes place in two phases. First, the program is compiled, translated into machine language-- the numerical codes which control the computer's hardware. Then the machine language is executed. With an interpretive language, such as PIL, however, each statement is decoded as it is executed.

   ```
   =1.5 FOR i = 1 to 100: do part 4.

   =4.1 SET s = s+n(i).
   ```

   Statement 4.1 would be decoded 100 times (provided n(1)...n(100) were defined). (This feature of PIL permits modifying programs in the midst of execution.)

   It is therefore desirable to write statements compactly, to reduce the number of characters that must be scanned in the decoding process. Use only the required blanks in statements. Use short names for variables. Combine two or more statements into one where possible, to save having to scan some statement numbers.

   Example 1:
   ```
   =1.50  SET a = y(j)*z(j).
   =1.51  SET s = s+a.
           becomes
   =1.5   SET s = s+y(j)*z(j).
   ```

   Example 2:
   ```
   =2.5   IF i < j, to step 2.9.
   =2.51  SET k = 3.
           becomes
   =2.5   IF i < j, to step 2.9; SET k = 3.
   ```

   Example 3:
   ```
   =3.1   SET i = 2.
   =3.11  SET j = 3.
   =3.12  SET k =0.
   =3.13  SET m = 0.
           becomes
   =3.1   FOR i = 2: FOR j = 3:
          FOR k = 0: SET m = 0.
   ```

2. The common compilers take advantage of the <u>random access</u> feature of current computer: any cell in the main memory can be accessed as quickly as any other. This means that X can be accessed as quickly as Y, and Z(9) can be obtained as quickly as Z(99). With two or more subscripts there can be slight variations. Also, the more subscripts, the longer it takes.

   Since in PIL, variables can be added or deleted at any time, a different internal addressing system had to be used. The variables are linked in alphabetic order. Thus, to access the variable e, it would be necessary to search through all the variable names beginning with a,b,c and d. Accordingly, it is suggested that the most frequently used variables be given names beginning with the first letters of the alphabet and in lower case. Also, deleting a

variable which is no longer needed removes a "way station" on the search. (Alternative searching schemes are being investigated.)

The defined members of singly-subscripted array are linked together in order of increasing subscripts. For example, if X(1), X(2), X(3), X(5), and X(8) are defined, and X(8) is to be accessed, first the variable names are searched until X if found. Then the search goes through X(1), X(2), X(3), and X(5) until X(8) is reached. The value of deleting entries which are no longer needed is now as apparent as the penalty incurred for developing long subscripted arrays. For example, if one is evaluating an iterative expression of the form

$$x_{i+1} = g(x_i)$$

efficiently, as soon as x(i) is no longer needed, it should be deleted.

When a multiply-subscripted element is to be accessed, first the variable name is located. Then a list of pointers is scanned until the first subscript is matched. Then the second subscript is matched, and so on. It is suggested that for n sufficiently large, an array X(1)...X(n*n) with all elements defined should be converted to an array XX(1,1)...XX(n,n) for more rapid access.

Some of the coding tricks recommended to a compiler language user would backfire in PIL. For example, given a long series of positive integers i which are know to lie between 1 and 30, it is desired to have the cube of each i. In FORTRAN, it would pay to precompute an array of cubes:

```
      DO 10 I = 1,30
10    ICUBE(I) = I * I* I
```

Then the cube of I is accessed by ICUBE(I). In PIL, on the other hand, it would be better to write:

```
=SET ICUBE = I**3.
```

Note also that using a numerical constant in the statement saves a search through the variable list.

3. The terminal is the slowest piece of apparatus in the system. Giving it less work to do will speed matters up. For example, DEMAND IN FREE FORM, or DEMAND IN FORM, is a faster and more convenient method for input of several pieces of data than DEMAND. Iterated verbosity in output messages is also to be avoided. Putting several items on a line by using forms is recommended.

In conclusion, these observations and suggestions are based on the hardware and software currently in use. As improvements are made to either, it will become necessary to reevaluate these ideas.

# APPENDIX A: SUMMARY OF PIL STATEMENTS

| Direct or Indirect | Direct Only | Indirect Only |
|---|---|---|
| SET | GO | DONE |
| IF | CANCEL | TO |
| DO | PAGES | DEMAND |
| FOR | FORM | STOP |
| TYPE | | |
| DELETE | | |
| PAGE | | |
| LINE | | |
| SWAP | | |
| NEXT | | |
| LAST | | |
| END | | |
| ACQUIRE | | |

## APPENDIX B: PRECISION OF ARITHMETIC

All numbers internal to PIL are carried in a "software" floating point with no special form for integers. Numbers which can be represented are of the form:

$$+\text{frac} \times 10^{\text{exp}} \quad \text{or} \quad -\text{frac} \times 10^{\text{exp}}$$

Where frac is seven-decimal-place number between .1000000 and .9999999, and exp is an integer between -50 and +49. Thus, $10^{-51}$ is the smallest positive number that can be represented; the largest is $.9999999 \times 10^{49}$. If PIL generates a number not in the permissible range, an error report is given.

```
Eh? NUMBER EXCEEDS PERMISSIBLE RANGE
```

After each PIL arithmetic operation, the result is rounded. If the eighth decimal place of the result is 5 or greater, the magnitude of the seventh decimal place is increased by 1 and any carry is propagated.

Since rounding takes place after each addition, subtraction, multiplication, and division, an arithmetic expression may contain the cumulative effects of several rounding operations.

These next examples should illustrate these effects.

```
=SET a = 1000000.
=SET b = 0.5.
=SET c = 0.4.
=TYPE a+b+b, b+b+a, a+c+c.
 a+b+b = 1000002
 b+b+a = 1000001
 a+c+c = 1000000
=TYPE a+2*c, c+c+a, a+b-a, a+c -a.
 a+2*c = 1000001
 c+c+a = 1000001
 a+b-a = 1.0
 a+c-a = 0
```

Recall that the statements are processed left to right, and note that the result is dependent upon the order in which operations are performed, hence addition and multiplication are strictly commutative.

Exponentiation is performed in one of two ways. If the exponent is a positive integer from 1 to 16, exponentiation is achieved through repeated multiplication. Otherwise, exponentiation is performed in accordance with the following identity:

$$x^y = e^{(y \ln x)}$$

where e is the Napierian base (2.7182818...) and ln denotes the Napierian base logarithm. Since the functions EXP OF a and the LN OF a are used; some error may be introduced:

```
=TYPE 10**20.
 10**20 = 9.999999E+19
```

The transcendental functions, such as sine and cosine, are evaluated making use of the "hardware" floating point features and the approximations are carried with the equivalent of nine decimal places for intermediate calculations and the result is rounded to the seventh place.

MTS-560-0

12-1-67

# S N O B O L 4

## SNOBOL4

The SNOBOL4 language is quite new and still under development. As a result, in an effort to produce adequate documentation in a hurry, all current documentation, including the following writeup, assumes a knowledge of SNOBOL3. In addition to the reference listed in the following writeup, a writeup on SNOBOL3 exists in Volume II of the current UMES manual. Work is underway to produce an introductory section to replace the first section of this writeup.

The following writeup is reproduced essentially as received from the authors. Information on usage of the SNOBOL4 language in MTS may be found in the writeup on *SNOBOL4 in section MTS-280 of this manual. Note that since input and output is done via the FORTRAN IV I/O routines, details on I/O might be found in the Fortran Users Guide, section MTS-520.

MTS-560-0

12-1-67

# Preliminary Report on the SNOBOL4 Programming Language II

R. E. Griswold, J. F. Poage, and I. P. Polonsky

Bell Telephone Laboratories, Inc.
Holmdel, New Jersey

November 22, 1967

S4D4

MTS-560-0

12-1-67

## 1. INTRODUCTION

This memorandum is a revision and elaboration of a previous report on SNOBOL4 [1]. A number of new language features are described, and some material is covered in more detail. A knowledge of SNOBOL3 is assumed.

The SNOBOL4 language is still under development. While the basic structure of the language will remain as described in this paper, further additions and changes may be expected from time to time.

While the specification of the language is as independent of a particular implementation as possible, certain dependencies are inevitable. The particular characters used in the syntax, and input-output specifications are examples. In such areas, this paper describes the implementation of SNOBOL4 for the IBM System/360 operating under OS.

## 2. DIFFERENCES BETWEEN SNOBOL3 AND SNOBOL4

The basic structure of SNOBOL4 is much the same as that of SNOBOL3 [2] The pattern matching facility has been revised and extended, and several ne features have been added. A number of minor changes have been made both i the syntax and in names and functions. These minor changes are described i the following two sections.

### 2.1 Changes in Syntax

The principal changes are:

1. Identifiers (string names or function names) must begin with a letter, and may not contain colons.

2. A colon rather than a slash separates the rule portion of a statement from the goto. Gotos may be separated from the colon by blanks, or the colon may stand alone. No colon is required unless there is a goto.

3. Complicated expressions do not require parentheses. Arithmetic operators have the usual precedences and associate in the usual way. Thus

```
X   =   N / M * Q - P ** A ** B
```

is equivalent to

```
X   =   (N / (M * Q)) - (P ** (A ** B))
```

Other binary operators are described elsewhere in this paper. A table of operator precedence is given in Appendix A.

4. Several unary operators may be written consecutively without parentheses. Thus

```
X   =   $$$N
```

is acceptable. The unary arithmetic operators + and - are included in the language. Other unary operators are described elsewhere in this paper.

Warning: Binary operators must be surrounded by blanks to distinguish them from unary operators. For example

```
M   =   N - P
```

is a difference, while

```
M   =   N -P
```

is a concatenation.

5. Either double or single quotation marks may be used for literals. Either

```
EXP   =   'A-(B+C)'
```

or

```
EXP   =   "A-(B+C)"
```

may be written. Quotation marks must be used in like pairs. Single quotation marks may occur in literals surrounded by double quotation marks, and conversely. Thus

```
QUOTE   =   "'"
```

assigns a single quotation mark as the value of QUOTE.

6. Integers are literals and need not be enclosed in quotation marks. For example

```
NEXTN   =   N + 1
SQUARE   =   M ** 2
RESULT   =   PROD * -3
```

are acceptable. The enclosing quotation marks may be used if desired.

7. Many expressions which are illegal in SNOBOL3 are syntactically correct in SNOBOL4. For example

```
LSON(N1)   =   N2
```

is syntactically correct in SNOBOL4. Depending on the definition of LSON, this expression may or may not be semantically correct when executed.

8. Pattern matching is significantly changed in SNOBOL4. The string variables of SNOBOL3 are no longer used. In fact

```
TEXT   *WORD*
```

is syntactically incorrect in SNOBOL4. See Section 3 which describes pattern matching.

9. The semicolon may be used to terminate statements in SNOBOL4. For example

```
LOOP   TEXT   ','   =   :S(LOOP);        OUTPUT   =   TEXT
```

is a line consisting of two statements. As in SNOBOL3, the end of a card

terminates a statement unless the next card begins with the continuatic mark period.

## 2.2 Changes in Names and Functions

The principal changes are:

1. INPUT, OUTPUT and PUNCH replace SYSPIT, SYSPOT and SYSPPT as th names associated with input, output, and punch respectively. The I/ association functions READ and PRINT have been somewhat modified. Se Section 11.

2. QUOTE does not have a preassigned value. See Section 2.1.

3. LT, LE, EQ, NE, GE, GT and INTEGER replace the numerical predicate .LT, .LE, .EQ, .NE, .GE, .GT, and .NUM of SNOBOL3.

4. IDENT and DIFFER replace the comparison predicates EQUALS and UNEQ of SNOBOL3.

5. The format of the function DEFINE is slightly modified. Loca arguments are listed after the function prototype, rather than in a thir argument. The entry label may be omitted (i.e. null) in which case th label is taken to be the same as the name of the function. For example

```
DEFINE('F(X,Y)N,M','FENTR')
```

defines a function F with formal arguments X and Y and two local variables I and M. The entry label is FENTR. On the other hand

```
DEFINE('CPY(Z)')
```

defines a function CPY with formal argument Z and entry label CPY.

There is no intrinsic limit on the number of formal arguments or local variables which may be specified for a defined function.

6. In addition to the functions described above, several new functions are described elsewhere in this paper. The functions OPSYN, SIZE and TRI of SNOBOL3 are also available. The functions MODE, ANCHOR and UNANCHOR are not included in SNOBOL4.

# 3. PATTERN MATCHING

SNOBOL4 departs radically from SNOBOL3 in the area of pattern matching. While the same rule formats are used to perform pattern matching, the facilities for creating patterns are quite different. In SNOBOL3, the structure of a pattern can only be specified by a particular syntactic configuration and remains fixed throughout program execution. In SNOBOL4, a pattern is a data object which may be constructed and changed during program execution.

## 3.1 Pattern Construction

There are several facilities for constructing patterns. These may be used in combination to construct very elaborate patterns. The following sections describe the fundamentals of pattern construction.

### 3.1.1 ALTERNATION

A pattern which will match any one of a number of alternatives may be formed by use of the binary operator | . Thus

```
VOWEL  =  'A' | 'E' | 'I' | 'O' | 'U'
```

assigns to VOWEL a pattern which will match any single vowel.

Operands may be other patterns as well as strings:

```
EVOWEL  =  VOWEL | 'Y'
```

creates a pattern which will match a Y as well as the other vowels.

The operator | has the lowest precedence of all operators so that

```
A B | C D
```

is equivalent to

```
(A B ) | (C D)
```

See Appendix A.

### 3.1.2 CONCATENATION

Concatenation in SNOBOL4 is the same as in SNOBOL3, but patterns may be concatenated as well. Thus

```
VPAIR   =   VOWEL VOWEL
```

creates a pattern which will match any two vowels in succession.

### 3.1.3 ARBITRARY STRINGS

In SNOBOL4 there is a primitive pattern which will match any string o characters. This pattern corresponds to the 'arbitrary string variable' o SNOBOL3. The variable ARB has this primitive pattern as value at th beginning of program execution. The SNOBOL4 pattern

```
ITEM    =   ARB ","
```

has the SNOBOL3 equivalent

```
** ","
```

When ARB is the last element in a pattern, it does not automatically matc the rest of the string. In this respect it differs from the arbitrar string variable of SNOBOL3. REM, described in Section 3.1.8, is used fo this purpose.

### 3.1.4 BALANCED STRINGS

The variable BAL has as its initial value a primitive pattern which wil match any nonnull string of characters which is balanced with respect t parentheses. BAL is equivalent to the balanced string variable of SNOBOL3 Thus

```
NEST    =   "(" BAL ")"
```

is equivalent to the SNOBOL3 pattern

```
"(" *()* ")"
```

### 3.1.5 FIXED-LENGTH STRINGS

There are several primitive functions in SNOBOL4 which return patterns a value. One of these, LEN, corresponds to the fixed-length string variabl of SNOBOL3. The value of LEN(N) is a pattern which will match any strin which is N characters long.

```
CARDLENGTH   =   LEN(72)
```

is equivalent to the SNOBOL3 pattern

```
*/"72"*
```

### 3.1.6 FIXED POSITIONS IN STRINGS

Two functions, POS and RPOS, have patterns as value which specify fixe positions within strings.

The value of POS(N) is a pattern which will match a null string immediately after the Nth character of a string. For example

```
OPFIELD   =   POS(7) LEN(8)
```

will match eight characters following the seventh character of a string.

The value of RPOS(N) is a pattern which will match a null string N characters from the end of a string. For example

```
LASTCHAR   =   RPOS(1) LEN(1)
```

will match the last character of a string.

### 3.1.7 TABULATION

Two functions, TAB and RTAB, have patterns as value which specify tabulation to fixed positions within a string.

The value of TAB(N) is a pattern which will match up to and including the Nth character. For example

```
OPFIELD1   =   POS(7) TAB(15)
```

is equivalent to the pattern OPFIELD given in Section 3.1.6.

The value of RTAB(N) is a pattern which will match up to the last N characters of a string. For example

```
LAST5   =   RPOS(5) RTAB(0)
```

will match the last five characters of a string.

### 3.1.8 REMAINDER

The variable REM has as its initial value a primitive pattern which will match the remainder of any string.

```
LAST5R   =   RPOS(5) REM
```

is equivalent to the pattern LAST5 in the previous section.

### 3.1.9 ALTERNATIVE CHARACTERS

Two primitive functions, ANY and NOTANY, have pattern values which permit the specification of alternative characters.

The value of ANY(CS) is a pattern which will match any character in the string CS.

```
AVOWEL   =   ANY('AEIOU')
```

is equivalent to the pattern VOWEL in Section 3.1.1. However, ANY is more

efficient and compact than the explicit alternation of the individual characters.

The value of NOTANY(CS) is a pattern which will match any single character which does not occur in the string CS.

```
NOTVOWEL  =  NOTANY('AEIOU')
```

will match any character which is not a VOWEL.

### 3.1.10 RUNS OF CHARACTERS

Two primitive functions, SPAN and BREAK, have patterns as value which permit the specification of runs of characters.

The value of SPAN(CS) is a pattern which will match a string composed of characters which appear in CS. SPAN(CS) will not match the null string.

```
INTEGER  =  SPAN('0123456789')
```

will match any string consisting of digits.

Patterns resulting from SPAN(CS) match the longest possible run of characters from CS, and do not back up to match shorter strings. Thus the pattern

```
M10  =  INTEGER '0'
```

will always fail to match.

The value of BREAK(CS) is a pattern which will match any string up to, but not including any character in CS. BREAK(CS) will match a null string.

```
WORD  =  BREAK(' ,.:;')
```

will match any string of symbols followed by a blank, comma, period, colon, or semicolon. The match fails if none of these symbols occurs.

### 3.1.11 REPETITIONS

The primitive function ARBNO permits the specification of an arbitrary number of repetitions of a pattern. The value of ARBNO(P) is a pattern which will match any string that would be matched by an arbitrary number of consecutive occurrences of the pattern P. For example

```
MUL5  =  ARBNO(LEN(5))
```

will match any string whose length is a multiple of five (including the null string).

Patterns resulting from ARBNO(P) first match the null string (corresponding to zero occurrences of P). If an alternative match is requested as the result of backup, a match for P is attempted. Each

subsequent request as a result of backup results in an attempt to match one more occurrence of P. Thus

```
      M10A    =    ARBNO(ANY('0123456789')) '0'
```

will match the shortest string which consists of digits followed by a zero, including a string consisting of a single zero.

ARBNO(P) may be thought of as an alternation of the form

```
      NULL | P | P P | P P P | P P P P | ...
```

### 3.1.12 SIGNALLING FAILURE

Three primitive patterns permit the specification of various kinds of failure during pattern matching.

The variable FAIL has as its initial value a primitive pattern which will always signal failure to match. FAIL, when encountered, does not necessarily cause the pattern match to fail, but requests the pattern matching algorithm to seek an alternative. FAIL can be used to initialize a pattern to be constructed from many alternatives, much as the null string can be used to initialize the concatenation of many patterns. For example, if INPUT is attached to an input stream

```
      ALTS    =    FAIL
LOOP  ALTS    =    ALTS | TRIM(INPUT) :S(LOOP)
```

builds a pattern consisting of alternatives taken from INPUT.

During pattern matching, the first alternative, FAIL, will request an alternative immediately.

The variable FENCE has as its initial value a primitive pattern which always matches a null string, but causes the entire pattern match to fail if an alternative for it is requested in backup. Thus in

```
      CARD    =    LABEL FENCE OP
```

if LABEL successfully matches, but OP fails to match, the pattern match will fail without seeking alternatives for LABEL. The main use of FENCE is to improve the efficiency of pattern matching by avoiding attempts to match which would be futile.

The variable ABORT has as its initial value a primitive pattern which, if encountered, causes the entire pattern match to fail. Thus

```
      CARDFORM    =    '*' ABORT | CARD
```

will fail if matched against a string which starts with an asterisk, and otherwise will match for the pattern CARD.

## 3.2 The Order of Pattern Matching

The actual process of pattern matching is performed in much the same fashion as in SNOBOL3. In SNOBOL4 the order of pattern matching is much more important because alternatives can be specified.

The two structural aspects of patterns result from alternation and concatenation. Matches for alternatives are attempted from left to right. Thus in the pattern

```
DOTS    =   '....' | '..'
```

an attempt is first made to match for four dots. If this fails, a match for two dots is tried. Thus if DOTS is to be used to compress multiple occurrences of dots, as in

```
RDOTS  TEXT   DOTS   =   '.'   :S(RDOTS)
```

the order in which the alternatives are specified is important for efficiency. In fact, if the alternatives were specified in the other order

```
SDOTS   =   '..' | '....'
```

the second alternative would never match.

Matches for successive components in a concatenation proceed from left to right. If a component fails to match, the matching process backs up to the preceding component and tries a new match for it. Thus

```
BALPER   =   BAL '.'
```

will first match a balanced string. If the next character is a period, the match will succeed. If not, another attempt for BAL will be made, extending the string previously matched if possible.

## 3.3 Deferred Pattern Definition

The creation of a pattern is like the creation of a string in the sense that the current values of its components are used when the pattern is constructed. As a result of executing

```
X   =   'A+B'
X   =   '(' X '*' X ')'
```

the final value of X is (A+B*A+B) . The same principle applies to pattern construction. As a result of executing

```
P   =   'X'
P   =   'Y' | P 'Z'
```

the final value of P is the same as it would be if

```
        P   =   'Y' | 'X' 'Z'
```

were executed.

On the other hand, it is sometimes desireable to defer the evaluation of a component of a pattern until pattern matching takes place. Consider the following section of program.

```
        N   =   0
DELETE LIST '(' N ')'   =     :F(OUT)
        N   =   N + 1     :(DELETE)
OUT
```

Since the value of N changes through a series of pattern matches, the pattern

```
        '(' N ')'
```

must appear explicitly and be reconstructed for each new value of N.

Deferred pattern definition, implemented by the unary operator * overcomes this difficulty. When applied to a variable which appears in a pattern, * causes the evaluation of this component to be deferred until pattern matching actually occurs. Thus

```
        NEST   =   '(' *N ')'
```

can be used as in the previous example:

```
        N   =   0
DELETE LIST   NEST   =   :F(OUT)
        N   =   N + 1     :(DELETE)
OUT
```

In this case, a new value of N is used in each pattern match.

Deferred pattern definition may also be used to achieve recursive patterns. If the example at the beginning of this section is revised so that

```
        P   =   'Y' | *P 'Z'
```

then P will match strings of the form

```
        Y
        YZ
        YZZ
        YZZZ
        .
        .
        .
```

Similarly, the following statements create patterns which will match a simple class of arithmetic expressions.

```
VARIABLE   =   ANY('XYZ')
ADDOP   =   ANY('+-')
MULOP   =   ANY('*/')
FACTOR   =   VARIABLE | '(' *EXP ')'
TERM   =   *FACTOR | *TERM MULOP *FACTOR
EXP   =   ADDOP *TERM | *TERM | *EXP ADDOP *TERM
```

Warning: *N is a pattern whose value is determined when pattern matching is performed. As a consequence,

```
P   =   POS(*N)
```

is illegal, since the argument of POS must be a number.

## 3.4 Value Assignment

In SNOBOL3, a string name can be associated with a string variable so that if a pattern match is successful, the name is given a new value corresponding to the string matched by the variable. SNOBOL4 has two constructions which permit such a value assignment as a result of pattern matching.

### 3.4.1 POST-MATCHING VALUE ASSIGNMENT

In SNOBOL3, value assignment takes place only after the entire pattern has successfully matched. In SNOBOL4, the binary operator . is used to associate a variable with a pattern component for this type of value assignment. Such an association has the form

```
P . V
```

where P is a pattern and V is a variable to be associated. For example

```
P   =   '(' BAL . B ')'
```

is equivalent to the SNOBOL3 pattern

```
'(' *(B)* ')'
```

A variable may be associated with any component of a pattern (including a literal). Value assignment is made only to the components of a pattern which match. Thus

```
LETTER   =   (VOWEL . V | LEN(1) . C) . L
```

creates a pattern which will match any single character. If the character is a vowel it will become the new value of V. If the character is not a

vowel, it will become the new value of C instead. In either case the character will be the new value of L.

The operator . has the highest precedence of all operators and associates to the left. See Appendix A.

```
P   =   '(' BAL . B . OUTPUT ')'
```

would assign the value matched by BAL to both B and OUTPUT. Assignment to an output-associated variable results in output just as if an explicit assignment had been made.

The same variable may appear in value associations more than once in a pattern. Value assignment is done from left to right and from the inside to outside in nestings. A value may be assigned to a variable more than once in this process, and the final value is determined by the order in which the assignments are done. For example, the pattern

```
NEST   =   '(' *NEST . OUTPUT ')' | '(' BAL . OUTPUT ')'
```

when used in

```
'((((X))))' NEST
```

would print

```
X
(X)
((X))
(((X)))
```

### 3.4.2 DYNAMIC VALUE ASSIGNMENT

Whereas the value assignment described in the previous section occurs only on successful completion of pattern matching, there is another type of association which results in value assignment whenever a component of a pattern successfully matches. The binary operator $ associates a variable with a component of a pattern for this dynamic value assignment. The pattern

```
FULLBAL   =   BAL $ OUTPUT   RPOS(0)
```

when used in

```
'A+(B+C)*(D/E)' FULLBAL
```

would print

```
A
A+
A+(B+C)
A+(B+C)*
A+(B+C)*(D/E)
```

Since dynamic value assignment occurs whenever an associated component matches, values of associated variables may be changed even if the pattern match eventually fails. Thus the pattern

```
BALPAIR   =   BAL $ B1   FENCE   BAL $ B2
```

would fail when used in

```
'(A+B))' BALPAIR
```

but (A+B) would be assigned as a new value of B1. The value of B2 would remain unchanged since no successful match would be made for the component with which it is associated.

Dynamic naming, together with deferred pattern definition, can also be used to achieve the effect of SNOBOL3 backreferencing. The pattern

```
L3PAIR   =   LEN(3) $ V   ARB *V
```

will successfully match any string having at least two nonoverlapping identical substrings of length 3. If this pattern matches successfully, the desired substring will be the new value of V. If the pattern does not match, the new value of V will be the last three characters of the subject string.

The pattern FAIL can be used to force a pattern through all possible matches. Used in conjunction with dynamic value assignment, a listing of all the alternatives may be obtained. The pattern

```
ALLBAL   =   BAL $ OUTPUT FAIL
```

when used in

```
'((A+(B*C))+D)' ALLBAL
```

would ultimately fail, but print

```
((A+(B*C))+D)
(A+(B*C))
(A+(B*C))+
(A+(B*C))+D
A
A+
A+(B+C)
+
+(B*C)
(B*C)
B
B*
B*C
*
*C
C
```

```
+
+D
D
```

This device may be used to explicate the exact order in which pattern matching is attempted for any pattern.

## 4. ARRAYS

Arrays may be created by execution of the primitive function ARRAY. ARRAY(P,V) creates an array described by the prototype P and gives each element of the array the value V.

The prototype P describes the indexing and dimensionality of the array. For example

```
VECTOR   =   ARRAY(10)
```

assigns a one-dimensional array of length 10 to VECTOR. Since the second argument is omitted, all elements of the array have null strings as value. Indexing ordinarily starts at 1. Other lower bounds may be specified by using a colon to separate the upper and lower limits:

```
LINE   =   ARRAY('-5:5')
```

creates an array with a lower bound of -5 and an upper bound of 5.

Additional dimensions in a prototype are separated by commas. Thus

```
BOARD   =   ARRAY('8,8','X')
```

defines an eight by eight array where all elements have the value X.

There is no intrinsic limit on the size or dimensionality of an array.

Warning: The first argument of ARRAY is the prototype and the second is a value which is given to each element of the resulting array. Thus,

```
A   =   ARRAY('8,8')
```

creates a two-dimensional array with each element having the null string as value. On the other hand

```
A   =   ARRAY(8,8)
```

creates a one-dimensional array with each element having the value 8.

It is also important to realize that each element of an array is given the same object as value. Consequently

```
A1   =   ARRAY(10)
A2   =   ARRAY(10,A1)
```

creates only two arrays. Each element of A2 has the same array, A1, as value.

If the value of a variable is an array, an element in the array may be referred to through the variable. Angular brackets following any array-valued variable are used to specify the element.

```
VECTOR<2>    =    EXP
```

assigns the value of EXP to the second element of VECTOR. There is no requirement that all values of an array be the same kind of object.

If an index referring to an element of an array falls outside the range of the array, the array reference fails. Thus

```
OUTPUT    =    VECTOR<12>
```

would fail. This failure may be used to iterate through the elements of an array without knowing the size of the array. A function SUM, whose value is the sum of all the elements of an array, could have the defining statement

```
DEFINE('SUM(ARRAY) N')
```

with the definition

```
SUM     N    =    N + 1
        SUM    =    SUM + ARRAY<N>      :S(SUM)F(RETURN)
```

The summation loop continues until N exceeds the extent of ARRAY. This function does not need to know the size of ARRAY, but only that it is a one-dimensional array with lower bound one.

The primitive function PROTOTYPE may be used to get an explicit representation of structure of an array. The value of PROTOTYPE(A) is the prototype of the array A. Thus

```
STRUCTURE    =    PROTOTYPE(BOARD)
```

assigns the value 8,8 to STRUCTURE.

In some cases an array may not be the value of an explicitly known name. The primitive function ITEM permits reference to elements of such an array. The value of ITEM(A,I1,...,In) is the (I1,...,In)th element of the array A. For example

```
X    =    VECTOR<5>
```

and

```
X    =    ITEM('VECTOR',5)
```

are equivalent.

It is important to realize that an array is a data object. The same array may be the value of more than one variable. In this case, a change in

the value of an element affects both variables which have this array as value. For example, as a result of executing the statements

```
A    =   ARRAY(5)
A<2>   =   'TWO'
B    =   A
B<2> 'W'   =
```

the value of A<2> will be TO .

A copy of an array may be made using the function COPY. The copy is not changed by changing elements in the original, and conversely. Consequently, as a result of

```
A    =   ARRAY(5)
A<2>   =   'TWO'
B    =   COPY(A)
B<2> 'W'   =
```

the value of B<2> will be TO and the value of A<2> will be TWO .

## 5. REAL NUMBERS

SNOBOL4 provides a limited facility for real (floating point) arithmetic. Real numbers may appear in the program as literals. For example

```
X   =   72.1527
```

Such literals must begin with a digit and contain a decimal point. Thus

```
N   =   0.01
```

is acceptable, while

```
N   =   .01
```

is erroneous (see Section 12.2).

Addition, subtraction, multiplication, and division (but not exponentiation) may be performed on real numbers. Thus, as a result of

```
M   =   X * 3.24561
```

the value of M becomes 234.1794.

An attempt to perform mixed arithmetic between real numbers and integer strings will result in error termination. Explicit conversion may be made using the function CONVERT. The value of CONVERT(V,T) is the result of converting V to type T. Thus

```
OUTPUT   =   CONVERT(M,'STRING')
```

prints 234.1794 . Similarly, strings can be converted to real numbers.

```
SUM   =   N + CONVERT(5,'REAL')
```

assigns the value 5.01 to SUM. In converting strings to real numbers, either real literals, such as 57.42, or integers such as 5 may be specified.

# 6. DATA TYPES

In SNOBOL3 there is only one kind of data object, the string. SNOBOL4 has many data types. Four types have already been described: STRING, PATTERN, ARRAY, and REAL. Others are described in the following sections.

## 6.1 Data Types in Operations

The data type of an object is used by the SNOBOL4 system to verify that appropriate types are given to procedures, or to select an appropriate procedure, depending on type. For example, the argument of the SIZE function must be a string.

```
VALUE   =   SIZE(ANY('0123456789'))
```

results in error termination since the argument of SIZE is a pattern. Similarly

```
SUM   =   3 + ARRAY(7)
```

is erroneous. In other cases, different procedures are required for different data types. For example

```
SELECT   =   'BIN' SIZE(TRIM(INPUT))
```

is the concatentaion of two strings and the result is a string. On the other hand

```
BINO   =   'BIN' (3 | 4)
```

is the concatenation of a string with a pattern. A different procedure is used and the result is a pattern.

## 6.2 Concatenation with the Null String

In concatenation, the null string is handled as a special case. If one of the two operands in concatenation is the null string, no concatenation is

```
POINTER   =   'X' ARRAY(10)
```

is erroneous since a string cannot be concatenated with an array,

```
POINTER   =   IDENT(MARK) ARRAY(10)
```

is acceptable, since IDENT returns a null string if it succeeds. Thus predicates may be used to achieve conditional expressions without interfering with the results of computation.

## 6.3 Data Type Determination

The programmer usually knows the data types of the objects which occur in his program. Sometimes, however, it is necessary to make an explicit determination. The function DATATYPE serves this purpose. The value of DATATYPE(V) is the data type of V. Thus

```
OUTPUT   =   DATATYPE(SPAN('01'))
```

would print PATTERN.

## 7. PROGRAMMER-DEFINED DATA TYPES

The programmer may define new data types by means of the function DATA. The result of executing DATA(P) is to create a data type and define field functions as given in the prototype P. For example

```
DATA('NODE(FATHER,LSON,RSIB,VALUE)')
```

creates a new data type NODE with four fields: FATHER, LSON, RSIB and VALUE. A NODE may be visualized as shown in Figure 7.1.

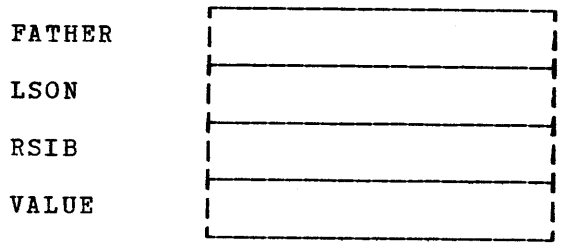



Figure 7.1 Structure of a NODE

Execution of this DATA function defines a function NODE which creates objects of data type NODE. Hence

```
N1 = NODE()
```

creates a NODE which becomes the value of N1. The NODE function has four arguments corresponding to the fields FATHER, LSON, RSIB and VALUE. These fields may be assigned value when a node is created.

```
N2 = NODE(N1,,,'X')
```

creates a node with the node N1 as the value of its FATHER field and X as the value of its VALUE field. The LSON and RSIB fields are null.

Execution of the DATA function also creates field functions FATHER, LSON, RSIB and VALUE which refer to the fields of a NODE. Thus

```
LSON(N1) = N2
```

assigns the node N2 to the LSON field of N1.

Using these functions, nodes can be created and trees constructed from them. The fields FATHER, LSON and RSIB permit representation of the structure of the trees. The VALUE field permits the assignment of contents of the nodes. For example

```
N1    =    NODE(,,,'*')
N2    =    NODE(N1,,,'Y')
N3    =    NODE(N1,,N2,'-')
N4    =    NODE(N3,,,'X')
LSON(N1)    =    N3
LSON(N3)    =    N4
```

creates a tree as illustrated in Figure 7.2.

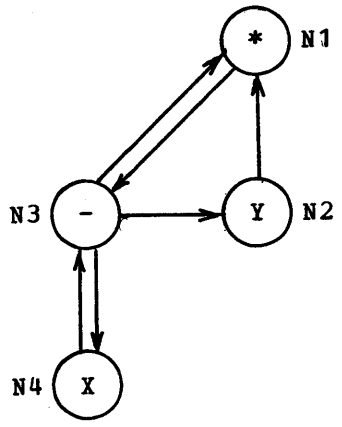



Figure 7.2 Representation of a Tree

Subsequently, executing

```
N5    =    NODE(N3,,,VALUE(N2))
RSIB(N4)    =    N5
VALUE(RSIB(FATHER(N4)))    =    VALUE(N4)
```

produces the tree illustrated in Figure 7.3

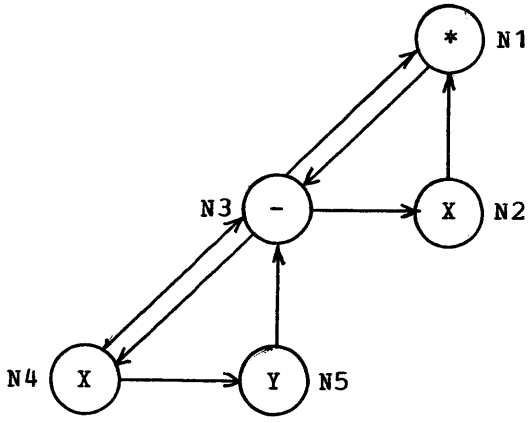



Figure 7.3 The Modified Tree

This facility may be used to implement elaborate data structures. An example given in Appendix C illustrates how a set of tree functions [4] may be implemented.

There is no intrinsic limit on the number of data types which may be defined. The same field function may be defined on several data types. Hence

```
DATA('ITEM(FLINK,BLINK,VALUE)')
```

creates a data type ITEM which has the same field, VALUE, as the data type NODE.

As with arrays, a programmer-defined data object may be the value of more than one variable. A change in the value of a field through one variable will change the field value for the other variable. The function COPY may be used for programmer-defined data objects as well as arrays. See Section 4.

# 8. COMPILATION DURING EXECUTION

The first phase of a SNOBOL4 run is 'compilation' in which the source program is converted into intermediate object code which is then interpreted in an execution phase.

## 8.1 Creating Object Code

A program can compile more object code during execution and then execute this new code. Compilation is accomplished by using the CONVERT function to convert a string to data type CODE. The string to be converted should consist of SNOBOL4 statements terminated by semicolons. For example

```
        NEWS    =   'NEW OUTPUT = SUM; SUM = SUM + 1 :(OLD);'
        NEWCODE =   CONVERT(NEWS,'CODE')      :(NEW)
OLD
```

creates two new statements. One of these statements contains a label NEW. The goto then transfers to this new block of CODE. The two new statements are then executed and transfer made to OLD.

Blanks are as important in strings to be converted to code as they are in the program itself. A statement without a label must begin with a blank. The string to be converted must end with a semicolon.

## 8.2 Direct Gotos

The goto field specifies a variable which occurs as a program label. The result of converting a string to object code is a data object. In the previous example, this data object became the value of the variable NEWCODE. A special type of goto permits transfer directly to a block of object code, rather than through a label. This type of goto uses enclosing angular brackets rather than parentheses. The previous example could have been

```
        NEWS    =   ' OUTPUT = SUM; SUM = SUM + 1 :(OLD);'
        NEWCODE =   CONVERT(NEWS,'CODE')      :<NEWCODE>
OLD
```

In this case transfer is made directly to the value of NEWCODE, and the label NEW is not needed.

Execution-time compilation permits a programmer-defined function CALL similar to the primitive function CALL of SNOBOL3. The defining statement can be

```
      DEFINE('CALL(FORM)S')
```

with the definition

```
CALL  S  =  ' CALL = '  FORM  ' :S(RETURN)F(FRETURN);'
      S  =  CONVERT(S,'CODE')    :<S>
```

The first statement creates the string which will become the code to be executed. The statement is then converted to code and executed. When executed, it evaluates FORM and assigns the result to CALL, through which the value is returned.

# 9. KEYWORDS

Keywords provide an interface between the SNOBOL4 program and certain internal symbols in the SNOBOL4 system. Using keywords the program may determine, for example, how many statements have been executed. Keywords also permit the program to change the value of certain internal symbols, such as the limit on the number of statements which may be executed.

Keywords begin with an ampersand (&) which distinguishes them from program identifiers which may not contain ampersands.

An example is &STCOUNT whose value is the number of statements which have been executed. Similarly, the value of &STLIMIT is the limit on the number of statements which may be executed. Using these two keywords, the program may at some point limit further execution to 100 statements by executing

```
&STLIMIT  =  &STCOUNT + 100
```

&STLIMIT, whose value may be changed, is called an unprotected keyword. The value of &STCOUNT may not be changed, however, and is called a protected keyword. An attempt to change the value of a protected keyword results in error termination. The following sections describe the available keywords.

## 9.1 Protected Keywords

There are two types of protected keywords. The first type includes values internal to the SNOBOL4 system. The second type includes strings and primitive patterns which are predefined in the SNOBOL4 language.

### 9.1.1 INTERNAL VALUES

1. &STCOUNT. The value of &STCOUNT is the number of statements which have been entered during program execution.

2. &STFCOUNT. The value of &STFCOUNT is the number of statements which have failed.

### 9.1.2 PREDEFINED VALUES

Certain keywords have values which are predefined in the SNOBOL4 language. These include the alphabet for the machine, and the primitive patterns. These protected keywords are provided so that their values will always be available.

1. &ALPHABET. The value of &ALPHABET is a string containing all the characters of the machine on which SNOBOL4 is implemented. The characters are ordered according to their internal coding.

2. &ARB. The value of &ARB is the primitive pattern which matches any string of characters. &ARB and ARB have the same value at the beginning of program execution. The value of ARB may be changed, however, while the value of &ARB is protected.

3. &ABORT. &ABORT has the same value as ABORT at the beginning of program execution. See &ARB.

4. &BAL. As above.

5. &FAIL. As above.

6. &FENCE. As above.

7. &REM. As above.

## 9.2 Unprotected Keywords

There are two types of unprotected keywords. The first type includes internal switches. The second type includes internal parameters which may be varied by the program.

### 9.2.1 INTERNAL SWITCHES

Keyword switches controlling the anchored pattern matching mode and the post-mortem dump replace the MODE function of SNOBOL3 [2]. Switches are off if their value is 0 or the null string, and on otherwise. All switches are off at the beginning of program execution.

1. &ANCHOR. If &ANCHOR is on, the pattern matching is anchored. That is, all patterns must match beginning with the first character of the subject string. Thus, e.g.

```
&ANCHOR = 'ON'
```

sets the anchored mode.

2. &DUMP. If &DUMP is on, a post-mortem dump of variable storage will be given.

### 9.2.2 INTERNAL PARAMETERS

1. &MAXLNGTH. The value of &MAXLNGTH is the limit on the length of strings that may be formed. The initial value of &MAXLNGTH is 5000, but this may be changed. Thus

```
&MAXLNGTH   =   1000
```

limits the maximum length of subsequent strings to 1000 characters. An attempt to form a string longer than the limit results in error termination of the program. All types of string formation are included in this limit: concatenation, value assignment as a result of pattern matching, and string input.

2. &STLIMIT. The value of &STLIMIT is the limit on the number of statements which may be executed (see &STCOUNT). The initial value of &STLIMIT is 50000. Exceeding the limit on statement execution results in error termination.

# 10. TRUTH PREDICATES

Two predicates, implemented by unary operators, are available for testing the success or failure which may result from evaluating expressions.

## 10.1 Negation

The unary operator ¬ fails if its operand succeeds and succeeds if its operand fails. A null string value is returned on success facilitating its use among other constructions. Thus

```
M<0>  =  ¬M<N> ¬M<-N> 0
```

assigns 0 to M<0> only if both N and -N are <u>out</u> of range of the array M.

Similarly

```
N  =  ¬F(N) N + 1
```

increments N only if F(N) fails.

## 10.2 Affirmation

The unary operator ? is the converse of ¬ . It succeeds if its operand succeeds and fails if its operand fails. A null string is returned on success permitting its insertion in other constructions without affecting their values. Thus

```
M<0>  =  ?M<N> ¬M<-N> 1
```

assigns 1 to M<0> only if N is <u>in</u> range and -N is <u>out</u> of range of the array M.

# 11. INPUT AND OUTPUT

As in SNOBOL3, input and output are accomplished by associating variables with data sets. Three variables have standard associations:

1. INPUT is associated with the normal input data set.
2. OUTPUT is associated with the normal print data set.
3. PUNCH is associated with the normal punch data set.

Input occurs whenever the value of the associated variable is used. Thus

```
CARD    =    INPUT
```

results in reading from the normal input data set. The resulting string becomes the value of CARD.

Similarly, output occurs whenever the value of the associated variable is changed. Thus

```
OUTPUT    =    CARD
```

causes the value of CARD to be printed.

## 11.1 I/O Association Functions

Other variables may be associated with other data sets using the primitive functions PRINT and READ. These functions have the form

```
READ(V,N,L)
PRINT(V,N,F)
```

where

1. V is the variable to be associated.
2. N is the data set reference number (symbolic unit number) with which the association is to be made.
3. L is the length of a string to be read on input.
4. F is a format to be used for output.

The three standard I/O variables have associations corresponding to

```
READ('INPUT',5,80)
PRINT('OUTPUT',6,'(1X,131A1)')
PRINT('PUNCH',7,'(80A1)')
```

FORTRAN IV conventions for data set reference numbers apply. Data se reference numbers from 1 through 99 are usually available. The data se reference number may not be omitted.

## 11.2 Output

Valid FORTRAN IV formats must be used for output. A format may specii literals and the output of a string by A-conversion (using n A1 to output string of n characters). Numbers are strings in SNOBOL4 and must be put ou by A-conversion. For example

```
PRINT('CONTROL',6,'(132A1)')
```

associates control with the normal print data set. With the specifie format, the first character of CONTROL is used for carriage control. Thus

```
CONTROL  =  1
```

results in a page eject.

A literal may be included to provide other desired information: t identify the particular variable being printed, for example. The associa tion

```
PRINT('SUM',6,'(5H SUM=,120A1)')
```

would result in the variable SUM being printed with its name prefixed a given in the literal part of the format. Similarly

```
PRINT('TITLE',6,'(1H1,131A1/(1X,131A1))')
```

associates TITLE so that when a value is assigned to TITLE, a page i ejected and the value titles the next page of output.

If the format is omitted, and the data set reference number is 6, default format of (1X,131A1) is used. For all other data set referenc numbers, the default format is (80A1).

If output is requested for a data object which is not a string, the nam of the data type is printed. Thus

```
OUTPUT  =  3.5
```

would print REAL .

## 11.3 Input

Any positive number up to the maximum allowed string length may be used to specify input length. Thus

```
READ('CARD',5,72)
```

associates CARD with the normal input data set. Subsequently

```
IMAGE  =  CARD
```

reads in a string of 72 characters which becomes the value of IMAGE.

If the length specified is shorter than the record length on the input data set, the remainder of the record is lost. If the length is longer than the record length, enough records are read to satisfy the input request.

If an end of file (end of data set) is encountered on input, the statement which requested the input fails.

If the length is omitted, the default length is 80.

## 11.4 Rewind

The primitive function REWIND rewinds a file. REWIND(N) rewinds the data sets associated with the data set reference number N. The next input request will result in reading from the first data set (file) associated with N.

## 11.5 Back Space

The primitive function BACKSPACE backspaces a file. BACKSPACE(N) backspaces one record on the data set currently associated with the data set reference number N. If the data set is positioned at its first record, BACKSPACE(N) has no effect.

## 11.6 End of File

The primitive function ENDFILE ends a file. ENDFILE(N) writes an end of file on (closes) the data set associated with the data set reference number N. The next output request is written on a new data set (file).

# 12. NAMES

There are several circumstances in which explicit handling of names i useful. A name is any object which can have a value. In

```
WORD    =    'MAY'
```

WORD is a name which is given the value MAY . The basic relation betwee names and values is exhibited by the indirectness operator $ . Fc example, in

```
$WORD    =    2
```

the name MAY is given the value 2. Through indirectness, any string can b used as a name.

Objects other than strings may be used as names. Individual array item and fields of programmer-defined data objects are examples.

```
BOARD<-1,1>    =    'X'
```

and

```
LSON(ROOT)    =    HTREE
```

are examples of computed names to which values are assigned.

## 12.1 Passing Names

A number of functions interpret the values of their arguments as names For example

```
PRINT('SUM',N,F)
```

associates the name SUM in the output sense (see Section 11). Subsequently whenever the value of SUM is changed, output is performed.

String names are typically passed in this manner as literals. Compute names, such as BOARD<-1,1> cannot be passed as literals. Thus

```
PRINT(BOARD<-1,1>,N,F)
```

associates the value of BOARD<-1,1>, but does not associate the array ite BOARD<-1,1> . On the other hand

```
PRINT('BOARD<-1,1>',N,F)
```

just associates the string of symbols BOARD<-1,1> and not the array item, just as the statements

```
X   =   'BOARD<-1,1>'
$X  =   5
```

have no connection with the array BOARD.

## 12.2 The Name Operator

To overcome this difficulty and put computed names on a par with string names, the unary name operator . may be used. The value of

```
.BOARD<-1,1>
```

is the name of BOARD<-1,1>. Thus

```
PRINT(.BOARD<-1,1>,N,F)
```

associates the array item in the output sense, and output is performed whenever this array item gets a new value. Similarly

```
READ(.LSON(ROOT),M,F)
```

forms an input association with LSON(ROOT), so that whenever a value for LSON(ROOT) is requested, a new value is obtained by input.

The name operator serves much the same purpose for computed names as quotation marks do for string names. The name operator applied to a string name behaves the same as quotation marks. Thus

```
WORD   =   .MAY
$WORD  =   2
```

produces the same result as the example above. Indirectness may be applied to any value obtained by the name operator. Hence

```
MARKER   =   .LSON(ROOT)
$MARKER  =   HTREE
```

is equivalent to

```
LSON(ROOT)   =   HTREE
```

If the argument of the name operator is a string, the value returned by the name operator has data type STRING. If the argument of the name operator is a computed name, the value returned has data type NAME. If the argument of the name operator is not a name, error termination occurs. For example

```
.SIZE(WORD)
```

is erroneous.

## 12.3 Returning by Name

A programmer-defined function may return a computed name (rather th value) by transferring to the label NRETURN which signals return by name.

An example of this feature exists in the programming of tree function where a NODE may be defined by

```
DATA('NODE(FATHER,LSON,RSIB,VALUE)')
```

The field functions FATHER, LSON, RSIB, and VALUE are automatically define Additional functions may be desired, however. For example, a functic ROOTFATHER, which is the father field of a tree's root node might b defined. The defining statement could be

```
DEFINE('ROOTFATHER(NODE)','RTF')
```

with the definition

```
RTF     IDENT(FATHER(NODE))                    :S(RTF1)
        NODE   =   FATHER(NODE)                :(RTF)
RTF1    ROOTFATHER   =   NODE                  :(RETURN)
```

This function merely returns the node which is the root. The father fiel could be returned with the following definition:

```
RTF     IDENT(FATHER(NODE))                    :S(RTF1)
        NODE   =   FATHER(NODE)                :(RTF)
RTF1    ROOTFATHER   =   .FATHER(NODE)         :(NRETURN)
```

The naming operator assigns the computed field name to ROOTFATHER. Th transfer to NRETURN indicates the value of ROOTFATHER is to be returned as name. Thus

```
ROOTFATHER(TREE)   =   NEWNODE
```

assigns the value of NEWNODE to the father field of the root of TREE.

NRETURN can always be avoided by resorting to other constructions. I the definition were

```
RTF     IDENT(FATHER(NODE))                    :S(RTF1)
        NODE   =   FATHER(NODE)                :(RTF)
RTF1    ROOTFATHER   =   .FATHER(NODE)         :(RETURN)
```

the corresponding assignment statement would be

```
$ROOTFATHER(TREE)    =    NEWNODE
```

NRETURN permits ROOTFATHER to be used on a par with FATHER, LSON, RSIB, and VALUE without the need for indirectness.

## 13. ADDITIONAL FUNCTIONS

In addition to the functions described earlier in this paper, there ar two functions derived from supplementary functions developed for SNOBOL These are REPLACE, corresponding to the SNOBOL3 function RPLACE [3], ar LGT, corresponding to LEXGT [5].

### 13.1 Character Replacement

One-to-one character replacement in a string may be accomplished usin the function REPLACE. The value of REPLACE(S,CS1,CS2) is the result c replacing in S characters in CS1 by corresponding characters in CS2. Fo example, as a result of

```
TEXT  =  REPLACE(TEXT,',.','.,')
```

all commas in TEXT are replaced by periods, and conversely.

### 13.2 Lexicographical Comparison

Two strings may be compared according to their lexicographic (alphabetic position by using the function LGT ('lexicographically greater than') LGT(A,B) succeeds and returns a null value if A follows B in alphabeti order, and fails otherwise. Thus

```
LGT('ARMY','AIR FORCE')
```

succeeds, while

```
LGT('ARMY','NAVY')
```

fails.

In the case that a string is an initial substring of another, the longe string is lexicographically greater. consequently

```
LGT('AIR FORCES','AIR FORCE')
```

succeeds.

The order of the characters in lexicographical ordering is given in the keyword &ALPHABET. See Section 9.1.2.

## ACKNOWLEDGEMENTS

The SNOBOL4 language was developed over a period of time, and the authors are indebted to many people for their suggestions. The contributions of Messrs. B. N. Dickman, D. J. Farber, P. D. Jensen, M. D. McIlroy and R. F. Rosin have been particularly significant.

Contributions to the implementation have been made by Messrs. I. Benyacar, A. R. Breithaupt, B. N. Dickman, R. S. Gaines, Mrs. M. R. Hawkins, Messrs. P. D. Jensen, A. M. Jones, Mrs. D. F. Teitelbaum, and Mr. L. C. Varian. The authors wish to thank Mr. Varian in particular for his valuable assistance in preparing the IBM System/360 implementation.

The authors gratefully acknowledge the assistance of Mr. J. F. Gimpel in the preparation of this paper. In addition, Mr. Gimpel contributed four of the programs in Appendix C: character conversion, random number generation, 'Typeset', and 'The Towers of Hanoi'.

## REFERENCES

1. Griswold, R. E., Poage, J. F., and Polonsky, I. P., Preliminary Description of the SNOBOL4 Programming Language, Unpublished, July 6, 1967.

2. Farber, D. J. Griswold, R. E. and Polonsky, I. P., The SNOBOL3 Programming Language. Bell System Technical Journal 45,6 (July-August 1966), pp. 895-943.

3. Manacher, G. K., A Package of Subroutines for the SNOBOL Language. Unpublished, July 1, 1964.

4. Griswold, R. E., and Polonsky, I. P., Tree Functions for SNOBOL3. Unpublished, February 1, 1965.

5. Griswold, R. E., Special Purpose SNOBOL3 Functions - II. Unpublished, April 18, 1966.

# APPENDIX A: OPERATOR PRECEDENCE

The relative precedence of the binary operators is listed below in order of decreasing precedence. Operators with the same precedence are listed on the same line. Exponentiation associates to the right. All other operators associate to the left.

| Operator | Name |
|---|---|
| $ . | Value Assignment |
| ** | Exponentiation |
| * | Multiplication |
| / | Division |
| + - | Addition and Subtraction |
| | Concatenation |
| \| | Alternation |

# APPENDIX B: LIST OF FUNCTIONS WITH SECTION REFERENCES

| Function | Section |
|---|---|
| ANY(CS) | 3.1.9 |
| ARBNO(P) | 3.1.11 |
| ARRAY(P,V) | 4 |
| BACKSPACE(N) | 11.5 |
| BREAK(CS) | 3.1.10 |
| CONVERT(V,T) | 5, 8.1 |
| COPY(S) | 4, 7 |
| DATA(P) | 7 |
| DATATYPE(X) | 6.3 |
| DEFINE(P,L) | 2.2 |
| DIFFER(X,Y) | 2.2 |
| ENDFILE(N) | 11.6 |
| EQ(N,M) | 2.2 |
| GE(N,M) | 2.2 |
| GT(N,M) | 2.2 |
| IDENT(X,Y) | 2.2 |
| INTEGER(X) | 2.2 |
| ITEM(A,I1,...,In) | 4 |
| LEN(N) | 3.1.5 |
| LE(N,M) | 2.2 |
| LGT(A,B) | 2.2 |
| LT(N,M) | 2.2 |
| NE(N,M) | 2.2 |
| NOTANY(CS) | 3.1.9 |
| OPSYN(F,G) | 2.2 |
| POS(N) | 3.1.6 |
| PRINT(V,N,F) | 11.1, 11.2 |
| PROTOTYPE(A) | 4 |
| READ(V,N,L) | 11.1, 11.3 |
| REWIND(N) | 11.4 |
| REPLACE(S,CS1,CS2) | 13.2 |
| RPOS(N) | 3.1.6 |
| RTAB(N) | 3.1.7 |
| SIZE(S) | 2.2 |
| SPAN(CS) | 3.1.10 |
| TAB(N) | 3.1.7 |
| TRIM(S) | 2.2 |

# APPENDIX C: SAMPLE PROGRAMS

This appendix contains twelve sample SNOBOL4 programs and their printed output. These programs illustrate various features and uses of the language from the simplest character manipulation through the most complicated recursive pattern matching.

The sample programs are:

1. Character Conversion
2. Word Counting
3. Bubble Sort
4. Random Number Generation
5. 'Typeset'
6. Column Justification
7. 'The Towers of Hanoi'
8. Theorem Proving
9. Magic Square Generation
10. Regular Expression Recognition
11. Phrase Structure Grammar Recognition
12. Tree Functions

## SAMPLE PROGRAM 1: CHARACTER CONVERSION

```
SNOBOL4 (PRELIMINARY VERSION, 9.18.67)

*************************************************************************
*                                                                       *
*          BCD  TO   EBCDIC                                             *
*                                                                       *
*         THE FOLLOWING ONE-LINE PROGRAM CAN BE USED BY                 *
*    INSTALLATIONS UNDERGOING A CHANGE FROM IBM'S SECOND GENERATION     *
*    TO THIRD GENERATION HARDWARE.  THE PROGRAM CONVERTS FROM THE OLD   *
*    BCD CODE FOR SCIENTIFIC CHARACTERS TO THE NEW EBCDIC CODE.  IN     *
*    PARTICULAR, IF INPUT IS THE CARD READER AND IF PUNCH IS THE CARD   *
*    PUNCH, AS IS USUALLY THE CASE, THEN THE PROGRAM CONVERTS A DECK    *
*    OF CARDS FROM 026 KEY PUNCH CODE TO 029 KEY PUNCH CODE.            *
*                                                                       *
*************************************************************************
*
L       PUNCH  =  REPLACE(INPUT, "#@%<&" , "='()+" )      :S(L)          1
END                                                                      2

SUCCESSFUL COMPILATION
```

MTS-560-0

12-1-67

## SAMPLE PROGRAM 2: WORD COUNTING

```
SNOBOL4 (PRELIMINARY VERSION, 9.18.67)

*               THIS PROGRAM COMPUTES THE NUMBER OF USES OF
*       EACH WORD IN A BODY OF TEXT.  LIMITED PUNCTUATION AND
*       INDENTING OF PARAGRAPHS IS RECOGNIZED.
*
        SEPARATOR  =  ' '  |  ', '  |  '. '  |  '.'                    1
READ    OUTPUT     =   TRIM(INPUT)  ' '                  :F(NEXT)     2
        TEXT       =   TEXT  OUTPUT                      :(READ)      3
*
NEXT    TEXT       ARB . WORD  SEPARATOR  =              :F(PRINT)    4
        IDENT(WORD,NULL)                                 :S(NEXT)     5
        $(WORD ':')  =  $(WORD ':') + 1                               6
        NE($(WORD ':'),1)                                :S(NEXT)     7
        LIST       =   LIST  WORD  ','                   :(NEXT)      8
*
PRINT   OUTPUT     =                                                  9
        OUTPUT     =   'COUNT   WORD'                                10
        OUTPUT     =                                                 11
MORE    LIST       BREAK(',') . WORD  ','  =             :F(END)     12
        OUTPUT     =   '  ' $(WORD ':')  '     '  WORD   :(MORE)     13
END                                                                  14

SUCCESSFUL COMPILATION
```

MTS-560-0

12-1-67

THE IDEAL COMPUTING MACHINE MUST THEN HAVE ALL ITS DATA INSERTED AT THE BEGINNING, AND MUST BE AS FREE AS POSSIBLE FROM HUMAN INTERFERENCE TO THE VERY END. THIS MEANS THAT NOT ONLY MUST THE NUMERICAL DATA BE INSERTED AT THE BEGINNING, BUT ALSO THE RULES FOR COMBINING THEM, IN THE FORM OF INSTRUCTIONS COVERING EVERY SITUATION WHICH MAY ARISE IN THE COURSE OF THE COMPUTATION.

| COUNT | WORD |
|---|---|
| 9 | THE |
| 1 | IDEAL |
| 1 | COMPUTING |
| 1 | MACHINE |
| 3 | MUST |
| 1 | THEN |
| 1 | HAVE |
| 1 | ALL |
| 1 | ITS |
| 2 | DATA |
| 2 | INSERTED |
| 2 | AT |
| 2 | BEGINNING |
| 1 | AND |
| 2 | BE |
| 2 | AS |
| 1 | FREE |
| 1 | POSSIBLE |
| 1 | FROM |
| 1 | HUMAN |
| 1 | INTERFERENCE |
| 1 | TO |
| 1 | VERY |
| 1 | END |
| 1 | THIS |
| 1 | MEANS |
| 1 | THAT |
| 1 | NOT |
| 1 | ONLY |
| 1 | NUMERICAL |
| 1 | BUT |
| 1 | ALSO |
| 1 | RULES |
| 1 | FOR |
| 1 | COMBINING |
| 1 | THEM |
| 2 | IN |
| 1 | FORM |
| 2 | OF |
| 1 | INSTRUCTIONS |
| 1 | COVERING |
| 1 | EVERY |
| 1 | SITUATION |

```
MTS-560-0

12-1-67


 1      WHICH
 1      MAY
 1      ARISE
 1      COURSE
 1      COMPUTATION
```

## SAMPLE PROGRAM 3: BUBBLE SORT

```
SNOBOL4 (PRELIMINARY VERSION, 9.18.67)

*
*          BUBBLE SORT PROGRAM
*
          DEFINE('SORT(N)I')                                              1
          DEFINE('SWITCH(I)TEMP')                                         2
          DEFINE('BUBBLE(I)')                                             3
*
*
*                    GET THE NUMBER OF ITEMS TO BE SORTED.
*
          N         =  TRIM(INPUT)                                        4
          A         =  ARRAY(N)                                           5
*
*                    READ IN ITEMS.
*
READ      I         =  I + 1                                              6
          A<I>      =  TRIM(INPUT)                :F(GO)                  7
          OUTPUT    =  A<I>                       :(READ)                 8
*
GO        SORT(N)                                                         9
          OUTPUT    =                                                    10
          OUTPUT    =                                                    11
          OUTPUT    =  'SORTED LIST'                                     12
          OUTPUT    =                                                    13
          I         =  1                                                 14
PRINT     OUTPUT    =  A<I>                       :F(END)                15
          I         =  I + 1                      :(PRINT)               16
*
*
*                    FUNCTIONS
*
*
SORT      I    =    LT(I,N - 1)  I + 1            :F(RETURN)             17
          LGT(A<I>,A<I + 1>)                      :F(SORT)               18
          SWITCH(I)                                                      19
          BUBBLE(I)                               :(SORT)                20
*
SWITCH    TEMP      =  A<I>                                              21
          A<I>      =  A<I + 1>                                          22
          A<I + 1>  =  TEMP                       :(RETURN)              23
*
*
BUBBLE    I         =  GT(I,1)  I - 1             :F(RETURN)             24
          LGT(A<I>,A<I + 1>)                      :F(RETURN)             25
          SWITCH(I)                               :(BUBBLE)              26
*
```

```
MTS-560-0

12-1-67

END                                                              27
SUCCESSFUL COMPILATION
```

GETLTH
EQUAL
GENVAR
ADJUST
DVREAL
END
ETIME
EXPINT
ENDJOB
FSHRTN
ADJTTL
BKSPCE
GETBAL
BUKINT
CHKVAL
CLERTB
COMPAR
BRANCH
COMPLG
COPPLX
FATHER
GETDT
GETLG
ADREAL
GETBLK
EQDT
GETCL
GETCLI
BUFFER
ARRAY
BKSIZE
DESCR
EQU
FETCH
APDSP
EQUIV
COPY
CPEQV
DIVIDE
FNDRES
DECRC
DECR
BUCKET
FORMAT
ALTERN
DIVINT
ADDLG
ADDSIB
ADDSON

SORTED LIST

MTS-560-0

12-1-67

```
ADDLG
ADDSIB
ADDSON
ADJTTL
ADJUST
ADREAL
ALTERN
APDSP
ARRAY
BKSIZE
BKSPCE
BRANCH
BUCKET
BUFFER
BUKINT
CHKVAL
CLERTB
COMPAR
COMPLG
COPPLX
COPY
CPEQV
DECR
DECRC
DESCR
DIVIDE
DIVINT
DVREAL
END
ENDJOB
EQDT
EQU
EQUAL
EQUIV
ETIME
EXPINT
FATHER
FETCH
FNDRES
FORMAT
FSHRTN
GENVAR
GETBAL
GETBLK
GETCL
GETCLI
GETDT
GETLG
GETLTH
```

## SAMPLE PROGRAM 4: RANDOM NUMBER GENERATION

```
SNOBOL4 (PRELIMINARY VERSION, 9.18.67)

*****************************************************************************
*                                                                           *
*                                                                           *
*                    A  RANDOM  NUMBER  GENERATOR                           *
*                                                                           *
*                                                                           *
*     RANDOM(N)  RETURNS A VALUE UNIFORMLY DISTRIBUTED OVER THE             *
*     INTEGERS  0,1,2,...,N-1                                               *
*                                                                           *
*          THE PSEUDO-RANDOM NUMBER GENERATION IS ACCOMPLISHED BY THE       *
*     SO-CALLED POWER RESIDUE METHOD.  THE VARIABLE RAN.VAR                 *
*     CYCLES THROUGH ALL NONNEGATIVE INTEGERS BELOW 100,000.                *
*     THE INITIAL VALUE OF RAN.VAR WILL DETERMINE THE SEQUENCE              *
*     OBTAINED AND IS CALLED THE WARM-UP CONSTANT.                          *
*                                                                           *
*     REFERENCE:                                                            *
*     J. M. HAMMERSLEY AND D. C. HANDSCOMB, 'MONTE CARLO METHODS',          *
*     METHUEN & CO. LTD., LONDON, 1965; PP. 27-29.                          *
*                                                                           *
*****************************************************************************
        DEFINE('RANDOM(N)')                              :(RANDOM.END)        1
RANDOM                                                                        2
        RAN.VAR  =  RAN.VAR * 1061  +  3251                                   3
        RAN.VAR    ARB  RPOS(5)  =                                            4
        RANDOM  =  (RAN.VAR * N) / 100000                :(RETURN)            5
RANDOM.END                                                                    6
*****************************************************************************
*                                                                           *
*      TO ILLUSTRATE ITS USE WE WILL GENERATE AND PRINT A FEW               *
*      'RANDOM' NUMBERS.                                                    *
*                                                                           *
*****************************************************************************
        N = 50                                                                7
        RAN.VAR = 0                                                           8
        RANGE = 100                                                           9
        OUTPUT = '    THE FIRST ' N ' RANDOM NOS.'                           10
        OUTPUT = '    WITH WARM-UP CONSTANT '  RAN.VAR                       11
        OUTPUT  =  '    UNIFORMLY DISTRIBUTED BETWEEN 0 AND '                 12
         (RANGE - 1)  ' ARE:'                                                 12
       OUTPUT  =                                                              13
DEMO                                                                          14
       OUTPUT  =  '                    '  RANDOM(RANGE)                      15
        N  =  GT(N,1)  N - 1                             :S(DEMO)             16
END                                                                           17

SUCCESSFUL COMPILATION
```

MTS-560-0

12-1-67

THE FIRST 50 RANDOM NOS.
WITH WARM-UP CONSTANT 0
UNIFORMLY DISTRIBUTED BETWEEN 0 AND 99 ARE:

3
52
71
99
52
82
36
17
45
17
63
10
44
63
43
95
27
3
2
80
28
35
43
12
79
14
84
15
45
89
0
25
68
48
61
38
8
53
73
44
78
82
19
70
89
69
97
17

MTS-560-0

12-1-67

88
46

MTS-560-0

12-1-67

## SAMPLE PROGRAM 5: 'TYPESET'

SNOBOL4 (PRELIMINARY VERSION, 9.18.67)

```
*************************************************************************
*                                                                       *
*          PARAGRAPH()  IS A FUNCTION WHICH SCANS THE INPUT TEXT UP     *
*      TO THE FIRST LINE OF A NEW PARAGRAPH (INDICATED BY INDENTATION   *
*      I.E. A BLANK IN COLUMN 1).  IT STRINGS ALL OF THE WORDS OF THE   *
*      PARAGRAPH INTO ONE LONG STRING.  BLANKS ARE INSERTED BETWEEN     *
*      LINES (ONE BLANK NORMALLY AND 2 BLANKS IF THE FIRST LINE ENDS    *
*      IN A PERIOD).  WHEN NO MORE PARAGRAPHS REMAIN, PARAGRAPH FAILS.  *
*                                                                       *
*************************************************************************
      DEFINE('PARAGRAPH(X)')                              :(PARA.END)      1
PARAGRAPH                                                                  2
* THIS IS THE ENTRY POINT FOR THE FIRST TIME PARAGRAPH IS CALLED.
* SUBSEQUENT CALLS ENTER AT PARA.1
      DEFINE('PARAGRAPH(X)','PARA.1' )                                     3
      PARA.LINE  =  TRIM(INPUT)                                            4
PARA.1                                                                     5
     OUTPUT  =                                                             6
     OUTPUT = PARA.LINE                                                    7
      PARAGRAPH  =  PARA.LINE                                              8
PARA.2                                                                     9
      PARA.LINE  =  TRIM(INPUT)                          :F(PARA.3)       10
* CHECK FOR LEADING BLANK
      PARA.LINE  POS(0)  ' '                             :S(RETURN)       11
      PARAGRAPH  '.'  RPOS(0)  =  '. '                                     12
                                                                           13
     OUTPUT = PARA.LINE
      PARAGRAPH  =  PARAGRAPH  ' '  PARA.LINE            :(PARA.2)        14
                                                                           15
PARA.3
      DEFINE('PARAGRAPH(X)','PARA.4')                    :(RETURN)        16
PARA.4                                                   :(FRETURN)       17
PARA.END                                                                   18
*************************************************************************
*                                                                       *
*          THIS IS THE MAIN PROGRAM.  ITS NAME IS TYPSET AND ITS        *
*       MAIN PURPOSE IS TO PRINT OUT A PARAGRAPH WHICH IT HAS READ IN   *
*      SUCH THAT BOTH LEFT AND RIGHT COLUMNS ARE ADJUSTED (SEE EXAMPLE  *
*      BELOW).  IT DOES THIS BY PADDING OUT BLANK AREAS WITHIN A LINE   *
*          IF NO SUCH HOLES ALREADY EXIST WITHIN THE LINE, THEN         *
*      THE PROGRAM BEGINS TO SEPARATE THE LETTERS OF INDIVIDUAL WORDS   *
*                                                                       *
*************************************************************************
      INDENTATION  =  '     '                                            19
      LINE.WIDTH  =  60                                                    20
      NB  =  NOTANY(' ')                                                   21
      HOLE  =  NB . A  *BLANK . B  NB . C                                  22
      UNPADDED.LINE  =  ARB . LINE  ' '  ARBNO(' ')  ARBNO(NB) . Y        23
```

```
                                                          POS(LINE.WIDTH + 1)    23
       LONG.WORD = (NB ARB) . LINE SPAN(' ')   NULL . Y                          24
TYPSET                                                                           25
       P  =  PARAGRAPH()                                  :F(END)                26
      OUTPUT  =                                                                  27
TS.0                                                                             28
       LE(SIZE(P),LINE.WIDTH)                             :F(TS.1)               29
       OUTPUT  =  INDENTATION  P                          :(TYPSET)              30
TS.1                                                                             31
       P  UNPADDED.LINE  |  LONG.WORD   =                 :F(ERROR)              32
       P  =  Y  P                                                                33
       BLANK  =                                                                  34
       LINE   NB  SPAN(' ')  NB                           :F(TS.3)               35
TS.2   BLANK  =  LT(SIZE(BLANK),LINE.WIDTH)  BLANK  ' '   :F(NEXT)               36
TS.3                                                                             37
       GE(SIZE(LINE) , LINE.WIDTH)                        :S(NEXT)               38
       LINE   HOLE   =   A B ' ' C                        :F(TS.2)S(TS.3)        39
NEXT                                                                             40
       OUTPUT   =   INDENTATION LINE                      :(TS.0)                41
END                                                                              42
```

SUCCESSFUL COMPILATION

MTS-560-0

12-1-67

"SCIENTIFIC MEN MUST
OFTEN EXPERIENCE A FEELING NOT FAR REMOVED FROM ALARM, WHEN WE CONTEMPLATE
THE FLOOD OF NEW KNOWLEDGE WHICH EACH YEAR BRINGS WITH IT. NEW
SOCIETIES SPRING INTO EXISTENCE, WITH THEIR PROCEEDINGS
AND TRANSACTIONS, LADEN
WITH THE LATEST DISCOVERIES, AND NEW JOURNALS CONTINUALLY APPEAR
IN RESPONSE TO THE GROWING DEMAND FOR POPULAR SCIENCE. EVERY YEAR
THE ADDITIONS TO THE COMMON STOCK OF KNOWLEDGE BECOME MORE BULKY, IF
NOT MORE VALUABLE; AND ONE IS IMPELLED TO ASK, WHERE
IS THIS TO END?"
. . . LORD JOHN WILLIAM STRUTT RAYLEIGH, 1874

"SCIENTIFIC MEN MUST OFTEN EXPERIENCE A FEELING NOT FAR REMOVED FROM ALARM, WHEN WE CONTEMPLATE THE FLOOD OF NEW KNOWLEDGE WHICH EACH YEAR BRINGS WITH IT. NEW SOCIETIES SPRING INTO EXISTENCE, WITH THEIR PROCEEDINGS AND TRANSACTIONS, LADEN WITH THE LATEST DISCOVERIES, AND NEW JOURNALS CONTINUALLY APPEAR IN RESPONSE TO THE GROWING DEMAND FOR POPULAR SCIENCE. EVERY YEAR THE ADDITIONS TO THE COMMON STOCK OF KNOWLEDGE BECOME MORE BULKY, IF NOT MORE VALUABLE; AND ONE IS IMPELLED TO ASK, WHERE IS THIS TO END?"
. . . LORD JOHN WILLIAM STRUTT RAYLEIGH, 1874

## SAMPLE PROGRAM 6: COLUMN JUSTIFICATION

SNOBOL4 (PRELIMINARY VERSION, 9.18.67)

```
*           KEYPUNCH OPERATORS FREQUENTLY HAVE DIFFICULTY RIGHT
*       ADJUSTING VARIABLE LENGTH DATA WITHIN FIELDS ON DATA CARDS.
*       KEYPUNCH ERRORS MAY BE REDUCED IF THE DATA ARE KEYED ONTO
*       THE CARDS WITH AN ARBITRARY NUMBER OF BLANKS BETWEEN DATA
*       ITEMS.  THE FOLLOWING PROGRAM WILL THEN RIGHT JUSTIFY THE
*       DATA IN SPECIFIED FIELDS AND FLAG CARDS THAT HAVE TOO MANY
*       FIELDS, TOO FEW FIELDS, OR FIELDS THAT ARE TOO LONG FOR THE
*       SPACE PROVIDED.
*
*           THE FIRST CARD OF THE INPUT DECK LISTS THE RIGHT-
*       MOST COLUMNS OF ALL FIELDS ON THE OUTPUT CARDS.  THE NUMBER OF
*       FIELDS WHICH MAY BE SPECIFIED IS LIMITED BY A TOTAL OF
*       80 CHARACTERS.  THUS,
*
*               8,16,24,48,72
*
*       LEFT JUSTIFIED ON THE FIRST CARD SPECIFIES 5 OUTPUT FIELDS
*       OF SIZE 8, 8, 8, 24, AND 24 COLUMNS RESPECTIVELY.
*
        M           =       ARRAY(30)                                       1
        BLANKS      =       '                                       '       2
.                           '                                       '       2
        PAT         =       POS(0) BREAK(' ') . FIELD SPAN(' ')             3
*
*       INITIALIZE MATRIX WITH SIZE OF FIELDS
*
        COLS        =       TRIM(INPUT) ','                                 4
INIT    COLS        ARB . COL ',' =                 :F(READ)                5
        L           =       L + 1                                           6
        M<L>        =       COL - LINE                                      7
        LINE        =       COL                     :(INIT)                 8
*
*       READ AND REFORMAT LINE
*
READ    N           =       1                                               9
        LINE        =                                                       10
        CARD        =       TRIM(INPUT) ' '         :F(END)                 11
        BADCARD     =       CARD                                            12
        CARD        POS(0)  SPAN(' ')       =                               13
LOOP    CARD        PAT     =                       :F(BAD)                 14
        BLANKS  GE(M<N>,SIZE(FIELD))  LEN(M<N> - SIZE(FIELD)) . BL          15
                                                    :F(BAD)                 15
        LINE        =       LINE  BL  FIELD                                 16
        N           =       LT(N,L)  N + 1          :S(LOOP)                17
        IDENT(CARD)                                 :F(BAD)                 18
        OUTPUT      =       LINE                    :(READ)                 19
```

MTS-560-0

12-1-67

```
*
*       FLAG AND PRINT ORIGINAL OF BAD LINES
*
BAD     OUTPUT    =     '******    ' BADCARD  '    *******'                20
.                                                      :(READ)              20
END                                                                         21
```

SUCCESSFUL COMPILATION

| | DECIMAL | EXP | HEX | |
|---|---|---|---|---|
| | 1 | 2**0 | 1 | |
| | 2 | 2**1 | 2 | |
| | 4 | 2**2 | 4 | |
| | 8 | 2**3 | 8 | |
| | 16 | 2**4 | 10 | |
| | 32 | 2**5 | 20 | |
| | 64 | 2**6 | 40 | |
| | 128 | 2**7 | 80 | |
| | 256 | 2**8 | 100 | |
| | 512 | 2**9 | 200 | |
| | 1024 | 2**10 | 400 | |
| | 2048 | 2**11 | 800 | |
| | 4096 | 2**12 | 1000 | |
| | 8192 | 2**13 | 2000 | |
| | 16384 | 2**14 | 4000 | |
| | 32768 | 2**15 | 8000 | |
| | 65536 | 2**16 | 10000 | |
| | 131072 | 2**17 | 20000 | |
| ****** | 2621442**18 | 40000 | ****** | |
| ****** | 524288 | 2**19 80000 / | ****** | |

MTS-560-0

12-1-67

## SAMPLE PROGRAM 7: 'THE TOWERS OF HANOI'

```
SNOBOL4  (PRELIMINARY VERSION, 9.18.67)

*
*                    THE TOWERS OF HANOI
*
*                      ||                    ||                    ||
*                    _||_                    ||                    ||
*                   |_||_|                   ||                    ||
*                  ___||___                  ||                    ||
*                 |___||___|                 ||                    ||
*                _____||_____                ||                    ||
*               |_____||_____|               ||                    ||
*              _______||_______              ||                    ||
*             |_______||_______|             ||                    ||
*            _________||_________            ||                    ||
*           |_________||_________|           ||                    ||
*
*                POLE A                POLE B                POLE C
*
*
*     THIS EXAMPLE ILLUSTRATES THE USE (AND POWER) OF RECURSIVE
*     PROGRAMMING.
*
*     'THE TOWERS OF HANOI' IS AN ANCIENT GAME CONSISTING OF 3 POLES AND
*     A NUMBER OF DIFFERENT SIZED RINGS, AS INDICATED ABOVE.  THE OBJECT
*     OF THE GAME IS TO MOVE THE RINGS FROM POLE A TO POLE C SUCH THAT
*        (1) ONLY ONE RING IS MOVED AT A TIME AND
*        (2) A LARGER RING IS NEVER ALLOWED TO LAY UPON A SMALLER RING.
*          IN THE PROGRAM BELOW, A FUNCTION IS DEFINED WHICH MOVES N
*     RINGS FROM POLE P1 TO POLE P2 USING POLE P3 AS TEMPORARY STORAGE.
*     THE FUNCTION CAN BE DEFINED IN TERMS OF MOVING N-1 RINGS BETWEEN
*     APPROPRIATE POLES AND A SINGLE MOVE OF THE NTH RING.  FINALLY, THE
*     MAIN ROUTINE CONSISTS MERELY OF A SINGLE CALL TO THE FUNCTION
*     ALREADY DEFINED.
*
*
        DEFINE('HANOI(N,P1,P2,P3)')                          :(HANOI.END)     1
HANOI                                                                         2
        EQ(N,0)                                              :S(RETURN)       3
        HANOI(N - 1,P1,P3,P2)                                                 4
        OUTPUT  =  'MOVE RING ' N ' FROM ' P1 ' TO ' P2                       5
        HANOI(N - 1,P3,P2,P1)                                                 6
                                                             :(RETURN)        7
HANOI.END                                                                     8
        HANOI(5,'POLE A','POLE C','POLE B')                                   9
END                                                                          10

SUCCESSFUL COMPILATION
```

MTS-560-0

12-1-67

```
MOVE RING 1 FROM POLE A TO POLE C
MOVE RING 2 FROM POLE A TO POLE B
MOVE RING 1 FROM POLE C TO POLE B
MOVE RING 3 FROM POLE A TO POLE C
MOVE RING 1 FROM POLE B TO POLE A
MOVE RING 2 FROM POLE B TO POLE C
MOVE RING 1 FROM POLE A TO POLE C
MOVE RING 4 FROM POLE A TO POLE B
MOVE RING 1 FROM POLE C TO POLE B
MOVE RING 2 FROM POLE C TO POLE A
MOVE RING 1 FROM POLE B TO POLE A
MOVE RING 3 FROM POLE C TO POLE B
MOVE RING 1 FROM POLE A TO POLE C
MOVE RING 2 FROM POLE A TO POLE B
MOVE RING 1 FROM POLE C TO POLE B
MOVE RING 5 FROM POLE A TO POLE C
MOVE RING 1 FROM POLE B TO POLE A
MOVE RING 2 FROM POLE B TO POLE C
MOVE RING 1 FROM POLE A TO POLE C
MOVE RING 3 FROM POLE B TO POLE A
MOVE RING 1 FROM POLE C TO POLE B
MOVE RING 2 FROM POLE C TO POLE A
MOVE RING 1 FROM POLE B TO POLE A
MOVE RING 4 FROM POLE B TO POLE C
MOVE RING 1 FROM POLE A TO POLE C
MOVE RING 2 FROM POLE A TO POLE B
MOVE RING 1 FROM POLE C TO POLE B
MOVE RING 3 FROM POLE A TO POLE C
MOVE RING 1 FROM POLE B TO POLE A
MOVE RING 2 FROM POLE B TO POLE C
MOVE RING 1 FROM POLE A TO POLE C
```

MTS-560-0

12-1-67

## SAMPLE PROGRAM 8: THEOREM PROVING

SNOBOL4 (PRELIMINARY VERSION, 9.18.67)

```
*           THIS PROGRAM IS THE ALGORITHM BY HAO WANG (CF. 'TOWARD
*         MECHANICAL MATHEMATICS', IBM JOURNAL OF RESEARCH AND
*         DEVELOPMENT 4(1) JAN 1960 PP.2-22.)  FOR A PROOF-DECISION
*         PROCEDURE FOR THE PROPOSITIONAL CALCULUS.  IT PRINTS OUT A
*         PROOF OR DISPROOF ACCORDING AS A GIVEN FORMULA IS A THEOREM
*         OR NOT.  THE ALGORITHM USES SEQUENTS WHICH CONSIST OF TWO
*         LISTS OF FORMULAS SEPARATED BY AN ARROW (--*).  INITIALLY, FOR
*         A GIVEN FORMULA F THE SEQUENT
*
*                   --* F
*
*         IS FORMED.  WANG HAS DEFINED RULES FOR SIMPLIFYING A FORMULA
*         IN A SEQUENT BY REMOVING THE MAIN CONNECTIVE AND THEN
*         GENERATING A NEW SEQUENT OR SEQUENTS.  THERE IS A TERMINAL
*         TEST FOR A SEQUENT CONSISTING OF ONLY ATOMIC FORMULAS:
*
*              A SEQUENT CONSISTING OF ONLY ATOMIC FORMULAS IS VALID IF
*              THE TWO LISTS OF FORMULAS HAVE A FORMULA IN COMMON.
*
*         BY REPEATED APPLICATION OF THE RULES, ONE IS LED TO A SET OF
*         SEQUENTS CONSISTING OF ATOMIC FORMULAS.  IF EACH ONE OF THESE
*         SEQUENTS IS VALID THEN SO IS THE ORIGINAL FORMULA.
*
*
*
        UNOP      =  'NOT'                                              1
        BINOP     =  'AND' | 'IMP' | 'OR' | 'EQU'                       2
        FORMULA   =  ' ' UNOP . OP '(' BAL . PHI ')'  |                 3
.                    ' ' BINOP . OP '(' BAL . PHI ',' BAL . PSI ')'     3
        ATOM      =  ( ' ' BAL ' ' ) . A                                4
*
        DEFINE('WANG(ANTE,CONSEQ)PHI,PSI')                             5
*
READ    EXP       =  TRIM(INPUT)                  :F(END)               6
        OUTPUT    =                                                     7
        OUTPUT    =  'FORMULA: ' EXP                                    8
        OUTPUT    =                                                     9
*
        WANG(,' ' EXP)                            :F(INVALID)           10
        OUTPUT    =  'VALID'                      :(READ)               11
INVALID OUTPUT    =  'NOT VALID'                  :(READ)               12
*
WANG    OUTPUT    =  ANTE ' --* ' CONSEQ                                13
        ANTE      FORMULA  =                      :S( $('A' OP) )       14
        CONSEQ    FORMULA  =                      :S( $('C' OP) )       15
        ANTE      =  ANTE  ' '                                          16
```

```
        CONSEQ    =  ' '  CONSEQ  ' '                                    1
TEST    ANTE     ATOM  =  ' '                  :F(FRETURN)              18
        CONSEQ    A                              :S(RETURN)F(TEST)        1
*
*
ANOT    WANG(ANTE,CONSEQ ' ' PHI)               :S(RETURN)F(FRETURN)     2
*
AAND    WANG(ANTE ' ' PHI ' ' PSI,CONSEQ)       :S(RETURN)F(FRETURN)     2
*
AOR     WANG(ANTE ' ' PHI,CONSEQ)               :F(FRETURN)              2
        WANG(ANTE ' ' PSI,CONSEQ)               :S(RETURN)F(FRETURN)     2
*
*
*
*
*
AIMP    WANG(ANTE ' ' PSI,CONSEQ)               :F(FRETURN)              2
        WANG(ANTE,CONSEQ ' ' PHI)               :S(RETURN)F(FRETURN)     2
*
AEQU    WANG(ANTE ' ' PHI ' ' PSI,CONSEQ)       :F(FRETURN)              2
        WANG(ANTE,CONSEQ ' ' PHI ' ' PSI)       :S(RETURN)F(FRETURN)     2
*
CNOT    WANG(ANTE ' ' PHI,CONSEQ)               :S(RETURN)F(FRETURN)     2
*
CAND    WANG(ANTE,CONSEQ ' ' PHI)               :F(FRETURN)              2
        WANG(ANTE,CONSEQ ' ' PSI)               :S(RETURN)F(FRETURN)     3
*
COR     WANG(ANTE,CONSEQ ' ' PHI ' ' PSI)       :S(RETURN)F(FRETURN)     3
*
CIMP    WANG(ANTE  ' ' PHI,CONSEQ ' ' PSI)      :S(RETURN)F(FRETURN)     3
*
CEQU    WANG(ANTE ' ' PHI,CONSEQ ' ' PSI)       :F(FRETURN)              3
        WANG(ANTE ' ' PSI,CONSEQ ' ' PHI)       :S(RETURN)F(FRETURN)     3
END                                                                      3
```

SUCCESSFUL COMPILATION

```
MTS-560-0

12-1-67



FORMULA: IMP(NOT(OR(P,Q)),NOT(P))

 --*  IMP(NOT(OR(P,Q)),NOT(P))
 NOT(OR(P,Q)) --*  NOT(P)
 --*  NOT(P) OR(P,Q)
 P --*  OR(P,Q)
 P --*  P Q
VALID

FORMULA: NOT(IMP(NOT(OR(P,Q)),NOT(P)))

 --*  NOT(IMP(NOT(OR(P,Q)),NOT(P)))
 IMP(NOT(OR(P,Q)),NOT(P)) --*
 NOT(P) --*
 --*  P
NOT VALID

FORMULA: IMP(AND(NOT(P),NOT(Q)),EQU(P,Q))

 --*  IMP(AND(NOT(P),NOT(Q)),EQU(P,Q))
 AND(NOT(P),NOT(Q)) --*  EQU(P,Q)
 NOT(P) NOT(Q) --*  EQU(P,Q)
 NOT(Q) --*  EQU(P,Q) P
 --*  EQU(P,Q) P Q
 P --*  P Q Q
 Q --*  P Q P
VALID

FORMULA: IMP(IMP(OR(P,Q),OR(P,R)),OR(P,IMP(Q,R)))

 --*  IMP(IMP(OR(P,Q),OR(P,R)),OR(P,IMP(Q,R)))
 IMP(OR(P,Q),OR(P,R)) --*  OR(P,IMP(Q,R))
 OR(P,R) --*  OR(P,IMP(Q,R))
 P --*  OR(P,IMP(Q,R))
 P --*  P IMP(Q,R)
 P Q --*  P R
 R --*  OR(P,IMP(Q,R))
 R --*  P IMP(Q,R)
 R Q --*  P R
 --*  OR(P,IMP(Q,R)) OR(P,Q)
 --*  OR(P,Q) P IMP(Q,R)
 --*  P IMP(Q,R) P Q
 Q --*  P P Q R
VALID
```

## SAMPLE PROGRAM 9: MAGIC SQUARE GENERATION

SNOBOL4 (PRELIMINARY VERSION, 9.18.67)

```
*
*       THIS PROGRAM GENERATES A MAGIC SQUARE OF ODD ORDER.  THE SIZE
*       OF THE SQUARE IS READ IN AS N. FOR DETAILS OF THE ALGORITHM SEE
*       JACM, AUGUST 1962, ALGOTITHM 118.
*
*       READ IN SIZE OF SQUARE AND DEFINE ARRAY.
*
        N = TRIM(INPUT)
        MAGIC = ARRAY(N ',' N)
*
*       I AND J ARE ROW AND COLUMN COORDINATES.  K IS THE NUMBER CUR-
*       RENTLY BEING PLACED INTO THE SQUARE.  LIM IS THE UPPER BOUND
*       ON K.
*
        I = (N + 1) / 2
        J = N
        LIM = N * N
        K = 1
*
*       MAIN PROGRAM LOOP WHICH MAKES A SINGLE ENTRY INTO THE ARRAY.
*
KLOOP   IDENT(MAGIC<I,J>,NULL)                          :S(ASSIGN)
        I = I - 1
        J = J - 2
        I = LE(I,0) I + N                                                1
        J = LE(J,0) J + N                                                1
ASSIGN  MAGIC<I,J> = K                                                   1
        I = I + 1                                                        1
        I = GT(I,N) I - N                                                1
        J = J + 1                                                        1
        J = GT(J,N) J - N                                                1
        K = LT(K,LIM) K + 1                             :S(KLOOP)        1
*
*       OUTPUT ROUTINE.
*
        OUTPUT = 'MAGIC SQUARE OF SIZE  ' N                              1
        OUTPUT =                                                         1
        I = 1                                                            2
OUT     ROW =                                                            2
OUT1    J = 1                                                            2
OUT2    '       ' LEN(SIZE(MAGIC<I,J>)) RTAB(0) . REST                   2
        ROW = ROW REST MAGIC<I,J>                                        2
        J = LT(J,N) J + 1                               :S(OUT2)         2
        OUTPUT =                                                         2
        OUTPUT = ROW                                                     2
        I = LT(I,N) I + 1                               :S(OUT)          2
```

```
 MTS-560-0

 12-1-67

END                                                              29
SUCCESSFUL COMPILATION
```

MTS-560-0

12-1-67

MAGIC SQUARE OF SIZE 5

| | | | | |
|---|---|---|---|---|
| 11 | 10 | 4 | 23 | 17 |
| 18 | 12 | 6 | 5 | 24 |
| 25 | 19 | 13 | 7 | 1 |
| 2 | 21 | 20 | 14 | 8 |
| 9 | 3 | 22 | 16 | 15 |

MTS-560-0

12-1-67

## SAMPLE PROGRAM 10: REGULAR EXPRESSION RECOGNITION

SNOBOL4 (PRELIMINARY VERSION, 9.18.67)

```
*           THE FOLLOWING PROGRAM DETERMINES WHETHER A GIVEN STRING IS A
*       MEMBER OF A SPECIFIED REGULAR SET OF STRINGS.  SINCE LAMBDA IS
*       UNAVAILABLE TO DENOTE THE NULL STRING, () CAN BE USED.
*
*               THE PROGRAM BEGINS BY DEFINING FOUR PATTERNS:  SUM,
*       ARBNO1, ARBNO2, AND TERM.  AT NEXTSET, A REGULAR EXPRESSION IS
*       READ IN AND PRINTED.  NEXT, KLEENE IS CALLED TO CONVERT THE
*       REGULAR EXPRESSION INTO A SNOBOL4 PATTERN USING THE FOUR PRE-
*       VIOUSLY CONSTRUCTED PATTERNS.  FINALLY, THE PATTERN RETURNED AS
*       THE VALUE OF KLEENE IS USED TO TEST IF SENTENCES ARE VALID.
*
*
        &ANCHOR   =   'ON'                                              1
        DEFINE('KLEENE(SPEC)EXP1,EXP2,RUN,REP')                        2
*
*           DEFINE BASIC PATTERNS USED IN THE KLEENE FUNCTION
*
        SUM     =  BAL . EXP1  ' V '  RTAB(0) . EXP2                    3
        ARBNO1 = ARB . RUN (LEN(1) . REP '*' | '(' ABORT)               4
        ARBNO2 = ARB . RUN '(' FENCE BAL . REP ')*'                     5
        TERM = ARB . RUN '(' FENCE BAL . REP ')'                        6
*
*           READ IN THE SPECIFICATION OF THE REGULAR SET.
*
NEXTSET   SETSPEC   =  TRIM(INPUT)                      :F(END)         7
        OUTPUT  =                                                       8
        OUTPUT  =  '     SET: '  SETSPEC                               9
*
*           CONSTRUCT A PATTERN CORRESPONDING TO THE SET.
*
        SET     =  KLEENE(SETSPEC)  RPOS(0)                             10
*
*           READ IN STRINGS AND TEST THEM.
*
TEST    TEST = TRIM(INPUT)                              :F(END)         11
        IDENT(TEST)                                     :S(NEXTSET)     12
        TEST    SET                                     :F(FAIL)        13
        OUTPUT  =  TEST  '  IS A MEMBER.'               :(TEST)         14
FAIL    OUTPUT  =  TEST  '  IS NOT A MEMBER.'           :(TEST)         15
*
KLEENE  SPEC    SUM                                     :F(K1)          16
        KLEENE  =  KLEENE(EXP1)  |  KLEENE(EXP2)        :(RETURN)       17
K1      SPEC ARBNO1 =                                   :F(K2)          18
        KLEENE = KLEENE RUN ARBNO(REP)                  :(K1)           19
K2      SPEC ARBNO2 =                                   :F(K3)          20
        KLEENE = KLEENE RUN ARBNO(KLEENE(REP))          :(K1)           21
```

```
K3      SPEC TERM =                                    :F(K4)          2
        KLEENE = KLEENE RUN KLEENE(REP)                :(K1)           2
K4      SPEC = IDENT(SPEC,'()')                                        2
        KLEENE = KLEENE SPEC                           :(RETURN)       2
END                                                                    2

SUCCESSFUL COMPILATION
```

MTS-560-0

12-1-67

```
     SET: (01 V 10)(00 V 11)*01
0101  IS A MEMBER.
1001  IS A MEMBER.
01000001  IS A MEMBER.
01110001  IS A MEMBER.
10001101  IS A MEMBER.
101001  IS NOT A MEMBER.
0001  IS NOT A MEMBER.
011100  IS NOT A MEMBER.

     SET: 1*(0 V 000)(10 V 11)*00 V 0101(111)*0
1000  IS A MEMBER.
00000  IS A MEMBER.
1010111000  IS A MEMBER.
01010  IS A MEMBER.
01011111110  IS A MEMBER.
0101111110  IS NOT A MEMBER.
01101  IS NOT A MEMBER.

     SET: (() V 11)(() V 00)1
1  IS A MEMBER.
111  IS A MEMBER.
001  IS A MEMBER.
11001  IS A MEMBER.
101  IS NOT A MEMBER.

     SET: (() V 11)*1
1  IS A MEMBER.
111  IS A MEMBER.
11111  IS A MEMBER.
10  IS NOT A MEMBER.
111111  IS NOT A MEMBER.
```

## SAMPLE PROGRAM 11: PHRASE STRUCTURE GRAMMAR RECOGNITION

SNOBOL4 (PRELIMINARY VERSION, 9.18.67)

```
*
*       THIS PROGRAM RECOGNIZES CONTEXT FREE PHRASE STRUCTURE GRAMMARS
*
*               GRAMMARS ARE SPECIFIED AS SHOWN IN THE OUTPUT.  A CARD
*       WITH 'EOF' TERMINATES THE GRAMMAR.  THE NEXT INPUT SPECIFIES THE
*       SYNTACTIC TYPE WHICH IS TO BE RECOGNIZED.  NEXT ARE THE SENTEN-
*       CES TO BE TESTED, ONE TO A CARD.  IF MORE GRAMMARS ARE TO BE
*       TRIED, AN 'EOF' SEPARATES THE SENTENCES FROM THE NEXT GRAMMAR.
*
        DEFINE('XLATE(S)TYPE','XLATE')                                  1
*
*               DEFINITION OF CONSTANTS AND BASIC PATTERNS
*
        RP = ')'                                                        2
        LP = '('                                                        3
        SL = '/'                                                        4
        ALTPAT = ARB . LIT '(' BAL . TYPE ')'                           5
        CLAUSEPAT = ARB . CLAUSE '/'                                    6
        TYPEPAT = '(' BAL . TYPE ')='                                   7
*
*               CONSTRUCTION OF PATTERNS FOR SYNTACTIC TYPES            8
*
READ    OUTPUT = 'GRAMMAR:'                                             9
READ1   CARD = TRIM(INPUT)                          :F(END)            10
        IDENT(CARD,'EOF')                           :S(RECOG)          11
        OUTPUT = CARD                                                  12
        CARD TYPEPAT =                              :F(ERR)            13
        $TYPE = FAIL                                                   14
NEXTC   CARD CLAUSEPAT =                            :F(ENDC)           15
        $TYPE = $TYPE | XLATE(CLAUSE)               :(NEXTC)           16
ENDC    $TYPE = $TYPE | XLATE(CARD)                 :(READ1)           17
*
*               XLATE FUNCTION TO CONSTRUCT PATTERN FOR A CLAUSE
*
XLATE                                                                  18
XL1     S ALTPAT =                                  :F(XL3)            19
        DIFFER(LIT)                                 :S(XL2)            20
        XLATE = XLATE *$TYPE                        :(XL1)             21
XL2     XLATE = XLATE LIT *$TYPE                    :(XL1)             22
XL3     IDENT(S)                                    :S(RETURN)         23
        XLATE = XLATE S                             :(RETURN)          24
*
*               RECOGNIZER TO READ AND TEST SENTENCES
*
RECOG   TYPE = TRIM(INPUT)                          :F(ERR)            25
        OUTPUT =                                                       26
```

MTS-560-0

12-1-67

```
        OUTPUT = 'TEST FOR SENTENCES OF TYPE ' TYPE                          27
        OUTPUT =                                                             28
        PAT = POS(0) $TYPE RPOS(0)                                           29
RCG1    CARD = TRIM(INPUT)                              :F(END)              30
        IDENT(CARD,'EOF')                               :S(RCG4)             31
        CARD PAT                                        :S(RCG2)F(RCG3)      32
RCG2    OUTPUT = CARD ' IS OF TYPE ' TYPE               :(RCG1)              33
RCG3    OUTPUT = CARD ' IS NOT OF TYPE ' TYPE           :(RCG1)              34
RCG4    OUTPUT =                                                             35
        OUTPUT =                                        :(READ)              36
END                                                                          37

SUCCESSFUL COMPILATION
```

```
GRAMMAR:
(A)=(B)/(C)
(C)=A/A(C)
(B)=A/AB(C)

TEST FOR SENTENCES OF TYPE A

A IS OF TYPE A
ABA IS OF TYPE A
ABAA IS OF TYPE A
ABAAA IS OF TYPE A
ABBA IS NOT OF TYPE A
AB IS NOT OF TYPE A


GRAMMAR:
(S)=(S)A/C
(T)=(S)T

TEST FOR SENTENCES OF TYPE T

CT IS OF TYPE T
CAT IS OF TYPE T
CAAT IS OF TYPE T
CAAAT IS OF TYPE T
CCT IS NOT OF TYPE T
CATT IS NOT OF TYPE T


GRAMMAR:
(IDENT)=X/Y/Z
(AREX)=(ADOP)(TERM)/(TERM)/(AREX)(ADOP)(TERM)
(TERM)=(FACTOR)/(TERM)(MULOP)(FACTOR)
(FACTOR)=(IDENT)/(LP)(AREX)(RP)
(ADOP)=+/-
(MULOP)=*/(SL)

TEST FOR SENTENCES OF TYPE AREX

X+Y*(Z+X)  IS OF TYPE AREX
X+Y+Z IS OF TYPE AREX
XY(Z+X)  IS NOT OF TYPE AREX
```

MTS-560-0

12-1-67

## SAMPLE PROGRAM 12: TREE FUNCTIONS

```
SNOBOL4 (PRELIMINARY VERSION, 9.18.67)

*
*       THIS PROGRAM IS DESIGNED TO ILLUSTRATE THE USE OF PROGRAMMER-
*       DEFINED DATA TYPES TO ADD TREE FUNCTIONS TO THE SNOBOL4 LANGUAGE.
*       THE FUNCTIONS GIVEN HERE ARE A SUBSET OF THOSE DESCRIBED IN
*       'TREE FUNCTIONS FOR SNOBOL3'.  THE FUNCTIONS INCLUDED WERE
*       CHOSEN TO INDICATE HOW TYPICAL FUNCTIONS MIGHT BE PROGRAMMED,
*       AND NOT ALL ARE USED IN THE EXAMPLE GIVEN.
*
*       THIS PROGRAM CONSISTS OF THREE PARTS:
*
*               1.  THE DEFINITION AND CODING OF THE TREE FUNCTIONS.
*
*               2.  THE CODING OF A FUNCTION TO CONVERT ALGEBRAIC
*               EXPRESSION INTO TREES.
*
*               3.  A TEST PROGRAM WHICH CONVERTS SAMPLE ALGEBRAIC
*               EXPRESSIONS INTO CANONICAL FORM.
*
PART1   &ANCHOR   =   'ON'                                              1
*
*       PART 1:  DEFINITION OF A NODE AND THE CODING OF TREE FUNCTIONS
*       WHICH MANIPULATE NODES TO FORM TREES AND OPERATE ON THESE TREES.
*
*       (THE VALUE OF A NODE MAY BE SET BY USE OF THE 'VALUE' FIELD
*       DEFINED ON A NODE, RATHER THAN BY USING INDIRECTNESS AS REQUIRED
*       BY THE SNOBOL3 TREE FUNCTIONS.)
*
        DATA('NODE(VALUE,L,R,F)')                                       2
*
        DEFINE('ADDSIB(NODE1,NODE2)')                                   3
        DEFINE('ADDSON(NODE1,NODE2)')                                   4
        DEFINE('FATHER(NODE)')                                          5
        DEFINE('LCTR(NODE)')                                            6
        DEFINE('LSIB(NODE)')                                            7
        DEFINE('LSON(NODE)')                                            8
        DEFINE('LTREE(CTR,FATHER)Y,Z')                                  9
        DEFINE('NXNODE(NODE)')                                         10
        DEFINE('OHUNT(NODE,STRING)')                                   11
        DEFINE('PRUNE(NODE)X')                                         12
        DEFINE('RSIB(NODE)')                                           13
        DEFINE('TCOPY(NODE,FATHER)')                                   14
                                                  :(PART2)             15
*
ADDSIB  PRUNE(NODE1)                                                   16
        R(NODE1)    =    R(NODE2)                                      17
        R(NODE2)  = NODE1                                              18
```

```
        F(NODE1)    =    F(NODE2)                    :(RETURN)                  19
*
ADDSON  PRUNE(NODE1)                                                            20
        R(NODE1)    =    L(NODE2)                                               21
        F(NODE1)  = NODE2                                                       22
        L(NODE2)  = NODE1                           :(RETURN)                  23
*
FATHER  FATHER = F(NODE)                                                        24
        DIFFER(FATHER)                              :S(RETURN)F(FRETURN)       25
*
*
*
*
*
LCTR    LCTR = VALUE(NODE)                                                      26
        LCTR = LCTR '(' LCTR(LSON(NODE))  ')'                                   27
        LCTR = LCTR ',' LCTR(RSIB(NODE))            :(RETURN)                  28
*
LSIB    LSIB    =    L(FATHER(NODE))                :F(FRETURN)                29
        IDENT(LSIB,NODE)                            :S(FRETURN)                30
LSIB1   IDENT(R(LSIB),NODE)                         :S(RETURN)                 31
        LSIB    =    R(LSIB)                        :S(LSIB1)                  32
*
LSON    LSON = L(NODE)                                                          33
        DIFFER(LSON)                                :S(RETURN)F(FRETURN)       34
*
LTREE   CTR    BAL . CTR ',' RTAB(0) . Y                                        35
        CTR    ARB . CTR '(' BAL . Z ')'                                        36
        LTREE = NODE(CTR,,,FATHER)                                              37
        L(LTREE) = LTREE(DIFFER(Z)  Z,LTREE)                                    38
        R(LTREE) = LTREE(DIFFER(Y)  Y,FATHER)       :(RETURN)                  39
*
NXNODE  NXNODE    =    LSON(NODE)                   :S(RETURN)                 40
NX1     NXNODE    =    RSIB(NODE)                   :S(RETURN)                 41
        NODE    =    FATHER(NODE)                   :S(NX1)F(FRETURN)          42
*
OHUNT   OHUNT    =    LSON(NODE)                    :F(FRETURN)                43
OHUNT1  IDENT(VALUE(OHUNT),STRING)                  :S(RETURN)                 44
        OHUNT    =    RSIB(OHUNT)                   :S(OHUNT1)F(FRETURN)       45
*
PRUNE   X    =    LSIB(NODE)                        :S(PRUNE1)                 46
        L(FATHER(NODE))    =    R(NODE)                                         47
PRUNE2  F(NODE)    =                                                            48
        R(NODE)    =                                :(RETURN)                  49
PRUNE1  R(X)    =    R(NODE)                        :(PRUNE2)                  50
*
RSIB    RSIB = R(NODE)                                                          51
        DIFFER(RSIB)                                :S(RETURN)F(FRETURN)       52
*
TCOPY   TCOPY = NODE(VALUE(NODE),,,FATHER)                                      53
        L(TCOPY) = TCOPY(LSON(NODE),TCOPY)                                      54
        R(TCOPY) = TCOPY(RSIB(NODE),FATHER)         :(RETURN)                  55
*
```

```
PART2                                                                   56
*
*       PART 2:  FUNCTION WHICH CONVERTS FULLY-PARENTHESIZED ALGEBRAIC
*                EXPRESSIONS INTO TREES.  SUCH A TREE REPRESENTATION
*                IS USEFUL IN CODE OPTIMIZATION.  THE RESTRICTION THAT
*                THE EXPRESSIONS BE FULLY PARENTHESIZED WAS INCLUDED
*                TO SIMPLIFY THE PROGRAM SINCE THE PURPOSE WAS TO ILLUS-
*                THE TREE FUNCTIONS.
*
        OP    =    '+' | '-' | '**' | '*' | '/'                        57
        DEFINE('TREE(EXP)T1,T2,E1,E2,O')                               58
                                                  :(PART3)             59
*
TREE    EXP    '(' BAL . EXP ')' RPOS(0)          :S(TREE)             60
        EXP    ('(' BAL . E1 ')' | BAL . E1) OP . O RTAB(0) . E2       61
                                                  :F(TREE1)            61
        TREE    =   LTREE(O)                                           62
        T1     =   TREE(E1)                                            63
        T2     =   TREE(E2)                                            64
        ADDSON(T1,TREE)                                                65
        ADDSIB(T2,T1)                             :(RETURN)            66
TREE1   TREE    =   LTREE(EXP)                    :(RETURN)            67
*
*
*
*
*
PART3                                                                  68
*
*       PART 3:  TEST PROGRAM WHICH READS IN ALGEBRAIC EXPRESSIONS,
*                CONVERTS THEM INTO TREES, AND PRINTS THE RESULT IN
*                CANONICAL FORM.
*
TEST    EXP    =   TRIM(INPUT)                    :F(END)              69
          OUTPUT   =   'EXPRESSION: ' EXP                              70
        T    =   TREE(EXP)                                             71
        OUTPUT    =   'CANONICAL FORM: ' LCTR(T)                       72
        OUTPUT    =                               :(TEST)              73
*
END                                                                    74
```

SUCCESSFUL COMPILATION

```
EXPRESSION: ((A+B)*C)/((D-E)*F)
CANONICAL FORM: /(*(+(A,B),C),*(-(D,E),F))

EXPRESSION: (A**2)+(C*(D**3))
CANONICAL FORM: +(**(A,2),*(C,**(D,3)))

EXPRESSION: (COUNT+(TOTAL-DELTA))*SCALE
CANONICAL FORM: *(+(COUNT,-(TOTAL,DELTA)),SCALE)
```

# APPENDIX D: TRACE FACILITY

There is in SNOBOL4 a very powerful and flexible tracing facility for debugging. A separate writeup on this is in progress at Bell Labs, Holmdel. Until this is available, the following brief description is offered:

## &FTRACE

To trace calls and returns of all functions, the value of &FTRACE should be made non-null. E.g.,

```
&FTRACE = "ON"
```

To turn this off, the value of &FTRACE should be made null. E.g.,

```
&FTRACE =
```

## &TRACE

This is an enabling switch, which if non-null, allows the tracing specified by the TRACE function to occur. This is provided so that all necessary calls to TRACE can occur once at the beginning of the program and then &TRACE can be used to switch things on and off.

```
TRACE(name,respect,tag,function)
TRACE(name,respect)
```

This function sets up an action to occur when something happens for a specified entity. name is the name of the entity being traced. respect designates in what respect it is being traced. The following are legal for respect:

| | |
|---|---|
| "CALL" | means trace function calls |
| "RETURN" | means trace returns from functions |
| "FUNCTION" | is the same as "CALL" plus "RETURN" |
| "VALUE" | means trace change of value |
| "LABEL" | means trace transfers to this label |
| "KEYWORD" | means trace this keyword |

The name and respect designate when tracing is to occur. At this time, if all four arguments to TRACE were specified, a call to function is made, giving name and tag as arguments. If the last two arguments were omitted, a call is made to a built-in function which prints out a trace event.

Example

Program doing the tracing:

```
        TRACE("ADD","FUNCTION")                                  1
        TRACE("START","LABEL")                                   2
        &TRACE = "ON"                                            3
        TRACE("STCNT","KEYWORD")                                 4
        DEFINE("ADD(A1,A2)")              :(START)               5
ADD     ADD = A1 + A2            :(RETURN)                       6
START                                                            7
        DO NOTHING                                               8
        OUTPUT = ADD(2,2)                                        9
END                                                              10
```

Output

```
  STATEMENT 5: &STCNT = 5
  STATEMENT 5: TRANSFER TO START
  STATEMENT 8: &STCNT = 6
  STATEMENT 9: &STCNT = 7
  STATEMENT 9: LEVEL 0 CALL OF ADD(2,2)
  STATEMENT 6: &STCNT = 8
  STATEMENT 6: LEVEL 0 RETURN OF ADD = '4'
4
```

MTS-570-0

12-1-67

# UMIST

## The University of Michigan Interpretive String Translator

Computing Center
Ann Arbor, Michigan
April, 1967

# PREFACE

The UMIST processor described in this publication was designed and written by Tom O'Brien and Tad Pinkerton. Its development at Michigan was motivated by its utility for text formatting, language preprocessing, and manipulation of data for graphical display purposes.

UMIST was made possible in part by support extended to the University by the Advanced Research Projects Agency of the Department of Defense (Contract #DA-49-083 OSA-3050, ARPA Order #716 administered through the Office of Research Administration, Ann Arbor).

Correspondence concerning UMIST and this section of the manual should be address to the author: Tad B. Pinkerton, Computing Center, University of Michigan, Ann Arbor, Michigan.

# CHAPTER I: INTRODUCTION

## TRAC

The TRAC language was developed by Calvin N. Mooers of the Rockford Research Institute, Cambridge, Massachusetts. It is a refinement and extension of an assembler macro language, and is designed specifically for use as an on-line text processor. A level of the TRAC language called "TRAC 64" is described in [1]. It is the basic standard and point of reference for this language. A good discussion of its design goals and principles is given in reference [2]. Much of the motivation for the development of TRAC came from the work of Eastwood and McIlroy [3,4] at Bell Laboratories. A system similar to TRAC which was developed independently in Great Britain is described by Strachey [5].

## UMIST

UMIST is a similar language available at the University of Michigan. It is written in System/360 Assembler Language and currently runs under the Michigan Terminal System (MTS) in UMMPS (University of Michigan Multi-Programming System). Its dependence on MTS is restricted to the use of input-output routines and the MTS file system for external storage. It is designed to be used with a wide range of I/O equipment, including unit record devices and those with upper/lower case capabilities.

UMIST can currently be described as follows in relation to the standard TRAC benchmarks: it includes the "Level Zero TRAC Language" as a subset, for any TRAC procedure using only the nine primitive functions PS, RS, DS, SS, CL, CC, CR, EQ, and DD will run in UMIST as specified in reference [1]. But UMIST does not contain "TRAC 64" as a subset, in the sense that some of the corresponding primitives (e.g. the arithmetic and Boolean functions) differ in detail in their actions, and some of the capabilities (e.g. external storage management) are provided in different ways by experimental primitives. UMIST also contains a number of extensions of the "TRAC 64" language beyond the addition of more primitive functions. For the most part, ignorance of these extended capabilities will do the UMIST user no harm. Since these facilities are new and largely unproven, they are subject to change as experience dictates.

## Guide to this Manual

This publication is intended to be a self-sufficient programming text for UMIST. Chapters II and III summarize the syntax and processing action of "TRAC 64" and the "Level Zero TRAC" primitive functions. This material is given in greater detail in reference [1]. Chapter IV explains the variations with which UMIST provides the full "TRAC 64" capabilities. Chapter V discusses the features new in UMIST.

Finally, a chapter is included which briefly explains the internal organization of the processor. An appendix is provided to guide the TRAC user in understanding the interactions of UMIST with MTS, and the necessary details of the commands in the latter. A complete list of primitive functions and their descriptions is given in Appendix B. Many of the primitives are not otherwise mentioned in the manual.

# CHAPTER II: THE UMIST PROCESSOR

## Mode of Operation

UMIST input and output each consist of single character strings. All processing of the input string is controlled by explicit calls, embedded in the input string, to functions whose arguments are substrings and whose values, possibly null strings, replace the function calls in the string being processed. For example, the input character string

```
The string ABC is #(LEN,ABC) characters long#(NL,X).
```

is changed, after processing by UMIST, to

```
The string ABC is 3 characters long.
```

There were two function calls in the input string, the first of which had the value 3, the second a null value (no characters).

There are two kinds of functions: primitives, or machine-language subroutines, that support the system in its environment and are the basis for constructing forms, or named procedures in UMIST storage, which are character strings written like macro definitions and are expanded, interpretively, when called. When writing a function call, one specifies whether its value (replacing the call) is to be processed again as part of the input string (active call), or that processing is to continue starting with the portion of the string to the right of the value returned (neutral call). A single processing cycle is completed when the scanning and evaluating process reaches the right-hand end of the string.

Sequencing and evaluation in UMIST are inherently recursive: function calls are evaluated from left to right but may be nested to any depth in the arguments of other calls. Each function call is evaluated when, and only when, all of its arguments have been completely processed. Thus the string being processed is divided logically into two parts: the active string, consisting of input text (possibly preceded by inserted functional values) which is yet to be scanned and evaluated, and the neutral string, containing the scanned and evaluated arguments of function calls which are not completely ready for evaluation. This mode of operation, based on the

completely interpretive execution of function calls, eliminates the distinction between program and data.

## Syntax

Each function call in UMIST has the form of a specially delimited argument list, in which the name of the function is always the first argument. Calls may be open (a variable number of arguments) or closed. A function call may be protected from evaluation by the use of literal delimiters. Another delimiter signals the right-hand end of the input string. These considerations lead to a syntax in which there are seven special symbols, whose occurrences are deleted from the string during syntax scanning and whose presence indicates the beginning or end of a substring. The character strings enclosed in brackets below are the UMIST special symbols:

1. Beginning of neutral function call [##(]
2. Beginning of active function call [#(]
3. End of argument [,]
4. End of call [)]
5. Beginning of literal [(]
6. End of literal [)]
7. End of input string [']

Note that the three beginning of substring symbols ##( and #( and ( are terminated by the occurrence of the same end of substring character, ). UMIST has a "parenthesis balanced" syntax, in the sense that an occurrence of the right parenthesis matches only the last previous occurrence of any one of the beginning of substring special symbols. Whenever a literal substring is encountered, the UMIST processor removes the enclosing parentheses, but only the outer set is removed if more than one matching pair occur. Thus a string initially protected from evaluation may be evaluated if scanned a second time, and in general evaluation can be controlled to occur the nth time the substring is scanned.

# CHAPTER III: UMIST PRIMITIVES

All of the primitives of "TRAC 64" have two-character names. These will be given in the sequel in capital letters. The complete descriptive name of the function will be enclosed in single quote, or apostrophe, marks. Since any function can be called both neutrally and actively, prototype calls in this manual will use the more common active notation, with the implicit understanding that neutral calls are also possible.

## Read String and Print String

The value of a 'read string' function call

```
#(RS)
```

is an input string accepted from the current input device. The 'print string' function

```
#(PS,X)
```

causes the display of the second argument, here symbolized by X, on the current output device, and has a null value.

When the UMIST processor is first given control, and at the end of every processing cycle, the idling procedure

```
##(PS,#(RS))
```

is automatically loaded as an input string. This procedure first causes a read from the input device, with the input string becoming the second argument of the 'print string' call. Thus the string, if any, remaining when the input string has been completely processed, is finally printed before the idling procedure is again loaded. for example, if the input string is

```
#(PS,ABC)'
```

then after the 'read string' has been evaluated the processor is scanning the string

```
#(PS,#(PS,ABC))
```

and the inner call produces the output ABC; the outer call nothing, since the inner 'print string' has a null value.

## Define, Call and Segment String

Any character string in UMIST can be given a name and placed in storage, from whence it can be called by using its name. The null-valued 'define string' function

```
#(DS,A,B)
```

places the string B in storage with the name A. A is called a form with value B. At most one string can be defined with a given name at any one time: use of the same name replaces a former definition. The value is retrieved with the 'call string' function

```
#(CL,A).
```

A form name, like a value, is any character string. The only restriction on length is that of the total string capacity of the processor.

The occurrence of strings in storage is deleted with the 'delete definition' function

```
#(DD,N1,N2,...).
```

This null-valued function removes the names N1,N2,... as forms and discards their values.

Once defined, a form can be "parameterized," or segmented, using the 'segment string' function:

```
#(SS,A,X1,X2,...).
```

This null-valued function scans the form A, searching for an occurrence of the string X1 as a substring. If X1 matches a part of A, that part is excluded from further matching, creating a "formal variable", or segment gap. The rest of the form is also compared with X1 to create, if possible, more segment gaps, all of which are assigned the ordinal value one, identifying the argument matched. The (separate) substrings of the form not already taken for segment gaps are next scanned with respect to the string X2, and any occurrences of the latter substring in A create segment gaps of ordinal value two, etc. The 'segment string' function may be called repeatedly for the same form, using differing arguments and resulting in the creation of additional gaps of ordinal values one, two, etc., among those already there. for example the sequence

```
#(DS,A,1234567890)
#(SS,A,23,6,1)
#(SS,A,0)
```

creates four segment gaps, two of which have ordinal value one, so that

#(CL,A)

produces the value

45789.

Thus the 'define string' and 'segment string' functions together create a "macro" in which the segment gaps locate the "formal parameters". The "macro" is expanded by supplying the "actual parameters" in a call on the 'call string' function mentioned above:

#(CL,A,Y1,Y2,...).

The value of the 'call' is generated by returning the form A with all the segment gaps of ordinal value 1,2,... replaced by Y1,Y2,... respectively. If extra arguments are given in a CL, they are ignored. If some are missing, null strings are used as their values. Thus using the above segmentation of the form A,

#(CL,A,X,Y,Z)

produces the value

ZX45Y789X.

## The Form Pointer

The 'call string' function, and other UMIST primitives whose values are generated by reading the text of a form, begin their value generation at a point in the form indicated by a form pointer. This mechanism attached to each form is initially set to the first character, and is moved forward (and back) by some of the more specialized "call functions" which read out part of a form. (The 'call string' function does not alter the position of the form pointer.)

The 'call character' function, for example, has as its value the character at the form pointer of the form named N

#(CC,N,Z).

Then the form pointer is moved ahead one character. Segment gaps are skipped: the value of CC is always a character not in one of the segment gaps. If the form pointer has reached the end of the form or the form is null, the value of the function is the third argument, Z. Since one may wish to call a procedure in this "failure" case, the alternative values for this type of function are always moved to the active string, i.e., treated as if the mode of the call had been active. The 'call restore' function,

#(CR,N),

restores the form pointer in the form named N to the first character.

## The Equal Function

A decision function is provided for character strings:

#(EQ,A,B,T,F).

If the string A is identical to the string B, then the value of this function is the third argument, T, otherwise the value is the fourth argument, F. Since the strings T and F may be any UMIST procedures, this primitive is the one normally used for branching.

## Other Language Features

Additional UMIST primitive functions and facilities are provided, primarily in the following areas:

1. Arithmetic functions.

   There are primitives for the four basic arithmetic operations on decimal integers, and a decision function for comparing signed decimal values.

2. Boolean functions.

   Boolean functions apply the logical operations and shifting to strings of binary digits, written in octal.

3. External storage functions.

   UMIST primitives exist to "fetch", "store" and "erase" groups of forms residing in secondary storage. These blocks have names which are treated like form names.

4. Diagnostic functions

   Using primitive functions, one may list the names of all his forms and print out the text of a form, complete with segment gap indications. A trace mode of operation may be invoked, in which the arguments of each function are displayed before its evaluation.

# CHAPTER IV: UMIST VARIATIONS

This chapter explains some of the ways in which UMIST functions differ from those "TRAC 64" functions with the same general purpose. "TRAC 64" primitives which are not implemented in UMIST cause an appropriate error comment if called.

## Input Functions

'Read character' and 'read N characters' functions are available in UMIST as well as 'read string.' Since an input device may, in an operating system outside of UMIST, respond to a read function with an independently determined string length, all input is buffered. A physical read occurs at the device when and only when the buffer does not contain enough characters to satisfy the UMIST read function. for example, if the procedure

```
#(RC)#(RS)#(RN,5)#(RC)
```

is initially supplied with the input string

```
XZBC'12345ZZZ
```

then only the single physical read is necessary, and after evaluating the procedure the string

```
ZZ
```

remains in the input buffer. The input buffer is cleared whenever UMIST is reinitialized.

## Arithmetic Functions

UMIST provides functions for both decimal and hexadecimal integer computations and comparisons. The decimal operands are assumed to contain only legal base ten digits, and the result may not exceed 16 digits, including the sign. Results have no leading zeros and are unsigned if positive. Hexadecimal arithmetic is unsigned base-complement (actually it is hex-coded binary two's-complement arithmetic), and the operands and result are no longer than 8 digits. (e.g. subtracting one from zero produces FFFFFFFF).

The decimal and hexadecimal decision functions include all of the standard relations, with a relation denoted by symbols constructed from the characters >, <, =, and /. E.g. the relation "greater than or equal to" is expressed either by >= or =>.

## Boolean Functions

Logical operations are carried out in UMIST on character strings consisting of T's (for true) and F's (for false). An unrecognizable value is assumed to be false. Operands may be of any length and need not, in the case of binary operations, be of the same length.

## External Storage Functions

There are no "fetch", "store" and "erase" block functions in UMIST, and in fact there is no UMIST management of external storage: the file system for data management is presupposed as a part of the operating system in which the UMIST processor resides, and appropriate functions have been added to UMIST to make use of external files. The 'parameter set' function, described in the next chapter, can be used to shift the input or output device for UMIST to another logical unit, which may be another typewriter, a unit-record device, or a data set in a secondary storage file. The 'print form' function produces an executable description of a form which may be used to reconstruct it later. Thus a rough equivalent of 'store block' is the following:

1. A call on PAR to shift the output designation to a specified file.
2. A call on PF to place descriptions of the desired forms in the file.
3. A (possible) call on PS to insert a PAR call at the end of the file to switch the input designation elsewhere after this file has been used as input.
4. A call on PAR to return the output device to its previous designation.

The analogue of 'fetch block' is simply

1. A call on PAR to shift the input designation to the desired file.

Once the input device is shifted, the output of PF stored in the file is executed, redefining the specified forms. When an end-of-file condition occurs at an input device, the designation is shifted to the original, standard device. If step 3 in the 'store block' sequence is used to specify a device change in the file input, more UMIST functions may be executed from another file before returning to the standard device, etc.

There is no equivalent of the "TRAC 64" 'erase block' function, since changes to files are a function of the operating system. However, since the files created by UMIST contain only character strings, they can be manipulated at any time outside of the UMIST system.

## Other Differences

Two significant differences in the implementation of UMIST affect the overall operation of the processor. First, the names of forms and primitive functions are stored in the same symbol table. This implies that it is not possible in UMIST to have both a form and a primitive function with the same name. (Other advantages of this approach outweigh this disadvantage--see the next chapter.)

Secondly, the action taken in the event of an error is usually to print an error comment and reinitialize, rather than to ignore the error condition. The user may choose, however, to override these actions and have only very serious errors reported. This is accomplished by setting a system parameter.

# CHAPTER V: UMIST EXTENSIONS

## Special Symbols

In UMIST there is an additional terminating special symbol:

8. End all parentheses [)...].

The effect of this symbol is to "balance the parenthesis count" at the place the symbol occurs, supplying enough right parentheses to match all unclosed left parentheses (except the one belonging to the PS call in the idling procedure, which is not typed by the user).

Each of the eight special symbols can be redefined to be any string of not more than four characters. (Appendix B. explains the scanning algorithm which "recognizes" the current special symbols). Redefinition is accomplished with the 'define special symbol' function,

#(DSS,S1,T1,S2,T2,...)

where each Si is a special symbol name from the following list,

| | |
|---|---|
| MNF | Begin neutral function |
| MAF | Begin active function |
| MARG | End argument |
| MEF | End function |
| MLL | Begin literal |
| MRL | End literal |
| MEP | End all parentheses |
| MES | End input string |

and the corresponding Ti is the
soon as the DSS call has been completely evaluated. Each of the Si above is also the name of a primitive function whose value is the corresponding current special symbol, e.g.

#(MAF)

produces the value

#(

if the active function symbol has not been redefined.

## Set Definition Function

Names of forms and primitives may be redefined in UMIST. The function

#(SET,A1,B1,A2,B2,...)

assigns the meaning of the name Bi to the name Ai, for each i. Ai may or may not have been previously defined: the definition is created if not, and altered by discarding the old definition and substituting the new one if so. Either or both names may have referred to primitives or forms. For example, one can redefine the primitive 'read string' to save the input string in a string called READ in addition to its normal function with the following procedure:

```
#(SET,*RS,RS)
#(DS,RS,(#(DS,READ,(#(*RS)...
```

Because of the possibility of "losing" a valuable primitive or form by unintentionally redefining it, the redefinition may in some cases be prevented (see the next section).

## Class Membership

Each name in UMIST is assigned to one or more of a fixed set of classes, used primarily for protection. Class assignments have been made in advance for primitive functions, and are made by default for forms when they are defined unless specific classes are assigned by the user. At any given time a subset of the set of classes has protection in force, in the sense that if a name A is a member of a protected class, a call on SET to redefine A will fail.

Membership in classes is also used to specify sets of names to be listed--see the LST function description.

## Parameter Setting

During its execution UMIST operates under the control of a number of global parameters, whose values can be changed by executing the 'parameter set' function. Each parameter has a one to four letter name and a variety of possible values. These parameters will be discussed in groups in the sequel according to the types of values they acquire. The call on the parameter setting function is of the form

#(PAR,N1,V1,N2,V2,...)

where each Ni is a parameter name and each Vi the corresponding new parameter value.

PROTECTION PARAMETERS

Parameters named PROT and STD are provided for setting class protection in force, and for default assignments of classes to new names, respectively. The values of PROT and STD are sequences of class names, separated by a single character, e.g. *. If a class name is given in a value sequence for PROT, then names belonging to that class are henceforth protected. Likewise, a class name appearing in the value sequence for STD causes subsequent default assignments of class membership to include membership in the specified class. For example, the execution of

```
#(PAR,PROT,US4*US2,STD,US3)
```

causes members of classes US4 and US2 to be the (only) names protected against redefinition, and all new names to be assigned membership in US3 (only) unless other class membership is specified. Names of the seven existing membership classes are given below. These names suggest predefined assumptions about the way in which the classes might be used, but the assignment of members to classes is completely unrestricted.

| | |
|---|---|
| BAS | Basic functions (RS,PS) |
| BLT | Built-in primitives |
| EXT | External functions |
| US1 | User class 1 forms |
| US2 | User class 2 forms |
| US3 | User class 3 forms |
| US4 | User class 4 forms |
| US5 | User class 5 forms |

The predefined setting of PROT is BAS*BLT, and the predefined value of STD is US1.

PARAMETER SWITCHES

Several modes of operation are controlled by global switches which may be turned on and off. These are the trace (TR), implicit call (CL), fold (FOLD), and translate (TRAN) switches. In each case, the given abbreviation is the switch name, and the value is either ON or OFF. If any other value is given the current setting of the switch is inverted, except that a null value is ignored.

(a) TR: Whenever the trace switch is on, the arguments of each function are displayed before its evaluation, and the input string reaching the UMIST processor is printed whenever an input function is called.

(b) CL: The meaning of the implicit call switch will be described in a later section of this manual.

(c) FOLD: Whenever the FOLD switch is on, all lower case alphabetic characters in the input string are converted to upper case before UMIST processing. When this switch is off, upper and lower case letters are treated as distinct characters.

(d) TRAN: If the TRAN switch is on, a uniform one-for-one replacement of characters in the input string is made according to previously specified substitutions: See the description of the 'translate' function, TRN.

SPECIAL CHARACTER PARAMETERS

Two (usually non-printing) character codes are preempted for internal use by the UMIST processor. They are denoted by STOP and IGNR. All other 254 printing and non-printing codes are legal symbols in UMIST strings. The STOP and IGNR characters may be set to any codes by giving one of the above names and the corresponding new value character in a PAR call. E.g.

```
#(PAR,STOP,*)
```

sets the * as the new stop character.

INTEGER PARAMETERS

Two UMIST parameters are given integer values.

Each error condition in UMIST has a predefined severity level, ranging from 0 to 99. When one of these error conditions is discovered, the processor either prints an error message and reinitializes or tries to recover and continue, depending on the severity code (SVCD). The error message and reinitialization occurs only if the severity level of the error is at least as great as the current value of SVCD. The predefined value of SVCD is zero.

The second integer parameter (LINE) determines the maximum length of a line printed by UMIST. Any printed string exceeding this length is put on more than one line. The predefined value of LINE is 72.

NAME PARAMETERS

The value of the file-or-device-in (FDI) and file-or-device-out (FDO) parameters is the name of a device or line file which is to be made the current input or output device, respectively. Whenever a new input or output device is specified, UMIST releases the previous device and attaches the new device.

## Implicit Calling and Call Procedure

It is permissible in some implementations of UMIST to have a form, rather than a primitive function, name as a first argument in a function call. For example, if A is a form,

```
#(A)
```

could be taken to be equivalent to the call

#(CL,A).

This feature is termed a "suppressed" or implicit call on the form, and i makes a call on a form look the same as a call on a primitive function Suppose, however, that one wished to simulate a UMIST primitive by writin a procedure in UMIST and defining a form whose value is that procedure E.g., suppose users of a small machine wished to write 'read string' as UMIST procedure which repeatedly used 'read character'. Then UMIST progra interchange is possible: with the implicit call feature any active call o a primitive function at one installation can be treated as an implicit cal on a form at another installation, assuming a form has been written to d the same thing. But the above statement is only true for active functio calls: A neutral function call on the primitive merely specifies that th value not be rescanned, whereas a neutral implicit call on the correspond ing form does not return the same value at all, but instead returns th definition of the procedure to obtain the value.

A new function in UMIST, referred to as 'call procedure', is useful i solving the above problem:

#(CP,A,Y1,Y2,...).

The form named A is expanded by the CP function just as in CL, bu regardless of the mode of the CP call, the expanded form A is placed on th active string. When that value has been completely rescanned and evaluat ed, its value is moved to the neutral or active string according to th mode of the CP function call. In effect, the neutral/active cal distinction is applied one level down from the point at which it wa specified.

Both the CL and CP functions exist in UMIST and may be calle explicitly. But the meaning of an implicit call depends on the setting o the CL parameter at the time the given form was defined: if CL is ON when form N is defined, then

#(N)

is equivalent to

#(CL,N).

Otherwise it is equivalent to

#(CP,N).

Note that in the latter case there is no observable difference between primitive function call and an implicit call on a form defined to do th same thing. The predefined value of the CL parameter is ON, and it is o course changed by

#(PAR,CL,OFF).

## External Functions

Public and private libraries of machine-coded functions may also be added to the repertoire of UMIST during execution. Such primitives are termed external functions, and they are added to the processor functions with the 'load external functions' function:

#(LEF,FDNAME)

This null-valued function loads external primitive functions from the file or device named as the second argument, and links them into the UMIST system. After the LEF function has been evaluated, the primitives defined in the load module at the given source are indistinguishable in their behavior from the built-in primitives.

## Status Recording

A programmer using UMIST can, through the use of facilities already discussed in this chapter, operate in a environment quite different from that of "TRAC 64". He may be using different special symbols, parameter settings other than the default specifications, input character translations, and external functions. Thus, in order to perform effectively with this "personal system", he ought to be able to

(a) discover the state of any such variable in the system at any time, and

(b) easily switch the processor to that state from the normal state when he returns to use the system the next time.

For this purpose a number of "status recording" functions are included in UMIST. They all produce values which

(1) describe the current state of system variables and

(2) are themselves executable to restore the system variables to the current state from some other state.

For example, the value produced by the 'print form' function is a complete description of one or more forms expressed as a call on the 'define form' function, which, if executed, would redefine the form to its current value. Another example is given by the special symbol functions MAF, MNF, etc.: writing #(MAF) causes the current beginning of active function special symbol to appear. After redefining special symbols the programmer can leave a procedure in a file to be executed when he signs on the next day, defining the special symbols he wishes to use. There are several other such status recording functions:

(a) The value of the 'translate print' function, TRP, is a call on the

'translate' function, TRN, to specify the current input characte translations.

(b) The value of the 'print parameter' function is a call on th 'parameter setting' function to specify the current values of all th parameters.

In addition, the 'list names' function, LST, has as its value the list o names which belong to certain of the "protection" classes:

#(LST,S,X)

The second argument, S, is a sequence of class names separated by th character *, denoting intersection, and/or the character +, denoting union. The subset of names whose class membership satisfies the given set-theoretic expression is listed as the value of this function, with eac name preceded by the third argument character string. For example, th value produced by

#(LST,BAS*US1+US4,...)

is the set of all names belonging either to both of the classes BAS an US1, or to the class US4, with each each name preceded by three periods.

# CHAPTER VI: INTERNAL STRUCTURE

Highlights of the UMIST implementation are mentioned here for those interested. The given design sacrifices storage whenever it is possible to reduce execution time.

## Pushdown Stack

A fixed, contiguous area of storage is reserved for use as a stack. The stack is used for

(a) argument identification

(b) general register storage

(c) temporary text storage

Each normal stack entry is a three-word block containing pointers to the beginning and end of an argument, and to the stack entry for the first argument of this call if the given argument is not the first. The stack entry for a first argument specifies the mode of the call (neutral or active). A stack entry is created for an argument as soon as its beginning is identified. Each function call is evaluated as soon as the stack entry for its last argument has been completed. After a function has been evaluated its entries are removed from the stack.

Whenever a machine language function or utility routine in the processor calls another, the general purpose registers used by the former are saved on the stack. If the called routine in turn calls another, the stack is pushed to cover the saved registers. It should be noted, however, that it is not necessary for the processor to work with recursive subroutine calls: use of the stack permits a calling depth of just one level. Arbitrarily nested subroutine calls were simply a convenience provided for the construction of the processor.

Finally, a single subroutine may use a piece of the stack to accumulate a value string, or save other temporary pointers, in lieu of having a separate temporary storage area. Such temporary storage is covered by pushing and popping the stack during subroutine calls along with the general register storage.

The stack structure just described allows each primitive function and subsidiary routine to be coded as an independent unit, which may be called by any other such unit. This organization also applies to the main syntax scanning and stack-building routine.

## Scanning Algorithm

The following procedure is used to locate special symbols in the activ string.

1. A search for a special symbol is undertaken whenever a character i found which is the first character of a current special symbol.

2. The longest possible special symbol is matched with the string a that point, and successively shorter ones until and unless a specia symbol is found.

3. If more than one special symbol is of a given length, recognition i attempted in the following fixed order (i.e. if A is above B in th list and A and B are the same length then A will be matched first):

MARG
MAF
MNF
MEF
MLL
MRL
MEP

4. A recognized special symbol is deleted from the string and scanning resumes with the first character following it.

5. The symbols MEF and MRL could be different. In this case the MEF special symbol still balances both the MAF and MNF symbols, but does not balance the MLL symbol.

## Storage Management

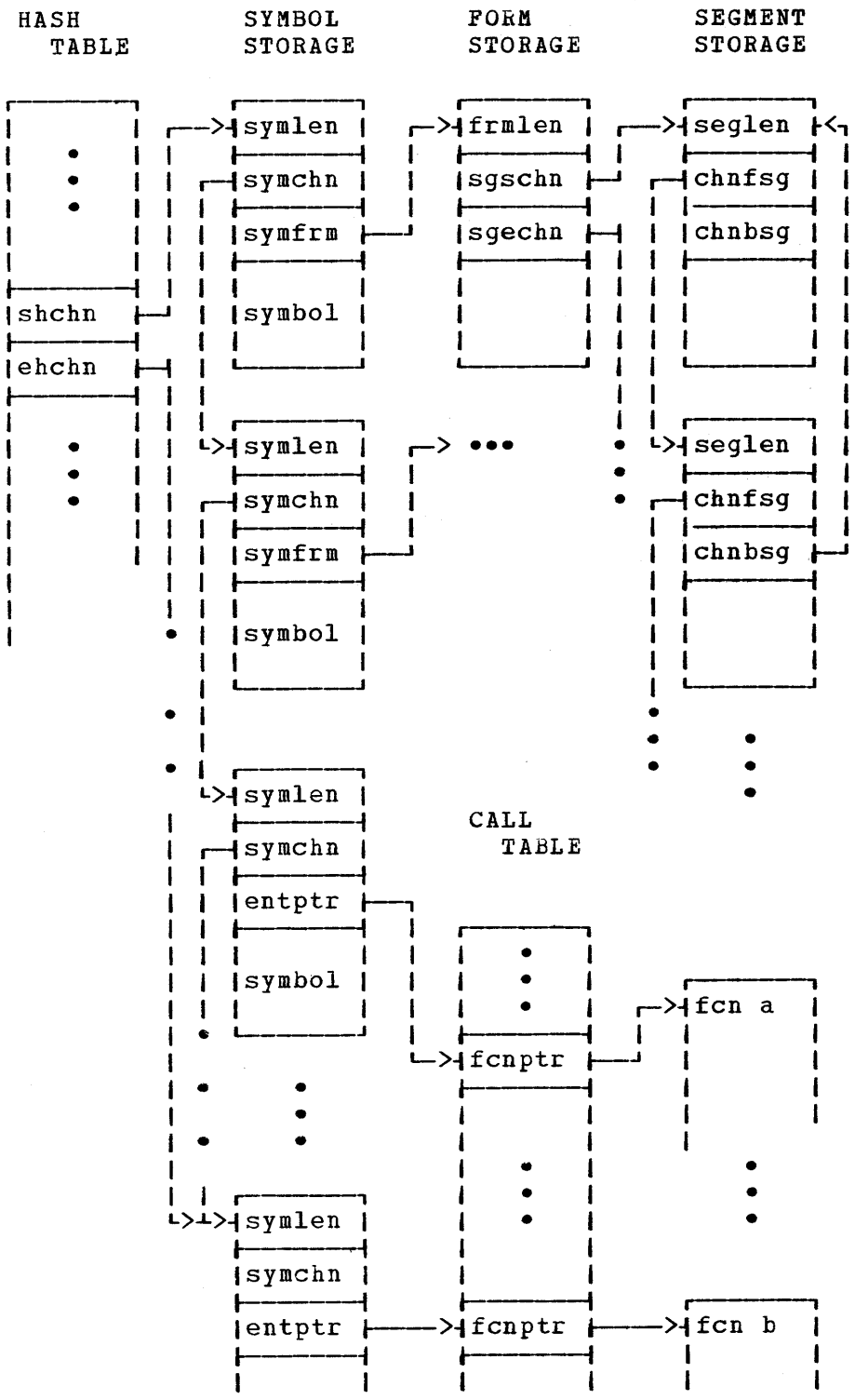
This section describes the way in which storage is allocated and managed for form and function names, segment gap information, and form values in UMIST.

Each name, form value, and segment (string between segment gaps) is described by a block of not less than 8 words of storage. The first 6 words of each block are used to describe the entry, and the value, if any, appears beginning in the 7th word. Blocks are allocated on demand from an area called free storage in predefined fixed sizes. If no blocks of a given size are available to satisfy a demand, a new one is generated. When a block is released it is placed on a queue with all other free blocks of the same size to be available for use again.

The various storage blocks are linked together as described below. The symbol table consists of a set of 64 unordered chains of symbol (name) blocks, with each name placed on a single chain by following hash scheme:

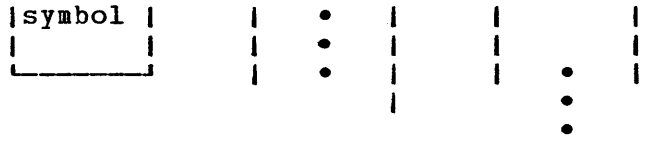
The first and last characters of the name are multiplied using their character codes as binary integers. The result is divided by 127, and the remainder is divided in two. The resulting quotient, whose value is in the range 0-63, is the index of the chain on which the symbol belongs.

As pictured in the following diagram, each symbol block points to a primitive function to be evaluated, and possibly a form block. The form block contains the text of the form (whose name appears in the symbol block), and points to a two-way ordered chain of segment blocks.


HASH TABLE
SYMBOL STORAGE
FORM STORAGE
SEGMENT STORAGE
shchn
ehchn
symlen
symchn
symfrm
symbol
frmlen
sgschn
sgechn
seglen
chnfsg
chnbsg
entptr
CALL TABLE
fcnptr
fcn a
fcn b

MTS-570-0

12-1-67



BIBLIOGRAPHY

[1] Mooers, Calvin N., "TRAC, A Procedure-Describing Language for the Reactive Typewriter", Comm. ACM 9 (March, 1966) p. 215

[2] Mooers, Calvin N., and L. P. Deutsch, "TRAC, A Text Handling Language," Proc. ACM Nat. Conf., Cleveland (August, 1965) p. 229

[3] Eastwood, D. E., and M. D. McIlroy, "Macro Compiler Modification of SAP, " Computer Lab. Memo., Bell Telephone Labs., Murray Hill, N. J. (Sept, 1959)

[4] McIlroy, M. D., "Using SAP Macro Instructions to Manipulate Symbolic Expressions," Computer Lab. Memo., Bell Telephone Labs., Murray Hill, N. J. (1960)

[5] Strachey, C., "A General Purpose Macro-generator," Computer Journal 8,3 (1966)

## APPENDIX A. A GUIDE TO USING UMIST IN MTS

The UMIST processor is stored in a file named *UMIST in the Michigan Terminal System. Its initial settings for input and output devices are the SCARDS and SPRINT logical units. Thus a user signing on at a terminal might issue the following MTS command to obtain the UMIST processor:

```
$RUN  *UMIST
```

If one wishes to use files or other devices for input-output as well as the terminal, he specifies these devices in a parameter set function call. For example, the function call

```
#(PAR,FDO,SAVNEW)
```

switches output to the file SAVNEW, and the call

```
#(PAR,FDI,DFF)
```

switches input to the file DFF. An end-of-file at any input device other than SCARDS switches input to the SCARDS device. An end-of-file at the SCARDS device terminates the execution of UMIST and returns the user to MTS.

The initial settings of all parameters are listed below.

| Name | Value |
|---|---|
| PROT ..... | BAS*BLT |
| STD ..... | US1 |
| TR ..... | OFF |
| CL ..... | ON |
| FOLD ..... | OFF |
| TRAN ..... | OFF |
| SVCD ..... | 0 |
| LINE ..... | 72 |
| FDI ..... | SCARDS |
| FDO ..... | SPRINT |
| STOP ..... | The character whose hex code is C0 |
| IGNR ..... | The character whose hex code is 80 |

Note: to change one of the character parameters STOP or IGNR to another non-printing code, the function XTC must be used to convert its hex code to a single character, e.g.

```
#(PAR,STOP,##(XTC,BA))
```

changes the STOP character to the character whose hex code is BA.

The XTC function may also be used to insert special edit and control characters into UMIST strings without causing the edit or control effect at the time of insertion. E.g. #(XTC,15) has as its value one carriage return character.

Since the FOLD parameter is initially OFF, and UMIST primitive function names are all upper case, a person using a device supporting lower case must be sure to type his function, parameter, class, etc. names in upper case, at least until he has turned ON the FOLD switch.

# APPENDIX B. PRIMITIVE FUNCTIONS

## print_string_function

#(ps,a)

this null valued function prints out the string a given as the second argument. more than one line may be printed if the length of a exceeds the current line parameter value.

## read_string_function

#(rs)

returns as value the beginning of the input string up to but not including the first end of string special symbol, which is eliminated.

## signoff_function

#(bye,note)

the bye function causes normal termination of UMIST execution after printing the optional second argument.

## define_string_function

#(ds,n,v)

this null valued function defines the string v as a form with the name n. the form pointer is set to the first character, and membership in the current standard protection classes is assigned. the meaning of an implicit call is set according to the current parameter value.

define form function

#(df,name,prot,call,value,fptr,g1a,g1b,g1c,g2a,...)

this null valued function defines a form completely, specifying membership classes, the meaning of implicit call, form pointer and segment gaps as well as the name and value strings. the prot argument is a class expression. the call argument is either cl or cp, and the fptr argument the ordinal position (from 0) of the character under the form pointer. the segment gap arguments come in triples, indicating ordinal value, position and length of gap, respectively.

segment string function

#(ss,n,a,b,c,...)

segment gaps in the form n are created by this null-valued function wherever a character scan with the arguments a, b, c,... finds string matches in the segments remaining in the form. the arguments a,b,... are taken in order and define gaps of ordinal value 1,2,... respectively. the form pointer is reset to the first character by this function.

call function

#(cl,n,a,b,c,...)

the value of cl is the expanded form whose name is the second argument: specifically, starting at the form pointer, each segment gap is filled with the i+2nd argument in the call, if present, where i is the ordinal value of the segment gap.

call procedure function

#(cp,n,a,b,c,...)

this function calls the form named as the second argument in exactly the manner specified for the cl function, and places its value on the active string.

after the form itself has been completely evaluated, its value is moved to the neutral or active string according to the mode of the call procedure function call. in effect, the neutral/active call distinction is applied by this function one level down from the point at which it was given.

print form function

#(pf,a,b,c,...)

the value of this function is a representation of each of the forms named as second and succeeding arguments. the location and ordinal value of each segment gap is shown, as well as the protection class membership, mode of implicit call, and form pointer. each form is shown by a neutral call on the define form function, which, if executed, would define the same form. name and value in the call constructed are protected by a set of literal symbols.

initial function

#(in,n,a,fail)

starting at the form pointer, the form named by the second argument is searched for the first occurrence of the third argument as a substring. segment gaps are skipped. the value of this function is the substring from the form pointer to the character just before the matching string. if a match is not found, the value is the fourth argument, which is always placed on the active string. the form pointer is moved to the first character following the matching string, or is not moved at all if there is no successful match.

call segment function

#(cs,n,e)

the value of this function is the substring of the form named by the second argument beginning at the form pointer and continuing to the segment gap which follows it. the form pointer is moved to the first character

after the gap. if the form is empty the third argument is returned to the active string.

call_character

#(cc,n,e)

the value of this function is the character under the form pointer in the form named n. if n is empty, the value is e (always moved to the active string). the form pointer, which always skips segment gaps, is moved ahead one character.

call_n_characters

#(cn,n,cnt,e)

this function reads characters from the form pointer to a total length specified by the third argument, which is a decimal integer. if the number is positive, the string is read from the pointer to the right, if negative, to the left (keeping the original character sequence). if there are not enough characters, the value is e (moved to active string). the form pointer is moved (right or left) to the first unread character. segment gaps are skipped.

call_restore_function

#(cr,n)

this null valued function restores the form pointer of the form whose name is the second argument to the first character.

set_function

#(set,a1,b1,a2,b2,...)

this null valued function forms an equivalence between the symbols ai and bi in pairs. ai is assigned

the meaning of bi, for each i, unless the former is in a protected class. any symbol may rename any other.

delete_definition_function

#(dd,a,b,c,...)

this null valued function deletes the definitions of all of the forms given as the second and succeeding arguments.

delete_all_function

#(da)

this null valued function deletes the definitions of all the forms and external functions.

equal_function

#(eq,a,b,yes,no)

this function returns the fourth argument if the second and third arguments are identical character strings, and the fifth argument otherwise.

decimal_arithmetic_functions

these functions operate on decimal character strings, assumed to be numeric only and no more than 16 digits long, including the optional sign. a decimal value is returned unsigned if positive, and leading zeros are deleted. the value zero is one digit long.

add_decimal

#(ad,a,b)

the value is a+b

subtract_decimal

#(su,a,b)

the value is a-b

multiply_decimal

#(ml,a,b)

the value is a*b

divide_decimal

#(dv,a,b)

the value is a/b

test_decimal

#(td,a,r,b,t,f)

compares a with b. if the relation r between them is satisfied, the value is t, otherwise the value is f.

special_symbol_functions

#(marg)
#(maf)
#(mnf)

#(mef)
#(mll)
#(mrl)
#(mep)
#(mes)
each of these functions has a value which is the corresponding current special symbol.

parameter set function

#(par,n1,v1,n2,v2,...)

this null valued function alters the settings of global parameters in the umist system, such as the trace switch and input/output devices, the error severity tolerance, and the protection classes.

print parameter function

#(ppr)

this function returns as its value the current values of all of the global umist parameters. they are given in the form of a call on the parameter setting function (par).

load external functions function

#(lef,dev)

this null valued function calls the loader to load and link external functions from the named device to the umist system, and then makes the table entries necessary to complete the acquisition.

read character function

#(rc)

returns the next input character as its value. may be the end of input string special symbol.

read_n_characters_function

#(rn,l)

returns as a value the beginning of the input string of the length given in decimal as the second argument. end of string special symbols may be included in this form of input.

dump_function

#(dmp)

this null valued function dumps all non-reentrant parts of UMIST on the current output device.

null_function

#(nl,a,b,c,...)

the null function does absolutely nothing but 'discard' its variable number of arguments: its value is null.

restart_function

#(res)

this null valued function completely restores the status of umist at startup time, including the predefined values of all parameters, function names and special symbols.

reinitialize_function

#(rin)

this null valued function reinitializes umist with the current print string/read string sequence and empties the input buffer.

set form pointer function

#(sfp,a,n)

this null valued function sets the form pointer of the form named by the second argument to the character displaced from the first by the number of characters given by the third argument, a decimal integer.

call form pointer function

#(cfp,n)

the value of this function is a decimal integer giving the number of characters which the form pointer is now displaced from the first character.

call gap function

#(cg,n,e)

the value of this function is the first segment gap to the right of the form pointer of the form n. the form pointer is set to the first character following the gap. if there are no more segment gaps, the value is e and the form pointer is not moved.

call ordinal value

#(cv,n,e)

the value of this function is the decimal integer ordinal value of the next segment gap to the right of the form pointer of the form n. the form pointer is not altered by this function. if there are no (more) segment gaps the value is the third argument.

erase segment gaps function

#(es,n)

removes all segmentation for the form given as the second argument. the form pointer is not moved.

set protection classes function

#(spc,n,s)

the name in the second argument is assigned to the protection classes in the expression which is the third argument.

list selected names function

#(lst,s,x)

the value of this function is a list of names of forms and functions, each name preceded by the string x. s is a parameter name expression specifying the names to be listed: legal operations are *, for intersection, and +,denoting union.

test character function

#(tc,a,r,b,yes,no)

a and b are character strings to be compared. the third argument, r, is one of the relations <, <=, >, >=, =, or/=, which stand for "properly contained in", "contained in", "properly contains", "contains", "identical to", and "not identical to", respectively. the argument yes is the value of the function when the relation is true, otherwise no is the value.

length function

#(len,s)

the value of this function is the length of the second argument, as an unsigned decimal integer.

hexadecimal to character function

#(xtc,a)

the value of this function is the character string resulting from converting the second argument from its hex character codes.

character to hexadecimal function

#(ctx,a)

the value of the ctx function is the string of hex character codes for the second argument. codes for leading null characters are deleted.

hexadecimal arithmetic functions

these functions operate on unsigned hexadecimal character strings, assumed to be not more than 8 legal hexadecimal digits long. a hex value is returned with no sign and no leading zeros.

add hex

#(ax,a,b)

the value is a+b

subtract_hex

#(sx,a,b)

the value is a-b

test_hex

#(tx,a,r,b,t,f)

compares a with b. if the relation r between them is satisfied the value is t, otherwise it is f.

if function

#(if,a,t,f)

the value of this function is t if the second argument, as a boolean string, contains at least one true, and f if not (or if a is null).

not function

#(not,a)

the value of this function is the characterwise complement of the second argument, a boolean character string.

and, or, and xor functions

#(or,a,b)
#(and,a,b)
#(xor,a,b)

the values of these functions are the characterwise logical "and", "or" and "exclusive or" of a and b. a and b are boolean character strings. if they are not of the same length the value has the length of the shorter for 'and' and the length of the longer for 'or'

and 'xor'. in the former case the leftmost part of the longer is used, and in the latter case the shorter is extended on the right with false values.

define special symbol function

#(dss,a,aval,b,bval,...)

this function redefines the special symbols. the arguments occur in pairs, denoting special symbol name and new special symbol value, respectively. the change is effective as soon as the function has been all evaluated. new special symbols may not exceed four characters in length.

date function

#(dt)

the value of this function is the current date, given in the form mm-dd-yy.

time of day function

#(tm)

the value of this function is the current time of day, given in the form hh:mm.ss.

translate function

#(trn,f1,t1,f2,t2,...)

after this null valued function has been executed, each of the characters fi is replaced in the input string by the corresponding ti before reaching the umist system

translate_print_function

#(trp)

this function returns as its value a call on the trn function, describing exactly those input character transformations currently specified in the table.

hash_history_function

#(hsh)

this null valued function dumps the hash history table and then clears it out. the latter contains a count, by hash chain, of the number of times symbols are referenced on that chain.

## APPENDIX C. UMIST LINE EDITOR

The purpose of the UMIST procedure given below is to insert, delete, and replace text fragments in the lines of a line-numbered auxiliary storage file. A single execution of the procedure alters the first occurrence in the given line of the specified context. The destination of a text fragment to be inserted (or used as a replacement) is specified by giving enough text on either side of the location to uniquely identify it.

A restriction imposed by the syntax of UMIST, but avoidable with some additional effort is that the text fragments given to the editing procedure contain no unbalanced parentheses.

The action to be taken in editing a line is given implicitly by the presence or absence of one or more of the text and context fragments in the procedure call:

#(edit,<file>,<line>,<left>,<text>,<right>)

where:

<file> denotes the name of the file to be edited

<line> denotes the (integer) line number

<left> denotes the context to the left of the location desired

<text> is the fragment to be inserted or deleted

<right> denotes the context to the right of the desired location

The following combinations of specified fragments give the indicated actions: (X denotes present)

| | <left> | <text> | <right> | Action |
|---|---|---|---|---|
| (a) | X | X | X | Replace whatever lies between <left> and <right> with <text> |
| (b) | X | X | | Insert <text> to the right of <left> |
| (c) | | X | X | Insert <text> to the left of <right> |
| (d) | | X | | Delete <text> |

Examples

Assume the file named ENGLISH contains the following lines of text:

```
271     With a filter such as P2 it is possible
272     to severely attenuate the high frequency
273     noise without distorting the low
274     frequency information excessively. If the
```

After the sequence of procedure calls shown below, the text will have been modified as indicated. _ denotes a space character.

```
#(edit,ENGLISH,271,a_,smoothing_)
#(edit,ENGLISH,274,,_excessively)
#(edit,ENGLISH,273,,_excessively,_dis)
#(edit,ENGLISH,272,to_, strongly,_att)
```

```
271     With a smoothing filter such as P2 it
272     is possible to strongly attenuate the high
273     frequency noise without excessively
274     distorting the low frequency information.  If the
```

A "higher-level" procedure for simplifying commands would be:

```
#(ds,EDIT,(#(edit,<file>,#(rs))#(EDIT,<file>)))
#(ss,EDIT,<file>)
```

The line editor can now be invoked by:

```
#(EDIT,<file>)'
```

with the editing commands typed one after the other in the form

```
<line>,<left>,<text>,<right>'
```

e.g. (from example above)

```
#(EDIT,ENGLISH)'
```

and then

```
271,a ,smoothing'
274,,  excessively'
       .
       .
       .
```

To stop the above procedure, insert a UMIST procedure call in the command line, such as one of the following:

```
#(rin)'          to reinitialize the UMIST system

#(bye)'          to return to the operating system
```

Elements of the Definition of Edit:

(1) to read the line from the desired file:

#(par,fdi,<file>((<line>,<line>)))##(rs)

Action:

Set the "file-or-device in" parameter to the unit named <file> and read from <line> to <line> before returning, then read string

(2) to write the output of the procedure back into the line of the file:

#(par,fdo,<file>((<line>)))#(ps,output)

#(par,fdo,*sink*)

Action:

Set the "file-or-device output" designation to the given <file> and <line>. print the string of output into the file return the fdo designation to the standard output device.

(3) Constructed procedures for the actions of inserting, deleting and replacing use the functions below:

#(in,form,string)

Action:

The form named is searched for the first occurrence of string and returns the form from the form pointer up to (but not including) string as a value. The form pointer is reset to the character following string.

#(nl,A,B,C,...)

Action:

This function always has a null value.

(4) To choose the appropriate procedure to construct and apply, the following functions are used:

#(len,string)

Action:

The value of len is the decimal integer number of characters in string.

#(mef)

Action:

The value is an end-of-function symbol (right parentheses)

#(eq,A,B,C,D)

Action:

The value of eq is C if A and B are identical, otherwise the value is D.

Remarks

A form called by giving its name as the first argument returns the value of the form.

The output line is produced by concatenating values of procedure calls and the text fragments given in the command line.

MTS-570-0

12-1-67

Definition of the Procedure:

```
#(ds,edit,(
   #(par,fdo,((<file>,<line>)))
        #(ds,old,#(par,fdi,<file>((<line>,<line>)))
   #(ps,
            #(rs))
   (ds,temp)#(in,old,#(eg,#(len,<left>),0,
            (#(eg,#(len,<right>),0,
                 (<text>#(mef)),
                 (<right>#(mef)<text><right>))),
            (<left>#(mef)<left><text><right>
                 #(nl,#(in,old,<right>))))
        #(old)))
   #(ps,#(temp)
   #(par,fdo,*sink*)))'
```

And finally to create the "formal parameters":

```
             #(ss,edit,<file>,<line>,<left>,<text>,<right>)'
```

The procedure edit produces one of the following procedures and the evaluates it:

(a) for delete:

```
#(in,old,<text>)#(old)
```

(b) for insert before:

```
#(in,old,<right>)<text><right>#(old)
```

(c) for insert after and replace:

```
#(in,old,<left>)<left><text><right>
     #(nl,#(in,old,<right>)#(old)
```

MTS-580-0

12-1-67

# W A T F O R

## WATFOR

WATFOR, the University of Waterloo Fortran translator, is available i MTS in the file *WATFOR. Consult the library file description of *WATFO in section MTS-280/66635 for details on I/O unit and parameter specification.

### I WATFOR CONTROL CARDS

Since execution of the object program commences immediately upo completion of compilation, and since WATFOR can process more than on FORTRAN-IV job on a single run, the mechanics of running in WATFOR diffe from standard MTS FORTRAN-G. Only one MTS $RUN command is needed to invok both the translator and the final object program for several separat fortran programs. WATFOR reads its source cards from SCARDS and places th program listing on SPRINT. The object program cannot be saved. WATFO recognizes certain control cards which are not MTS commands.

#### $COMPILE

The $COMPILE card must appear immediately before each job[1] to b processed by WATFOR. The control character ($) must appear in colum one and must be followed immediately by the word "COMPILE". A list o programmer-specified parameters may be included and must be separate from the "$COMPILE" by at least one blank and from each other by comma. They may not contain imbedded blanks. These optional parameter are as follows:

| | |
|---|---|
| LINES | The number of lines per page. Default is 59. |
| PAGES | The number of pages of output allowed. Default is 100. |
| RUN | "CHECK", "NOCHECK", or "FREE". Default is "CHECK". (See section entitled, "Error Diagnostics and Runnin Modes" for explication.) |
| KP | "26" or "29". Default is "29". Specifies type o keypunch used for the source program. |

Following the $COMPILE card comes the main program and its subroutines. No $COMPILE cards may be inserted between the main program an its subroutines, or between the subroutines themselves.

---

[1]A "job" is defined to be a main program, its subroutines and data.

Examples: $COMPILE PAGES=25,LINES=30,RUN=NOCHECK
$COMPILE PAGES=200,RUN=FREE

$DATA

The $DATA card signals end of source decks for a job. Upon detecting $DATA, WATFOR enters the main program which it has translated. Cards following the $DATA control card are treated as data for the current job. End of data is signalled by the next card with a "$" in column one, or an end of file.

$STOP

A $STOP card will return control to MTS. An end of file on SCARDS causes a "$STOP" card to generated.

This example indicates the type of deck set-up for running under WATFOR.

```
$SIGNON XXXX
$RUN *WATFOR
$COMPILE        PAGES=150
        {main program source deck}
        {source deck for subroutines}
$DATA
        {data cards}
$COMPILE        RUN=FREE
        {source cards}
$STOP
$SIGNOFF
```

## II ERROR DIAGNOSTICS AND RUNNING MODES

WATFOR classifies compile-time errors under three types:

| | |
|---|---|
| EXTENSION | A non-standard FORTRAN language feature has been used. (See section entitled, "Language Extensions.") |
| WARNING | A non-fatal error has been detected and a forgiving assumption has been made by the compiler; e.g. truncation of a name of more than six characters. |
| ERROR | A serious error for which no reasonable corrective action may be taken. Most serious errors generate object code which will terminate execution of the object program when the statement in error is reached. |

All execution errors are fatal and the job is terminated with a subprogram traceback.

WATFOR will allow a program to go into execution even with some serious source errors. To give the programmer control over this feature, WATFOR

provides three modes of running which may be specified on the $COMPIL card. These are CHECK, NOCHECK and FREE.

CHECK means execute the translated program if no serious errors have been detected and check for all execution-time errors. This is the normal mode.

NOCHECK is the same as CHECK, but undefined variables[2] are ignored. (That is, the object code necessary to perform checking for undefined variables is not generated.) This results in somewhat faster execution.

FREE is the same as CHECK, but execution is initiated in spite of any serious source errors.

MTS WATFOR has CHECK as default and an alternate mode may be specified on the $COMPILE card.

WATFOR will generate a "core constant" for each variable and variable array, depending on the mode of the variable. Each fixed point variable is preset to -2139062144, and each real variable is preset to -0.4335017E-77.

III SUBROUTINE REFERENCES.

Any subroutines referenced in WATFOR must come from one of three possible sources: user supplied (i.e., with his source deck), the internal WATFOR function library, or a source library on disk. *WATLIB is the name of the WATFOR library maintained by the Computing Center. The user may supply his own library, provided it is correctly formatted (see Section IV). WATFOR searches for subroutines in the following order: user-supplied, built-in function library, and pre-stored source library. Logical unit 0 is used to reference the pre-stored library.

In the following example, a user wishes to use *WATLIB.

```
$RUN *WATFOR; 5=*SOURCE* 6=*SINK* 0=*WATLIB
```

IV WATFOR SUBROUTINE LIBRARY STRUCTURE.

A WATFOR subroutine library consists of a directory and the WATFOR source code for the subroutines. The structure of this library is identical to the structure of a macro library (See Section MTS-255).

A. The directory:

1. Each entry in the directory contains the name of a subroutine in columns 1-8 and the line-number of the first WATFOR statement of the

---

[2]An "undefined variable" is a variable whose value has not been set at either compile or execution time.

subroutine in columns 10-16. (Both the name and the line-number must be left justified with trailing blanks.)

2. The line-number of the first entry in the directory must be 1.

3. The terminating entry in the directory is a string of eight zeros in columns 1-8.

B. The subroutines:

1. The line-number of the first statement in the subroutine must be a positive integer.

2. The first subroutine follows the last entry in the directory.

3. Each subroutine must be followed by a line with $TERM in positions 1 through 5.

## V LANGUAGE EXTENSIONS

WATFOR provides a number of extensions to the standard FORTRAN-IV language. Warning messages are generated in the source listing for all such extensions so the programmer may eliminate them should he desire to run his program through other compilers. Following is a list of extensions currently available in WATFOR.

1. Free I/O. This allows the programmer to do input/output without reference to a format statement. For example, the statement PRINT,ALPHA,X will cause the values of ALPHA and X to be output on sprint. Free I/O has been implemented to function only on statements of the form

```
READ,  list

PRINT, list

PUNCH, list
```

Input is done through SCARDS, punch through SPUNCH, and output through SPRINT. The comma after the READ, PRINT, or PUNCH is mandatory.

For input, data items must be separated by a comma and/or one or more blanks or a card boundary.

A duplication factor may be given to avoid punching the same constant may times. For example, if A were dimensioned 25, the statement

```
READ,A
```

with data 25*2. would free-read 2 into all the elements of A.

Enough cards are read to satisfy the requirements of the I/O lis and each free input statement starts a new card. The forms whic may be used for input are:

Integer - signed or unsigned integer constant

Real - signed or unsigned real constant in F, E or formats

Complex - 2 real numbers enclosed in parentheses and sepa rated by a comma.

Logical - T or F

Data items must match in type the variables they are being rea into.

Free output uses fixed formats for the various types of variable and line overflow is automatically accounted for (since severa lines may result from one PRINT statement.) The formats used are:

| | |
|---|---|
| Integer | I12 |
| Real*4 | E16.7 |
| Real*8 | D28.16 |
| Complex*8 | 2E16.7 |
| Complex*16 | 2D28.16 |
| Logical | L8 |

No extension message is given for the use of Free I/O.

2. Multiple assignment statements of the form

V1 = V2 = V3 = ... = Vn = expression

are allowed, where the Vi represent variable names or arra elements. This is treated exactly like the sequence of statements

Vn = expression

Vn-1 = Vn

•

•

•

V1 = V2

3. Expressions may be placed in output statements. For example

```
WRITE (6,10)  SIN(X)**2, A*X+(B-C)/2.
```

The expression may not, however, start with a left parenthesis, since the compiler uses this as a signal that an implied DO in the I/O list is to be used. For example PRINT,(A+B)/2. would give an error message.

4. Common blocks (including blank common) may be initialized in other than BLOCK DATA subprograms.

5. Implied DO's are allowed in DATA statements. For example:

```
DATA (A(I), I=1,5,2)/3*.25/
```

is valid. In general, DATA (A(I),I=L,M,N)/ constant list / is valid, provided L, M and N have been initialized and at least

$$\frac{(L-M)}{N} + 1$$

constants are present in the constant list.

6. Common blocks may be given different lengths in different subprograms, but the length of the longest block of any label is used.

7. Subscripts may be used on the right hand side of a function definition. For example:

```
F(X) = A(I) * X + B(I)
```

8. Hollerith constants may be used in function references. In general, Hollerith constants are treated by WATFOR as singly subscripted, real arrays and are right-padded with blanks to a multiple of four, if necessary. Thus, 'ABC' is stored as 'ABC '. No all-bits fence is provided. Since WATFOR forces the user to have subroutine arguments agree in number, type, and mode, a Hollerith constant passed to a subroutine must be passed to a dummy real vector.

9. Boundary violations due to EQUIVALENCE or COMMON are diagnosed at compile time and corrected at execution time by moving the data to a valid boundary, performing the arithmetic operation, and moving any result back to the original boundary.

10. The character sequence FORMAT( is a reserved word when it appears as the first seven characters of a statement. Thus

```
FORMAT(I1) = 10.
```

is illegal, whereas

```
X = FORMAT(I1)
```

is valid.

11. The programmer may bypass floating point overflow and underflow interrupts by calling a WATFOR built-in routine named TRAPS, which functions as follows:

    The statement

    ```
    CALL TRAPS(I,J,K)
    ```

    placed in the source program will allow "I" fixed point overflows, "J" exponent overflows, or "K" exponent underflows. If any of these is exceeded execution is terminated. I, J, and K must all be given and must be INTEGER*4 expressions.

## VI LANGUAGE RESTRICTIONS.

A number of restrictions are present in the language; anyone attempting to use WATFOR should peruse the next section carefully.

1. WATFOR is strictly a FORTRAN compiler; no assembly language subprograms may occur in a WATFOR job.

2. The compiler does not produce object decks.

3. NAMELIST and direct access I/O features have not been implemented.

4. The debug language instructions as described in form TNL N28-2147 have not been implemented.

MTS-580-0

12-1-67

# WATFOR COMPILER ERROR MESSAGES

## ASSIGN STATEMENTS AND VARIABLES

AS-2 ATTEMPT TO REDEFINE AN ASSIGNED VARIABLE IN AN ARITHMETIC STATEMENT
AS-3 ASSIGNED VARIABLE USED IN AN ARITHMETIC EXPRESSION
AS-4 ASSIGNED VARIABLE CANNOT BE HALF WORD INTEGER
AS-5 ATTEMPT TO REDEFINE AN ASSIGN VARIABLE IN AN INPUT LIST

## BLOCK DATA STATEMENTS

BD-0 EXECUTABLE STATEMENT IN BLOCK DATA SUBPROGRAM
BD-1 IMPROPER BLOCK DATA STATEMENT

## CARD FORMAT AND CONTENTS

CC-0 COLUMNS 1-5 OF CONTINUATION CARD NOT BLANK
PROBABLE CAUSE - STATEMENT PUNCHED TO LEFT OF COLUMN 7

CC-1 TOO MANY CONTINUATION CARDS (MAXIMUM OF 5)
CC-2 INVALID CHARACTER IN FORTRAN STATEMENT "?" INSERTED IN SOURCE LISTING
CC-3 FIRST CARD OF A PROGRAMME IS A CONTINUATION CARD
PROBABLE CAUSE - STATEMENT PUNCHED TO LEFT OF COLUMN 7
CC-4 STATEMENT TOO LONG TO COMPILE (SCAN-STACK OVERFLOW)
CC-5 BLANK CARD ENCOUNTERED
CC-6 KEYPUNCH USED DIFFERS FROM KEYPUNCH SPECIFIED ON JOB CARD
CC-7 FIRST CHARACTER OF STATEMENT NOT ALPHABETIC
CC-8 INVALID CHARACTERS(S) CONCATENTATED WITH FORTRAN KEYWORD
CC-9 INVALID CHARACTERS IN COL 1-5. STATEMENT NUMBER IGNORED
PROBABLE CAUSE - STATEMENT PUNCHED TO LEFT OF COLUMN 7

## COMMON

CM-0 VARIABLE PREVIOUSLY PLACED IN COMMON
CM-1 NAME IN COMMON LIST PREVIOUSLY USED AS OTHER THAN VARIABLE
CM-2 SUBPROGRAMME PARAMETER APPEARS IN COMMON STATEMENT
CM-3 INITIALIZING OF COMMON SHOULD BE DONE IN A BLOCK DATA SUBPROGRAMME
CM-4 ILLEGAL USE OF COMMON BLOCK OR NAMELIST NAME

## FORTRAN TYPE CONSTANTS

CN-0 MIXED REAL*4,REAL*8 IN COMPLEX CONSTANT
CN-1 INTEGER CONSTANT GREATER THAN 2,147,483,647 ($2^{31}-1$)
CN-3 EXPONENT ON REAL CONSTANT GREATER THAN 99
CN-4 REAL CONSTANT HAS MORE THAN 16 DIGITS, TRUNCATED TO 16
CN-5 INVALID HEXADECIMAL CONSTANT
CN-6 ILLEGAL USE OF DECIMAL POINT
CN-8 CONSTANT WITH E-TYPE EXPONENT HAS MORE THAN 7 DIGITS, ASSUME D-TYPE
CN-9 CONSTANT OR STATEMENT NUMBER GREATER THAN 99999

## COMPILER ERRORS

CP-0 DETECTED IN PHASE RELOC
CP-1 DETECTED IN PHASE LINKR

CP-2 DUPLICATE PSEUDO STATEMENT NUMBERS
CP-4 DETECTED IN PHASE ARITH

DATA STATEMENT
DA-0 REPLICATION FACTOR GREATER THAN 32767, ASSUME 32767
DA-1 NON-CONSTANT IN DATA STATEMENT
DA-2 MORE VARIABLES THAN CONSTANTS IN DATA STATEMENT
DA-3 ATTEMPT TO INITIALIZE A SUBPROGRAMME PARAMETER IN A DATA STATEMENT
DA-4 NON-CONSTANT SUBSCRIPTS IN A DATA STATEMENT INVALID IN /360 FORTRAN
DA-5 EXTENDED DATA STATEMENT NOT IN /360 FORTRAN
DA-6 NON-AGREEMENT BETWEEN TYPE OF VARIABLE AND CONSTANT IN DATA STATEMENT
DA-7 MORE CONSTANTS THAN VARIABLES IN DATA STATEMENT
DA-8 VARIABLE PREVIOUSLY INITIALIZED. LATEST VALUE USED
CHECK COMMON/EQUIVALENCED VARIABLES
DA-9 INITIALIZING BLANK COMMON NOT ALLOWED IN /360 FORTRAN
DA-A INVALID DELIMITER IN CONSTANT LIST PORTION OF DATA STATEMENT
DA-B TRUNCATION OF LITERAL CONSTANT HAS OCCURRED

DIMENSION STATEMENTS
DM-0 NO DIMENSIONS SPECIFIED FOR A VARIABLE IN A DIMENSION STATEMENT
DM-1 OPTIONAL LENGTH SPECIFICATION IN DIMENSION STATEMENT IS ILLEGAL
DM-2 INITIALIZATION IN DIMENSION STATEMENT IS ILLEGAL
DM-3 ATTEMPT TO RE-DIMENSION A VARIABLE
DM-4 ATTEMPT TO DIMENSION AN INITIALIZED VARIABLE

DO LOOPS
DO-0 ILLEGAL STATEMENT USED AS OBJECT OF DO
DO-1 ILLEGAL TRANSFER INTO THE RANGE OF A DO-LOOP
DO-2 OBJECT OF A DO STATEMENT HAS ALREADY APPEARED
DO-3 IMPROPERLY NESTED DO-LOOPS
DO-4 ATTEMPT TO REDEFINE A DO-LOOP PARAMETER WITHIN RANGE OF LOOP
DO-5 INVALID DO-LOOP PARAMETER
DO-6 TOO MANY NESTED DO'S (MAXIMUM OF 20)
DO-7 DO-PARAMETER IS UNDEFINED OR OUTSIDE RANGE
DO-8 THIS DO LOOP WILL TERMINATE AFTER FIRST TIME THROUGH
DO-9 ATTEMPT TO REDEFINE A DO-LOOP PARAMETER IN AN INPUT LIST

EQUIVALENCE AND/OR COMMON
EC-0 TWO EQUIVALENCED VARIABLES APPEAR IN COMMON
EC-1 COMMON BLOCK HAS A DIFFERENT LENGTH THAN A PREVIOUS SUBPROGRAMME
EC-2 COMMON AND/OR EQUIVALENCE CAUSES INVALID ALIGNMENT. EXECUTION SLOWED
REMEDY - PUT DOUBLE WORD QUANTITIES FIRST
EC-3 EQUIVALENCE EXTENDS COMMON DOWNWARDS
EC-7 COMMON/EQUIVALENCE STATEMENT DOES NOT PRECEDE PREVIOUS USE OF VARIABLE
EC-8 VARIABLE USED WITH NON-CONSTANT SUBSCRIPT IN COMMON/EQUIVALENCE LIST
EC-9 A NAME SUBSCRIPTED IN AN EQUIVALENCE STATEMENT WAS NOT DIMENSIONED

END STATEMENTS
EN-0 NO END STATEMENT IN PROGRAMME -- END STATEMENT GENERATED

```
EN-1      END STATEMENT USED AS STOP STATEMENT AT EXECUTION
EN-2      IMPROPER END STATEMENT
EN-3      FIRST STATEMENT OF SUBPROGRAMME IS END STATEMENT

 EQUAL SIGNS
EQ-6      ILLEGAL QUANTITY ON LEFT OF EQUALS SIGN
EQ-8      ILLEGAL USE OF EQUAL SIGN
EQ-A      MULTIPLE ASSIGNMENT STATEMENTS NOT IN /360 FORTRAN

 EQUIVALENCE STATEMENTS
EV-0      ATTEMPT TO EQUIVALENCE A VARIABLE TO ITSELF
EV-1      ATTEMPT TO EQUIVALENCE A SUBPROGRAMME PARAMETER
EV-2      LESS THAN 2 MEMBERS IN AN EQUIVALENCE LIST
EV-3      TOO MANY EQUIVALENCE LISTS (MAX = 255)
EV-4      PREVIOUSLY EQUIVALENCED VARIABLE RE-EQUIVALENCED INCORRECTLY

 POWERS AND EXPONENTIATION
EX-0      ILLEGAL COMPLEX EXPONENTIATION
EX-2      I**J WHERE I=J=0
EX-3      I**J WHERE I=0,J < 0
EX-6      0.0**Y WHERE Y //G 0.0
EX-7      0.0**J WHERE J=0
EX-8      0.0**J WHERE J < 0
EX-9      X**Y WHERE X < 0.0, Y ≠ 0.0

 ENTRY STATEMENT
EY-0      SUBPROGRAMME NAME IN ENTRY STATEMENT PREVIOUSLY DEFINED
EY-1      PREVIOUS DEFINITION OF FUNCTION NAME IN AN ENTRY IS INCORRECT
EY-2      USE OF SUBPROGRAMME PARAMETER INCONSISTENT WITH PREVIOUS ENTRY
          POINT
EY-3      ARGUMENT NAME HAS APPEARED IN AN EXECUTABLE STATEMENT BUT WAS NOT A
          SUBPROGRAMME PARAMETER
EY-4      ENTRY STATEMENT NOT PERMITTED IN MAIN PROGRAMME
EY-5      ENTRY POINT INVALID INSIDE A DO-LOOP
EY-6      VARIABLE WAS NOT PREVIOUSLY USED AS A PARAMETER - PARAMETER ASSUMED

 FORMAT
  SOME FORMAT ERROR MESSAGES GIVE CHARACTERS IN WHICH ERROR WAS DETECTED
FM-0      INVALID CHARACTER IN INPUT DATA
FM-2      NO STATEMENT NUMBER ON A FORMAT STATEMENT
FM-5      FORMAT SPECIFICATION AND DATA TYPE DO NOT MATCH
FM-6      INCORRECT SEQUENCE OF CHARACTERS IN INPUT DATA
FM-7      NON TERMINATING FORMAT

FT-0      FIRST CHARACTER OF VARIABLE FORMAT NOT A LEFT PARENTHESIS
FT-1      INVALID CHARACTER ENCOUNTERED IN FORMAT
FT-2      INVALID FORM FOLLOWING A SPECIFICATION
FT-3      INVALID FIELD OR GROUP COUNT
FT-4      A FIELD OR GROUP COUNT GREATER THAN 255
FT-5      NO CLOSING PARENTHESIS ON VARIABLE FORMAT
FT-6      NO CLOSING QUOTE IN A HOLLERITH FIELD
FT-7      INVALID USE OF COMMA
FT-8      INSUFFICIENT SPACE TO COMPILE A FORMAT STATEMENT (SCAN-STACK
```

```
          OVERFLOW)
FT-9      INVALID USE OF P SPECIFICATION
FT-A      CHARACTER FOLLOWS CLOSING RIGHT PARENTHESIS
FT-B      INVALID USE OF PERIOD(.)
FT-C      MORE THAN THREE LEVELS OF PARENTHESES
FT-D      INVALID CHARACTER BEFORE A RIGHT PARENTHESIS
FT-E      MISSING OR ZERO LENGTH HOLLERITH ENCOUNTERED
FT-F      NO CLOSING RIGHT PARENTHESIS

 FUNCTIONS AND SUBROUTINES
FN-0      NO ARGUMENTS IN A FUNCTION STATEMENT
FN-3      REPEATED ARGUMENT IN SUBPROGRAM OR STATEMENT FUNCTION DEFINITION
FN-4      SUBSCRIPTS ON RIGHT HAND SIDE OF STATEMENT FUNCTION
             PROBABLE CAUSE - VARIABLE TO LEFT OF = NOT DIMENSIONED
FN-5      MULTIPLE RETURNS ARE INVALID IN FUNCTION SUBPROGRAMMES
FN-6      ILLEGAL LENGTH MODIFIER IN TYPE FUNCTION STATEMENT
FN-7      INVALID ARGUMENT IN ARITHMETIC OR LOGICAL STATEMENT FUNCTION
FN-8      ARGUMENT OF SUBPROGRAMME IS SAME AS SUBPROGRAMME NAME

 GO TO STATEMENTS
GO-0      STATEMENT TRANSFERS TO ITSELF OR A NON-EXECUTABLE STATEMENT
GO-1      INVALID TRANSFER TO THIS STATEMENT
GO-2      INDEX OF COMPUTED 'GO TO' IS NEGATIVE OR UNDEFINED
GO-3      ERROR IN VARIABLE OF 'GO TO' STATEMENT
GO-4      INDEX OF ASSIGNED 'GO TO' IS UNDEFINED OR NOT IN RANGE

 HOLLERITH CONSTANTS
HO-0      ZERO LENGTH SPECIFIED FOR H-TYPE HOLLERITH
HO-1      ZERO LENGTH QUOTE-TYPE HOLLERITH
HO-2      NO CLOSING QUOTE OR NEXT CARD NOT CONTINUATION CARD
HO-3      HOLLERITH CONSTANT SHOULD APPEAR ONLY IN CALL STATEMENT
HO-4      UNEXPECTED HOLLERITH OR STATEMENT NUMBER CONSTANT

 IF STATEMENTS (ARITHMETIC AND LOGICAL)
IF-0      STATEMENT INVALID AFTER A LOGICAL IF
IF-3      ARITHMETIC OR INVALID EXPRESSION IN LOGICAL IF
IF-4      LOGICAL, COMPLEX, OR INVALID EXPRESSION IN ARITHMETIC IF

 IMPLICIT STATEMENT
IM-0      INVALID MODE SPECIFIED IN AN IMPLICIT STATEMENT
IM-1      INVALID LENGTH SPECIFIED IN AN IMPLICIT OR TYPE STATEMENT
IM-2      ILLEGAL APPEARANCE OF $ IN A CHARACTER RANGE
IM-3      IMPROPER ALPHABETIC SEQUENCE IN CHARACTER RANGE
IM-4      SPECIFICATION  MUST  BE  SINGLE ALPHABETIC CHARACTER, 1ST CHARACTER
          USED
IM-5      IMPLICIT STATEMENT DOES NOT PRECEDE OTHER SPECIFICATION STATEMENTS
IM-6      ATTEMPT TO ESTABLISH THE TYPE OF A CHARACTER MORE THAN ONCE
IM-7      /360 FORTRAN ALLOWS ONE IMPLICIT STATEMENT PER PROGRAMME
IM-8      INVALID ELEMENT IN IMPLICIT STATEMENT
IM-9      INVALID DELIMITER IN IMPLICIT STATEMENT

 INPUT/OUTPUT
IO-0      MISSING COMMA IN I/O LIST OF I/O OR DATA STATEMENT
```

IO-2 STATEMENT NUMBER IN I/O STATEMENT NOT A FORMAT STATEMENT NUMBER
IO-3 BUFFER OVERFLOW - LINE TOO LONG FOR DEVICE
IO-6 VARIABLE FORMAT NOT AN ARRAY NAME
IO-8 INVALID ELEMENT IN INPUT LIST OR DATA LIST
IO-9 TYPE OF VARIABLE UNIT NOT INTEGER IN I/O STATEMENTS
IO-A HALF-WORD INTEGER VARIABLE USED AS UNIT IN I/O STATEMENTS
IO-B ASSIGNED INTEGER VARIABLE USED AS UNIT IN I/O STATEMENTS
IO-C INVALID ELEMENT IN AN OUTPUT LIST
IO-D MISSING OR INVALID UNIT IN I/O STATEMENT
IO-E MISSING OR INVALID FORMAT IN READ/WRITE STATEMENT
IO-F INVALID DELIMITER IN SPECIFICATION PART OF I/O STATEMENT
IO-G MISSING STATEMENT NUMBER AFTER END= OR ERR=
IO-H /360 FORTRAN DOESN'T ALLOW END/ERR RETURNS IN WRITE STATEMENTS
IO-J INVALID DELIMITER IN I/O LIST
IO-K INVALID DELIMITER IN STOP, PAUSE, DATA, OR TAPE CONTROL STATEMENT

JOB CONTROL CARDS
JB-1 COMPILE CARD ENCOUNTERED DURING COMPILATION
JB-2 INVALID OPTION(S) SPECIFIED ON COMPILE CARD
JB-3 UNEXPECTED CONTROL CARD ENCOUNTERED DURING COMPILATION

JOB TERMINATION
KO-0 JOB TERMINATED IN EXECUTION BECAUSE OF COMPILE TIME ERROR
KO-1 FIXED-POINT DIVISION BY ZERO
KO-2 FLOATING-POINT DIVISION BY ZERO
KO-3 TOO MANY EXPONENT OVERFLOWS
KO-4 TOO MANY EXPONENT UNDERFLOWS
KO-5 TOO MANY FIXED-POINT OVERFLOWS
KO-6 JOB TIME EXCEEDED
KO-7 COMPILER ERROR - INTERRUPTION AT EXECUTION TIME, RETURN TO SYSTEM

LOGICAL OPERATION
LG-2 .NOT. USED AS A BINARY OPERATOR

LIBRARY ROUTINES
LI-0 ARGUMENT OUT OF RANGE DGAMMA OR GAMMA. (1.382E-76 < X < 57.57)
LI-1 ABSOLUTE VALUE OF ARGUMENT > 174.673, SINH,COSH,DSINH,DCOSH
LI-2 SENSE LIGHT OTHER THAN 0,1,2,3,4 FOR SLITE OR 1,2,3,4 FOR SLITET
LI-3 REAL PORTION OF ARGUMENT > 174.673, CEXP OR CDEXP
LI-4 ABS(AIMAG(Z)) > 174.673 FOR CSIN, CCOS, CDSIN OR CDCOS OF Z
LI-5 ABS(REAL(Z)) ≥ 3.537E15 FOR CSIN, CCOS, CDSIN OR CDCOS OF Z
LI-6 ABS(AIMAG(Z)) ≥ 3.537E15 FOR CEXP OR CDEXP OF Z
LI-7 ARGUMENT > 174.673, EXP OR DEXP
LI-8 ARGUMENT IS ZERO, CLOG, CLOG10, CDLOG OR CDLG10
LI-9 ARGUMENT IS NEGATIVE OR ZERO, ALOG, ALOG10, DLOG OR DLOG10
LI-A ABS(X) ≥ 3.537E15 FOR SIN, COS, DSIN OR DCOS OF X
LI-B ABSOLUTE VALUE OF ARGUMENT > 1, FOR ARSIN, ARCOS, DARSIN OR DARCOS
LI-C ARGUMENT IS NEGATIVE, SQRT OR DSQRT
LI-D BOTH ARGUMENT OF DATAN2 OR ATAN2 ARE ZERO
LI-E ARGUMENT TOO CLOSE TO A SINGULARITY, TAN, COTAN, DTAN OR DCOTAN
LI-F ARGUMENT OUT OF RANGE DLGAMMA OR ALGAMA. (0.0 < X < 4.29E73)
LI-G ABSOLUTE VALUE OF ARGUMENT ≥ 3.537E15, TAN, COTAN, DTAN, DCOTAN
LI-H FEWER THAN TWO ARGUMENTS FOR ONE OF MIN0, MIN1, AMIN0, ETC.

MIXED MODE
MD-2 RELATIONAL OPERATOR HAS A LOGICAL OPERAND
MD-3 RELATIONAL OPERATOR HAS A COMPLEX OPERAND
MD-4 MIXED MODE - LOGICAL WITH ARITHMETIC
MD-6 WARNING - SUBSCRIPT IS COMPLEX

MEMORY OVERFLOW
MO-0 SYMBOL TABLE OVERFLOWS OBJECT CODE. SOURCE ERROR CHECKING CONTINUES
MO-1 INSUFFICIENT MEMORY TO ASSIGN ARRAY STORAGE. JOB ABANDONED
MO-2 SYMBOL TABLE OVERFLOWS COMPILER, JOB ABANDONED
MO-3 DATA AREA OF SUBPROGRAMME TOO LARGE -- SEGMENT SUBPROGRAMME

PARENTHESES
PC-0 UNMATCHED PARENTHESIS
PC-1 INVALID PARENTHESIS NESTING IN I/O LIST

PAUSE, STOP STATEMENTS
PS-0 STOP WITH OPERATOR MESSAGE NOT ALLOWED. SIMPLE STOP ASSUMED
PS-1 PAUSE WITH OPERATOR MESSAGE NOT ALLOWED. TREATED AS CONTINUE

RETURN STATEMENT
RE-0 FIRST CARD OF SUBPROGRAM IS A RETURN STATEMENT
RE-1 RETURN I, WHERE I IS ZERO, NEGATIVE, OR TOO LARGE
RE-2 MULTIPLE RETURN NOT VALID IN FUNCTION SUBPROGRAMME
RE-3 VARIABLE IN MULTIPLE RETURN IS NOT A SIMPLE INTEGER VARIABLE
RE-4 MULTIPLE RETURN NOT VALID IN MAIN PROGRAMME

ARITHMETIC AND LOGICAL STATEMENT FUNCTIONS
PROBABLE CAUSE OF SF ERRORS - VARIABLE ON LEFT OF = WAS NOT DIMENSIONED
SF-1 PREVIOUSLY REFERENCED STATEMENT NUMBER ON STATEMENT FUNCTION
SF-2 STATEMENT FUNCTION IS THE OBJECT OF A LOGICAL IF STATEMENT
SF-3 RECURSIVE STATEMENT FUNCTION, NAME APPEARS ON BOTH SIDES OF =

SUBPROGRAMMES
SR-0 MISSING SUBPROGRAMME
SR-2 SUBPROGRAMME ASSIGNED DIFFERENT MODES IN DIFFERENT PROGRAMME SEGMENTS
SR-4 INVALID TYPE OF ARGUMENT IN SUBPROGRAMME REFERENCE
SR-5 SUBPROGRAM ATTEMPTS TO REDEFINE A CONSTANT, TEMPORARY OR DO PARAMETER
SR-6 ATTEMPT TO USE SUBPROGRAMME RECURSIVELY
SR-7 WRONG NUMBER OF ARGUMENTS IN SUBPROGRAMME REFERENCE
SR-8 SUBPROGRAM NAME PREVIOUSLY DEFINED -- FIRST REFERENCE USED
SR-9 NO MAIN PROGRAM
SR-A ILLEGAL OR BLANK SUBPROGRAMME NAME

SUBSCRIPTS
SS-0 ZERO SUBSCRIPT OR DIMENSION NOT ALLOWED
SS-1 SUBSCRIPT OUT OF RANGE
SS-2 INVALID VARIABLE OR NAME USED FOR DIMENSION

STATEMENTS AND STATEMENT NUMBERS

ST-0 MISSING STATEMENT NUMBER
ST-1 STATEMENT NUMBER GREATER THAN 99999
ST-3 MULTIPLY-DEFINED STATEMENT NUMBER
ST-4 NO STATEMENT NUMBER ON STATEMENT FOLLOWING TRANSFER STATEMENT
ST-5 UNDECODEABLE STATEMENT
ST-7 STATEMENT NUMBER SPECIFIED IN A TRANSFER IS A NON-EXECUTABLE STATEMENT
ST-8 STATEMENT NUMBER CONSTANT MUST BE IN A CALL STATEMENT
ST-9 STATEMENT SPECIFIED IN A TRANSFER STATEMENT IS A FORMAT STATEMENT
ST-A MISSING FORMAT STATEMENT

SUBSCRIPTED VARIABLES
SV-0 WRONG NUMBER OF SUBSCRIPTS
SV-1 ARRAY NAME OR SUBPROGRAMME NAME USED INCORRECTLY WITHOUT LIST
SV-2 MORE THAN 7 DIMENSIONS NOT ALLOWED
SV-3 DIMENSION TOO LARGE
SV-4 VARIABLE WITH VARIABLE DIMENSIONS IS NOT A SUBPROGRAMME PARAMETER
SV-5 VARIABLE DIMENSION NEITHER SIMPLE INTEGER VARIABLE NOR S/P PARAMETER

SYNTAX ERRORS
SX-0 MISSING OPERATOR
SX-1 SYNTAX ERROR-SEARCHING FOR SYMBOL,NONE FOUND
SX-2 SYNTAX ERROR-SEARCHING FOR CONSTANT, NONE FOUND
SX-3 SYNTAX ERROR-SEARCHING FOR SYMBOL OR CONSTANT, NONE FOUND
SX-4 SYNTAX ERROR-SEARCHING FOR STATEMENT NUMBER, NONE FOUND
SX-5 SYNTAX ERROR-SEARCHING FOR SIMPLE INTEGER VARIABLE, NONE FOUND
SX-C ILLEGAL SEQUENCE OF OPERATORS IN EXPRESSION
SX-D MISSING OPERAND OR OPERATOR

I/O OPERATIONS
UN-0 CONTROL CARD ENCOUNTERED ON SCARDS DURING EXECUTION
PROBABLE CAUSE - MISSING DATA OR IMPROPER FORMAT STATEMENTS
UN-1 END OF FILE ENCOUNTERED
UN-2 I/O ERROR
UN-3 ILLEGAL DATA SET REFERENCE NUMBER. NOT SPECIFIED ON "$RUN"
UN-4 REWIND, ENDFILE, BACKSPACE REFERENCES UNIT SCARDS, SPRINT, SPUNCH
UN-5 ATTEMPT TO READ ON UNIT SCARDS AFTER IT HAS HAD END-OF-FILE
UN-6 UNIT NUMBER IS NEGATIVE, ZERO, OR GREATER THAN 9
UN-7 TOO MANY PAGES
UN-8 ATTEMPT TO DO SEQUENTIAL I/O ON A DIRECT ACCESS FILE
UN-9 WRITE REFERENCES SCARDS OR READ REFERENCES SPRINT OR SPUNCH

UNDEFINED VARIABLES
UV-0 UNDEFINED VARIABLE - SIMPLE VARIABLE
UV-1 UNDEFINED VARIABLE - EQUIVALENCED, COMMONED, OR DUMMY PARAMETER
UN-2 UNDEFINED VARIABLE - ARRAY MEMBER
UV-3 UNDEFINED VARIABLE - ARRAY NAME WHICH WAS USED AS A DUMMY PARAMETER
UV-4 UNDEFINED VARIABLE - SUBPROGRAMME NAME USED AS DUMMY PARAMETER
UV-5 UNDEFINED VARIABLE - ARGUMENT OF THE LIBRARY SUBPROGRAMME NAMED
UV-6 VARIABLE FORMAT CONTAINS UNDEFINED CHARACTER(S)

VARIABLE NAMES

VA-0 ATTEMPT TO REDEFINE TYPE OF A VARIABLE NAME
VA-1 SUBROUTINE NAME OR COMMON BLOCK NAME USED INCORRECTLY
VA-2 VARIABLE NAME LONGER THAN SIX CHARACTERS. TRUNCATED TO SIX
VA-3 ATTEMPT TO REDEFINE THE MODE OF A VARIABLE NAME
VA-4 ATTEMPT TO REDEFINE THE TYPE OF A VARIABLE NAME
VA-6 ILLEGAL USE OF A SUBROUTINE NAME
VA-8 ATTEMPT TO USE A PREVIOUSLY DEFINED NAME AS FUNCTION OR ARRAY
VA-9 ATTEMPT TO USE A PREVIOUSLY DEFINED NAME AS A STATEMENT FUNCTION
VA-A ATTEMPT TO USE A PREVIOUSLY DEFINED NAME AS A SUBPROGRAMME NAME
VA-B NAME USED AS A COMMON BLOCK PREVIOUSLY USED AS A SUBPROGRAM NAME
VA-C NAME USED AS SUBPROGRAMME PREVIOUSLY USED AS A COMMON BLOCK NAME

EXTERNAL STATEMENT
XT-0 INVALID ELEMENT IN EXTERNAL LIST
XT-1 INVALID DELIMITER IN EXTERNAL STATEMENT
XT-2 SUBPROGRAMME PREVIOUSLY EXTERNALLED

MTS-590-0

12-1-67

# 8 A S S

# 8ASS -- PDP-8 ASSEMBLER

## INTRODUCTION

The following sections describe the PDP-8 Assembler (8ASS), which is a collection of programs written mostly in FORTRAN IV (G) and operating under the Michigan Terminal System (MTS) on the IBM 360/67. 8ASS assembles programs for the Digital Equipment Company (DEC) PDP-5 and PDP-8 computers. Once a program has been assembled, it may be punched on cards, saved in a file, or transmitted through the Data Concentrator over data lines. It is also possible to obtain binary paper tapes by use of the Data Concentrator.

The reader is assumed to be familiar with the reference manual for the PDP-8 available from DEC. ("Programmed Data Processor-8 Users Handbook," (DIGITAL F-85), Digital Equipment Corporation, Maynard, Mass., 1964.) For the description and use of assemblers in general the reader is referred to the description of the PAL-III assembler for the PDP-8 available from DEC ("PAL-III Symbolic Assembler Programming Manual," (DIGITAL 8/3/S), Digital Equipment Corporation, Maynard, Mass., 1965). 8ASS follows the PAL-III operation code and addressing conventions. The input format and program listing conventions of 8ASS are slightly different from those PAL-III, however, since 8ASS is organized around a line format while PAL-III is organized around a paper tape format.

## ASSEMBLY PROCESSING

An assembler is a vehicle for the transformation of symbolic source programs into the internal representation of machine instructions and data. Each PDP-8 machine instruction occupies exactly one location in its memory. The assembly language program is a sequence of input lines to the assembler which specifies these instructions in symbolic form. The assembler reads these lines and constructs, or assembles, corresponding PDP-8 binary words.

Symbolic names for the PDP-8 memory locations are defined by the appearance of a name at the beginning of an input line. Symbolic names for operation codes appear next, sometimes followed by operands. The assembler lists a value corresponding to the value of the operator, augmented by the value of the operand. Each such value is associated with a PDP-8 address by means of the instruction location counter (ILC). The ILC contains a value which is incremented modulo 4096 after each PDP-8 word is generated. Normally, therefore, assembled words are placed in sequentially ascending locations in PDP-8 memory.

Some input lines do not generate PDP-8 words, but activate internal procedures in 8ASS. Several names which may appear in the operand are not operation codes but procedure calls. For example the procedure call ORG resets the value of the ILC, allowing the programmer to control the starting location of a block of words.

The symbolic information on each assembly language line is grouped into four fields: the label, operation code (opcode), operand and comment fields. These fields are delimited by blanks.

The label field starts at character 1 and is terminated by the first blank. If it is non-empty it may contain a name of up to eight characters, beginning with a letter. Any variable used in the program must be defined by its appearance in the label field, and the variables used with some procedure calls must be predefined, that is, defined at some point before the procedure call is processed.

The opcode field is the expression starting with the first non-blank character after the label field, and ending with the next blank. Any variable appearing in the opcode field must be an operation code.

If the operation code is a microinstruction or a self-defining expression, the operand field is empty. Otherwise, the operand field starts with the first non-blank character after the opcode field, and ends with the next blank. Any variable appearing in the operand field must be a label.

The three fields discussed above may extend to the 72nd character. The comment field starts at the end of the operand field and may extend through the 80th character. It has no effect on the binary output of the assembler--it is merely copied onto the assembly listing--but is useful to the programmer as a method of documentation. If the first character of the source line is an asterisk (*), the label, opcode and operand field are all empty and the card is just copied onto the output listing.

There are two kinds of output from the assembler, a binary "deck" and an assembly listing. The former is a list of the machine program in a form appropriate for loading into the PDP-8 computer. The latter, the listing, not only provides the programmer and operator of the PDP-8 with what can be an invaluable guide to the operation of the program, but also indicates some types of possible programming errors.

## 8ASS IN MTS

The PDP-8 Assembler is available as a library file in the Michigan Terminal System (MTS). Its use is invoked by the $RUN command, with the following logical devices specified:

1 The assembly language input lines.

2 A table of opcodes (the library file *8OPS).

6 A tape or file (rewindable) for intermediate storage.

8 The assembly listing (output).

SPUNCH The binary output (card format).

Example:

```
$RUN *8ASS;1=*SOURCE* 2=8OPS 6=-F 8=*SINK* SPUNCH=*PUNCH*
```

Due to internal size limitations, the size of program which can be assembled is limited. If a program defines S symbols and refers to symbols R times (including uses of operation codes and procedure calls), S and R must satisfy:

$$10(S + 65) + 2R < M$$

For *8ASS , M=30,000.

## NAMES AND EXPRESSIONS

A program name is a symbol which stands for a numeric value. It may stand for a self-defining value, in which case it is called a constant, or it may stand for a value which is defined elsewhere, in which case it is called a variable. A variable may be an opcode, in which case it is defined from the input table *8OPS (see previous section) or by use of the procedure calls OPD or OPDM, or it may be label, in which case it is defined by its appearance in the label field of some input line. If this line corresponds to a PDP-8 memory location, the defined value of the label is the address of the location; if the operation field of the line is the procedure call EQU, the defined value of the label is the value of the expression in the operand field.

The special program name, *, is self-defining. Its value is the current contents of the ILC (the value "here").

The following EBCDIC characters may be used in the formation of names and expressions:

| | |
|---|---|
| Alphabetic | upper case letters A-Z |
| Numeric | digits 0-9 |

| | |
|---|---|
| Operators | + - (plus, minus) |
| Delimiters | expression field delimiter (blank); comment field delimiter ; (seimicolon) |
| Literal prefix | = |

Program names must be less than nine characters long. Variables may contain alphabetic and numeric characters, but must begin with a letter. Constants must start with a digit and may contain only digits.

An expression is a sequence of program names, separated by the operators + and -, and is delimited by blanks. In the opcode field, any variables must be opcodes or procedure calls; in the operand field, any variables must be labels. The assembler evaluates the expression from left to right by combining the values of the names according to the operators. In the opcode field, and in the operand field of an OPD or OPDM line, the operator + combines values by the logical OR operation. In the operand field of other procedure calls and memory-reference instructions, the values are combined arithmetically (+ for addition, - for 2's complement subtraction) modulo 4096.

An operand-field expression may be prefixed with an equal sign (=) which designates an occurrence of a literal. The value of the expression itself is termed the value of the literal, and the location to which it is assigned is termed its address. All such literal occurrences are saved in a special pool during assembler processing. When a LIT procedure call is encountered, this pool is assigned machine locations while multiple occurrences of the same value are suppressed. All literal occurrences up to this point are replaced with addresses which point to the assigned value. All symbols used in a literal expression must be predefined.

When an expression is evaluated in the operand field of a memory-reference instruction, a check is made to determine whether the value of the expression is within the current memory page. If it is then the same-page bit of the assembled instruction is set to one. If a memory-reference instruction opcode expression is immediately followed by an asterisk *, then the indirect bit of the assembled instruction is set to one. The I and Z conventions of PAL-III are invalid in 8ASS.

## INSTRUCTIONS AND PROCEDURE CALLS

A standard set of PDP-8 instruction codes is defined into the *8ASS internal symbol table from an external table such as *8OPS. The opcodes in the list *8OPS include the memory-reference instructions; microinstructions (Group 1, and Group 2 operate instructions, the extended arithmetic (EAE) instructions, the Teletype IOT instructions); and a number or procedure

calls. The machine instruction codes and their values are listed in Appendix I.

Combined microinstructions can be written as an opcode expression of microinstructions separated by + operators. This has the effect of forming the inclusive OR of the respective values. New instructions can be defined with the OPD and OPDM procedure calls.

Procedure calls are opcodes which do not represent PDP-8 machine instruction, but are signals to the assembler to invoke special procedures. The procedure calls (also know as pseudo-operations, or pseudo-ops) of *8ASS and the effects of their procedures are summarized below.

DC - defined constant

Define the (optional) symbol in the label field to have a value equal to the current contents of the instruction location counter (ILC). Then substitute the value of the expression in the operand field itself for the memory location signified by the current ILC. (The DC pseudo-op provides the facility for defining decimal, octal, or address constants in a fashion paralleling the PAL-III custom of placing the name of the constant itself in the operation field.)

DECMOD - define constant conversion mode decimal

Set constant conversion to the decimal radix (normal mode is octal). May be used alternately with the OCTMOD procedure call any number of times in a program. Note - If any constant is followed by one of the letters K or D, then that constant is assumed of radix eight or ten, respectively, regardless of the current mode.

DS - define storage

Define the (optional) symbol in the label field to have a value equal to the current instruction location counter (ILC). Then add the value of the expression (predefined) in the operand field to the ILC.

END - end assembly

(Identical to the $ function of PAL-III) Define the (optional) symbol in the label field to have a value equal to the current instruction location counter (ILC). If the operand expression is non-null, then its value will be punched on a binary transfer card as the starting address of the program.

EQU - symbolic equivalence

Define the name in the label field to have a value equal to that of the expression (predefined) in the operand field. (Similar to the = function of PAL-III)

LIT - begin literal pool

Begin assignment of literals collected so far in the program.

OCTMOD - define constant conversion mode

Set constant conversion to the octal radix (normal mode). May be used alternately with the DECMOD procedure call any number of times in a program.

OPD - operation code definition

Define the name in the label field to designate an instruction which has an operation code equal to the value of the expression (predefined) in the operand field. (Note - The operation and symbol tables of the 8ASS assembler are disjoint so that name conflict can be avoided. In the PAL-III assembler this is not the case - operation names used in the operand fields must be disjoint.)

OPDM - memory-reference instruction code definition

Operates identically to the OPD pseudo-op except that the operation code is presumed to designate a memory-referenced instruction.

ORG - reset instruction location counter

Reset instruction location counter (ILC) to the value of the expression (predefined) in the operand field. (Identical to the * function in PAL-III)

## DEBUGGING AIDS

When the assembler can detect an irresolvable ambiguity or inconsistency, it prints error comments on the assembly listing. Typical comments and their meanings are listed below.

"MULTIPLY DEFINED SYMBOL nnnnnnnn xxxx VARIABLE" or "...OPCODE". The name nnnnnnnn was defined more than once as a variable by its appearance in

the label field and/or by the EQU procedure call, or more than once as an opcode by its appearance in the standard instruction table (*8OPS) and/or by the procedure call OPD or OPDM. In any case, the line is printed, with xxxx equal to the defined value, once for each definition. These comments are printed before the assembly listing; the four listed below are printed just before the line to which they apply. The value punched and listed for the appropriate ILC value is probably wrong.

"UNDEFINED PROGRAM NAME." During the evaluation of an expression, a name was encountered which was not defined in the program. Note that names in some procedure calls, and in literal expressions, must be predefined.

"OFF-PAGE REFERENCE." The value of the operand expression of a memory-reference instruction is neither an address on page 0 nor an address on the current page.

"INVALID OPERATOR EXPRESSION." The expression in the operator field is invalid. For example, there may be a label in the expression.

"OPERATOR-OPERAND CONFLICT." The opcodes given are incompatible, or the operator and operand are. For example, the invalid operator expression "OSR+RAR" has a group 2 and a group 1 opcode.

A cross-reference table is printed at the end of the assembly. It lists each variable (label or opcode) used by the program, along with its value and the contents of the ILC at each time it was used.

A summary of the number of error comments printed, the number of source lines processed, the number of symbols defined (including the standard table), the number of references to defined symbols, and the number of card images produced follows the cross-reference table.

## OBJECT DECKS

8ASS produces column binary card images suitable for punching and/or loading into a PDP-8. Text cards contain numbers to be loaded into PDP-8 memory. A transfer card is produced by the END procedure call, if its operand field is nonempty. The transfer card is usually used to specify a starting address for the PDP-8 program. The format of a text card is, by column:

col. 1 — a 6-7-9 punch, indicating a text card

col. 2 — N , the number of contiguous PDP-8 words specified by this card (N≤68)

col. 3. — the address of the first word in the block

| | |
|---|---|
| col. 4 | consecutive PDP-8 word values |
| col. 3+N | same as above |
| col. 4+N | a checksum, the arithmetic sum of columns 2 through 3+N, modulo 4096. |

The format of a transfer card is:

| | |
|---|---|
| col. 1 | a 5-7-9 punch, indicating a transfer card |
| col. 2 | 0 |
| col. 3 | the starting address of the program |
| col. 4 | a checksum |

## APPENDIX 1: 8ASS STANDARD OPCODES

The following opcodes are defined as standard from the table *8OPS. The codes are octal.

### I. Memory Reference Instructions

These opcodes may carry the indirect reference modifier, *, and take an operand in which any name must be a label.

| NAME | CODE |
|---|---|
| AND | 0000 |
| DCA | 3000 |
| ISZ | 2000 |
| JMP | 5000 |
| JMS | 4000 |
| TAD | 1000 |

### II. Microinstructions.

#### A. Input-Output Instructions (IOT'S)

| NAME | CODE |
|---|---|
| IOF | 6002 |
| ION | 6001 |
| IOT | 6000 |
| KCC | 6032 |
| KRB | 6036 |
| KRS | 6034 |
| KSF | 6031 |
| TCF | 6042 |
| TLS | 6046 |
| TSF | 6041 |

#### B. Group I Operate Instructions

| NAME | CODE |
|---|---|
| CIA | 7041 |
| CLA | 7200 |
| CLL | 7100 |
| CMA | 7020 |
| GLK | 7204 |
| IAC | 7001 |
| NOP | 7000 |
| OPR | 7000 |
| RAL | 7004 |
| RAR | 7010 |
| RTL | 7006 |

| | |
|---|---|
| RTR | 7012 |
| STA | 7240 |
| STL | 7120 |

C. Group II Operate Instruction

| NAME | CODE |
|---|---|
| CLA | 7600 |
| HLT | 7402 |
| LAS | 7604 |
| OSR | 7404 |
| SKP | 7410 |
| SNL | 7420 |
| SMA | 7500 |
| SNA | 7450 |
| SPA | 7510 |
| SZA | 7440 |
| SZL | 7430 |

D. Extended Arithmetic Element

| NAME | CODE | |
|---|---|---|
| ASR | 7415 | |
| CAM | 7621 | |
| CLA | 7601 | (when combined with other EAE'S) |
| DVI | 7407 | |
| LSR | 7417 | |
| MQA | 7501 | |
| MQL | 7421 | |
| MUY | 7405 | |
| NMI | 7411 | |
| SCA | 7441 | |
| SHL | 7413 | |

Indexed Words and Phrases

Indexed Words and Phrases

Indexed Words and Phrases

Indexed Words and Phrases

Indexed Words and Phrases

Indexed Words and Phrases