M T S

The Michigan Terminal System

Volume 9: SNOBOL4 in MTS

September 1975

Updated June 1979 (Update 1)
Updated May 1984 (Update 2)

The University of Michigan Computing Center
Ann Arbor, Michigan

DISCLAIMER

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

September 1975 Page Revised May 1984

PREFACE

The software developed by the Computing Center staff for the operation of the high-speed IBM 370-compatible computers can be described as a multiprogramming supervisor that handles a number of resident, reentrant programs. Among them is a large subsystem, called MTS (Michigan Terminal System), for command interpretation, execution control, file management, and accounting maintenance. Most users interact with the computer's resources through MTS.

The MTS Manual is a series of volumes that describe in detail the facilities provided by the Michigan Terminal System. Administrative policies of the Computing Center and the physical facilities provided are described in a separate publication entitled Introduction to Computing Center Services.

The MTS volumes now in print are listed below. The date indicates the most recent edition of each volume; however, since volumes are periodically updated, users should check the file *CCPUBLICATIONS, or watch for announcements in the Computing Center Newsletter, to ensure that their MTS volumes are fully up to date.

| Volume 1: The Michigan Terminal System, January 1984
| Volume 2: Public File Descriptions, April 1982
| Volume 3: System Subroutine Descriptions, April 1981
| Volume 4: Terminals and Networks in MTS, March 1984
| Volume 5: System Services, May 1983
| Volume 6: FORTRAN in MTS, October 1983
| Volume 7: PL/I in MTS, September 1982
| Volume 8: LISP and SLIP in MTS, June 1976
| Volume 9: SNOBOL4 in MTS, September 1975
| Volume 10: BASIC in MTS, December 1980
| Volume 11: Plot Description System, August 1978
| Volume 12: PIL/2 in MTS, December 1974
| Volume 13: The Symbolic Debugging System, November 1980
| Volume 14: 360/370 Assemblers in MTS, May 1983
| Volume 15: FORMAT and TEXT360, April 1977
| Volume 16: ALGOL W in MTS, September 1980
| Volume 17: Integrated Graphics System, December 1980
| Volume 18: The MTS File Editor, September 1982
| Volume 19: Tapes and Floppy Disks, February 1983

Other volumes are in preparation. The numerical order of the volumes does not necessarily reflect the chronological order of their appearance; however, in general, the higher the number, the more specialized the volume. Volume 1, for example, introduces the user to MTS and describes in general the MTS operating system, while Volume 10 deals exclusively with BASIC.

Page Revised May 1984 September 1975

The attempt to make each volume complete in itself and reasonably independent of others in the series naturally results in a certain amount of repetition. Public file descriptions, for example, may appear in more than one volume. However, this arrangement permits the user to buy only those volumes that serve his or her immediate needs.

Richard A. Salisbury

General Editor

Contents

OVERVIEW OF SNOBOL4 IN MTS

The SNOBOL4 language was developed at Bell Telephone Laboratories as a tool for string and list-processing applications. As a consequence, the language contains many powerful facilities for solving problems which involve text analysis, text formatting, etc. On the other hand, SNOBOL4 is a poor choice for solving problems which are primarily numeric in nature. Such problems are handled much more efficiently with mathematically oriented languages such as FORTRAN.

In MTS, there are four distinct SNOBOL4 language processors which are available in the public files *SNOBOL4, *SNOBOL4B, *SPITBOL, and *SNOSTORM. Each public file is described in detail in the appendix to this volume. A brief summary is given here.

*SNOBOL4 contains the language processor developed at Bell Telephone Laboratories and accepts the basic SNOBOL4 language. This processor is an interpreter and is implemented in a machine-independent macro language. This processor tends to be rather slow and expensive to use.

*SNOBOL4B contains the language processor developed at Bell Telephone Laboratories for an upward-compatible extension of the basic SNOBOL4 language. This extension is called SNOBOL4 with Blocks and provides additional facilities for producing 2- and 3-dimensional printed output, such as is required for graphs and flowcharts. This processor is also implemented as an interpreter and tends to be expensive to use.

*SPITBOL contains the language processor for a nearly compatible extension of the basic SNOBOL4 language which was developed at the Illinois Institute of Technology. This processor is implemented as a true compiler and produces fast and efficient object module programs. Programs compiled with *SPITBOL can be run as much as ten times cheaper than programs run under *SNOBOL4.

*SNOSTORM contains a preprocessor for an extended version of the SNOBOL4 language allowing structured-programming types of control structures. This preprocessor passes the SNOBOL4 program it generates to the *SPITBOL processor for completion of compilation and execution.

Documentation for the various SNOBOL4 languages and their associated language processors is available from several sources. The following information is included in this volume:

(1) Introduction to SNOBOL4

This section is a brief introduction to the basic SNOBOL4 language. It was written by Fred G. Swartz and and modified by Kenneth A. DeJong, both of the University of Michigan Computing Center.

(2) SPITBOL

This section is the only SPITBOL reference manual available. It was written at the Illinois Institute of Technology and adapted for use at the University of Michigan. It assumes a working knowledge of the basic SNOBOL4 language.

(3) SNOBOL4 Blocks

This section is the only description of the SNOBOL4 Blocks extension available. It was written at Bell Telephone Laboratories and adapted for use at the University of Michigan. It assumes a working knowledge of the basic SNOBOL4 language.

(4) SNOSTORM

This section describes the language extensions to SNOBOL4 for structured programming provided by the SNOSTORM preprocessor. This section is the only description of SNOSTORM available. It assumes a working knowledge of the basic SNOBOL4 language.

(5) Public File Descriptions

*CONVSNOBOL - a conversational SNOBOL4 program.
*SNOBOL4 - a SNOBOL4 processor.
*SNOBOL4B - an extended SNOBOL4 processor.
*SNOSTORM - a SNOBOL4 preprocessor.
*SPITBOL - a fast SNOBOL4 processor.
*SPITDEBUG - a debugging package for SPITBOL programs.
*SPITERR - SPITBOL error messages.
*SPITLIB - object-code support for SPITBOL programs.
*TRANSNOBOL - a SNOBOL3 to SNOBOL4 conversion program.

In addition, the following documentation is available through the local bookstores:

The SNOBOL4 Programming Language, R. E. Griswold, J. F. Poage, and I. P. Polonsky. Prentice-Hall (1971).

This is the standard reference manual for the basic SNOBOL4 language.

A SNOBOL4 Primer, R. E. Griswold and M. T. Griswold. Prentice-Hall (1973).

This provides an introduction to SNOBOL4 without assuming any previous programming experience.

SNOBOL - An Introduction to Programming, Peter R. Newsted. Hayden Book Co. (1975).

This also provides an introduction to SNOBOL4 without assuming any previous programming experience.

List and String Processing in SNOBOL4, R. E. Griswold. Prentice-Hall (1975).

This explores applications of the more sophisticated facilities in the SNOBOL4 language. It presumes a good working knowledge of the basic SNOBOL4 language.

Algorithms in SNOBOL4, J. F. Gimpel. John Wiley & Sons (1976).

This is an excellent text for the advanced SNOBOL4 user.

Page Revised June 1979 September 1975

INTRODUCTION TO SNOBOL4

This introduction is very limited in scope. It covers only a few of the very basic features of the SNOBOL4 language and leaves many areas uncovered. It is intended only to start the beginning user along the path of learning SNOBOL4.

The philosophy of SNOBOL4 differs from that of other programming languages such as FORTRAN or PL/I. All of the functions that most programming languages perform are crowded into a single statement format: computation, decision making, and branching are all parts of the one SNOBOL4 statement type. The strength of SNOBOL4 lies primarily in two areas: dynamic control (e.g., storage allocation) and pattern-matching (character manipulation) operations.

SNOBOL4 has dynamic storage allocation, something which few of today's programming languages have. Each SNOBOL4 variable does not represent a fixed area of storage; instead, as each variable needs more storage, it is acquired from a large pool of storage reserved for that purpose. If a variable no longer requires as much storage, the unneeded storage is released into the pool. The SNOBOL4 programmer remains blissfully ignorant of storage allocation problems and may assume that a variable may be of any size.

In most programming languages, both the storage and the data type (e.g., once an integer, always a integer) are assigned to a variable in a fixed manner. This is certainly not the case in SNOBOL4. A variable may have an integer value at one point in the program and a character string value in another part of the program. Moreover, not all the elements of an array need be of the same data type.

Another feature, one which the novice might hardly be expected to use, is that of converting character strings representing SNOBOL4 statements into executable SNOBOL4 code.

Arithmetic processing in SNOBOL4 is very slow. It would be sheer economic folly to attempt to solve a primarily numeric problem in SNOBOL4. The primary use of SNOBOL has been and will continue to be the manipulation of symbolic or structural data.

Page Revised June 1979 September 1975

SNOBOL4 CONSTANTS

There may be several types of constants in a SNOBOL4 program. The two most commonly used constant types are integer and string. The user is referred to the discussion of real constants in The SNOBOL4 Programming Language by Griswold, Poage, and Polonsky.

Integers in SNOBOL4 look very much like they do in any programming language: they consist of a string of digits.

```
EXAMPLES:  0   12345   0909   2
Illegal:   '234'  (this is a string, not an integer)
           1.3  (this is a real number, not an integer)
```

The string is the most important element in SNOBOL4. A string is a sequence of characters. The characters may be any that can be represented on any of the various computer input devices attached to the system. The examples contained in this writeup are comprised only of those characters which can be punched on an IBM 029 keypunch. In a string, all characters are treated identically. No characters have special meaning. Even blanks are treated the same as other characters. Since SNOBOL4 statements are themselves strings of characters, there must be some way to separate string constants from the rest of the SNOBOL4 statement. To do this, a string constant (sometimes the word "literal" is used to mean a string constant) is surrounded by quotation marks--either single or double, but the ending delimiter must be the same as the starting delimiter. One must remember that the enclosing quotes are not part of the string.

The delimiting character of a string constant may not occur within the constant. Since there are two possible delimiters, one can be used to enclose a string containing the other.

EXAMPLES

```
'THIS IS A STRING'

"THIS IS ALSO A STRING"
```

Something which hardly seems worthy of the name string is the null string. This is the string of no characters and can be written "". However, it turns out to be a very common string.

September 1975

QUESTIONS

Which of the following are legal string constants?

a) ’234’

b) "ABC’

c) "’"

d) ’DON’T QUOTE ME.’

e) " "

f) ’@#$%&*()_+:-=;,./"’

g) 945

h) ""

ANSWERS

a) OK.

b) Beginning and ending delimiters are not identical.

c) OK.

d) The delimiting character may not occur within the string.

e) OK; it is a string of two blanks.

f) Yes; all of these characters are legal in a string.

g) This is an integer, not a string.

h) The famous null string.

VARIABLES

A programming language containing only constants would be of little value and so, as would be expected, SNOBOL4 also has variables. A variable name must begin with an alphabetic character and may be continued with a sequence of alphabetic or numeric characters or periods. The initial contents of all variables are the null string. Later we will see how any string of characters (except the null string) may be used for a name.

EXAMPLES

```
ABC

DOT.DOT

EXPO67

S.O.S.
```

September 1975

QUESTIONS

Which of the following is a legal string name?

a) ALPHA-BETIC

b) PART.NO.

c) A

d) THIS.IS.A.NAME

e) .LT.

f) 1FOR.YOUR.MONEY

g) A...AL5)

h) LAPHN@

ANSWERS

a) Illegal character.

b) OK.

c) OK.

d) OK.

e) A variable name may not begin with a period.

f) A variable name must begin with an alphabetic character.

g) Illegal character.

h) Illegal character.

ASSIGNMENT

There are several ways in which values may be assigned to variables. One way is by means of a notation that looks very much like the assignment statement of FORTRAN, PL/I, etc. This is the form:

VARIABLE = EXPRESSION

For now, let us consider the simplest forms of expression, the variable and the constant.

The value[1] of the expression is computed and this value replaces any previous value that the variable may have had. The expression may be of any type and need not be of the same type as the current value of the variable. SNOBOL4 is very strict in its punctuation requirements. The equal sign must have one or more blanks immediately on each side of it.

EXAMPLES

```
A = 3 assigns the integer value 3 to A
A = '3'  assigns a one-character string to A
```

---

[1]The words "contents" and "value" are used interchangeably in this writeup.

September 1975

QUESTIONS

Which of the following assignments is legal?

a) GAMMA2 = 'TRALALA'

b) BETA2=GAMMA2

c) NOW = TIME

d) ETERNITY = ""

e) "THIS" = "THAT"

f) VAR = 5

g) 1HALF = ONEHALF

ANSWERS

a) OK.

b) There must be at least one blank on each side of the =.

c) OK.

d) OK. This replaces any previous contents of ETERNITY by the null string. Another way to do this is to write ETERNITY = . If nothing is written to the right of the =, the null string is assigned to the variable on the left.

e) You cannot assign a new value to a constant.

f) OK.

g) 1HALF is an illegal variable name.

ARITHMETIC OPERATIONS

The assignments we have seen so far tend to be rather dull. More interesting, although perhaps not more exciting, are expressions formed by combining several elements by means of operations. As might be expected from any reasonable compiler, SNOBOL4 allows the normal arithmetic operations; they mean what one would expect and they are done in the order which one would expect (e.g., multiplication and division before addition and subtraction). It is possible to group the elements of an expression in parentheses to override or to make explicit the order of evaluation. It is also possible to use a string as an operand in arithmetic operations as long as the string represents a legal SNOBOL4 integer or real number. The null string can also be used arithmetically and is equivalent to zero.

As another example of SNOBOL4's picayunishness we find that all binary operators must have one or more blanks on each side of them. This distinguishes the binary operators from the unary operators, which cannot have blanks separating them from their operands and which must be preceded by either one or more blanks or another unary operator. Both + and - may be used as unary operators.

EXAMPLES

```
X = 2 + 3
```
assigns the integer value 5 to X.

```
Y = 2 * '3'
```
assigns the integer value 6 to Y.

```
Z = '15' / '2'
```
assigns the integer value 7 to Z. Notice that this is integer division and always gives an integer result.

Although it is possible to do arithmetic using numeric strings, the conversion from numeric strings to integers requires considerable computation and should be avoided whenever possible.

September 1975

QUESTIONS

Which of the following assignments are legal and what do they do?

a) `SUM = -SUM`

b) `TOTAL = 67 + A/2`

c) `FOUR = TWO + TWO`

d) `PI = 3.0 + 0.14159`

e) `PI = 3 + .14159`

f) `B = 2 + 4 * 5`

g) `C = (8 - 1) * (5 + 2) + 2 ** 2`

ANSWERS

a) This assigns the negative of SUM to SUM. This is legal only if SUM contains an integer, a real number, or a string of digits.

b) Binary operators must have one or more blanks on each side of them.

c) Legal only if TWO contains a number representation.

d) Yes, these are legal real numbers.

e) Another example of SNOBOL4's eccentricity. It turns out that, although one could not have been expected to know, real numbers must begin and end with a digit.

f) The result is 22.

g) The result is 53.

CONCATENATION

One operation SNOBOL4 allows us to perform on character-string data is the combination of two or more strings to make one longer string. This process is called concatenation. Concatenation is denoted, not by an explicit operator, but by the juxtaposition (placing adjacent) of two string expressions with one or more separating blanks.

EXAMPLES

```
STR = "ABC"  "DEFG"
```

assigns the string "ABCDEFG" to STR. Using this assignment of STR, the following statement

```
B = STR "X" STR
```

would assign the string "ABCDEFGXABCDEFG" to B.

Concatenation has lower precedence than arithmetic operations, e.g., 2 + ’3’ ’4’ is the same as 5 ’4’ which is the same as ’54’. If either or both of the operands of a concatenation is an integer, the result will be a string.

September 1975

QUESTIONS

Which of the following concatenations are legal and what is the value of each?

a) "A" 'A'

b) 5 + 5 5

c) ("-" "4") 3 - "1"

d) If the value of M is the string "BOOLA", what is M M?

ANSWERS

a) "AA"

b) "105"

c) "-42"

d) "BOOLABOOLA"

INDIRECTION

Because of the dynamic nature of the SNOBOL4 storage structure, it is possible to do some rather strange (i.e., powerful) things, strange at least by ordinary programming language standards. One operation is the creation of new variables during execution. This is useful if some of the data being read in is to be used as a variable name. Any string of characters may be used as a variable name although variable names which occur in the source program must conform to the restricted syntax specified above.

To use any string of characters as a variable name, the unary $ operator must be applied to it. This operation is called indirection. (Remember that unary operators have no blanks following them.) The argument of the indirection may be any string expression except the null string. The result of applying the indirection operator to a variable or to a constant whose contents are a string is the same as if that string had appeared in that position as a variable name.

EXAMPLES

```
$"X" = FFM is the same as writing X = FFM.
$('ALP' 'HA') = 2 is the same as writing ALPHA = 2.
```

September 1975

QUESTIONS

What is the result of each of the following assignments? Assume that the following assignments have already been made.

```
A = "B"
B = "B"
C = "A"
AB = "*"
```

a) `R = $C`

b) `R = $("A" $B)`

c) `R = $A B`

d) `$C = $A`

e) `$AB = $$$$$B`

f) `$( C $C) = "QQSV"`

g) `$ B = $ A`

ANSWERS

a) This is the same as writing R = A.

b) This is the same as R = $("A" B), which is the same as R = AB, which is the same as R = "*".

c) This is the same as R = B B, which is the same as R = "BB".

d) This is the same as A = B.

e) This creates a new string by the name of * and assigns to it the contents "B".

f) The same as AB = "QQSV".

g) Illegal. There cannot be blanks immediately following a unary operator.

INPUT-OUTPUT

While most programming languages provide separate statements for performing input and output, SNOBOL4, which has only one statement format, uses special variables for these operations. One variable, INPUT, always contains the next input line. If there are two references to the variable INPUT, then at the first reference the variable will contain the first input line and at the second reference it will contain the second input line.

EXAMPLES

```
NEXTSTM = INPUT
```

will read the next input line and leave it in NEXTSTM.

There are two predefined output variables: OUTPUT and PUNCH. Whenever either of these is assigned a new value, this value is written out as the next line. OUTPUT differs from PUNCH in that it puts a blank at the front of the line it is writing out; therefore, carriage control will not work with OUTPUT.

September 1975

SUCCESS AND FAILURE

In SNOBOL4, as in every programming language, there must be a way of testing certain conditions and thereby controlling the flow of execution in the program. The execution of every SNOBOL4 statement results in one of two conditions: "success" or "failure".

A failure condition can arise from a number of things. For example, a reference to INPUT after reaching an end-of-file (i.e., there are no more input lines) on input will result in a failure condition for that statement. Later, we will see other possible causes of failure. A statement which does not fail succeeds.

GOTOS

Now that we have seen that every SNOBOL4 statement may generate one of two conditions, let us see how this condition may be tested. The optional field on the end of every SNOBOL4 statement is called the GOTO field. It is separated from the rest of the statement by a colon. This field may contain one or more of the following:

1) A success goto. This consists of the capital letter S followed immediately by a statement label enclosed in parentheses. If the statement to which this is attached succeeds, a transfer to the statement label enclosed within the parentheses is made.

   EXAMPLES

   ```
   :S(HOME)

   : S(PRES.)
   ```

2) A failure goto. This consists of the capital letter F followed immediately by a statement label enclosed in parentheses. If the statement fails, a transfer is made to the statement label enclosed in the parentheses.

   EXAMPLES

   ```
   :F(BOTTOM)

   : F(CLERK)
   ```

3) An unconditional goto. This is simply a statement label enclosed in parentheses. Whether the statement fails or succeeds, the transfer will be made to the statement label.

   EXAMPLES

   ```
   :(ALWAYS)

   : ( IMMER )
   ```

More than just a simple statement label may occur between the parentheses of a goto. However, we shall not concern ourselves with the general form of the statement label field here. It will only be stated that any string expression to which the unary name operator may be applied can occur in place of a simple statement label. In fact, the transfer is made to the

statement whose label is the same as the value of the expression with the application of the unary name operator. But enough of this confusion.

Inside the parentheses in each case, there may be blanks between the statement label name and the parentheses; however, there may not be blanks between the S or F and the left parenthesis.

There may be both a success and a failure exit on any SNOBOL4 statement. There also may be a goto on the null statement. Let us consider some examples.

EXAMPLES

a) `LINE = INPUT :F(NO.DATA)`

If an end-of-file is reached on input, this statement will fail and a transfer will be made to the statement labeled NO.DATA. The contents of LINE will not be altered if INPUT fails. Assignment by means of the = operator is not made if any part of the right-hand expression fails.

b) `OUTPUT = "*" INPUT "*" : S(FINE)F(BAD)`

This statement will cause a line to be read in. If there is an end-of-file, the failure exit will be taken; otherwise, an asterisk will be concatenated to the beginning and end of the line and it will be printed, followed by a transfer to FINE.

PROGRAM FORMAT

We are now ready to write an entire, although simple, SNOBOL4 program. But first we must find out what the format of a program is.

Statement labels must start in column one if they are present. If there is no statement label on a statement, column one must contain a blank (exceptions: comment cards and continuation cards; see below). The body of a statement may start anywhere after column one, although it may not extend beyond column 72. If it is necessary to continue a statement from one line to the next, the statement must be broken at a place where a blank separating two elements may occur. This means, for example, that a statement could not be broken in the middle of a string constant, even though there may be a blank in that constant. All continuation cards must have a period or a plus sign in column one.

Comment lines begin with an asterisk in column one. It is possible to place more than one SNOBOL4 statement on a line by using semicolons to separate the statements. The column immediately following the semicolon is treated as column one of the new statement. An asterisk indicating a comment may occur only in the first column of a line and not following a semicolon.

The last statement of a SNOBOL4 program consists of the label END. If this statement is ever transferred to or reached by following the previous statement in execution, the execution of the program is terminated.

EXAMPLES

Let us write a simple program to read in a line and print it out; continuing this process until there are no more input lines, i.e., until INPUT fails.

```
STARTER     LINE = INPUT     :F(END)
            OUTPUT = LINE    :(STARTER)
END
```

This program could be written in even fewer statements as follows:

```
STARTER     OUTPUT = INPUT     :S(STARTER)
END
```

If we wanted to do the above with the addition of a line number at the beginning of each line, the program could be written:

```
BEG     LINE = INPUT    :F(END); NUM = NUM + 1
        OUTPUT = NUM "  " LINE :(BEG)
END
```

September 1975

SIMPLE PATTERN MATCHING

So far we have seen one implicit operation, concatenation. Another operation that is not explicitly expressed but is indicated by its position relative to other expressions is the operation of looking through a string for some pattern of characters. This operation is fundamental to SNOBOL4. In a later section we will discuss the construction of more complicated patterns, but in this section we will be satisfied with the simplest of all patterns: the lowly string.

Suppose we have a string which is the contents of the variable S. In order to search through the string S for some pattern, let us say the sequence of characters "THE", we simply write the string we are scanning followed by the pattern. In this example, we would write

```
S "THE"
```

If the pattern of characters is found, the scan will succeed, otherwise, it will fail. This particular scan will succeed only if the three characters "THE" occur together somewhere in the string. If these three characters do not appear in the string, or if they are not adjacent and in the proper order, the scan will fail. For example, if S contained the string "BROTHER", the scan would succeed. If S contained the string "MOTH EATEN", it would fail.

We may write the following program to read in lines and print only those lines which contain the sequence of characters "END".

```
BEG     STM = INPUT     :F(END)
        STM "END"      :F(BEG)
        OUTPUT = STM    :(BEG)
END
```

The string being scanned is called the subject. The string we are scanning for is called the pattern. The subject and pattern must always be separated by one or more blanks. Either of these may be more complicated than just a simple string name. Since both pattern matching and concatenation are indicated by juxtaposition, there is a potential ambiguity in the order of evaluation of expressions. The ambiguity is resolved by the SNOBOL4 compiler's placing a higher priority on the evaluation of concatenation than it does on scanning. However, we may always use parentheses to group the operands to indicate the order of evaluation. Thus, the statement

```
STM " " "END" " "
```

is the same as

```
STM (" " "END" " ")
```

or

```
STM " END "
```

If we had three strings A, B, and C and wished to scan the contents of A for the contents of B concatenated with the contents of C, we could write

```
A B C
```

However, if we wished to scan the contents of A concatenated with the contents of B for the contents of C, we would write

```
(A B) C
```

September 1975

REPLACEMENT

Now that we have seen that we can scan for a particular string of characters, we may wish to change that portion of the subject string which has been matched. SNOBOL4 allows the replacement of the matched substring in the subject string by any string expression. We would write this as the subject string, followed by the pattern, followed by an equal sign, followed by the string that is to replace the matched portion of the subject string. Let us consider an example. The following statement will replace the first occurrence of an "A" in the string named TEXT by an "E":

```
TEXT "A" = "E"
```

Since the SNOBOL4 scan proceeds from left to right, the first occurrence, if indeed there is more than one, is the leftmost. If TEXT contains the string "GREAT GREAT SCOTT" before this statement is executed, it will contain "GREET GREAT SCOTT" afterwards. The replacement string need not be restricted to the same length as the matched substring. To replace the first occurrence of the string "ALL" by the string "NONE" in the string named UNI, we would write

```
UNI "ALL" = "NONE"
```

If UNI contained the string "SHALLOW" at the beginning, it would contain "SHNONEOW" after the pattern match and replacement had been performed.

Another example is a program which will read input lines, delete all blanks from each, and print them out.

```
     START  IN = INPUT     :F(END)

     HERE   IN " " =       :S(HERE)

            OUTPUT = IN    :(START)

     END
```

Note: The null string need never be written when it appears directly to the right of an = as in the second statement of the above example.

QUESTIONS

What will be the effect of each of the following replacements? Assume the following statements have been executed previous to each example.

```
A = "AEIOU"
B = "BOOGALOO"
C = "CONCUBINE"
D = "A+(B-23)/OVER"
```

a) `B "OO" = "U"`

b) `B "OO" = "OO"`

c) `C "CON" = "ANTI"`

d) `A $B = A`

e) `"THERAPIST" "THE" =`

f) `D "(" ")" = ")" "("`

ANSWERS

a) B will contain "BUGALOO"

b) B will contain "BOOGALOO".

c) C will contain "ANTICUBINE"

d) The null string occurs in infinitely many places in any string. If we assume that the contents of the string BOOGALOO are the null string, then the null string will match in front of the first character and will be replaced by the contents of A. A will now have the contents "AEIOUAEIOU".

e) It is not possible to change the value of a literal.

f) There are not two adjacent parentheses in D. It will fail. Two concatenated literals may always be written as one, i.e., this statement could be written as `D "()" = ")("`

September 1975

FUNCTION CALLS

There are a number of predefined functions available to the SNOBOL4 programmer. In addition, the programmer is allowed to define his own functions by means of the DEFINE function. Function calls may either fail or succeed. If a function call succeeds, it returns a value even if this value is just the null string. The item returned by a function call may be of any data type, not just string mode. Most of the predefined functions are concerned with pattern matching, and these will be discussed in a later section. We will consider a few of the more useful predefined functions here.

Function calls are written in SNOBOL4 much as they are written in FORTRAN. The function name is immediately followed by a parenthesized list of arguments. The left parenthesis must immediately follow the function name. For example, let us consider the predefined function SIZE. SIZE returns an integer which is the number of characters in its argument. An example call might look like the following

```
N = SIZE( "ABC" )
```

which is rather useless, since we already know that the size of the string "ABC" is three. The arguments to a function may be arbitrarily complicated. We could write

```
SIZE("X" SIZE($T) NNN)
```

In this case, the inner function call would be evaluated first. The function SIZE never fails. Let us look at one which does.

The function IDENT takes two string arguments. It fails if they are not identical and succeeds if they are identical. It returns the null string on success. The function call

```
IDENT("STRING1","STRING2")
```

would fail because the two strings are not identical. The function call

```
IDENT("ABC","A" "BC")
```

would succeed and return the null string. If an argument to a function is omitted, the null string is substituted for the missing argument. For example, to test the string name SDF to see if it contains the null string, we could write the function call either as IDENT(SDF) or as IDENT(,SDF). The null string is substituted for the missing second argument in the first case and the missing first argument in the second case. A function call is indicated by the presence of the parentheses which must be present even if there are no arguments.

The function TRIM takes one string argument and returns a string which is the same as that argument with all trailing blanks removed.

The functions which are used to test some condition and which return the null string are called predicate functions. The predicate functions may be used to cause the conditional execution of a statement, for as soon as one function call in a statement fails, the rest of the statement is not evaluated. Take for example, the statement

```
A = IDENT(A,"CHANGE") INPUT
```

If the previous contents of the variable A are the string "CHANGE", then the new contents of A are the null string concatenated with the contents of INPUT, i.e., the next input line. If the old contents of A are not identical to the string "CHANGE", then the rest of the statement is not evaluated and, in this example, the next input line will not be read.

There are predefined predicate functions to test for numeric equality and the other algebraic relations. The function names are LT, LE, EQ, NE, GE, and GT. The names should adequately suggest the relations that they test. Each takes two arguments which are integers, numeric strings, or real numbers and compares the first argument to the second. The function EQ differs from IDENT in two ways: first, IDENT takes string arguments and not numeric arguments; second, it is possible for two strings to represent the same number and yet be different strings. For example, IDENT("002", "2") would fail, but EQ("002","2") would succeed.

There are two more predicate functions of interest. The function DIFFER takes two string arguments. It succeeds if they are not identical and fails if they are. Another function which is useful when manipulating integers is the function INTEGER. This function takes one argument, a string, and succeeds if the string consists only of digits with an optional preceding sign. It fails in all other cases.

Note: No SNOBOL4 function changes its arguments!!!! In particular, the TRIM function does <u>not</u> change its argument: it returns a value.

September 1975

QUESTIONS

Which of the following function calls are legal and what effect will they have?

a)
```
A = " BLANKS "
TRIM(A)
```

b)
```
B =
A = IDENT(B) "SEVEN"
```

c)
```
A = IDENT(B,C) DIFFER(C,B)
```

d)
```
EQ("-2",6*8-50)    :S(S)
```

e)
```
N = SIZE  ()
```

f)
```
SIZE(S) = 10
```

ANSWERS

a) TRIM(A) will return the string " BLANKS" and will leave the string A unaltered. Since the returned string is ignored, the function call effectively does nothing.

b) A will be assigned the string "SEVEN".

c) Since both IDENT and DIFFER cannot succeed with the same arguments, the statement will fail.

d) The binary operators must be surrounded by blanks! If it weren't for this error, the statement would have succeeded.

e) There must not be blanks between the function name and the following left parenthesis.

f) Only functions defined by means of the DATA function may occur to the left of the =.

THE COMPLETE STATEMENT FORMAT

We can now define the statement format. As we have already seen, there is only one statement format in SNOBOL4. To make this a workable scheme, it was necessary to make many parts of the statement optional. In fact, all parts of the statement are optional. We can use the following representation of the SNOBOL4 statement (the brackets around an item indicate that this item is optional):

```
[ label ] [ subject [ pattern ]  [ = replacement ] ] [ :goto ]
```

This representation does not tell the entire story. What form each of the parts may take may still be unclear. This will be left as an exercise for the reader.

EXAMPLES

I

```
L5      A = A

                :(DEF)

        FCN(ARG)

        LIT PAT = TAP

        STRUNG STRING  :F(GH)

        INPUT INPUT INPUT INPUT
```

September 1975

MORE SOPHISTICATED PATTERN MATCHING

Much thought is not required before one realizes that simply scanning one string for a pattern which consists of another string is neither very useful nor very exciting. Perhaps we would like to look for two strings separated by any arbitrary string, or perhaps for a string of a fixed length. These and many more things are possible in SNOBOL4. It is also possible to assign a pattern to a variable. Along with the data types real, integer, and string, there is a pattern data type.

Many patterns do not vary during execution. If there is such a pattern and it contains a function call or more than one element, execution time can be saved by assigning this pattern to a variable at the beginning of the program and using this variable in place of the pattern throughout the remainder of the program.

Since the pattern matching capabilities are described in detail in The SNOBOL4 Programming Language, only a selected subset will be covered here to provide some examples of relatively sophisticated patterns.

THE ARBITRARY PATTERN - ARB

Variables may have patterns as their values. We have seen a trivial case of this, one in which the pattern has the form of a string. A pattern may be far more complex and can be thought of as something which matches a string of characters. The predefined pattern variable ARB contains a pattern that will match any sequence of characters. As with any pattern match, if more than one substring would satisfy the pattern, it is the leftmost shortest such substring that will be matched. That is, ARB will match the leftmost string of appropriate characters, and, if there are two or more such substrings beginning with the same character, it will match the shorter one. In the following example

```
"ABRACADABRA"  "B" ARB "B"
```

the first "B" of the pattern will match the second character of the subject string and the second "B" of the pattern will match the ninth character of the subject string. The ARB matches the third through the eighth character, i.e., "RACADA". In all pattern matches, one element of a pattern must immediately follow another to constitute a match. Therefore, the pattern "B" "B" will match two adjacent "B"'s and not just any two "B"'s.

In the previous example, we could have assigned the pattern to a variable before performing the pattern match. For example,

```
PAT = "B" ARB "B"

"ABRACADABRA" PAT
```

would have yielded the same result. We could also have defined only part of the pattern previously, as in the following statements.

```
PAT1 = "B" ARB ;  "ABRACADABRA" PAT1 "B"
```

Since ARB matches the leftmost shortest string, it is possible for it to match the null string. If the previous pattern had been

```
MAGIC = "A" ARB "B" ARB "C"
"ABRACADABRA"  MAGIC
```

the first ARB would have matched the null string and the second ARB would have matched the two characters RA.

September 1975

QUESTIONS

What will the predefined pattern ARB match in the following statements? (Many of the examples in this writeup use a pattern match on a literal. This is a rather unusual practice and is used here only to show conveniently the contents of the subject string.)

a)
```
QQSV = "Q" ARB "Q"
"QQSV" QQSV
```

b)
```
"1...2...3" ".." ARB ".."
```

c)
```
X = ARB "C"; "THE TIME HAS COME" X
```

d)
```
"12 O'CLOCK" ARB "'"
```

e)
```
"9.5" ARB "." ARB
```

f)
```
ARB = "BABA"
SIC = "A" ARB "B"
"ABABAB"  SIC
```

ANSWERS

a) ARB matches the null string.

b) ARB matches the string ".2".

c) ARB will match "THE TIME HAS ".

d) ARB matches "12 O". Although a shorter pattern match is possible, this is the leftmost.

e) The first ARB matches "9" and matches the second the null string.

f) ARB is a variable like any other, and the value may be changed as this example shows. With its new value, ARB matches "BABA", of course.

CONDITIONAL VALUE ASSIGNMENT

In a pattern match containing an arbitrary pattern element, the substring matched by the arbitrary element is often of interest. For example, if we wished to scan a string S for matched parentheses enclosing any string of characters, we might write a statement like

```
S "(" ARB ")"
```

How can we retain the string of characters matched by ARB? The binary period operator will do this for us. It takes as its left operand a pattern and as its right operand a variable name. It will assign to the variable name on its right a copy of the substring matched by the pattern element on its left. Let us take the previous example and suppose that we wish to assign the string of characters occurring between the parentheses to the variable OP. Let us also suppose that the value of S is the string "IF (A .LT. 50) GOTO 5". The pattern match

```
S "(" ARB . OP ")"
```

which matches the string S for a string of characters composed of a "(" followed by an arbitrary string followed by a ")" will succeed. The previous value of OP will be replaced by "A .LT. 50". The contents of S will be unchanged. If the pattern match had failed, no new value assignment would have been made. The binary period, called the conditional value assignment operator, has the highest precedence of all operators. Thus, if the left operand is more than a single pattern element, it is necessary to group the pattern elements in parentheses.

If we take the previous example and wish to assign not only everything between the parentheses but also the parentheses to the variable OP, we would write the statement as follows:

```
S ( "(" ARB ")" ) . OP
```

This pattern match will succeed and the string "(A .LT. 50)" will be assigned to OP.

September 1975

QUESTIONS

What assignments will be made as a consequence of the following pattern matches?

a)
```
SENT = "NOW IS THE TIME ..."
SENT " " ARB . WORD " " =
```

b)
```
SPQR = "CAVEAT EMPTOR"
SPQR ARB . W " "
```

c)
```
"A...Z"  ARB . LETTERS
```

d)
```
EXP = "P+((9-M)/F)"
EXP "(" ARB . IE ")"
```

e)
```
"THE COST IS $54.02 " "$" ARB . DOLLARS "." ARB . CENTS " "
```

f)
```
"THERE IS ONLY ONE ."  "." ARB . F "."
```

ANSWERS

a) WORD will be assigned "IS".
SENT will be assigned "NOWTHE TIME ..."

b) W will be assigned "CAVEAT".

c) LETTERS will be assigned the null string.

d) IE will be assigned "(9-M" .

e) DOLLARS will be assigned "54" and CENTS "02".

f) The pattern match will fail, and therefore no assignment will be made.

THE BALANCED PATTERN - BAL

Since many of the problems to which SNOBOL4 is applied have a portion of their data which is algebraic in nature, a special pattern, BAL, is provided, which matches any parenthesis-balanced string. To be parenthesis-balanced, a string must consist of at least one character and, if there are any parentheses in it, the parentheses must be paired in the usual manner.

EXAMPLES of parenthesis-balanced strings:

```
"()"  "A"   "NO PARENS"   "A(5)"   "(Q/2-(J*J))"
```

EXAMPLES of parenthesis-unbalanced strings:

```
""    "("  ")("   "A(5"   "(A/2-(J*J)))"
```

EXAMPLES of pattern matches using BAL:

a) `"IF (P .LT.  (Q-J)) GO TO 23"  BAL .  X ")"`
The variable X will be assigned the string "P .LT. (Q-J)".

b) `")(()))))("  BAL .  P`
P will be assigned the string "(())" .

c) What will the following program produce as output?

```
      STRING = ")(()))))("
H     STRING BAL = "("   :F(END)
      OUTPUT = STRING    :(H)
END
```

The output will be

```
)())(
)()(
)((
```

September 1975

THE FIXED-LENGTH PATTERN FUNCTION - LEN(N)

One of the most useful of the pattern-valued functions is LEN. It takes one argument, an integer, and returns a pattern which will match that integer number of characters. The argument must be nonnegative.

EXAMPLES

```
"ABC" LEN(1) .  LET
```

will cause the string "A" to be assigned to LET.

Here is a program which will read in a series of strings, trim them, reverse them, and print the reversed strings.

```
        L1 = LEN(1) .  C
BEG     IN = TRIM(INPUT)    :F(END)
        OUT =
NC      IN L1 =             :F(PRNT)
        OUT = C OUT          :(NC)
PRNT    OUTPUT = OUT        :(BEG)
END
```

QUESTIONS

What will the following programs and statements do?

a)
```
     NL   VOWEL = "AEIOU"
          TEXT = INPUT      :F(END)
     J    VOWEL LEN(1) . V =   :F(OL)
     H    TEXT V =          :S(H)F(J)
     OL   OUTPUT = TEXT     :(NL)
     END
```

b)
```
     "**(OPEN(5 - 2))**" (LEN(2) BAL) . X
```

c)
```
     P = BAL LEN(1) BAL
     "(THIS)(HOME)" P . V
```

d)
```
     X LEN(SIZE(X) - 1) . X
```

ANSWERS

a) This program will read input lines, delete all vowels in each line, and print the lines.

b) The LEN(2) will match the first "**" and BAL will match "(OPEN(5 - 2))". Therefore, X will be assigned the string "**(OPEN(5 - 2))".

c) The first BAL will match "(THIS)", LEN(1) will match "(", and the second BAL will match "H". Therefore, V will be assigned the string "(THIS)(H".

d) The new contents of X will be the previous contents of X with the rightmost character removed. If X initially contains the null string, this will result in a fatal error.

September 1975

FUNCTION DEFINITION

One of the most useful features of SNOBOL4 is its flexible function definition capability. A significant difference between SNOBOL4 and other languages is that SNOBOL4 function definition is dynamic. That is to say, a function may be defined and redefined during execution. Although this is a powerful feature, it is used only rarely in actual programming.

All functions are defined by means of the DEFINE function. The DEFINE function takes two string arguments. The first string has the following format:

fcn(fplist)lvlist

fcn is the function name.

fplist is a list of the formal parameters separated by commas. A formal parameter is the name by which the value of the actual parameter is referenced. The actual parameters are the particular arguments to the function at a particular function call. In the function definition, all references to the actual parameters are made through the formal parameters. At the time of the function call, the current values of the formal parameters are saved and the value of each actual parameter is assigned to the corresponding formal parameter. When the function returns, the previous values of the formal parameters are reinstated.

lvlist is a list of the local variables separated by commas. It is often necessary to use variable names to contain intermediate results during the execution of the program. To avoid any conflict with variable names occurring in that section of the SNOBOL4 program which made the function call, the variable names which are used for temporary storage in the function may be declared local. When a function is entered, the current values of all local variable names are saved and the variable names are assigned null values. When the function returns, the old values are restored. An analogous treatment is made of the variable whose name is the same as the function name. As one can see, these procedures are similar to those in the case of the formal parameters, with the exception that the variable names which are the same as the formal parameters are given as contents not the null string, but the values of the actual parameters; all three processes occur immediately upon entry to the function.

The second argument to the DEFINE function is a string which is the name of the entry point of the function (i.e., the label at which to start execution of the function). If the second argument is omitted, it will be assumed that the entry point will have a label which is the same as the name of the function.

EXAMPLES

```
DEFINE("F(X)","ENT")
```

This defines a function named F with one formal parameter X and no local variables. The entry point is labeled ENT.

```
DEFINE("SIZE(S)N")
```

This defines a function named SIZE with one formal parameter S and one local variable N. The entry is at the label SIZE. Although the function SIZE is predefined, it may be redefined like any other function.

```
DEFINE("QQSV(ABC,Z)Q1,Q2,THREE,UOI..." , "SESAME")
```

This will define a function named QQSV with two formal parameters, four local variables, and the entry point SESAME.

```
A = "Z)RN"
DEFINE("P(" A "G")
```

This will define a function named P with a formal parameter Z and a local variable RNG.

Notes:

1) There may not be blanks in either of the strings which are arguments to the DEFINE function.

2) Remember that the DEFINE function is executable. A function is not defined until the appropriate DEFINE function call has been made. It slows down, but otherwise does not impair the execution of the program to have a DEFINE function call in the middle of a loop. Once a function has been defined it is not necessary to define it again.

September 1975

FUNCTION EXECUTION

After a function has been entered, we are faced with some new problems. How do we indicate the success or failure of a function, and how do we indicate the value to be returned if the function succeeds?

A function return is made by transferring to one of two reserved labels. (We have already encountered the reserved label END.) If a transfer is made to the label RETURN, the function succeeds. If a transfer is made to the label FRETURN, the function fails.

The value that is to be returned by the function is the value of the variable with the same name as the function. (A name may be used as a variable name, a function name, and a label with no conflict.) When the function is entered, the current value of the variable with the same name as the function is saved, and the null string becomes the new contents. Therefore, if no assignment is made to this variable and the function returns successfully, the value of the function call is the null string.

Since, at every function call, the actual parameters are evaluated before the current values of variables with names the same as the formal parameters, the local variables, and the function name are saved, it is possible to write recursive functions, i.e., functions which call upon themselves.

UNDERSTAND THE FOLLOWING EXAMPLES!

a)
```
      * THIS PROGRAM WILL READ IN LINES AND PRINT THE TRIMMED LENGTH
      * THE SIZE AND TRIM FUNCTIONS ARE REDEFINED HERE.
           DEFINE("TRIM(S)")
           DEFINE("SIZE(S)")
            L1 = LEN(1)
      *
      T     OUTPUT = SIZE(TRIM(INPUT))  :S(T)F(END)
      *
      SIZE  SIZE  = 0
      ND    S L1 =      :F(RETURN)
            SIZE = SIZE + 1    :(ND)
      *
      TRIM TRIM = S
      H    TRIM DIFFER(TRIM) LEN(SIZE(TRIM) - 1) .  TRIM " "
      .          :S(H)F(RETURN)
      END
```

b) This function takes two arguments. All occurrences in the first string of any of the characters in the second string will be deleted. Only the statements relevant to the function definition and execution are shown here.

```
      DEFINE("DELETE(S1,S2)C","DEL")
          .
          .
          .
DEL   DELETE = S1
UT    S2 LEN(1) .  C =     :F(RETURN)
H    DELETE C =        :S(H)F(UT)
          .
          .
          .
```

c) Here are two examples of recursive functions. The first example is the SIZE function written recursively.

```
      DEFINE("SIZE(S)")
           .
           .
           .
SIZE  SIZE = 0
      S LEN(1) =        :F(RETURN)
      SIZE = 1 + SIZE(S) :(RETURN)
           .
           .
```

d) The next function takes one argument, a string containing a fully parenthesized arithmetic expression with one-character variables, constants, and binary operators, and returns the Polish prefix form of the expression.

```
     DEFINE("PPF(EXP)LOP,OP,ROP")
         .
         .
         .
PPF  EXP "(" BAL .  LOP LEN(1) .  OP BAL .  ROP ")"   :S(DEC)
     PPF = EXP         :(RETURN)
DEC  PPF = OP PPF(LOP) PPF(ROP)   :(RETURN)
```

SPITBOL

The SPITBOL compiler is available in the file *SPITBOL. SPITBOL (speedy implementation of SNOBOL4) was developed at the Illinois Institute of Technology as an alternative to the interpreter developed at Bell Telephone Laboratories (available in the public file *SNOBOL4). Programs run considerably faster under SPITBOL than they do under the BTL interpreter and can be saved in object module form. The language accepted by SPITBOL is very nearly a superset of SNOBOL4, but does have a few incompatibilities. The SPITBOL compiler itself is also more flexible than the BTL interpreters and has been implemented in MTS somewhat differently than the BTL interpreters.

The following documentation was provided by the authors of SPITBOL and has been edited here to reflect the MTS implementation. It was written with the assumption that the users would have a knowledge of SNOBOL4 as developed by Bell Telephone Laboratories and reflects, to the best of our knowledge, the current state of SPITBOL. It is designed to be a reference manual rather than a tutorial guide.

INTRODUCTION

SPITBOL is an implementation of the SNOBOL4 computer language for use on the IBM System 360/370. SPITBOL is considerably smaller than the implementation from Bell Telephone Laboratories (implemented by the designers of the SNOBOL4 language -- R. E. Griswold and I. Polonsky) and has execution speeds up to ten times faster. For certain programs, notably those with in-line patterns, the gain in speed may be even greater.

Unlike BTL SNOBOL4, SPITBOL is a true compiler which generates executable machine code. The generated code may be listed in assembly form. Of course, the complexity of the SNOBOL4 language dictates that system subroutines be used for many common functions. SPITBOL can be run as a compile-and-execute system like WATFIV, where jobs are executed as soon as they are compiled. Alternatively, the compiler can generate an object module for later execution.

This section assumes that the reader is familiar with the standard version of SNOBOL4 (referred to as BTL SNOBOL4 in the remainder of the section). Version 3.4 of SNOBOL4 is the reference version for comparison. There are several minor incompatibilities and some features are unimplemented. There are also several additions to the language in this implementation.

In general, an attempt has been made to retain upward compatibility wherever possible. Most SNOBOL4 programs which operate correctly using BTL SNOBOL4 should operate correctly when compiled and executed using SPITBOL.

SPITBOL was designed and implemented by Robert B. K. Dewar and Kenneth Belcher at The Illinois Institute of Technology.

SUMMARY OF DIFFERENCES

This section contains a summary of the significant differences between SPITBOL and BTL SNOBOL4.

Features Not Implemented

At the current time, the following features of BTL SNOBOL4 are not implemented.

(1) Array elements, table elements, and program-defined data types cannot be traced.

(2) OPSYN for operators (third argument) is permitted only for normally undefined operators.

(3) The BLOCK datatype (as implemented in SNOBOL4B) is not available.

(4) The &STFCOUNT keyword is not implemented.

Features Implemented Differently

The following features are implemented by SPITBOL, but the usage is different from that in BTL SNOBOL4, and changes in existing programs may be required.

(1) Recovery from execution errors (see the description of the SPITBOL function SETEXIT).

(2) I/O is somewhat different. The FORTRAN I/O routines are not used. However, a FORTRAN format-processing routine has been included for compatibility.

Additional Features

The following additional features (not in BTL SNOBOL4) are included in the SPITBOL system.

(1) The datatype DREAL (7-byte real).

(2) The additional functions BREAKX, LEQ, LGE, LLE, LLT, LNE, LPAD, REVERSE, RPAD, SETEXIT, and SUBSTR.

(3) Additional flexibility in I/O. Format-free, variable-record-length I/O for simple string input/output. FDnames and logical I/O unit names are allowed in INPUT and OUTPUT statements.

(4) The symbolic dump optionally includes elements of arrays, tables, and program-defined datatypes.

(5) Both the pattern-matching stack and the function-call push-down stack may expand to use all available dynamic memory if necessary.

Other Incompatibilities

(1) The value of a modifiable keyword can be changed only by direct assignment using "=". Pattern assignment cannot be used to change a keyword value, and the name operator cannot be applied to a keyword.

(2) SPITBOL allows some datatype conversions not allowed in BTL SNOBOL4. For example, a REAL value may be used in pattern alternation and is converted to a string. In general, SPITBOL converts objects to an appropriate datatype if at all possible.

(3) The unary . (name) operator applied to a natural variable yields a NAME rather than a STRING. Since this NAME can be converted to a STRING when required, the difference is normally not noticed. The only points at which the difference is apparent is in use of the IDENT, DIFFER, and DATATYPE functions and when used as a TABLE subscript.

(4) SPITBOL normally operates in an optimized mode which generates a number of incompatibilities. This mode can be turned off if necessary; see the description of the control cards -OPTIMIZE and -NOOPTIMIZE.

(5) SPITBOL permits leading and trailing blanks on numeric strings which are to be converted to STRING.

(6) Several of the built-in functions are different. These are identified by an * appended to their name in the section "Functions."

(7) SPITBOL does not permit exponentiation of two real numbers.

(8) The BACKSPACE function is not implemented.

DATATYPES AND CONVERSIONS

Datatypes in SPITBOL

STRING	strings range in length from 0 (null string) to 32758 characters (subject to the setting of &MAXLNGTH). Any characters from the EBCDIC set can appear.
INTEGER	integers are stored in 32-bit form, allowing a range of -2**31 to +2**31-1. There is no negative zero.
REAL	stored as a 32-bit, short-form, floating-point number.
DREAL	stored using long-form floating-point. The low-order byte is not available and is stored as zero, thus giving a 48-bit mantissa (15 decimal digits).
ARRAY	arrays may have up to 255 dimensions.
TABLE	a table may have any number of elements; see the description of the TABLE function in the section "Functions" for further details. Any SPITBOL entity may be used as the name of a table element, including the null string.
PATTERN	pattern structures may range up to 32768 bytes. This means there is essentially no limit on the complexity of a pattern.
NAME	a name can be obtained from any variable. Note that in SPITBOL, the name operator (unary dot) applied to a natural variable yields a name, not a string as in BTL SNOBOL4.
EXPRESSION	any expression may be deferred via the unary * operator.
CODE	a string representing a valid program can be converted to code at execution time. The resulting object, of type CODE, may be executed in the same manner as the original program.

Datatype Conversion

As far as possible, SPITBOL converts from one datatype to another as required. The following table shows which conversions are possible. A

dashed entry indicates that the conversion is never possible, X indicates that the conversion is always possible, and F indicates that conversion may be possible, depending on the value involved.

Convert from	Convert to: S	I	R	D	A	T	P	N	E	C
S	X	F	F	F	-	-	X	X	F	F
I	X	X	X	X	-	-	X	X	X	-
R	X	F	X	X	-	-	X	X	X	-
D	X	F	X	X	-	-	X	X	X	-
A	-	-	-	-	X	F	-	-	-	-
T	-	-	-	-	F	X	-	-	-	-
P	-	-	-	-	-	-	X	-	-	-
N	F	F	F	F	-	-	-	-	F	F
E	-	-	-	-	-	-	-	-	X	-
C	-	-	-	-	-	-	-	-	-	X

S -- STRING
I -- INTEGER
R -- REAL
D -- DREAL
A -- ARRAY
T -- TABLE
P -- PATTERN
N -- NAME
E -- EXPRESSION
C -- CODE

The detailed description of each of the possible conversions is given below.

STRING --> INTEGER

Leading and trailing blanks are ignored. A leading sign is optional. The sign, if present, must immediately precede the digits. A null string is converted to zero.

STRING --> REAL

Leading and trailing blanks are ignored. A leading sign, if present, must immediately precede the number. The number itself may be written in standard (FORTRAN-type) format with an optional exponent.

STRING --> DREAL

The rules are the same as for STRING to REAL. Note that a STRING is considered to represent a DREAL if more than eight significant digits are given, or if a D is used for the exponent instead of an E.

STRING --> PATTERN

A pattern is created which matches the string value.

STRING --> NAME

The result is the name of the natural variable with a name of the given string. This is identical to the result of applying the unary dot operator to the variable in question. The null string cannot be converted to a name.

STRING --> EXPRESSION

The string must represent a legal SPITBOL expression. The compiler is used to convert the string into its equivalent expression and the result can be used anywhere an expression is permitted.

STRING --> CODE

The string must represent a legal SPITBOL program, complete with labels, and using semicolons to separate statements. The compiler is used to convert the string into executable code. The resulting code can be executed by transferring to it with a direct GOTO or by a normal transfer to a label within the code.

INTEGER --> STRING

The result has no leading or trailing blanks. Leading zeros are suppressed. A preceding minus sign is supplied for negative values. Zero is converted to '0'.

INTEGER --> REAL

A real number is obtained by adding a zero fractional part. Note that significance is lost in converting integers whose absolute value exceeds 2**24-1.

INTEGER --> DREAL

A DREAL is obtained by adding a zero fractional part. Significance is never lost in this conversion.

INTEGER --> PATTERN

The integer is first converted to STRING and then treated as STRING to PATTERN.

INTEGER --> NAME

The integer is first converted to STRING and then treated as STRING to NAME.

INTEGER --> EXPRESSION

The result is an expression which, when evaluated, yields the INTEGER as its value.

REAL --> STRING

The real number is converted to its standard character representation. Fixed-type format is used if possible; otherwise, an exponent (using E) is supplied. Seven significant digits are generated, the last being correctly rounded for all cases. Trailing insignificant zeros are suppressed after rounding has taken place.

REAL --> INTEGER

This conversion is possible only if the REAL is in the range permitted for integers. In this case, the result is obtained by truncating the fractional part.

REAL --> DREAL

Additional low-order zeros are added to extend the mantissa.

REAL --> PATTERN

The integer is first converted to STRING and then treated as STRING to PATTERN.

REAL --> NAME

The integer is first converted to STRING and then treated as STRING to NAME.

REAL --> EXPRESSION

The result is an expression which, when evaluated, yields the REAL as its value.

DREAL --> STRING

The conversion is like REAL to STRING except that 15 significant digits are given and a D is used for the exponent if one is required.

DREAL --> INTEGER

This conversion is possible only if the DREAL is in the range permitted for integers. In this case, the result is obtained by truncating the fractional part.

DREAL --> REAL

The low-order digits of the mantissa are truncated to reduce the precision.

DREAL --> PATTERN

The integer is first converted to STRING and then treated as STRING to PATTERN conversion.

DREAL --> NAME

The integer is first converted to STRING and then treated as STRING to NAME conversion.

DREAL --> EXPRESSION

The result is an expression which, when evaluated, yields the DREAL as its value.

ARRAY --> TABLE

The array must be two-dimensional with a second dimension of two, or an error occurs. For each entry (value of the first subscript), a table entry using the (X,1) entry as the name and the (X,2) entry as the value is created. The resulting table has the same number of hash headers (see TABLE function) as the first dimension.

TABLE --> ARRAY

The table must have at least one element which is nonnull. The array generated is two-dimensional. The first dimension is equal to the number of non-null entries in the table. The second dimension is two. For each entry, the (X,1) element in the array is the name and the (X,2) element is the value. The order of the elements in the array is the order in which elements occurred in the table.

NAME --> STRING

A NAME can be converted to a STRING only if it is the name of a natural variable. The resulting string is the character name of the variable.

NAME --> INTEGER, REAL, DREAL, PATTERN, EXPRESSION, CODE

The NAME is first converted to a string (if possible) and then the conversion proceeds as described for STRING.

SYNTAX

This section describes differences between the syntax in SPITBOL and BTL SNOBOL4. These differences are minor and should not affect existing programs.

(1) Reference to elements of arrays which are themselves elements of arrays is possible without using the ITEM function. Thus the following are equivalent:

A<J><K> = B<J><K>

ITEM(A<J>,K) = ITEM(B<J>,K)

(2) Up to 255 columns of input may optionally be used -- see the description of the -INxxx control card in the section "Control Cards."

(3) The only way to change the value of a keyword is by direct assignment. It is not permissible to use a keyword in any other context requiring a name.

(4) The compiler permits real constants to be followed by a FORTRAN-style exponent E+xxx or D+xxx, the latter signifying a double-precision real (DREAL).

PATTERN MATCHING

Pattern matching is essentially compatible, however there are some minor differences and extensions as described in this section.

The stack used for pattern matching can expand to fill all available dynamic memory if necessary. Thus, the diagnostic issued for an infinite pattern recursion is simply the standard memory overflow message.

In SPITBOL, the values of &QUICKSCAN and &ANCHOR are obtained only at the start of the match. In BTL SNOBOL4, changing these values during a match can lead to unexpected results.

The BREAKX function allows construction of an extended break pattern. See the description in the section "Functions."

FUNCTIONS

This section defines the functions which are built-in to the SPITBOL system. The functions are described in alphabetical order. In most cases, the arguments are automatically preconverted to some particular datatype. This is indicated in the function header by the notation

FUNCTION(STRING,INTEGER,etc...)

If the corresponding argument cannot be converted to the indicated datatype, an error with major code 1 (illegal datatype) occurs (see the section "Error Codes"). In some cases, the range of arguments permitted is restricted. Arguments outside the permitted domain cause the generation of an error with major code 13 (incorrect value for function or operator). The usage 'ARGUMENT' implies that the argument can be of any datatype. 'NUMERIC' implies that any numeric datatype can occur (INTEGER, REAL, or DREAL).

In the following descriptions, a single asterisk following the name of the function indicates that the implementation of the function differs from that in BTL SNOBOL4, or that the function is not available in BTL SNOBOL4.

ANY -- Pattern to Match Selected Character

ANY(STRING) or ANY(EXPRESSION)

This function returns a pattern which will match a single character selected from the characters in the argument string. A null argument is not permitted.

If an expression argument is used, then the expression is evaluated during the pattern match and must give a nonnull string result.

APPLY* -- Apply Function

APPLY(NAME,ARG,ARG,...)

The first argument is the name of a function to be applied to the (possibly null) list of arguments following. Unlike BTL SNOBOL4, SPITBOL does not require the number of arguments to match. Extra arguments are ignored, and missing arguments are supplied as null strings.

ARBNO -- Pattern for Iterated Match

ARBNO(PATTERN)

This function returns a pattern which will match an arbitrary number of occurrences of the pattern argument, including the null string (corresponding to zero occurrences).

ARG -- Obtain Argument Name

ARG(NAME,INTEGER)

The first argument represents the name of a function. The integer is the number of a formal argument to this function. The returned result is the string name of the selected argument. ARG fails if the integer is out of range (less than one, or greater than the number of arguments).

ARRAY -- Generate Array Structure

ARRAY(STRING,ARG)

The string represents the prototype of an array to be allocated. This is in the format 'LBD1:HBD1,LBD2:HBD2,..'. The lower bound (LBD) may be omitted for some or all of the dimensions, in which case a lower bound of one is assumed. The second argument (of any datatype) is the initial value of all the elements in the array. If the second argument is omitted, the initial value of all elements becomes the null string.-

September 1975

BREAK -- Construct Scanning Pattern

BREAK(STRING) or BREAK(EXPRESSION)

This function returns a pattern which will match any string up to but not including a character in the string argument. A null argument is not permitted.

If an expression argument is given, the resulting pattern causes the string to be evaluated during pattern matching. In this case, the evaluated result must be a non-null string.

BREAKX* -- Construct Scanning Pattern

BREAKX(STRING) or BREAKX(EXPRESSION)

BREAKX returns a pattern whose initial match is the same as a corresponding BREAK pattern. However, BREAKX has implicit alternatives which are obtained by scanning past the first break character found and scanning to the next break character. In other words, should the pattern fail, BREAKX will force scanning past the current break character and match (like BREAK) at the next break character, etc.

Note that BREAKX may be used to replace ARB in many situations where BREAK cannot be used easily. For example, the following replacement can be made:

ARB ('CAT' | 'DOG') ---> BREAKX('CD') ('CAT' | 'DOG')

In the case of an expression argument, the expression is evaluated during pattern matching and must yield a nonnull string value. Note that the evaluation of the expression is not repeated on rematch attempts by extension.

CLEAR* -- Clear Variable Storage

CLEAR(STRING,ARGUMENT)

This function causes the values of variables to be set to null. In the simple case, where both arguments are omitted, the action is the same as in BTL SNOBOL4; i.e., all variables are cleared to contain the null string. Two extensions are available in SPITBOL. The first argument may be a string which is a list of variable names separated by commas. These represent the names of variables whose value is to be left unchanged. In addition, if a second nonnull argument is supplied, then all variables containing pattern values are left unchanged. For example,

CLEAR('ABC,CDE,GGG',1)

would cause the value of all variables to be cleared to null except for the variables ABC, CDE, GGG, and all other variables containing pattern values.

CODE -- Compile Code

CODE(STRING)

The effect of this function is to convert the argument to type CODE as described in the section "Datatype Conversion." The STRING must represent a valid SPITBOL program complete, with labels and using semicolons to separate statements. The call to CODE fails if these conditions are not met.

COLLECT -- Initiate Storage Regeneration

COLLECT(INTEGER)

The COLLECT function forces a garbage collection which retrieves unused storage and returns it to the block of available storage. The integer argument represents a minimum number of bytes to be made available. If this amount of storage cannot be obtained, the collect function fails. On successful return, the result is the number of bytes actually obtained.

Note that although the implementation of COLLECT is similar to that in BTL SNOBOL4, the values obtained will be quite different due to different internal data representations. Furthermore, the internal organization of SPITBOL is such that forcing garbage collections to occur before they are required always increases execution time.

CONVERT* -- Convert Datatypes

CONVERT(ARGUMENT,STRING)

The returned result is obtained by converting the first argument to the type indicated by the string name of the datatype given as the second argument. The section "Datatype Conversion" describes the permitted conversions. Any conversions which are not permitted cause failure of the CONVERT call.

An additional possibility for the second argument is 'NUMERIC', in which case the argument is converted to INTEGER, REAL, or DREAL according to its form.

COPY* -- Copy Structure

COPY(ARGUMENT)

The COPY function returns a distinct copy of the object which is its argument. This is useful only for arrays, tables, and program-defined datatypes. Note that SPITBOL does permit the copying of TABLES, unlike BTL SNOBOL4.

September 1975

DATA -- Create Datatype

```
DATA(STRING)
```

The argument to DATA is a prototype for a new datatype in the form of a function call with arguments. The function name is the name of the new datatype. The 'ARGUMENT' names are names of functions which represent the fields of the new datatype.

Note that in SPITBOL a significant increase in efficiency is obtained by avoiding the use of duplicate field names for different datatypes, although SPITBOL does allow such multiple use of field function names.

DATATYPE* -- Obtain Datatype

```
DATATYPE(ARGUMENT)
```

DATATYPE returns the formal identification of the datatype of its argument. In SPITBOL, the additional datatype names 'DREAL' and 'NAME' are included in the list of possible returned results.

DATE -- Obtain Date

```
DATE()
```

DATE returns an eight-character string of the form MM/DD/YY representing the current date.

DEFINE -- Define a Function

```
DEFINE(STRING,NAME)
```

The DEFINE function is used to define program-defined functions. The use of DEFINE is the same in SPITBOL as in BTL SNOBOL4.

DETACH -- Detach I/O Association

```
DETACH(NAME)
```

NAME is the name of a variable which has previously been input- or output-associated. Use of the DETACH function does not affect the file involved.

DIFFER* -- Test for Arguments Differing

```
DIFFER(ARGUMENT,ARGUMENT)
```

DIFFER is a predicate function which fails if its two arguments are identical objects. Note that DIFFER(.ABC,'ABC') succeeds in SPITBOL since .ABC is a NAME. DIFFER, IDENT, and DATATYPE are the only functions in which the different implementations of the name operator (unary dot) may give rise to problems.

DUMP* -- Dump Storage

DUMP(INTEGER)

The DUMP function causes a dump of the items specified by the integer argument. After the dump is complete, execution continues unaffected (the DUMP function returns the null string). The integer arguments are defined as follows:

DUMP(0) dumps nothing.
DUMP(1) dumps all nonconstant keywords and all nonnull natural variables.
DUMP(2) dumps all of the above plus the values of the elements of arrays, tables, and program-defined datatypes.
DUMP(3) causes a hexadecimal dump of the SPITBOL system and should be avoided. This is intended for system debugging.

DUPL -- Duplicate String

DUPL(STRING,INTEGER)

DUPL returns a string obtained by duplicating the first (STRING) argument the number of times indicated by the second argument.

ENDFILE* -- Close file

ENDFILE(STRING)

STRING is the name of a file (not the name of a variable associated with the file). The named file is closed, all associated storage is released, and all variables associated with the file are automatically detached. Thus, ENDFILE should be used only when no further use is to be made of the file. If the file is to be reread or rewritten, REWIND should be used rather than ENDFILE.

EQ -- Test for Equal To

EQ(NUMERIC,NUMERIC)

EQ is a predicate function which tests whether its two arguments are equal. DREAL arguments are permitted.

EVAL -- Evaluate Expression

EVAL(EXPRESSION)

EVAL returns the result of evaluating its expression argument. Note that a string can be converted into an expression by compiling it into code. Thus, EVAL in SPITBOL is compatible with BTL SNOBOL4 and handles strings in the same way.

FIELD -- Get Field Name

FIELD(NAME,INTEGER)

FIELD returns the name of the selected field of the program-defined datatype whose name is the first argument. If the second argument is out of range (less than one, or greater than the number of fields), the FIELD function fails.

GE -- Test for Greater or Equal To

GE(NUMERIC,NUMERIC)

GE is a predicate function which tests if the first argument is greater than or equal to the second argument.

GT -- Test for Greater Than

GT(NUMERIC,NUMERIC)

GT is a predicate function which tests if the first argument is greater than the second argument.

IDENT* -- Test for Identical

IDENT(ARGUMENT,ARGUMENT)

IDENT is a predicate function which tests if its two arguments are identical. Note that in SPITBOL, IDENT(.ABC,'ABC') fails since .ABC is a name in SPITBOL. Otherwise, IDENT is compatible with BTL SNOBOL4.

INPUT* -- Set Input Association

INPUT(NAME,STRING,INTEGER)

The first argument is the name of a variable which is to be input-associated. The second argument is the filename of the file to which the variable is to be associated. In MTS, a filename can be an FDname (MYFILE, *SOURCE*, etc.) or a logical I/O unit name (SCARDS, 7, etc.). If the second argument is omitted, it is assumed to be SCARDS.

The third argument is either zero, in which case it is ignored, or a positive nonzero integer, in which case input records longer than the given limit are truncated.

A restriction in SPITBOL is that only natural variables can be input-associated. It is not possible to input-associate array and table elements.

INTEGER* -- Test for Integer

INTEGER(NUMERIC)

INTEGER is a predicate function which tests whether its argument is integral. It fails if the argument cannot be converted to numeric, or if it has a nonintegral value.

ITEM -- Select Array or Table Element

ITEM(ARRAY,INTEGER,INTEGER,...) or ITEM(TABLE,ARGUMENT)

ITEM returns the selected array or table element by name. Note that the use of ITEM is unnecessary in SPITBOL because of the extended syntax for array references (see the section "Syntax").

LE -- Test for Less Than or Equal To

LE(NUMERIC,NUMERIC)

LE is a predicate function which tests whether the first argument is less than or equal to the second argument.

LEN -- Generate Specified-Length Pattern

LEN(INTEGER) or LEN(EXPRESSION)

LEN generates a pattern which will match any sequence of characters of length given by the argument, which must be a non-negative integer.

If the argument is an expression, it is evaluated during pattern matching and must yield a nonnegative integer.

LEQ* -- Test for Lexically Equal To

LEQ(STRING,STRING)

LEQ is a predicate function which tests whether its arguments are lexically equal. Note that LEQ differs from the IDENT function in that its arguments must be strings. Thus, LEQ(10,'10') succeeds, as does LEQ(.ABC,'ABC').

LGE* -- Test for Lexically Greater Than or Equal To

LGE(STRING,STRING)

LGE is a predicate function which tests whether the first argument is lexically greater than or equal to the second argument.

LGT -- Test for Lexically Greater Than

LGT(STRING,STRING)

LGT is a predicate function which tests whether its first string argument is lexically greater than the second string argument.

LLE* -- Test for Lexically Less Than or Equal To

LLE(STRING,STRING)

LLE is a predicate function which tests whether its first string argument is lexically less than or equal to the second argument.

LLT* -- Test for Lexically Less Than

LLT(STRING,STRING)

LLT is a predicate function which tests whether its first argument is lexically less than its second argument.

LNE* -- Test For Lexically Not Equal To

LNE(STRING,STRING)

LNE is a predicate function which tests whether its arguments are lexically unequal. LNE differs from the DIFFER function in that its arguments must be strings.

LOAD* -- Load External Function

LOAD(STRING,STRING)

LOAD is used to load an external function. The form of the LOAD function is the same as in BTL SNOBOL4 except that the datatype DREAL may be used and unconverted descriptors must be specified differently. In the case where the datatype is unspecified, the form of the descriptor passed is quite different from that in BTL SNOBOL4. The section "External Routines" describes the actual form that the arguments take.

LOCAL -- Get Name of Local Variable

LOCAL(NAME,INTEGER)

The value returned is the name of the indicated local variable of the function whose name is given by the first argument. LOC fails if the second argument is out of range (less than one, or greater than the number of local variables).

LPAD* -- Left Pad

LPAD(STRING,INTEGER,STRING)

LPAD returns the result obtained by padding out the first argument on the left to the length specified by the second argument, using the pad character supplied by the one-character-string third argument. If the third argument is null or omitted, a blank is used as the pad character. If the first argument is already long enough or too long, it is returned unchanged. LPAD is useful for constructing columnar output.

LT -- Test for Less Than

LT(NUMERIC,NUMERIC)

LT is a predicate function which tests whether the first argument is less than the second argument.

NOTANY -- Build Character Select Pattern

NOTANY(STRING) or NOTANY(EXPRESSION)

NOTANY returns a pattern which will match any single character not in the string argument given. A null argument is not permitted.

If the argument is an expression, then the expression is evaluated at pattern-match time and must yield a nonnull string.

OPSYN* -- Equate Functions

OPSYN(NAME,NAME,INTEGER)

The first argument is to have the same definition as the second argument. OPSYN may be used to redefine operators as in BTL SNOBOL4 using 1 or 2 as the third argument, subject to the following restrictions:

(1) Only the first argument can be an operator name.
(2) Only normally undefined operators can be redefined.

OUTPUT* -- Set Output Association

OUTPUT(NAME,STRING,STRING)

The first argument is the name of a variable to be output-associated. The second argument is the name of the file or unit to which the association is to be made. In MTS a filename can be an FDname (MYFILE, *PRINT*, etc.) or a logical I/O unit name (SPRINT, 6, etc.). If the second argument is omitted, SPRINT is assumed.

The third argument is the format. If it is omitted, the output record length is taken from the FDname definition. Strings are transmitted

directly. If a string exceeds the specified length (maximum record length for variable length records), then it is split into segments as required.

The second possibility for a format argument is a single character. This is used for print files. The character given is a control character which is appended to the start of each record. Thus, the definition of the standard print file is

```
OUTPUT(.OUTPUT,,' ')
```

A third possibility for the format argument is a FORTRAN format. This is supplied for compatibility with BTL SNOBOL4 and should not be used except where required since format processing is inherently time consuming.

A restriction on the output function in SPITBOL is that only natural variables may be associated. It is not possible to output-associate array and table elements.

POS -- Define Positioning Pattern

```
POS(INTEGER) or POS(EXPRESSION)
```

POS returns a pattern which matches the null string after the indicated number of characters has been matched. The argument must be a nonnegative integer.

If an expression argument is given it is evaluated during pattern matching and must yield a nonnegative integer.

PROTOTYPE -- Retrieve Prototype

```
PROTOTYPE(ARRAY) or PROTOTYPE(TABLE)
```

PROTOTYPE returns the first argument used in the ARRAY or TABLE function call which created the argument.

REMDR -- Remainder

```
REMDR(INTEGER,INTEGER)
```

REMDR returns the remainder obtained on dividing the first argument by the second. The remainder has the same sign as the first argument (quotient).

REPLACE -- Translate Characters

```
REPLACE(STRING,STRING,STRING)
```

REPLACE returns the result of applying the transformations represented by the second and third arguments to the first argument. REPLACE fails if the second and third arguments are unequal in length or null.

REVERSE* -- Reverse String

REVERSE(STRING)

REVERSE returns the result of reversing the order of the characters in its string argument. Thus, REVERSE('ABC') = 'CBA'.

REWIND -- Reposition File

REWIND(STRING)

STRING is the name of an external file (not the name of a variable associated with the file). The named file is repositioned so that the next read or write operation starts at the first record of the file. Existing associations to the file are unaffected.

RPAD* -- Right Pad

RPAD(STRING,INTEGER,STRING)

RPAD is similar to LPAD except that the padding is done on the right.

RPOS -- Create Positioning Pattern

RPOS(INTEGER) or RPOS(EXPRESSION)

RPOS creates a pattern which will match null when the indicated number of characters remains to be matched. The integer argument must be nonnegative.

If an expression argument is used, it is evaluated during the pattern match and must yield a nonnegative integer.

RTAB -- Create Tabbing Pattern

RTAB(INTEGER) or RTAB(EXPRESSION)

RTAB returns a pattern which matches from the current location up to the point where the indicated number of characters remains to be matched. The argument must be a nonnegative integer.

If an expression is used, it is evaluated during pattern matching and must yield a nonnegative integer.

SETEXIT* -- Set Error Exit

SETEXIT(NAME) or SETEXIT()

The use of SETEXIT allows interception of any execution error. The argument to SETEXIT is a label to which control is passed if a subsequent error occurs, providing that the value of the keyword &ERRLIMIT is nonzero. The value of &ERRLIMIT is decremented by 1 when the error trap occurs. The SETEXIT call with a null argument causes

cancellation of the intercept. A subsequent error will terminate execution as usual with an error message.

The result returned by SETEXIT is the previous intercept setting (i.e., a label name or null if no intercept is set). This can be used to save and restore the SETEXIT conditions in a recursive environment.

The error intercept routine may inspect the error code stored in the keyword &ERRTYPE (see the section "Keywords"), and take one of the following actions:

(1) Terminate execution by transferring to the special label ABORT. This causes error processing to resume as though no error intercept had been set.
(2) Branching to the special label CONTINUE. This causes execution to resume by branching to the failure exit of the statement in error.
(3) Continue execution elsewhere by branching to some other section of the program. Note that if the error occurred inside a function, we are still 'down a level'.

The occurrence of an error cancels the error intercept. Thus, the error intercept routine must reissue the SETEXIT if required.

SIZE -- Get String Size

```
SIZE(STRING)
```

SIZE returns an integer count of the length of its string argument.

SPAN -- Create Scanning Pattern

```
SPAN(STRING) or SPAN(EXPRESSION)
```

SPAN creates a pattern matching a nonnull sequence of characters contained in the first argument, which must be a nonnull string.

If an expression argument is used, it is evaluated during pattern matching and must yield a nonnull string value.

STOPTR* -- Stop Trace

```
STOPTR(NAME,STRING)
```

STOPTR terminates tracing for the name given by the first argument. The second argument designates the sense in which the trace is to be stopped as follows:

'VALUE' or 'V' or null (omitted)	value
'LABEL' or 'L'	label
'FUNCTION' or 'F'	function call & return
'CALL' or 'C'	function call

'RETURN' or 'R'	function return
'KEYWORD' or 'K'	keyword tracing

SUBSTR* -- Extract Substring

SUBSTR(STRING,INTEGER,INTEGER)

SUBSTR extracts a substring from the first argument, the second argument specifies the first character (1 = start of string), and the third argument specifies the number of characters. SUBSTR fails if the substring is not a proper substring.

TAB -- Create Tabbing Pattern

TAB(INTEGER) or TAB(EXPRESSION)

TAB creates a pattern which matches from the current position up to the point where the indicated number of characters has been matched. The argument to TAB is a nonnegative integer.

If an expression argument is used, it is evaluated during pattern matching and must yield a nonnegative integer.

TABLE* -- Create Table

TABLE(INTEGER)

The TABLE function creates an associative table as in BTL SNOBOL4. However, in SPITBOL, the table is implemented internally using a hashing algorithm. The integer argument to TABLE is the number of hash headers used. The average number of searches is about M/2N where M is the number of entries in the table, and N is the number of hash headers. Since the overhead for hash headers is small compared to the size of a table element, a useful guide is to use an argument which is an estimate of the number of entries to be stored in the table.

Since the use of even numbers of headers can cause anomalies in the hashing algorithm, TABLE forces its argument to be odd by incrementing even arguments by one.

Note that this implementation of TABLE is compatible in that the call used in BTL SNOBOL4 works, though possibly not with maximum efficiency.

TIME -- Get Timer Value

TIME()

TIME returns the integer number of milliseconds of processor time since the start of execution. Note that the values obtained will be different (smaller) than those obtained with BTL SNOBOL4, since most programs run faster under SPITBOL.

TRACE* -- Initiate Trace

TRACE(NAME,STRING,ARGUMENT,NAME)

The TRACE function initiates a trace of the item whose name is given by the first argument. The second argument specifies the sense of the trace as follows:

'VALUE' 'V' or null (omitted)	value
'LABEL' or 'L'	label
'FUNCTION' or 'F'	function call & return
'CALL' or 'C'	function call
'RETURN' or 'R'	function return
'KEYWORD' or 'K'	keyword tracing

Keyword tracing is available only for the keywords &STCOUNT, &FCNLEVEL, and &ERRTYPE.

The third and fourth arguments are optional and are used to specify programmer-defined trace functions as in BTL SNOBOL4.

Tracing of array elements, table elements, and program-defined data-
| types types is not currently allowed.

TRIM -- Trim Trailing Blanks

TRIM(STRING)

TRIM returns the result of trimming trailing blanks from the argument string.

UNLOAD* -- Unload Function

UNLOAD(STRING)

String is the name of an external function which is to be unloaded. The restriction in BTL SNOBOL4 concerning functions OPSYNed to loaded functions does not apply in SPITBOL. A function is not actually unloaded until all functions OPSYNed to it have been unloaded. SPITBOL also allows the names of ordinary functions to appear in calls to UNLOAD. In this case, the result is merely to undefine the function.

VALUE-- Value Function

VALUE(ARGUMENT)

As in BTL SNOBOL4, the VALUE function returns the value of a string, a name, or a programmer-defined data type.

KEYWORDS

The following is a list of the keywords implemented in SPITBOL. The notation (R) after the name indicates that the keyword is read-only, that is, its value may not be modified by assignment.

A restriction in SPITBOL is that the only way to change a keyword value is by a direct assignment. Keywords may not appear in any other context requiring a name (for example as the right argument of binary $).

&ABEND Initially set to zero. If it is set to one when execution terminates, a SPITBOL system dump is given. This is useful only for system checkout.

&ABORT(R) Contains the pattern ABORT.

&ALPHABET(R) Contains the 256 characters of the EBCDIC set in their natural collating sequence.

&ANCHOR Set to zero for unanchored mode and to one for anchored pattern matching mode.

&ARB(R) Contains the pattern ARB.

&BAL(R) Contains the pattern BAL.

&CODE The value in &CODE is used as a system return code if this job is the last in a batch. It is normally set to zero.

&DUMP The standard value is zero. If the value is zero at the end of execution, then no symbolic dump is given. A value of one gives a dump including values of keywords and natural variables. If the value is two, the dump includes nonnull array, table, and program-defined datatype elements as well. The dump format is self-explanatory and deals with the case of branched structures including circular lists. If the value is 3, a core dump of the SPITBOL system is given. This is intended for system debugging and should be avoided.

&ERRLIMIT The maximum number of errors which can be trapped using the SETEXIT function. &ERRLIMIT is initially zero and is decremented each time a SETEXIT trap occurs. SETEXIT has no effect on normal error processing if &ERRLIMIT is zero.

&ERRTYPE If an execution error is intercepted with the use of the SETEXIT function, then the error code is stored as an integer in &ERRTYPE. The value stored is 1000*majorcode+minorcode. Thus, the error code 13.026 is stored as the integer 13026. &ERRTYPE may be assigned a value, in which case an immediate error is signaled. This may be useful in signaling program-detected errors. If such an error is intercepted, then either the standard error message appropriate to the major

code assigned is printed, or a standard message USER ISSUED ERROR MESSAGE is printed if the major code is not in the standard range (1-14).

&FAIL(R) Contains the pattern FAIL.

&FENCE(R) Contains the pattern FENCE.

&FNCLEVEL(R) Contains the current function nesting level.

&FTRACE If the value is greater than zero, all function calls and returns are traced. If the value is zero or negative, no trace output is generated. Each line of the trace output decrements the value by one. The initial value is zero.

&FULLSCAN The standard value is zero (QUICKSCAN pattern matching mode). The value is set to one to obtain FULLSCAN mode.

&INPUT Set to one for normal input (standard value). If set to zero, all input associations are ignored.

&LASTNO(R) Contains the number of the last statement executed.

&MAXLNGTH Contains the maximum permitted string length. This value may not exceed 32758. The default is 5000.

&OUTPUT Set to one for normal output (standard value). If set to zero, all output associations are ignored.

&REM(R) Contains the pattern REM.

&RTNTYPE(R) Contains 'RETURN', 'FRETURN', or 'NRETURN' depending on the type of function return most recently executed.

&STCOUNT(R) The number of statements executed so far.

&STLIMIT The maximum number of statements allowed to be executed. The initial value is 50000. The maximum value allowed is 2**31-1 = 2,147,483,647.

&STNO(R) The number of the current statement.

&SUCCEED(R) Contains the pattern SUCCEED.

| &TRACE If the value of &TRACE is greater than zero, all tracing specified by the TRACE function is performed. Whenever a tracing event takes place, &TRACE is decremented by one. This self-extinguishing trace counter prevents excessive debugging output. &TRACE is unaffected by tracing events caused by &FTRACE. The initial value of &TRACE is zero.

&TRIM Set to zero for normal input mode (standard value). If the value is set to one, all input records are automatically

trimmed (trailing blanks removed) by the SPITBOL I/O routines. Note that MTS may or may not trim all but one trailing blank depending on which device is being read and how it was attached in MTS.

CONTROL CARDS

Control cards are identified by a minus sign in column one. They may occur anywhere in a source program and take effect when they are encountered. Most of these control card types are special features of SPITBOL and are not implemented in BTL SNOBOL4.

Listing Control Cards

Listing control cards are used to alter the appearance of the listing; they have no other effect on the compilation or execution of the program.

-EJECT

The -EJECT control card causes the compilation listing to skip to the top of the next page. The current title and subtitle (if any) are printed at the top of the page.

-SPACE

The -SPACE control card causes spaces to be skipped on the current page. If -SPACE occurs with no operand, then one line is skipped. Alternatively, an unsigned integer can be given (separated by at least one space from the -SPACE) which represents the number of lines to be skipped. If there is insufficient space on the current page, -SPACE acts like -EJECT and the listing is spaced to the top of the next page.

-TITLE

The -TITLE card is used to supply a title for the source program listing. The text of the title is taken from columns 8-72 of the -TITLE card. The subtitle (if any), is cleared to blanks, and an eject to the next page occurs.

-STITL

The -STITL card is used to supply a subtitle for the source program listing. An eject occurs to the top of the next page and the current title (if any) and the newly supplied subtitle are printed. The text for the subtitle is taken from columns 8-72 of the -STITL card. Note that if both title and subtitle are to be changed, then the -TITLE card should precede the -STITL card.

September 1975

Option Control Cards

The option control cards allow selection of various compiler options. In each case, there are two modes. Two control cards allow switching from one mode to the other. The mode may be flipped back and forth within a single program. The full names are given for each control card; however, only the first four characters are examined, and the names may thus be abbreviated to four characters. Several control options may be specified on the same control card by separating the names with commas (no intervening spaces should occur). For example

-CODE,LIST,PRINT

In each of the cases listed below, the default option is shown first in the control options listed.

-LIST/-NOLIST/-UNLIST

Normally, the source statements are listed (-LIST option). The -NOLIST option causes suppression of this printout. This may be useful for established programs known to work, or for terminal output. Note that line numbers are always listed on the left, which is convenient for terminal output. If compilation errors are detected, the offending statements are printed regardless of the setting of the list mode. -UNLIST is provided for compatibility with BTL SNOBOL4 and is equivalent to -NOLIST.

-NOCODE/-CODE

The -CODE option causes a printout of the generated code in assembly language type format. This listing may be useful in determining how SPITBOL handles the compilation of various types of statements. The -NOCODE control option resets the normal mode of no code listing. It is permissible to use these cards in combination to obtain listings for selected sections of the source program. The code listing occurs after the end of the source listing, starting on a separate page, so that the source listing is not affected.

-NOPRINT/-PRINT

Normally, control cards are not printed (-NOPRINT). The -PRINT option causes control cards to be listed (provided that the -LIST option is in effect). This option may be useful if serialization is used for updating purposes.

-SINGLE/-DOUBLE

The compilation listing is normally single-spaced (-SINGLE). The -DOUBLE option causes double-spacing to be used, with a blank line between each listed line.

-OPTIMIZE/-NOOPTIMIZE

The SPITBOL compiler normally operates in an optimized mode in which the following assumptions are made:

(1) The values of BAL, ARB, FENCE, ABORT, REM, FAIL, and SUCCEED are not modified during execution.
(2) The standard system functions (see the section "Functions" for a full list) are not redefined.
(3) Function calls in a statement do not result in modification of values of variables referenced elsewhere in the same statement.

Violating these assumptions in -OPTIMIZE mode will produce incorrect results in that references to those functions and variables listed above will not yield the current value. For example, the pattern match below will succeed when compiled in -OPTIMIZE mode because the value of ARB was compiled as a constant.

```
ARB = 'A'  |  'B'
'123'  ARB        :S(LOOP)
```

The -NOOPTIMIZE control card specifies that the compiler should not make the above assumptions. This results in a higher level of compatibility with BTL SNOBOL4 at the expense of both space and speed. In some cases, the loss of speed may be as much as a factor of ten. The optimizing mode may be switched on and off so that only isolated statements are compiled in nonoptimized mode. Note that it is the references to redefined functions which cause the trouble, not the actual definition itself.

-INxxx

The right margin for the input source scanner can be set to any value 1 $\leq$ xxx $\leq$ 255. The default value is the minimum of the pair (input record length for SCARDS, 255). Programs which use columns 73-80 for sequential IDs should set the margin at 72.

-NOSEQUENCE/-SEQUENCE

This option is relevant only if -IN72 is in effect. The normal mode (-NOSEQUENCE) ignores any serialization occurring in columns 73-80. If the -SEQUENCE option is taken, then the SPITBOL compiler tests to see whether the serialization is in correct ascending sequence. If an out-of-sequence card occurs, a message is printed, but no other action is taken (unless -NOERRORS is also specified at the time of the sequence error).

-ERRORS/-NOERRORS

Normally, execution is allowed even if compilation errors occur (-ERRORS). If a compilation error or a sequence error (with

-SEQUENCE on) occurs and the -NOERRORS option has been specified, then the execution of the program is suppressed.

-FAIL/-NOFAIL

In BTL SNOBOL4, and in SPITBOL with the -FAIL mode set, a failure in a statement with no conditional GOTO field is ignored, and the program execution resumes with the next statement in sequence. This convention often results in errors going undetected, particularly in the case of array references with out-of-range subscripts and pattern matches which are expected always to succeed. The -NOFAIL option changes this convention. If a statement having no conditional GOTO field is compiled under the -NOFAIL mode, and a failure occurs when the statement is executed, an execution error occurs and a suitable message is generated. The -NOFAIL option is particularly useful for student jobs and other situations where many small programs are being debugged.

-EXECUTE/-NOEXECUTE

Normally, execution is initiated following compilation. If the option -NOEXECUTE is set at the end of compilation, execution is inhibited. This is often useful in conjunction with the DECK and LOAD options used to generate object modules.

-COPY filename

The -COPY control card allows coding to be copied into the source stream from an external file. The compiler processes the text (lines) in the file, and then returns to the line following the -COPY card. In MTS the filename can be an FDname (MYFILE, *SOURCE*, etc.) or a logical I/O unit name (SCARDS, 7, etc.). The text copied may in itself contain -COPY cards up to a maximum nesting level of eight.

PROGRAMMING NOTES

The internal organization of SPITBOL is quite different from that of BTL SNOBOL4. Consequently, the relative speed of various operations differs. This section attempts to give some idea of what is going on inside, so the SPITBOL programmer can achieve maximum efficiency.

Space Considerations

The SPAN, BREAK, and BREAKX functions use translate-and-test tables. For one-character arguments, the tables are built into the system and require no additional space. For arguments longer than one character, tables must be built for each call. Each such table requires 260 bytes of storage. If the argument is deferred, no storage is required, but the execution of the pattern is much slower.

ANY and NOTANY allocate 16-byte tables (actually one bit position in a shared 256-byte table).

The space required for each element of an array is 8 bytes, in addition to storage required for a string or other structure. All numeric items require no additional space beyond the 8-byte item.

The space required for each nonnull element of a table is 24 bytes, in addition to space for a string or other structure. A table hash header is 4 bytes. Thus, the number of headers can be made reasonably large without using much additional space.

Program-defined datatypes require 8(F+1) bytes, where F is the number of fields. They are thus quite compact and can be used freely.

The memory required for dynamically compiled code (CODE function) is not reclaimed efficiently in the current version. Improvements will be attempted in future versions.

Each variable block requires 32 bytes. This space is a constant requirement, whether or not the variable name has a single use or multiple uses (label, function, variable, etc.). This space is never reclaimed once it has been allocated. Thus, it is inefficient to use variables to build a table with the $ operator. Instead, the TABLE datatype should be used.

The COLLECT function can be used to obtain more detailed information on memory utilization for various structures.

Speed Considerations

To a greater extent than is the case with BTL SNOBOL4, SPITBOL sacrifices some efficiency in encoding complex structures as strings. Arrays, tables, and program-defined datatypes should be used where possible. The latter are particularly efficient in SPITBOL.

A POS pattern may be used freely at the start of a pattern since SPITBOL optimizes this occurrence to prevent useless movements of the anchor point. This optimization (which is completely transparent) occurs in both QUICKSCAN and FULLSCAN modes.

Time for datatype conversions is relatively more noticeable in SPITBOL. Where efficiency is important, unnecessary conversions should be avoided.

For patterns which do not generate a large number of intermediate matches, the $ pattern assignment is, if anything, faster than the . pattern assignment and may be used freely.

SPITBOL precomputes all constant expressions before execution. When the OPTIMIZE mode is in effect (normal case), most patterns can be precomputed; thus, no efficiency is lost by writing patterns in line rather than

predefining them. Use of the unary * operator to defer computation is still useful in certain cases. For example, consider the following in-line pattern matches.

```
X POS(0) ARB N 'X'
X POS(0) ARB *N 'X'
```

The second form is more efficient, since the compiler can precompute the entire pattern. The use of deferred operands in QUICKSCAN mode when the pattern may back up may produce unexpected results (see The SNOBOL4 Programming Language for details).

BREAK, BREAKX, and SPAN are very fast, except that deferred arguments having more than one character are quite slow. ARBNO is quite slow.

ARB may be slow and should be avoided where fast patterns such as BREAK, BREAKX, and SPAN are possible.

The actual matching process can be much faster in FULLSCAN mode than in QUICKSCAN mode since the heuristics require time-consuming tests. If a match does not back up much, FULLSCAN may well be faster. A program should be run both ways to determine which is faster.

The SETEXIT error intercepts are fast and may be used for program control as well as debugging.

If a variable is traced or I/O-associated, references to the variable are substantially slowed down even if the trace and I/O associations are later removed.

The unary $ (indirect) operator applied to a string argument works differently in SPITBOL and corresponds to a hash search of existing variables. The process of applying $ to a name (including the name of a natural variable) is much faster, which is why SPITBOL returns a name instead of a string when the unary dot (name) operator is used with a natural variable. Thus, it is better to use names where possible, for example in passing labels indirectly.

The REPLACE function is optimized when the second argument is &ALPHABET. In this case, the third argument can be used as a translate table directly, and there is no need to construct a table dynamically. The REPLACE function itself can be used to construct the necessary third argument. Thus, the call

```
A = REPLACE(X,Y,Z)
```

may be replaced by the two calls

```
TBL = REPLACE(&ALPHABET,Y,Z)
A = REPLACE(X,&ALPHABET,TBL)
```

The first of these calls is slow and need only appear once. The second call is fast and could be executed repeatedly for various values of X.

RUNNING IN MTS

This section describes the implementation of SPITBOL in MTS and how this differs from the BTL SNOBOL4 implementation.

Parameters

In MTS the SPITBOL system is contained in two files: *SPITBOL and *SPITLIB. *SPITBOL contains the compiler for the SPITBOL language and *SPITLIB contains the execution-time routines required when running object modules produced by *SPITBOL. The behavior of *SPITBOL can be altered in two ways: via the PAR field on the $RUN command and via special control commands which appear as part of the source program (see the section "Control Cards"). The PAR field consists of a sequence of keywords and free verbs separated by commas or blanks. A semicolon may be used to terminate the PAR field; the text after the semicolon is not processed by the SPITBOL compiler but is available to the program via the SYSPAR external function. Free verbs may be negated by one of three prefixes: 'NO', '¬', or '-'. Any parameter may be abbreviated to a minimum substring (shown underlined below). The available parameters are:

ALIST or NOALIST

> If ALIST is specified, an object-code listing will be produced on SPRINT. The default is NOALIST. This parameter replaces the OLIST parameter.

BATCH or NOBATCH

> If BATCH is specified, the compiler will batch process input decks. The batch pseudo-end-of-file is "./*" in columns 1-3. The default is NOBATCH.

CSTAT or NOCSTAT

> If CSTAT is specified, compilation statistics are printed on SPRINT. The default is NOCSTAT.

DECK or NODECK

> If DECK is specified, an object module will be produced on SPUNCH. SPITBOL object modules must be run in concatenation with *SPITLIB. The default is NODECK.

DUMP=nnn

> At termination of execution, the SPITBOL dump function is called with "nnn" as the argument. The default is DUMP=0 which produces no dump.

ED<u>UMP=nnn</u>

If execution terminates abnormally, the SPITBOL dump function will be called with "nnn" as the argument. If SPRINT is assigned to a terminal, the default is EDUMP=0 which produces no dump; if SPRINT is not assigned to a terminal, the default is EDUMP=1 which generates a dump of natural variables and keywords.

ERRXEQ or NOERRXEQ

If ERRXEQ is specified, the compiled program is executed even if errors were detected during compilation. The default is ERRXEQ.

ESTAT or NOESTAT

If ESTAT is specified, execution statistics are produced on SPRINT. The default is NOESTAT.

EXECUTE or NOEXECUTE

If EXECUTE is specified, the compiled program is executed. The default is EXECUTE.

FAILCHK or NOFAILCHK

If FAILCHK is specified, execution is terminated with an error if a statement is executed that fails and there is no conditional "goto" field. This is a useful debugging tool. The default is NOFAILCHK.

INMGN=nnn

"nnn" is the right-hand margin for the compiler when scanning input source programs. The default is set to the minimum of the input record length for SCARDS and 255. Programs that have sequential IDs in columns 73-80 should compile with INMGN=72.

LINECNT=nnn

"nnn" is the number of lines per page for printed output. This must be in the range of $3 \leq nnn \leq 32767$. The default is 58.

LIST or NOLIST

If LIST is specified, a source listing of the compiled program is printed on SPRINT. The default is LIST unless SPRINT is assigned to a terminal. This parameter replaces the SLIST parameter.

LOAD or NOLOAD

If both LOAD and NODECK are specified, an object module will be produced on logical I/O unit 0. The default is NOLOAD.

OPTIMIZE or NOOPTIMIZE

If OPTIMIZE is specified, the object code produced will be optimized. This can reduce the CPU time and storage required for program execution. The default is OPTIMIZE.

PLIST or NOPLIST

If PLIST is specified, a list of parameter values will be printed on SPRINT. The default is NOPLIST.

SDUMP or NOSDUMP

If SDUMP is specified, a storage dump of the SPITBOL work areas will be produced on SPRINT if an internal SPITBOL error is detected. This is useful only for system debugging. The default is NOSDUMP.

SEQCHK or NOSEQCHK

If SEQCHK is specified and INMGN=72, the sequential ID field (columns 73-80) is checked for ascending order. Out-of-order cards will be flagged. The default is NOSEQCHK.

SIZE=nnn

"nnn" is the number of pages allocated for dynamic storage within the compiler. This must be in the range of $4 \leq nnn \leq 256$. The default is 20.

TIMECHK or NOTIMECHK

If TIMECHK is specified, SPITBOL will terminate execution of the user program 0.25 seconds before any specified time limit in order that a symbolic dump may be given. The default is TIMECHK.

XREF or NOXREF

A cross-reference listing of the symbols in the compiled program is printed on SPRINT. The default is NOXREF.

The default mode of operation for *SPITBOL is "compile and execute," exactly like *SNOBOL4. However, this can be altered to "compile and produce an object module" or "compile, produce an object module, and execute" as desired. The object modules produced are not stand-alone modules. They must be run with a collection of execution-time support routines which are located in *SPITLIB.

Example #1 - running in "compile and execute" mode with the source program in PROG.S:

```
$RUN *SPITBOL SCARDS=PROG.S
```

Example #2 - running in "compile and produce an object module" mode with the object module saved in PROG.O:

```
$RUN *SPITBOL SCARDS=PROG.S SPUNCH=PROG.O PAR=DECK,NOEX
```

Example #3 - running a previously compiled module:

```
$RUN PROG.O+*SPITLIB PAR=SIZE=4
```

A limited number of parameters are also available when running *SPITLIB. These are:

DUMP=nnn

At termination of execution, the SPITBOL dump function is called with "nnn" as the argument. The default is DUMP=0 which produces no dump.

EDUMP=nnn

If execution terminates abnormally, the SPITBOL dump function will be called with "nnn" as the argument. If SPRINT is assigned to a terminal, the default is EDUMP=0 which produces no dump; if SPRINT is not assigned to a terminal, the default is EDUMP=1 which generates a dump of natural variables and keywords.

ESTAT or NOESTAT

If ESTAT is specified, execution statistics are generated on SPRINT. The default is NOESTAT.

LINECNT=nnn

"nnn" is the number of lines per page for printed output. This must be in the range of 3 ≤ nnn ≤ 32767. The default is 58.

PLIST or NOPLIST

If PLIST is specified, a list of parameter values is printed on SPRINT. The default is PLIST.

SDUMP or NOSDUMP

If SDUMP is specified, a storage dump of the SPITBOL work areas will be produced on SPRINT if an internal SPITBOL error is detected. This is useful only for system debugging. The default is NOSDUMP.

SIZE=nnn

"nnn" is the number of pages of dynamic storage allocated. This must be in the range of 4 ≤ nnn ≤ 256. The default is 20.

TIMECHK or NOTIMECHK

If TIMECHK is specified, SPITBOL will terminate execution of the user program 0.25 seconds before any specified time limit in order that a symbolic dump may be given. The default is TIMECHK.

I/O in MTS

Both *SNOBOL4 and *SNOBOL4B were designed to use the FORTRAN library I/O routines. SPITBOL, however, has its own I/O routines. As a consequence, several features of I/O are different.

FIXED vs VARIABLE record format

*SNOBOL4 and *SNOBOL4B read input via a FIXED record format, the record length being specified via the input association function INPUT(,,). This means that all records read in are padded with blanks (if necessary) to the specified record length. In SPITBOL, records are read via VARIABLE record format which means that records of length less than or equal to the specified INPUT length are transmitted as-is (i.e., no padding occurs). Hence, programs which run correctly under *SNOBOL4 may not run correctly under *SPITBOL if they depend on a fixed input record length.

MTS logical I/O units

*SNOBOL4 and *SNOBOL4B were designed to do I/O via FORTRAN data set reference numbers (1,2,...). The FORTRAN library I/O routines assign MTS logical I/O units to FORTRAN data set reference numbers in the following way: if the MTS logical I/O unit of the same number has been assigned on the $RUN command, it should be used; otherwise SCARDS should be used for input and SPRINT for output.

SPITBOL uses MTS logical I/O units directly; no FORTRAN data set reference numbers are involved. The logical I/O units used are:

SCARDS - SNOBOL4 source program to be compiled followed by the data read via the default variable INPUT.

SPRINT - source listing, object listing, parameter listing, compiler diagnostics, execution-time diagnostics, and output via the default variable OUTPUT.

SPUNCH - object module if the DECK option was specified, and output from the default variable PUNCH.

0 - object module if NODECK and LOAD are specified.

SERCOM - prompting if a terminal.

GUSER - user responses if a terminal.

In *SNOBOL4 and *SNOBOL4B, the default associations for the pseudo-variables INPUT, OUTPUT, and PUNCH are FORTRAN data set reference numbers 5, 6, and 7. These in turn default to SCARDS and SPRINT if the MTS logical I/O units 5, 6, and 7 have not been specified on the $RUN command. In *SPITBOL, the default associations for INPUT, OUTPUT, and PUNCH are SCARDS, SPRINT, and SPUNCH, respectively. Note in particular that in batch mode, strings assigned to the pseudo-variable PUNCH will be written by default to the card punch in *SPITBOL and the printer in *SNOBOL4.

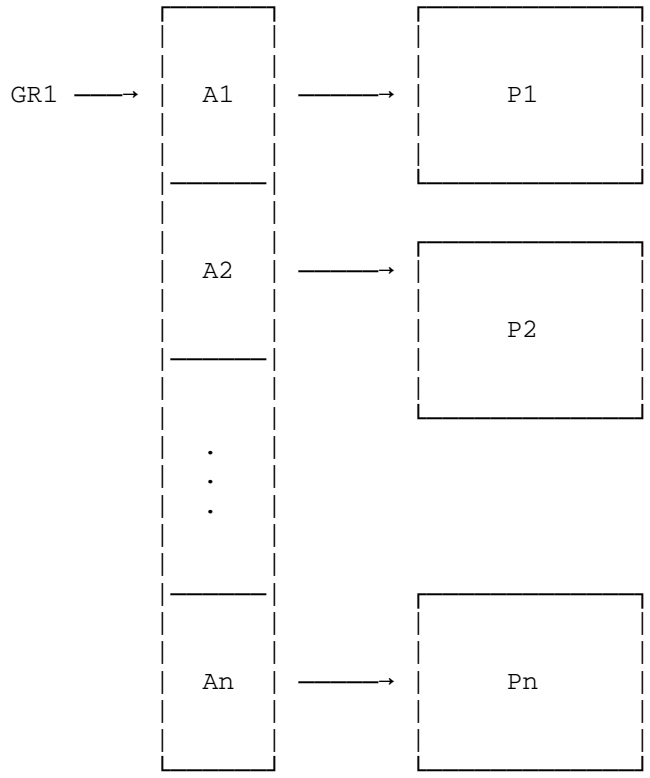
External Routines

As in SNOBOL4 there is no facility in SPITBOL for defining (and hence compiling) external routines. However, as in SNOBOL4 there is a facility for dynamically loading and linking to external routines which have been written in FORTRAN or assembly language.

When calling external functions, SPITBOL conforms to standard OS/360 S-type calling conventions, that is:

```
 GR1 = address of parameter list
GR13 = address of a save area
GR14 = return address
GR15 = function address
```

The external function being called must save and restore all general registers. For each argument being passed, the parameter list contains a pointer to the argument.



To cause an external function to be loaded, as well as to describe the characteristics and location of that function, it is necessary to call the SPITBOL LOAD function. The LOAD function takes two string parameters. The first gives the function name, parameter types, and returned-value type. The second is the name of a file containing the external function object code.

```
LOAD('fname(ptype,...)rtype','objectfile')
```

where "fname" is the name used both in the function calls in the SPITBOL program and the name of the entry point in the object module.

Parameters

Each "ptype" is the datatype that the corresponding actual parameter will be converted to for each call on "fname". Permissible values are given below, along with the corresponding internal representation for the call.

ptype	Representation
INTEGER	Fullword (4-byte) integer.
REAL	Fullword (4-byte) floating-point value.
DREAL	Doubleword (8-byte) floating-point value. The precision of this value is less than the normal 8-byte, floating-point value because the last byte is always zero.
STRING	Two words, the first of which is the address of the first character of the string and the second which is the length of the string.
INTERNAL	The doubleword internal SPITBOL descriptor. As there is no published description of the descriptor format, its use is limited.

This limited range of the parameter types restricts the number of existing functions that can be called. FORTRAN routines are restricted to those which take INTEGER*4, REAL*4, and REAL*8 parameters. An additional restriction on parameter passing is that no value may be returned by the called function in a parameter. A single returned value may be given as specified below.

Returned Values

"rtype" is the datatype of the value the function returns. It may be specified as one of the following:

rtype	Where Returned
INTEGER	GR0 contains the integer value.
REAL	FR0 contains the floating-point value.
DREAL	FR0 contains the floating-point value; however, the low-order byte will not be used by SPITBOL.
STRING	GR0 contains the address of a two-word region. The first word contains the address of the first character of the string and the second word contains the length of the string.
INTERNAL	GR0 contains the address of an 8-byte, internal SPITBOL descriptor.
omitted	If no return type is specified, the function call will always return the null string.

Success and Failure

Success or failure of an external function call may be indicated by making one of the two following return branches in the external function (after restoring the registers):

```
BR  R14      indicates success

B   4(,R14)  indicates failure
```

FORTRAN programs consequently are only able to make successful returns. The nonstandard nature of the failure return can be exploited only by assembly-language programs written explicitly for the purpose of interfacing to SPITBOL.

It should be noted that SPITBOL does not make use of return codes passed back in general register 15.

Loading a Function

The first step SPITBOL performs in loading an external function is to determine if it is already loaded. One reason that a function may already be loaded is that it is an alternate entry point to a function that was previously loaded. In this case no duplicate copy will be loaded. If one desires to load a new copy of a function, it is necessary to first unload the old copy with the built-in SPITBOL UNLOAD function.

If a function is not already loaded, SPITBOL calls the MTS loader, giving the MTS file specified in the second parameter "objectfile" to LOAD as the location of the corresponding object. If "objectfile" is not specified, by default, the resident-system library (LCSYMBOL) and the Elementary Function Library <EFL> will be searched.

Built-in External Functions

Two functions, SYSTOD and SYSPAR, are loaded with the SPITBOL system and may be activated by calling on the LOAD function without specifying an MTS file to load from.

The function SYSTOD may be used to obtain the time of day as a string of the form "HH:MM:SS". After LOADing, it may be called as follows:

```
LOAD('SYSTOD()STRING')
   .
   .
   .
TIME = SYSTOD()
```

The function SYSPAR may be used to obtain the text which follows a semicolon from the PAR field of the $RUN command. For example:

```
$RUN *SPITBOL ... PAR=SIZE=100;PLOUGH
   .
   .
   .
   LOAD('SYSPAR()STRING')
   .
   .
   .
   TEXT = SYSPAR()  will assign "PLOUGH" to TEXT
```

Other External Functions

There is currently no Computing Center supported library of functions written specifically to interface with SPITBOL. However, there is a collection of unsupported functions which have proved useful to many SPITBOL programmers. These functions are in the library UNSP:SPITLIB and are documented in UNSP:WRITEUPS.

Future development of the LOAD function is anticipated to extend the parameter types, interrogate return codes, and allow storing into parameters.

ERROR MESSAGES AND HANDLING

Compilation Error Messages

When the compiler detects an error, a flag is placed under the point in the statement where the error was discovered and processing of the statement in error is discontinued. Compilation continues with the next statement. Execution is not suppressed unless the -NOERRORS option has been set (see the section "Control Cards"). If an attempt is made to execute a statement found erroneous by the compiler, an execution error occurs. Compiler error messages are surrounded by ****** so they are easy to find. The following section describes the various error messages.

******ERROR IN GOTO FIELD******

The goto field is incorrectly formed.

Page Revised June 1979 September 1975

******ERROR IN NUMERIC ITEM******

A numeric item is illegally constructed.

******EXPRESSION IS TOO COMPLICATED FOR THE COMPILER******

The expression being compiled overflows work areas in SPITBOL. The expression must be broken into two or more statements.

******ILLEGAL CHARACTER******

The compiler detected a character which has no syntactic meaning in the SNOBOL4 language outside a string literal.

******ILLEGAL TRANSFER ADDRESS******

The operand on an END card is not a simple variable. The operand is ignored and execution starts with the first statement.

******ILLEGAL USE OF , ******

A comma has been used in an illegal context. The only legal uses of commas are to separate array subscripts and function arguments. Note that this error can be caused by accidentally inserting a blank between the function name and the left parenthesis.

******ILLEGAL USE OF < ******

The character < (array left bracket) has been used in a context where an array left bracket cannot legally occur.

******ILLEGAL USE OF) ******

A right parenthesis has been used in an illegal context.

******ILLEGAL USE OF > ******

An array right bracket has been used in an illegal context. This character can only be used to terminate a list of array subscripts.

******ILLEGAL USE OF = ******

An equal sign has been used in an illegal context. Only one equal sign may occur in a statement.

******INVALID -COPY CARD******

A -COPY card has an incorrect filename, or -COPY has been nested more than eight levels. Compilation proceeds after ignoring the erroneous card.

******LABEL HAS BEEN PREVIOUSLY DEFINED******

The statement has a label which has already been used. Compilation of the statement is discontinued and the earlier definition of the label is retained.

******MISSING END CARD SUPPLIED******

An end-of-file was read on the SPITBOL input file (SCARDS) during compilation. The compiler supplies an END card and initiates execution unless the -NOERRORS option is set.

******MISSING OPERAND******

This message is generated when the compiler expects an operand and does not find one. For example, A / / B, (C+)

******MISSING OPERATOR******

The compiler expected an operator and no operator was found. This occurs in situations like (X)A, where an operator is expected after the right parenthesis. This message is also given when the blanks surrounding a binary operator are omitted.

******NON-RECOVERABLE INPUT ERROR******

A nonrecoverable input error has been signaled on the SPITBOL input file (SCARDS). This is a fatal error which terminates compilation and prevents execution. Note that it also cancels any subsequent jobs when a batched run is being processed.

******PROGRAM TOO LONG FOR AVAILABLE STORAGE******

The storage required by the program exceeds available storage. Increase the region allocated and/or the H parameter in the compiler parameter field. Note that storage for execution time use has not yet been allocated. This must be taken into consideration in deciding how much additional memory to allocate. This is a fatal error which terminates compilation and prevents execution.

******UNBALANCED () OR <>******

This occurs if the parentheses or array brackets in a statement are not properly balanced.

******UNDEFINED TRANSFER ADDRESS******

The label used on an END card is not defined. The operand is ignored, and execution starts with the first statement.

******UNMATCHED QUOTE******

A string literal has been started but not properly terminated. Note that string literals cannot be split over continuation cards.

September 1975

Execution Error Messages

The execution package performs extensive error checking. When an error is detected, execution is terminated with an error message unless the error is intercepted by means of the SETEXIT function. The message is accompanied by an error code of the form AA.BBB, where AA is the major code and BBB is the minor code. The major code refers to the message given (see below). The minor code further identifies the specific error. The following is a list and explanation of the error messages together with their major codes.

MAJOR = 1 ILLEGAL DATATYPE

In a context where a definite datatype is required, a value of the wrong datatype is given and the attempt to convert it to the correct datatype fails.

MAJOR = 2 UNEXPECTED FAILURE

A statement having no conditional GOTO failed with the -NOFAIL option set. This usually corresponds to an error such as an unexpected out-of-range subscript.

MAJOR = 3 ERROR IN ARRAY REFERENCE

An array reference is incorrect. Either the object referenced is not an array or table, or the wrong number of subscripts is given.

MAJOR = 4 COMPILER DETECTED ERROR

An attempt was made to execute a statement found erroneous by the compiler.

MAJOR = 5 ERROR IN REFERENCE TO KEYWORD

An error was made in a keyword reference. Either the operand of & is incorrect, or the value assigned to the keyword is incorrect.

MAJOR = 6 MEMORY OVERFLOW

Dynamic memory is exhausted. Note that this can occur as a result of runaway recursion in function references or pattern matching.

MAJOR = 7 EVALUATION OF GOTO FAILED

If a complex expression is used in the GOTO field, it is not allowed to fail. Such a failure within a GOTO expression did occur.

MAJOR = 8 ERROR IN GOTO

The operand of a GOTO must be a natural variable which is a defined label. Some other value was given. This error message is also given on a return from level zero.

MAJOR = 9 CALL TO UNDEFINED FUNCTION OR OPERATOR

A reference was made to an undefined function, or an undefined operator was used.

MAJOR = 10 ERROR IN ARITHMETIC OPERATION

This message covers a variety of arithmetic errors such as overflow, division by zero, etc.

MAJOR = 11 KEYWORD OR SYSTEM LIMIT EXCEEDED

This message is issued when any of the following limits is exceeded: time, page, or card system limits, &MAXLNGTH, &STLIMIT keyword limits. The minor code distinguishes the specific limit which was exceeded. (see the section "Error Codes").

MAJOR = 12 INPUT/OUTPUT OR OTHER SYSTEM ERROR

An error has been signaled by one of the operating system routines. Some examples are nonrecoverable: I/O error, attempt to load a nonexistent function, etc. Note that the minor codes for this message may differ from operating system to operating system, and have been edited here to reflect the MTS environment.

MAJOR = 13 INCORRECT VALUE FOR FUNCTION OR OPERATOR

An argument to a function or operand of an operator was of the right datatype, but was outside the range of values permitted for some particular use. For example, the null string is an illegal argument for the BREAK function.

MAJOR = 14 VALUE RETURNED WHERE NAME IS REQUIRED

This will occur in a context requiring a name (left side of =, GOTO expression, or right argument of unary $ or unary dot operators).

If &ERRTYPE is assigned a value xxyyy greater than 14999 or less than 1000, then it is assumed to be a user-defined error and the following message is printed:

xx.yyy USER ISSUED ERROR MESSAGE

Error Codes

This section gives detailed descriptions of the minor codes for all major codes. The public file *SPITERR also contains each error message at an MTS line number equal to its error-code number.

1.001 Evaluated result of deferred argument to POS is not an INTEGER

1.002 Evaluated result of deferred argument to RPOS is not an INTEGER

1.003 Evaluated result of deferred argument to RTAB is not an INTEGER

1.004 Evaluated result of deferred argument to TAB is not an INTEGER

1.005 Evaluated result of deferred argument to LEN is not an INTEGER

1.006 Evaluated result of deferred argument to ANY is not a STRING

1.007 Evaluated result of deferred argument to NOTANY is not a STRING

1.008 Evaluated result of deferred argument to SPAN is not a STRING

1.009 Evaluated result of deferred argument to BREAKX is not a STRING

1.010 Evaluated result of deferred argument to BREAK is not a STRING

1.011 Evaluated result of deferred expression used in a pattern match is not a STRING or PATTERN

1.012 Value to be stored in a keyword is not an INTEGER

1.013 Real argument to loaded function is not a REAL

1.014 Integer argument to loaded function is not an INTEGER

1.015 String argument to loaded function is not a STRING

1.016 Dreal argument to loaded function is not a DREAL

1.017 Operand of unary $ is not a NAME

1.018 Replacing right side in a pattern replacement is not a STRING

1.019 Subject of a pattern match is not a STRING

1.020 The pattern in a pattern match is not a PATTERN

1.021 Subscript in reference to one-dimensional array is not an INTEGER

1.022 Subscript in reference to a multi-dimensional array is not an INTEGER

1.023 A field function was applied to an inappropriate program-defined datatype

1.024 The left operand for alternation or concatenation is not a STRING or PATTERN

1.025 The right operand for alternation or concatenation is not a STRING or PATTERN

1.026 The argument to a field function is not a program-defined datatype

1.027 An operand of binary + is nonnumeric

1.028 An operand of binary - is nonnumeric

1.029 An operand of binary * is nonnumeric

1.030 An operand of binary / is nonnumeric

1.031 An argument to NE, EQ, LE, GE, LT, GT is nonnumeric

1.032 An operand of binary ** is nonnumeric

1.033 The operand of unary + is nonnumeric

1.034 The operand of unary - is nonnumeric

1.035 First argument to LEQ, LNE, LGT, LLT, LGE, or LLE is not a STRING

1.036 Second argument to LEQ, LNE, LGT, LLT, LGE, or LLE is not a STRING

1.037 Argument to SIZE is not a STRING

1.038 Left operand of binary $ or . is not a PATTERN

1.039 Argument to LEN is not an INTEGER or EXPRESSION

1.040 Argument to POS is not an INTEGER or EXPRESSION

1.041 Argument to TAB is not an INTEGER or EXPRESSION

1.042 Argument to RPOS is not an INTEGER or EXPRESSION

1.043 Argument to RTAB is not an INTEGER or EXPRESSION

1.044 Argument to SPAN is not a STRING or EXPRESSION

1.045 Argument to BREAKX is not a STRING or EXPRESSION

1.046 Argument to BREAK is not a STRING or EXPRESSION

1.047 Argument to NOTANY is not a STRING or EXPRESSION

1.048 Argument to ANY is not a STRING or EXPRESSION

1.049 Argument to VALUE is not a STRING, NAME, or correct programmer-defined datatype

1.050 Argument to ARBNO is not a PATTERN

1.051 First argument to APPLY is not the name of a function

1.052 First argument to ARG is not a NAME

1.053 Second argument to ARG is not an INTEGER

1.054 First argument to ARRAY is not a STRING

1.055 First argument to CLEAR is not a STRING

1.056 Argument to CODE is not a STRING

1.057 Argument to COLLECT is not an INTEGER

1.058 Second argument to CONVERT is not a STRING

1.059 Argument to DATA is not a STRING

1.060 First argument to DEFINE is not a STRING

1.061 Second argument to DEFINE is nonnull and is not the name of a label

1.062 Argument to DETACH is not the name of a natural variable

1.063 Second argument to DUPL is not an INTEGER

1.064 First argument to DUPL is not a STRING

1.065 Argument to ENDFILE is not a STRING

1.066 Argument to EVAL is not an EXPRESSION (or a STRING which could be converted into an EXPRESSION)

1.067 First argument to FIELD is not a NAME

1.068 Second argument to FIELD is not an INTEGER

1.069 First argument to INPUT is not the name of a natural variable

1.070 Filename (second argument) to INPUT is not a STRING

1.071 Record length (third argument) to INPUT is not an INTEGER

1.072 Argument to LOAD is not a STRING

1.073 First argument to LOC is not a NAME

1.074 Second argument to LOC is not an INTEGER

1.075 Third argument to LPAD is not a STRING

1.076 Second argument to LPAD is not an INTEGER

1.077 First argument to LPAD is not a STRING

1.078 First argument to OPSYN is not the name of a natural variable

1.079 Second argument to OPSYN is not a function name

1.080 First argument to OUTPUT is not the name of a natural variable

1.081 Filename (second argument) for OUTPUT function is not a STRING

1.082 Format specification (third argument) for OUTPUT function is not a STRING

1.083 Argument to PROTOTYPE is not an ARRAY or TABLE

1.084 Second argument to REMDR is not an INTEGER

1.085 First argument to REMDR is not an INTEGER

1.086 Third argument to REPLACE is not a STRING

1.087 Second argument to REPLACE is not a STRING

1.088 First argument to REPLACE is not a STRING

1.089 Argument to REVERSE is not a STRING

1.090 Argument to REWIND is not a STRING

1.091 Third argument to RPAD is not a STRING

1.092 Second argument to RPAD is not an INTEGER

1.093 First argument to RPAD is not a STRING

1.094 Argument to SETEXIT is not a label name

1.095 First argument to SUBSTR is not a STRING

1.096 Second argument to SUBSTR is not an INTEGER

1.097 Third argument to SUBSTR is not an INTEGER

1.098 Argument to TABLE is not an INTEGER

1.099 Argument to TRIM is not a STRING

1.100 Argument to UNLOAD is not the name of a function

2.001 Failure of a statement having no conditional GOTO with -NOFAIL option in effect

3.001 Array reference with one subscript refers to an object which is neither a TABLE nor an ARRAY

3.002 Multi-dimensional array reference refers to an object which is not an array

3.003 Wrong number of subscripts in an array reference

4.001 Attempted execution of a statement found erroneous by the compiler

5.001 An attempt was made to reference the keyword attribute of a nonnatural variable

5.002 Reference to an undefined keyword

5.003 An attempt was made to change the value of a keyword associated with a nonnatural variable

5.004 Attempt to change the value of an undefined keyword

5.005 Attempt to change the value of a protected keyword

6.001 Overflow in main dynamic storage area. This can occur as a result of runaway recursion in pattern matching or function reference, as well as from generation of too much data.

7.001 The evaluation of a complex GOTO expression failed

8.001 RETURN from function level zero

8.002 Transfer to an undefined label

8.003 A transfer to the label CONTINUE occurred, but no previous error had been intercepted

8.004 A transfer to the label ABORT occurred, but no previous error had been intercepted

8.005 Name used as a GOTO operand is not the name of a natural variable

8.006 The operand of a direct GOTO is not code.

9.001 Reference to an undefined function

9.002 Use of the undefined operator -- unary /

9.003	Use of the undefined operator -- binary &
9.004	Use of the undefined operator -- binary ¬
9.005	Use of the undefined operator -- binary @
9.006	Use of the undefined operator -- unary |
9.007	Use of the undefined operator -- unary #
9.008	Use of the undefined operator -- binary #
9.009	Use of the undefined operator -- binary ?
9.010	Use of the undefined operator -- unary %
9.011	Use of the undefined operator -- binary %
9.012	Use of the undefined operator -- unary !
10.001	Overflow in + - / or * of two DREALs
10.002	Overflow in + - / or * of two REALs
10.003	REAL division by zero
10.004	DREAL division by zero
10.005	Overflow in exponentiation of the form REAL ** INTEGER or DREAL ** INTEGER
10.006	Integer division by zero
10.007	Integer addition overflow
10.008	Integer subtraction overflow
10.009	Integer multiplication overflow
10.010	Negative exponent for INTEGER ** INTEGER
10.011	Overflow in integer exponentiation
10.012	Exponentiation of the form DREAL ** DREAL is not permitted
10.013	Exponentiation of the form REAL ** REAL is not permitted
10.014	Integer overflow for unary minus (happens only with largest negative number)
10.015	Attempted division by zero in REMDR function

11.001 Page limit (P parameter) exceeded

11.002 Card limit (C parameter) exceeded

11.003 Input record longer than &MAXLNGTH

11.004 Attempt to set &MAXLNGTH to a value greater than the maximum allowed (32758)

11.005 &STLIMIT set to a value less than the number of statements already executed

11.006 Statement limit (&STLIMIT) exceeded

11.007 Attempt to form a string longer than &MAXLNGTH by concatenation

11.008 A pattern structure has exceeded the maximum permitted size (32K bytes)

11.009 Time limit (T parameter) exceeded

11.010 Attempt to form a string longer than &MAXLNGTH in call to DUPL function

11.011 Attempt to form a string longer than &MAXLNGTH in call to LPAD function

11.012 Attempt to form a string longer than &MAXLNGTH in call to RPAD function

12.001 Filename given is null

12.002 Referenced file is nonexistent or unavailable

12.003 Function name for LOAD is > 8 characters

12.004 Noncorrectable input error

12.005 Noncorrectable output error

12.006 Attempt to read past end-of-file

12.007 Noncorrectable input error during loading of external module

12.008 Module for external function not found

12.009 Module to be unloaded is not currently loaded (this is probably an error in the SPITBOL system)

12.010 Attempt to REWIND standard SPITBOL I/O units: SCARDS, SPRINT, SPUNCH

12.011 Attempt to read from a file previously written on with no intervening REWIND

12.012 Attempt to write on a file previously read from with no intervening REWIND

12.013 Duplication factor or T operand (tab location) in an output (FORTRAN) format is zero

12.014 Illegal character in output (FORTRAN) format

12.015 Too many levels of parentheses in output (FORTRAN) format -- the limit is 10

12.016 Too many right parentheses in output (FORTRAN) format

12.017 T operand (tab location) is missing in output (FORTRAN) format

12.018 Operand for H (string literal) in output (FORTRAN) format extends beyond the end of the format

12.019 Output format containing more than one character does not start with a left parenthesis and cannot be interpreted as a FORTRAN-type format

12.020 Output format containing more than one character does not start with a right parenthesis and cannot be interpreted as a FORTRAN-type format

12.022 Error in opening file for output

12.023 Error in opening file for input

13.001 Evaluated result of deferred argument to POS is negative

13.002 Evaluated result of deferred argument to RPOS is negative

13.003 Evaluated result of deferred argument to RTAB is negative

13.004 Evaluated result of deferred argument to TAB is negative

13.005 Evaluated result of deferred argument to LEN is negative

13.006 Evaluated result of deferred argument to ANY is null

13.007 Evaluated result of deferred argument to NOTANY is null

13.008 Evaluated result of deferred argument to SPAN is null

13.009 Evaluated result of deferred argument to BREAKX is null

13.010 Evaluated result of deferred argument to BREAK is null

13.011 Operand of unary $ is null

13.012 Argument for LEN is negative

13.013 Argument for POS is negative

13.014 Argument for TAB is negative

13.015 Argument for RPOS is negative

13.016 Argument for RTAB is negative

13.017 SPAN argument is null

13.018 Argument for BREAKX is null

13.019 Argument for BREAK is null

13.020 NOTANY argument is null

13.021 ANY argument is null

13.022 Null first argument in call to the ARRAY function

13.023 An array bound in a call to the ARRAY function is null

13.024 An array bound in a call to the ARRAY function is nonnumeric

13.025 In the first argument to ARRAY, a subscript bound has two colons

13.026 An array lower bound in a call to the ARRAY function is not in the range -32768 < LBD < +32768

13.027 An array dimension (HBD-LBD+1) in a call to the ARRAY function is not in the range 0 < DIM < 32768

13.028 Name in CLEAR first argument is null

13.029 Array argument for CONVERT to TABLE is not two-dimensional.

13.030 Argument to DATA is null

13.031 Datatype name in argument to DATA is null

13.032 Missing left parenthesis in DATA argument

13.033 Field name is null in DATA argument

13.034 DATA argument does not end with)

13.035 Too many fields (more than 30) in argument to DATA

13.036 First argument to DEFINE is null

13.037 Function name in first argument to DEFINE is missing (null)

13.038 First argument to DEFINE is missing a (

13.039 Argument name in first argument to DEFINE is null

13.040 First argument to DEFINE is missing a)

13.041 Null local name in first argument to DEFINE

13.042 Argument to ENDFILE is null

13.043 Argument to LOAD is null

13.044 Function name in argument to LOAD is null

13.045 Missing (in argument to LOAD

13.046 Missing argument to LOAD

13.047 Too many arguments (more than 64) in function to be LOADed

13.048 Argument to REWIND is null

13.049 Argument to TABLE is zero or negative

14.001 A function called by name returned a value

14.002 An expression other than a function call returned a value where a name was required

Calling External System Subroutines

The file *SPITSYSLIB (previously UNSP:SPITLIB) contains a library of SPITBOL-callable system subroutines. These subroutines may be used as the first argument of the SNOBOL4 LOAD statement. This file name may be specified as the second argument, or the program *OBJUTIL may be used to extract individual subroutines and place them in another file which can then be specified as the second argument.

The following subroutines are included in *SPITSYSLIB:

ITR, ITS, OSINT, RTI, RTS, SCMD, SCREATE, SCREPLY, SGDINFO, SGETFD, SGETRC, SGUINFO, SMTSCMD, SPITATTN, SPUTRC, SREAD, SSETPFX, SSNS, STI, STR, and SWRITE.

As stated above, the normal use of *SPITSYSLIB is to include the file name as the second argument in a call to the SPITBOL function LOAD. Documentation for these subroutines is given later in this section. The use of subroutine libraries is described in MTS Volume 3, System Subroutine Descriptions, and the MTS loader and the *OBJUTIL program are described in MTS Volume 5, System Services.

The file-name argument need not be used when the desired routines have all ready been loaded (by the MTS loader).

Page Revised May 1984 September 1975

Type Conversion Routines

Subroutine Description

Purpose: To change the type of a SPITBOL variable without performing any data conversion.

Location: *SPITSYSLIB

Calling Sequences:

```
real    = ITR(integer)
string  = ITS(string)
integer = RTI(real)
string  = RTS(real)
integer = STI(string)
real    = STR(string)
```

Parameters:

- real — a variable of type real.
- integer — a variable of type integer.
- string — a variable of type string.

Description: These functions always succeed. The functions that return strings, ITS and RTS, always return four characters. The functions that take string arguments, STI and STR, return a fullword zero if they are passed the null string, return the first four bytes of strings that are four characters long or longer, or return four bytes right-justified with leading binary zeros for strings with lengths of one, two, or three. None of these functions perform any data conversion; rather the data is moved unchanged from a variable of one type to a variable of another type.

SCMD

Subroutine Description

Purpose: To call the MTS subroutine CMD from a SPITBOL program.

Location: *SPITSYSLIB

Calling Sequence:

```
SCMD(command)
```

Parameter:

command is the character string that is to be passed to CMD as an MTS command.

Description: For a complete description of the CMD subroutine, see MTS Volume 3. SCMD normally succeeds and always returns the null string. A call to SCMD will fail if the command string is null or longer than 255 characters.

Example:

```
     LOAD('SCMD(STRING)','*SPITSYSLIB')
LOOP OUTPUT = "ENTER AN MTS COMMAND"
     COMMAND = TRIM(INPUT)        :F(END)
     SCMD(COMMAND)                :S(LOOP)
     OUTPUT = "COMMAND STRING TOO LONG OR SHORT"
END
```

SCREATE

Subroutine Description

Purpose: To call the MTS subroutine CREATE from a SPITBOL program.

Location: *SPITSYSLIB

Calling Sequence:

```
rc = SCREATE(file,size,type)
```

Parameters:

- file is a nonnull string containing the file name of the file to be created (it will be truncated to 17 characters if necessary).
- size is an integer specifying the size of the file to be created.
- type is an integer indicating the type of file to be created and the units associated with the size parameter.
- rc is a string that will be set to the null string to indicate a return code of zero from CREATE or to a four-character string representing the CREATE return code. A string of "0000" indicates that SCREATE was passed a null string for a file name.

Description: For a complete description of the CREATE subroutine, see MTS Volume 3. A maximum file size may be specified by the appropriate choice of a size value (size = fsize + maxsize * 2^{16}). The function SGETRC may also be used to obtain the CREATE return code.

Example:

```
     LOAD('SCREATE(STRING,INTEGER,INTEGER)STRING',
+'*SPITSYSLIB')
     DIFFER(SCREATE('-DATA',1,256))           :S(OOPS)
     OUTPUT = 'FILE "-DATA" HAS BEEN CREATED' :(END)
OOPS OUTPUT = 'SCREATE FAILED'
END
```

SCREPLY

Subroutine Description

Purpose: To call the MTS subroutine CANREPLY from SPITBOL programs.

Location: *SPITSYSLIB

Calling Sequence:

```
SCREPLY()
```

Description: For a complete description of the CANREPLY subroutine, see MTS Volume 3. SCREPLY always returns the null string. If the subroutine is called from batch mode it fails, otherwise it succeeds.

Example:

```
     LOAD('SCREPLY()','*SPITSYSLIB')
     OUTPUT = SCREPLY() "ENTER YOUR NAME"
     NAME = INPUT
       .
       .
       .
END
```

SGDINFO

Subroutine Description

Purpose: To call the MTS subroutine GDINFO from a SPITBOL program.

Location: *SPITSYSLIB

Calling Sequences:

```
string = SGDINFO(name)
string = SGDINFO2(unit)
```

Parameters:

name is a character string logical I/O unit name or number.

unit is an integer FDUB-pointer (as returned by SGETFD) or an integer logical I/O unit number (0 through 99).

Value Returned:

string is 44 bytes of information as described in MTS Volume 3.

Description: See MTS Volume 3 for a complete description of the GDINFO subroutine. SGDINFO normally succeeds but if name is passed as the null string or if GDINFO gives a nonzero return code, SGDINFO will fail. The function SGETRC may be used to obtain the GDINFO return code.

Example:

```
     LOAD('SGDINFO(STRING)STRING','*SPITSYSLIB')
     OUTPUT = "TYPE = " SUBSTR(SGDINFO('SCARDS'),5,4)
END
```

SGETFD

Subroutine Description

Purpose: To call the MTS subroutine GETFD from a SPITBOL program.

Location: *SPITSYSLIB

Calling Sequence:

```
fdub1 = SGETFD(fdname)
fdub2 = SGETFD2(fdname)
```

Parameter:

fdname is a character-string FDname terminated with a trailing blank.

Value Returned:

fdub1 is a FDUB-pointer returned as a string.
fdub2 is a FDUB-pointer returned as an integer.

Description: See MTS Volume 3 for a complete description of the GETFD subroutine. SGETFD normally succeeds and returns a FDUB-pointer. A failure return is made if the FDname string is null or GETFD gives a nonzero return code. The function SGETRC may be used to obtain the GETFD return code.

Example:

```
     LOAD('SGETFD(STRING)STRING','*SPITSYSLIB')
     FDUB = SGETFD('-LOAD ')  :F(OOPS)
       .
       .
       .
OOPS OUTPUT = "CALL TO SGETFD FAILED"
END
```

SGETRC, SPUTRC

Subroutine Description

Purpose: To allow SPITBOL programs to access the return code area used by several of the routines contained in *SPITSYSLIB.

Location: *SPITSYSLIB

Calling Sequences:

```
rc = SGETRC()
SPUTRC(newrc)
```

Parameters:

rc is the integer value that was stored in the return code area, SPITRCSA.

newrc is an integer to replace the current value stored in the return code area.

Description: Because external SPITBOL functions can only return a single value and a success/failure indication, it is not possible for many of the subroutines available in UNSP: SPITLIB to return both a value and a return code. To overcome this problem, the functions SGETRC and SPUTRC are provided. SGETRC always succeeds and returns the last return code stored in the return code area. SPUTRC normally succeeds and places an integer argument into the return code area. SPUTRC may fail if the return code area has not been defined.

The return code area is part of the SGETRC function and is defined when that function is loaded. For this reason it is necessary to load SGETRC before any functions that require the return code area are called. Subroutines that use the return code area will not fail if SGETRC has not been defined, but their return codes will not be available.

Example:

```
      LOAD('SGDINFO2(INTEGER)STRING','*SPITSYSLIB')
       LOAD('SGETRC()INTEGER','*SPITSYSLIB')
       INFO = SGDINFO2(0)
       RC = SGETRC()
       OUTPUT = 'TYPE = ' SUBSTR(INFO,5,4)  :S(END)
       OUTPUT = 'ERROR RETURN - RC=' RC
END
```

SGUINFO

Subroutine Description

Purpose: To call the MTS subroutine GUINFO from a SPITBOL program.

Location: *SPITSYSLIB

Calling Sequences:

```
string  = SGUINFO(name,length)
string  = SGUINFO2(number,length)
integer = SGUINFO3(name)
integer = SGUINFO4(number)
```

Parameters:

name is a one- to eight-character item name.
number is an integer item number.
length is an integer in the range 1 to 24 that specifies the length of the string to be returned.

Values Returned:

string is a string from 1 to 24 characters in length.
integer is an integer value.

Description: For a complete description of the GUINFO subroutine, see MTS Volume 3. This subroutine normally succeeds and returns the requested value. A failure return is made if a length of zero is specified, a null string is given for name, or GUINFO gives a nonzero return code. The function SGETRC may be used to obtain the GUINFO return code.

Example:

```
     LOAD('SGUINFO(STRING,INTEGER)STRING','*SPITSYSLIB')
     LOAD('SGUINFO4(INTEGER)INTEGER','*SPITSYSLIB')
     CCID = SGUINFO('SIGNONID',4)       :F(OOPS)
     PAGES = SGUINFO4(80)               :F(OOPS)S(END)
OOPS OUTPUT = "OOPS"
END
```

Page Revised May 1984 September 1975

SMTSCMD

Subroutine Description

Purpose: To call the MTS subroutine MTSCMD from a SPITBOL program.

Location: *SPITSYSLIB

Calling Sequence:

```
SMTSCMD(command)
```

Parameter:

command is a character string to be passed to MTSCMD as an command.

Description: For a complete description of the subroutine MTSCMD, see MTS Volume 3. This subroutine normally succeeds and always returns the null string. A failure return is made if the command string is null or longer than 255 characters.

Example:

```
     LOAD('SMTSCMD(STRING)','*SPITSYSLIB')
     OUTPUT = "USE $RES TO RESTART"
     SMTSCMD("$EDIT -SSF")
END
```

SPITATTN

Subroutine Description

Purpose: To detect attention interrupts from a SPITBOL program.

Location: *SPITSYSLIB

Calling Sequence:

SPITATTN('string')

Parameter:

string is one of three possible strings indicating the function to be performed.

Value Returned:

SPITATTN always returns the null string.

Description: Three calls to SPITATTN are possible:

SPITATTN('SET')

will establish an attention trap and will always succeed, even if a trap was already set.

SPITATTN('CLEAR')

will clear any trap set by SPITATTN and will always succeed, even of no trap was set.

SPITATTN('TEST')

will succeed if no attention interrupt has occurred and will fail if an attention interrupt has occurred. This call also clears the internal flag that indicates that an attention has occurred.

If an attention trap has been set using SPITATTN and a second attention interrupt is received before the first has been serviced by a call to SPITATTN('TEST'), MTS will process the interrupt as if no trap had been established.

Example:

```
      LOAD('SPITATTN(STRING)','*SPITSYSLIB')
      NO = 0
      SPITATTN('SET')
LOOP  SPITATTN('TEST')          :F(ATTN)
        .
        .
        .                       :S(DONE)F(DONE)
ATTN  OUTPUT = 'ATTN NO.  = ' NO
      NO = NO + 1               :(LOOP)
DONE  SPITATTN('CLEAR')
        .
        .
        .
END
```

SREAD

Subroutine Description

Purpose: To call the MTS subroutine READ from a SPITBOL program.

Location: *SPITSYSLIB

Calling Sequences:

```
string = SREAD(unit,line,mod)
string = SREAD2(unit2,line,mod)
```

Parameters:

unit is a string containing an MTS logical I/O unit name or FDUB-pointer.

unit2 is an integer FDUB-pointer or logical I/O unit number.

line is an integer MTS line number times 1000 to be used during indexed reads.

mod is an integer representing the I/O modifier bits (see the "I/O Modifiers" description in MTS Volume 3).

Values Returned:

SREAD returns a character representation of the MTS line number times 1000, concatenated to the special character '#', and followed by the actual input line (if any). SREAD succeeds if a return code zero is giveon, otherwise it fails. The function SGETRC may be used to obtain the return code from READ.

Description: See MTS Volume 3 for a complete description of the READ subroutine. If on a call to SREAD unit contains a null string, the unit, line, and mod values from the previous call will be reused. If there was no previous call to SREAD, MTS will print the message "Call to READ uses illegal parameters" and force execution to terminate.

Example:

```
     LOAD('SREAD(STRING,INTEGER,INTEGER)STRING',
+'*SPITSYSLIB')
     INLINE  = TRIM(SREAD('GUSER',))     :F(EOF)
     INLINE2 = SREAD()                   :F(EOF)S(END)
EOF  OUTPUT  = 'ENDFILE READ FROM GUSER'
END
```

SSETPFX

Subroutine Description

Purpose: To call the MTS subroutine SETPFX from a SPITBOL program.

Location: *SPITSYSLIB

Calling Sequence:

```
oldpfx = SSETPFX(newpfx)
```

Parameters:

newpfx is a character string, the first character of which will be used as the new user prefix character.

oldpfx is the previous user prefix returned as a one-character string.

Description: See MTS Volume 3 for a complete description of the SETPFX subroutine. SSETPFX normally succeeds, failing only if the argument string is null.

Example:

```
LOAD('SSETPFX(STRING)STRING','*SPITSYSLIB')
OLD = SSETPFX("?")
  .
  .
  .
SSETPFX(OLD)
  .
  .
  .
END
```

SSNS

Subroutine Description

Purpose: To return the "SNS" information for a given device.

Location: *SPITSYSLIB

Calling Sequence:

```
snsinfo = SSNS(name)
snsinfo = SSNS2(number)
```

Parameters:

name is a logical I/O unit name (e.g., SCARDS), or the character form of a logical I/O unit number (0 through 99).

number is an integer logical I/O unit number (0 through 99) or a FDUB-pointer (as returned by SGETFD).

Description: This subroutine calls the MTS CONTROL subroutine to obtain up to 72 bytes of "SNS" information. The information is returned as a character string. For information on the format of "SNS" data, see MTS Volume 4, Terminals and Networks in MTS, or MTS Volume 19, Tapes and Floppy Disks. For information on the CONTROL subroutine, see MTS Volume 3.

Calls to SSNS and SSNS2 normally succeed but will fail if the name string is null or a nonzero return code is given by the CONTROL subroutine. The function SGETRC may be used to obtain the return code from the control subroutine. The DSR (Device Support Routine) return code is not available.

Example:

```
LOAD('SSNS(STRING)STRING','*SPITSYSLIB')
ANSBACK = SUBSTR(SSNS("GUSER"),25,24)
  .
  .
  .
END
```

SWRITE

Subroutine Description

Purpose: To call the MTS subroutine WRITE from a SPITBOL program.

Location: *SPITSYSLIB

Calling Sequences:

```
SWRITE(unit,text,line,mod)
SWRITE2(unit2,text,line,mod)
```

Parameters:

unit is a string containing an MTS logical I/O unit name or FDUB-pointer.

unit2 is an integer FDUB-pointer or a logical I/O unit number.

text is the string to be written.

line is an integer MTS line number times 1000 to be used during indexed writes.

mod is an integer representing the I/O modifier bits (see the "I/O Modifiers" description in MTS Volume 3).

Values Returned:

SWRITE always returns the null string. SWRITE succeeds if a return code of zero is given, otherwise it fails. The function SGETRC may be used to obtain the return code from WRITE.

Description: See MTS Volume 3 for a complete description of the WRITE subroutine. If on a call to SWRITE, unit contains a null string, the unit, line, and mod values from the previous call will be reused. If there was no previous call to SWRITE, MTS will print the message "Call to WRITE uses illegal parameters" and force execution to terminate.

Example:

```
     LOAD('SWRITE(STRING,STRING,INTEGER,INTEGER)',
+'*SPITSYSLIB')
     SWRITE('SERCOM',OUTLINE)  :F(OOPS)
     SWRITE(,OUTLINE)          :F(OOPS)S(END)
OOPS OUTPUT = 'SWRITE FAILED'
END
```

SNOSTORM

INTRODUCTION

SNOSTORM is a SNOBOL (SPITBOL) preprocessor which accepts, in addition to standard SNOBOL statements, structuring statements (e.g., IF ELSE ENDIF). The SNOBOL language has extremely poor control structures and SNOSTORM adds a number of useful control statements to make structured programming much easier. There are also facilities for forming logical expressions and producing more readable listings.

DEFINITION OF TERMS

The following terms are used in the description of SNOSTORM:

sexp = a statement expression (explained below)
var = a variable
nexp = a numeric expression
ncon = a numeric constant
icon = an integer constant
xexp = any SNOBOL expression
xcon = any SNOBOL constant

Statement Expressions

There are no logical operators for combining logical results in SNOBOL. Often some tricks are used which make use of the fact that the predicate functions return the null string, but there is no general way to combine the many tests into one expression. To fix this omission, SNOSTORM allows the use of the logical operators NOT, AND, and OR. These may be used to form a statement expression (sexp) by the following rules:

sexp = any SNOBOL statement without a goto
| (sexp)
| NOT sexp
| sexp AND sexp
| sexp OR sexp

These operators have the following meaning:

NOT sexp	succeeds if sexp fails.
sexp1 AND sexp2	succeeds if both sexp1 succeeds and sexp2 succeeds.
sexp1 OR sexp2	succeeds if either sexp1 succeeds or sexp2 succeeds.

The precedence of these operators is such that NOT has the highest precedence, followed by AND, and then OR. The order of evaluation may be controlled with the use of parentheses.

Evaluation of these statement expressions takes place from left to right among equal precedence operations. However, the evaluation is optimized so that only the minimum number of SNOBOL statements need be evaluated. For example, if the first statement operand of an AND fails, there is no need to evaluate the second.

One or more blanks must surround each occurrence of AND, OR, and NOT.

Example:

```
       IF NOT (STM "WIMPY" OR LT(I,1000) AND X = A<I>), Y = 1
```

would generate

```
       STM "WIMPY" :S(aaa)
       LT(I,1000)  :F(bbb)
       X = A<I>    :S(aaa)
  bbb
       Y = 1
  aaa
```

CONTROL STRUCTURES

Four types of control structures are supplied in SNOSTORM:

(1) An IF structure for selecting groups of statements depending on one or more logical expressions.
(2) A LOOP structure which provides a means to repeat sections of code.
(3) A CASE structure which allows one of many sections of statements to be selected by some arbitrary value.
(4) A PROCEDURE structure which allows functions to be defined in a more obvious manner than is possible in SNOBOL.

The IF and LOOP statements may contain logical conditions which are expressed in terms of SNOBOL statements. The term sexp will be written to stand for these "statement expressions". The syntax of these is explained below.

IF STRUCTURES

Simple IF

The simple SNOSTORM IF is of the form

```
IF sexp, s
```

This is translated into

```
          sexp      :F(aaa)
          s
     aaa
```

where s is any SNOBOL or nonblock SNOSTORM statement. This performs s only if sexp succeeds. This statement eliminates the ugly practice of imbedding predicate functions in various parts of the SNOBOL statement to conditionally execute that statement.

Example:

```
IF IDENT(X,Y), S P = R
```

IF...ENDIF

The form of the simple block IF statement is:

```
IF sexp
   statements to be executed if sexp succeeds
ENDIF
```

This is translated into

```
          sexp :F(aaa)
             statements to be executed if sexp succeeds
     aaa
```

If the statement expression sexp succeeds, the statements enclosed between the IF and the ELSE will be executed, otherwise execution will proceed immediately after the ENDIF.

IF...ELSE...ENDIF

The form of this statement is:

```
        IF sexp
           statements to be executed if sexp succeeds.
        ELSE
           statements to be executed if sexp fails.
        ENDIF
```

This translates into

```
        sexp   :F(aaa)
           statements to be executed if sexp succeeds.
        :(bbb)
   aaa     statements to be executed if sexp fails.
   bbb
```

If sexp succeeds, the first clause will be executed, if sexp fails, the second clause will be executed. In both cases execution will then proceed with the first statement after the ENDIF.

IF...ELSEIF...ENDIF

A series of sequential decisions may be made using the following type of IF statement:

```
        IF sexp1
           performed if sexp1 succeeds.
        ELSEIF sexp2
           performed if previous test failed and sexp2 succeeds.
        ELSEIF sexp3
           performed if previous tests failed and sexp3 succeeds.
        ELSEIF sexpn
           performed if previous tests failed and sexpn succeeds.
        [ELSE]
           performed if none of previous tests succeeded.
        ENDIF
```

LOOP STRUCTURES

Many loops in SNOBOL are hidden within the execution of one statement. Nevertheless, it is often necessary to write multiple statement loops. SNOSTORM provides the LOOP...ENDLOOP statement for this. It is possible to add several clauses to these statements to control the type of looping.

The legal clauses are:

```
LOOP [for] [while] [until]
ENDLOOP [REPEAT [while] [until]]
```

Clauses specified on the LOOP statement are tested at the beginning of each iteration of the loop. Clauses on the ENDLOOP statement are tested at the end of each loop iteration. If more than one LOOP or ENDLOOP clause is given, the looping continues only if all clauses would continue execution. When multiple clauses are specified, the FOR clause is always processed first and WHILE and UNTIL clauses are processed in the order specified.

It is permissible to have clauses on both the LOOP and ENDLOOP statements.

LOOP

The unembellished LOOP statement makes no tests at the top of the loop. It is possible to specify conditions on the ENDLOOP to be tested at the bottom of the loop. Without termination conditions on either the LOOP or ENDLOOP statements, infinite looping must be avoided by a statement within the loop which branches out, such as: EXITLOOP or a goto. Of these, EXITLOOP would be preferable from a structured programming point of view.

Example:

```
LOOP
   *** COUNT TO INFINITY ***
   I = I + 1
ENDLOOP
```

LOOP FOR iteration

The LOOP...ENDLOOP structure with a for-clause provides a means of initializing an arithmetic variable, incrementing it each time through the | loop, and stopping when it goes beyond a specified value. The FOR clause | may also be used to traverse a list by making use of assignment in the BY | clause (see the BY clause description below). Because the termination of loops in SNOBOL is often made on the basis of some condition rather than upon a fixed bound, it is possible to omit the final value and use a WHILE clause to give the termination. If there is a final value specified, the iteration variable is compared to it before every iteration, including the first.

```
LOOP FOR var = nexp1 [BY nexp2] [TO nexp3]
   statements in the body of the loop
ENDLOOP
```

where nexp1 is the initial value, nexp2 is the increment (default 1), and nexp3 is the maximum value (default none).

The exact translation depends on the nature of nexp2 and nexp3. Changes to either of these values during the execution of the loop will have no effect on the number of times the loop is executed. That is, if either nexp2 or nexp3 references a variable, copies of the initial values of these expressions are made and then used in their places in the following generated code prototype.

The previous syntactic prototype would produce code something like the following:

```
        var = nexp1    :(aaa)
   bbb  var = var + nexp2
   aaa  GT(var,nexp3) :S(ccc)
           statements in the body of the loop
        :(bbb)
   ccc
```

Regardless of the order of the FOR, WHILE, and UNTIL clauses, the FOR clause is always processed first because it requires an initialization. Because of this initialization, the FOR clause is not allowed on the ENDLOOP statement.

The BY clause can have one of three forms:

(1) An expression. The value of the BY clause expression will be computed at the beginning of the loop and this value will be added on each iteration to the iteration variable. This is similar in action to iterated loops in other languages. This type of iteration would typically be used in traversing arrays.

(2) A negative integer constant. This special case of an expression can be used to cause the iteration variable to go from a high value to a low value. Just as with an expression, the value of the BY expression will be added to the iteration variable (thereby decrementing it). However, unlike an expression, if there is a TO value, loop termination will occur when the iteration variable is LESS THAN the final value. Specification of a negative CONSTANT is the only way to force the iteration variable to descend in value and to terminate when the iteration variable is less than the TO clause value. If a negative valued expression (not a constant) is specified, the iteration variable value will also descend, but the TO clause termination will only be made when the iteration variable value is greater than it (undoubtedly an error).

(3) An assignment statement. If the TO clause is a SNOBOL assignment, this assignment statement will be performed instead of any other form of incrementing. By also replacing the TO clause with a WHILE or UNTIL condition, this option allows for nonnumeric iterations, such as traversal of a linked list. A typical list traversal loop might look like the following:

```
|              LOOP  FOR X = HEAD BY X = NEXT(X) UNTIL IDENT(X,NULL)
|                ...
|              ENDLOOP
|
|         This would generate code like the following:
|
|                    X = HEAD       :(aaa)
|              bbb   X = NEXT(X)
|              aaa   IDENT(X,NULL)  :(zzz)
|                ...
|                    :(bbb)
|              zzz
```

Examples:

```
LOOP FOR I=1 TO N
   OUTPUT = NAME(A<I>)
   OUTPUT = '   ' ADDRESS(A<I>)
ENDLOOP

LOOP FOR I=1 WHILE ELEM = A<I>    Because subscript references
   OUTPUT = NAME(ELEM)            outside the bounds of an array
   OUTPUT = '   ' ADDRESS(ELEM)   cause a failure, this loop
ENDLOOP                           will stop after last entry.
```

LOOP WHILE sexp

The LOOP WHILE statement specifies a loop which is continued as long as the condition sexp succeeds at the beginning of each iteration.

```
LOOP WHILE sexp
   statements to be executed within loop
ENDLOOP
```

Page Revised May 1984 September 1975

is translated to

```
     aaa  sexp  :F(bbb)
             statements to be executed within loop
          :(aaa)
     bbb
```

LOOP UNTIL sexp

The LOOP UNTIL statement specifies a loop which is terminated at the beginning of any iteration in which the condition sexp succeeds.

```
          LOOP UNTIL sexp
             statements to be executed within loop
          ENDLOOP
```

is translated to

```
     aaa  sexp  :S(bbb)
              statements to be executed within loop
          :(aaa)
     bbb
```

ENDLOOP [REPEAT [while] [until]]

An ENDLOOP statement must be supplied to match each LOOP statement. The ENDLOOP statement may be written without any termination conditions, in which case it will generate a branch back to the top of the loop.

A termination condition may be specified with either a WHILE or UNTIL clause on the ENDLOOP. In this case, the word REPEAT must be inserted after the ENDLOOP keyword and before these clauses.

ENDLOOP REPEAT WHILE sexp tests the condition sexp at the end of each iteration and continues with the next iteration of the loop only if sexp succeeds.

ENDLOOP REPEAT UNTIL sexp tests the condition sexp at the end of each iteration and terminates execution of the loop only if sexp succeeds.

Both WHILE and UNTIL clauses may be specified on the ENDLOOP statement. The loop will be continued only if the WHILE sexp succeeds and the UNTIL sexp fails.

EXITLOOP

The EXITLOOP statement causes execution to continue with the statement immediately following the end of the immediately enclosing loop. The form of the EXITLOOP statement is:

```
EXITLOOP
```

NEXTLOOP

The NEXTLOOP statement will cause execution to proceed with the test for the next iteration of the immediately enclosing loop. The form of the NEXTLOOP statement is:

```
NEXTLOOP
```

CASE STRUCTURES

The DOCASE statement permits the selection of one of a number of groups of statements depending upon the value of an expression given in the DOCASE statement. Although there is no restriction on the datatype of the control expression, the only practical values to use are strings, integers, and reals because other values are very difficult to use correctly.

First, the control value specified on the DOCASE statement is evaluated. If this evaluation fails, control is passed to the ELSECASE clause. If the control value was successfully evaluated, a branch is made to the CASE clause which specifies the same value as the control value. If the control value is not specified in any CASE statement, execution proceeds with the ELSECASE clause.

The ELSECASE clause is required because an erroneous control value is often a source of programming errors. It is a good idea to put some error message in the ELSECASE clause if execution is never expected to reach that point.

Unlike a series of ELSEIF constructions, the DOCASE statement doesn't test for each case sequentially, but instead uses a table of labels to go directly to the correct case.

The DOCASE construction must always begin with a DOCASE statement. Next are one or more CASE statements, each preceding a group of statements to be executed for the control value specified on that CASE. The ELSECASE statement is next, and finally, the ENDCASE statement, terminating the DOCASE...ENDCASE structure.

No label may occur on either the CASE or ELSECASE statements.

Because the case structure is implemented using a table to select the appropriate clause to execute, SNOSTORM generates code to initialize this table at the beginning of execution. The method of making these initializations is described in the section "Initialization."

The form of this statement is:

```
DO CASE xexp
CASE xcon,...
   statements to be executed if IDENT(xexp,any xcon)
CASE xcon,...
   statements to be executed if IDENT(xexp,any xcon)
...
as many CASE statements as desired
...
ELSECASE
   statements to be executed if xexp was not identical to
   any of the xcons
ENDCASE
```

PROCEDURE STRUCTURES

PROCEDURE...ENDPROCEDURE

Procedures may be defined in SNOSTORM in such a way that the scope may be clearly indicated and the necessity of either separating the DEFINE from the procedure body or generating gotos around the body is eliminated. Procedure definitions may occur anywhere in the source text. SNOSTORM insures that they are defined during the initial phase of the program's execution. See the section "Initialization" for the details of this process. Additionally, SNOSTORM generates gotos around the body so it is not possible to accidentally flow into a procedure.

The form of a procedure definition is as follows:

```
  name PROCEDURE [([params])[locals]]
         .
         .
         .
       ENDPROCEDURE [{SUCCESS|SUCCEED|FAILURE|FAIL|NAME}]
```

where "name" is the procedure name, "params" the formal parameters, and "locals" the list of local variables. This is equivalent to the standard SNOBOL syntax for procedure definition of DEFINE("name(params)locals").

Execution of the ENDPROCEDURE statement causes one of three types of returns. If nothing is specified after the ENDPROCEDURE, or if SUCCESS (or

SUCCEED) is specified, a normal successful return is made (:(RETURN)). Specification of FAILURE (or FAIL) causes a failure return (:(FRETURN)) and specification of NAME causes a successful name return (:(NRETURN)).

Procedure definitions may not be imbedded in any other SNOSTORM block structures.

PROC and ENDPROC are the only acceptable abbreviations for PROCEDURE and ENDPROCEDURE, respectively.

EXITPROCEDURE

The EXITPROCEDURE statement provides a means of exiting a procedure without execution of the ENDPROCEDURE statement. The options specified on the EXITPROCEDURE act in the same way as on the ENDPROCEDURE statement. EXITPROC is the only acceptable abbreviation of EXITPROCEDURE.

```
EXITPROCEDURE [{SUCCESS|SUCCEED|FAILURE|FAIL|NAME}]
```

INITIALIZATION

Since there are no declarations in SNOBOL it is necessary for SNOSTORM to insure that certain things are dynamically defined at the beginning of execution.

Procedure and Case Initialization

Procedure (function) definitions are made by SNOSTORM at the beginning of execution of the program. Because SNOSTORM makes a single pass over the source program, it is not possible to produce these initializations at the beginning of the target program. Instead, a goto is produced at the beginning of the program that goes to the first place that needs initialization. From there a branch is made to the next place that is to be initialized, etc. Procedure definitions need the DEFINE to be executed at the beginning and case statements need a table defined during this process. At the end of the program a goto is generated back to a label at the beginning of the program.

INITIAL...ENDINITIAL

One of the frustrations of SNOBOL programming is that a pattern may be written in the pattern-matching statement, in which case one pays for the extra execution time necessary to build the pattern repeatedly, or a pattern may be preassigned to a variable at the beginning, in which case it is often difficult to remember what it is while looking at the pattern-matching statement. By allowing the programmer explicit access to the initializing chain used for the PROCEDURE and DOCASE constructions the conflict may be resolved.

An INITIAL construction may be used anywhere in the program and it is possible to specify as many INITIAL blocks as desired.

The INITIAL block is of the form:

```
INITIAL
   statements to be executed in the initialization phase
ENDINITIAL
```

Example:

```
INITIAL
   PARPAT = OPTB VARIABLE . V OPTB '=' OPTB CONPAT . C
ENDINITIAL
LOOP WHILE BACK PARPAT =
   ...
ENDLOOP
```

COMMENT STATEMENTS

In addition to the regular SNOBOL comment statement beginning with an '*' in column 1, two additional comment statements are provided in SNOSTORM. The SNOSTORM comments have three advantages: (1) they do not obscure program flow by cluttering the label field, (2) they do not interrupt the scope brackets which are printed when using the automatic indentation feature, and (3) they are indented to the current indentation level.

The COM and NOCOM options (see the "PAR Options" section) are used to control the passage of source program comments into the SNOBOL target program. The default value is NOCOM which inhibits the passage of any comments into the target program. If COM is specified, comments will be passed on into the target program, replacing the first character with an '*' if necessary.

SNOSTORM Comment Lines

Any statement that begins with an asterisk ('*') beyond column 1 is treated as a comment by SNOSTORM. The advantage of these comment lines over those which begin with an '*' in column 1 is that these comments are indented to the current structure nesting depth and they do not interrupt the scope brackets. Use of these comments will, in general, produce a more readable listing than the use of standard SNOBOL comments.

Blank lines are also considered to be comments. (SNOBOL treats blank lines as statements.) This means that blank lines may be used freely without causing additional execution. Blank lines do not interrupt the scope brackets which are printed in the listing to show the beginning and end of SNOSTORM structures.

LISTING CONTROL STATEMENTS

SNOSTORM interprets the following statements which help in producing more readable source listings. None of these statements generates any executable code.

None of the listing control statements may be labeled.

The SPITBOL listing control statements (-SPACE, -TITLE, and -STITL) are still processed by SNOSTORM for compatibility with an earlier version, but should be changed to the new form since they may not be processed indefinitely.

EJECT [icon]

The EJECT statement without the icon parameter causes the listing to continue at the top of the next page, if it is not already at the top of a page. The current title and subtitle, if any, will be printed at the top.

When EJECT has the icon parameter, the EJECT action is only taken if there are fewer than icon lines remaining on the current page. This is useful where there is a section of code or comments which should not be broken across a page boundary.

TITLE 'text of title'

The TITLE statement makes the text between the quotes into the current title and then causes the same action as the EJECT statement. The subtitle text is blanked by the occurrence of a TITLE statement.

SUBTITLE 'text of subtitle'

The SUBTITLE statement makes the text between the quotes into the current subtitle and then causes the same action as the EJECT statement. The EJECT action is not taken if the listing is already positioned at the top of the page as would be the case if a TITLE statement appeared immediately previous to this.

SPACE icon

The SPACE statement causes icon number of blank lines (or an EJECT to the top of the next page if there are fewer than icon lines remaining on the current page) to be generated at this point in the source listing.

An all-blank line is to be preferred to a SPACE 1 because, although both produce a blank line in the SNOSTORM listing, a blank line makes the text in the source file more readable.

Blank lines will not be printed at the top of a page.

LIST [keyword]

The LIST statement may be used to turn the source listing switch on or off. When the listing switch is on, a source listing will be printed; when off, it will not be printed. This listing switch is initially on. If no keyword follows LIST, ON is assumed. The keywords have the following meanings:

ON	the listing switch is turned on.
OFF	the listing switch is turned off.
PUSHON	the listing switch is saved on a stack and turned on.
PUSHOFF	the listing switch is saved on a stack and turned off.
POP	the listing switch is restored from the stack.

SNOSTORM LISTING

The listing produced by SNOSTORM contains the source line along with the statement number it will be given by SPITBOL and the line number of the source line.

Source Indentation

Quick recognition of the structure of the source program is essential to good programming. Each source line is indented to its proper structure level with a series of periods connecting the beginning and end of each structure. This scope bracket of periods may be changed to any arbitrary string by means of the INDENT option in the PAR field. Comments beginning with an '*' in column 1 are not indented when the automatic indentation option is in use; however, the SNOSTORM form of comments may be used to give uninterrupted scope brackets and indentation.

PAR OPTIONS

Options may be specified on the $RUN command for *SNOSTORM. If more than one option is specified, they should be separated by commas with no intervening blanks.

If specified, the SIZE option must appear first among the options. The order of the other options is irrelevant.

Examples:

```
$RUN *SNOSTORM ... PAR=;COM
$RUN *SNOSTORM ... PAR=;SIZE=100,INDENT='|   '
```

SIZE=icon

Default: SIZE=20

The SIZE parameter sets the amount of memory used by SNOSTORM and SPITBOL in the process of compiling the source program. It would not usually be necessary to increase this parameter, but it might be necessary for large source programs. The value of icon should not be less than 20 and may be increased as high as 256. If specified, this parameter must be placed before other parameters.

{COM|NOCOM}

Default: NOCOM

This option controls the passage of comments into the target module. If COM is specified, all comments in the source module are passed into the target module. If NOCOM is specified, no comments are passed to the target module.

CASE statements will also generate comments in the target module if the COM option is in effect.

{INDENT=string|NOINDENT}

Default: INDENT='. '

The INDENT option is followed by an '=' and a quoted string. This string is taken as the string to be replicated once for each indentation level. Automatic indentation may be turned off with the NOINDENT option.

{LIST|NOLIST}

Default: LIST

This option controls all further source program listing, either suppressing it with NOLIST or enabling it with LIST. It performs the same action as the LIST {ON|OFF} statement.

CONVERT

This option will cause SNOSTORM to convert obsolete SNOSTORM statements into a currently acceptable form. When this option is specified, SNOSTORM will read the old program from SCARDS and write a new version of the program on SPUNCH, preserving the line numbers above 1.000 as it does so. SNOSTORM will not translate during this process except for translating obsolete forms to new forms (see the section "Obsolete SNOSTORM Statements").

DEBUG

The DEBUG option is provided to aid in using *SPITDEBUG. This option causes SNOSTORM to put out a label for each unlabelled source statement. This label is based on the source statement file line number, and is only generated if the line number is at least one and is higher than any previously encountered file line number. The use of $CONTINUE WITH can cause labels not to be generated because of line number shifts. Leading zeros and trailing zeros and periods are omitted.

SNOSTORM generates an unconditional goto immediately preceding each label. This is necessary for the proper operation of the *SPITDEBUG BREAK command.

*SPITDEBUG will be included with the program with a -COPY *SPITDEBUG at the end of the initialization chain and a call on the *SPITDEBUG DEBUG function will be made immediately following that. Note that this call will be made after execution of the INITIAL structures, but before any execution of the rest of the program. This call will allow the user to make initializations or set breakpoints or enable tracing at the beginning of the program's execution.

See the *SPITDEBUG description for the details of its use.

The use of this option will consume additional processor time both for compilation and during the execution phase of the final object program. For this reason, it would generally be a good idea to recompile the program without the DEBUG option once the program is running correctly.

The execution time of an object program compiled with the DEBUG option will increase somewhat. However, there will be a substantial increase in the number of statements executed, about three times as many. This may necessitate an increase in &STLIMIT.

RUNNING SNOSTORM IN MTS

A SNOSTORM program may be translated to an object program as follows:

```
$RUN *SNOSTORM SCARDS=source SPRINT=listing SPUNCH=object
```

where "source" is the location of the SNOSTORM source program, "listing" is where the SNOSTORM listing should be written, and "object" is where the SNOBOL object program will be written.

Two processors are involved in the translation, first SNOSTORM, and then SPITBOL with the linkage automatically made between them if the SNOSTORM translation was without error. Using this combination of SNOSTORM and SPITBOL it is not possible to run the program in a "compile-and-execute" manner; an object program is always produced. The one difference in the object programs produced by *SNOSTORM and *SPITBOL is that a $CONTINUE WITH *SPITLIB will be appended to the object program produced by *SNOSTORM.

There are two temporary files created in the translation:

-PRINT* is a file which will contain the SPITBOL listing of the program produced by SNOSTORM. This name is chosen to correspond to the file that SPITDEBUG uses in printing the source file statement when errors are encountered. This file will be emptied before the listing is put into it.

-SNOOBJ is the file that is used to hold the output of the SNOSTORM translation. This file then becomes the input to SPITBOL.

Options may be passed to the SNOSTORM preprocessor in the PAR= field of the $RUN statement. See the section "PAR Options" for an explanation of how this is done.

It may be necessary to increase the SIZE parameter in the PAR field when translating large SNOSTORM programs.

RESTRICTIONS

There are several restrictions on the language accepted by SNOSTORM.

Multiple Statements

SNOSTORM doesn't allow multiple statements on input lines. Errors may be caused by doing so.

Reserved Words

Variable names which are identical to SNOSTORM keywords should not be used. Given below is a list of the reserved words and the context in which they are reserved. It is dangerous to use reserved words, even in a context in which they are not reserved. The use of such names may result in erroneous execution and may not necessarily be revealed by either SNOSTORM or SPITBOL compilation errors. Note that both parts of the two-word reserved words are reserved. The reserved words are:

Context: at the beginning of a statement

```
INITIAL
ENDINITIAL
IF
ELSE
ELSEIF
ENDIF
LOOP
ENDLOOP
DOCASE
CASE
ELSECASE
ENDCASE
DO
ENDDO
DOWHILE
ENDWHILE
DOUNTIL
ENDUNTIL
```

Context: in a sexp

NOT
AND
OR

Context: in a LOOP or ENDLOOP statement

FOR
BY
TO
WHILE
UNTIL
REPEAT

Labeled SNOSTORM Statements

Statement labels may not occur on the following SNOSTORM statements:

Listing control statements

EJECT
SPACE
TITLE
SUBTITLE
LIST

Statements with hidden branches

CASE
ELSE
ELSECASE
ELSEIF

A label would not be very meaningful on any of the listing control statements, and a label is confusing on each of the other statements because a GOTO is the first part of the generated code.

-COPY

SNOSTORM does not interpret -COPY statements in the source. Because the -COPY command is not processed, the source statement numbering in the SNOSTORM listing will be in error in programs which use -COPY. Additionally, the copied file may not contain any SNOSTORM statements. Both of these problems can be avoided with the use of $CONTINUE WITH f RETURN in place of -COPY f.

Error Handling

Error detection by the preprocessor is not performed on those portions of the source text which are presumed to be SNOBOL. This can result in SNOBOL errors on what were intended to be SNOSTORM statements because minor spelling or punctuation errors in the SNOSTORM statement were simply passed on to SNOBOL with no processing by SNOSTORM.

Errors in a SNOSTORM structure may result in further errors as a result of structure mismatch propagation.

Error messages are printed in the listing both at the point of detection and at the end. If SNOSTORM is being run in interactive mode, errors will also be listed on SERCOM.

OBSOLETE SNOSTORM STATEMENTS

The preliminary version of SNOSTORM handled the following statements. Although they will still be processed by SNOSTORM, they should be converted to the new forms to avoid warning messages. Old programs may be converted to new programs by running SNOSTORM with the CONVERT option in the PAR field. See the section "PAR Options" for a description.

Old Form	New Form
IFNOT s	IF NOT s
ELSEIFNOT s	ELSEIF NOT s
DO WHILE s	LOOP WHILE s
DO WHILENOT s	LOOP WHILE NOT s
ENDWHILE	ENDLOOP
EXITDO	EXITLOOP
EXITDO s	IF s, EXITLOOP
NEXTDO	NEXTLOOP
NEXTDO s	IF s, NEXTLOOP
DO UNTIL s...ENDUNTIL	LOOP ... ENDLOOP REPEAT UNTIL s
-TITLE	TITLE
-STITL	SUBTITLE
-SPACE	SPACE
-EJECT	EJECT

EXAMPLES

Example A:

The following is an example of a program to syllabify English words. (It needs a lot of work to be very useful).

```
 &ANCHOR = 1
 &TRIM = 1

 CONS = 'BCDFGHJKLMNPQRSTVWXZ'
 V = SPAN('AEIOUY')
 C = SPAN(CONS)

 SEP =  'B' . C1 NOTANY('LR')     . C2
+     | 'C' . C1 NOTANY('HLR')    . C2
+     | 'D' . C1 NOTANY('RW')     . C2
+     | 'F' . C1 NOTANY('LR')     . C2
+     | 'G' . C1 NOTANY('HLR')    . C2
+     | 'K' . C1 NOTANY('HLNR')   . C2
+     | 'M' . C1 NOTANY('N')      . C2
+     | 'P' . C1 NOTANY('FHLRST') . C2
+     | 'R' . C1 NOTANY('H')      . C2
+     | 'S' . C1 NOTANY('CHKLMNPTVW') . C2
+     | 'T' . C1 NOTANY('HRW')    . C2
+     | 'W' . C1 NOTANY('HR')     . C2
+     | ANY('JLNVX') . C1 LEN(1) . C2

 SUFFIX = ('TURE' | ANY('CGST') ('IAN' | 'ION' | 'IOUS') | 'MEN
+         | ANY('CDMNT') 'IAL' | 'FUL' | 'SHIP' | ANY('LN') 'ESS'
+         | ANY('CDGLMNTV') 'ENT' | ANY('AI') 'BLE' | 'LY'
+         | 'ES') ('S' | NULL) RPOS(0)

 PREFIX = ('IN' | 'CON' | 'DIS' | 'EX' | 'PRE' | 'NON' | 'SUB')
 REMWORD = (SEP ABORT | ((C V | V C) REM))

 AV1 = ARB ANY('AEIOUY') ARB
 AV2 = ARB ANY('AEIOUY') REM
     LOOP WHILE LINE = INPUT ' '
        LOOP WHILE LINE BREAK(' ') . WORD ' ' =
           IF DIFFER(WORD,NULL), OUTPUT = WORD ': ' SYL(WORD)
        ENDLOOP
     ENDLOOP

SYL PROCEDURE (WORD)X,Y,Z,Q,C1,C2,SUFF
       *** THIS PROCEDURE SYLLABIFIES A WORD USING THE RULES:
       *** (1) NONE IF WORD IS OF FORM V | VC | CV | CVC
       *** (2) AFTER PREFIXES E.G., IN-
       *** (3) BEFORE SUFFIXES E.G., -TION, -CIOUS
       *** (4) OTHERWISE AFTER A VOWEL
```

```
       ***     BUT DON'T LEAVE NEXT SYLLABLE STARTING WITH A
       ***     IMPOSSIBLE LETTER PAIR.

       IF WORD (V | V C | C V | C V C) RPOS(0)
          *** (1) NO SYLLABIFICATION IS NECESSARY ***
          SYL = WORD
       ELSE

          LOOP WHILE WORD PREFIX . X REMWORD . WORD
             *** (2) AFTER PREFIXES ***
             SYL = X '-'
          ENDLOOP

          LOOP WHILE WORD ARBWORD . WORD  SUFFIX . Y
             *** (3) BEFORE SUFFIXES ***
             SUF = '-' Y SUF
          ENDLOOP

          LOOP WHILE WORD (V | C V ) . X AV2 . WORD
             *** (4) - BREAK AFTER A VOWEL ***
             IF WORD C . Q =
                LOOP WHILE Q SEP =
                   *** SEPARATE THESE CONSTANTS ***
                   X = X C1
                   Q = C2 Q
                ENDLOOP
                WORD = Q WORD
             ENDIF
             SYL = SYL X '-'
          ENDLOOP

          IF DIFFER(WORD,NULL)
             SYL = SYL WORD SUF
          ELSE
             (SYL SUF) ARB . X '--' REM . Y
             SYL = X Y
          ENDIF
       ENDIF
    ENDPROCEDURE
END
```

Example B:

```
* From a program given in The SNOBOL4 Programming Language by
* Griswold, Poage, and Polonsky.
*
        ...
READ    OUTPUT    = INPUT                  :F(DISPLAY)
        TEXT      = OUTPUT
NEXT    TEXT      CHAR =                   :F(READ)
        COUNT<CH> = COUNT<CH> + 1          :(NEXT)
DISPLAY OUTPUT    =
LOOP    LETTERS   CHAR =                   :F(END)
```

```
        OUTPUT = NE(COUNT<CH>) CH ' OCCURS ' COUNT<CH> ' TIMES'
+                                             :(LOOP)
END

* Here is the same program, rewritten in SNOSTORM.  The horrid
* use of OUTPUT in the first two statements is also changed.
*
      ...
      LOOP WHILE TEXT = INPUT
         OUTPUT = TEXT
         LOOP WHILE TEXT CHAR =
            COUNT<CH> = COUNT<CH> + 1
         ENDLOOP
      ENDLOOP
      OUTPUT =
      LOOP WHILE LETTERS CHAR =
         IF NE(COUNT<CH>)
            OUTPUT = CH ' OCCURS ' COUNT<CH> ' TIMES'
         ENDIF
      ENDLOOP
END
```

September 1975

MTS 9: SNOBOL4 in MTS

September 1975

September 1975

MTS 9: SNOBOL4 in MTS

September 1975

September 1975

September 1975

September 1975

MTS 9: SNOBOL4 in MTS

September 1975

126.2 SNOBOL4 Blocks

September 1975

SNOBOL4 BLOCKS

INTRODUCTION

While strong in text analysis, SNOBOL4 has been traditionally weak in text composition (formatting, page layout, etc.). This section describes an extension to SNOBOL4 which is intended to provide a means of forming and manipulating printable output.

The methods for forming printable output center around a new data type called a block. A block is a three-dimensional (height, width, and depth) extension to a string (the third dimension, depth, is used for over-striking). Blocks may be printed, concatenated in any of three dimensions, and merged on the basis of programmer-defined connection points. Some blocks adapt in size and shape to their environment.

The extended version of SNOBOL4 is called SNOBOL4B. It is a fully supported, upward-compatible version of SNOBOL4.

Output is normally produced in SNOBOL4 by printing strings. Any string may be printed, and, since a string may be the result of an arbitrarily complex computation, any output configuration can be produced. As a practical matter, however, the programmer who must specify the formation of printable output often prefers to do so in a manner more closely aligned with the way he visualizes its construction. For example, it is normally a troublesome procedure to place two separately constructed images side by side on a printout page, whereas it would be easy and natural to express this construct as a concatenation.

As another example, consider writing a program which accepts a mathematical equation in linear form (say a FORTRAN expression) and prints it in conventional mathematical notation such that numerators are above denominators and exponents are superscripted. This represents a formidable programming problem when viewed as a sequence of printed lines. But, conceptually, the problem is neither intricate nor difficult. It consists of

(1) parsing the equation into operands and an operator

(2) obtaining the picture of each operand (by a recursive call)

(3) juxtaposing the component pictures apropos the operator.

Such problems can be solved relatively easily in a programming language which has data objects that can represent two-dimensional printable images and which has operations for directly manipulating these images. To correspond to the two-dimensionality of the medium, we extend the notation of a one-dimensional string of characters to two dimensions. To allow for

overstriking, an added dimension of depth is given to our now three-dimensional object, which we will call a block.

DEFINITION

A block is a three-dimensional aggregate of characters in the form of a right parallelepiped. It has a height H, a width W, and a depth D and it contains HxWxD characters (see Figure 1 below). A string is regarded as a special case of a block in which the height and depth are 1 and the width is the length of the string. The null string (the string of zero length) is the one exception. It is regarded as having a height, width, and depth of 0.

This definition of a block is overly simplistic in much the same way that it is an oversimplification to say that a river is a body of flowing water. It will be seen later how a block may have attributes in addition to its component characters. These additional properties need not be of concern immediately but will be introduced as they become relevant to this discussion. A block, whatever it precisely is, is intended to represent a parallelepiped of characters.

PRINTING

If "b" is an expression for a block, then the statement

```
PRINT(b)
```

will print the block. When a block is printed, a rectangular array of print positions is covered whose height is the height of the block and whose width is the width of the block. Each print position is struck a number of times equal to the depth of the block. The character printed at position "i,j" at strike "k" will be the character which appears at the "i,j,k"th position in the block. Very often blocks have only a depth of 1. Underlining requires a depth of at least 2. The depth of blocks is characteristically small but there is no intrinsic limit to the height, width, or depth of a block.

It should be emphasized at the outset that it was not intended that the third dimension of blocks provide a mechanism for representing three-dimensional objects in a computer. This is a natural and persistent misunderstanding. Rather, a block is primarily intended to represent a picture, i.e., a two-dimensional image. The third dimension provides an overstrike capability which considerably extends the variety of images which can be produced on a two-dimensional surface.

September 1975

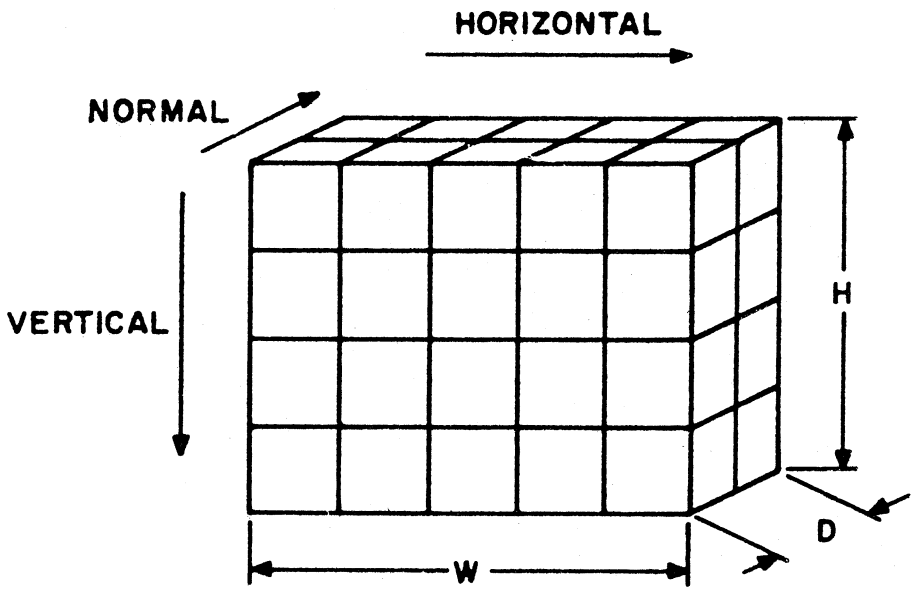


Figure 1: A block of height 4, width 5, and depth 2.

MTS 9: SNOBOL4 in MTS

September 1975

128.2 SNOBOL4 Blocks

September 1975

CONCATENATION

Analogous to the concatenation of strings, SNOBOL4B allows the concatenation of blocks, with the obvious extension that blocks can be concatenated in each of three dimensions.

Vertical concatenation is represented by the binary operator "%". Thus, the expression

$$b^1 \ \% \ b^2$$

where b^1 and b^2 are any block expressions, places block b^1 above block b^2. If the horizontal and/or normal dimensions of the two blocks disagree, the joining takes place through the centers. For example,

```
Q = '300' % '-------' % 'Z + 2'
```

will assign to Q a block of height 3, width 7, and depth 1. If Q is then printed as

```
PRINT(Q)
```

the characters

```
  300
-------
 Z + 2
```

will appear on the output page.

String concatenation is represented in SNOBOL4 by simply a blank binary operator. By extension, horizontal concatenation is denoted by an expression of the form

$$b^1 \ \langle blanks \rangle \ b^2$$

where b^1 and b^2 are any block expressions and where <blanks> is one or more blanks. Thus, if Q is the quotient as defined above, then

```
PRINT('X = ' Q)
```

will print

```
      300
X = -------
     Z + 2
```

Normal concatenation (normal to the plane of the paper) is represented by the binary operator "#". Thus

$$b^1 \ \# \ b^2$$

will place block b^1 in front of block b^2. As an example,

```
PRINT(('THIS' # '____') 'IS UNDERLINED')
```

will print

THIS IS UNDERLINED

on the output page.

Of the three concatenations, the operator with the highest precedence is <blanks>, followed by "%", followed by "#". Thus

A B % C # D E % F

is equivalent to

((A B) % C) # ((D E) % F)

Within the SNOBOL4 hierarchy of precedence operators, all three are higher than alternation "|" but lower than the arithmetics operators.

Notes on concatenation:

(1) The null string is ignored in concatenation.

(2) If blocks of unequal size are concatenated, a fill character is used to pad out the block to a full parallelepiped. The fill character is by default blank. It can be modified by the programmer if desired (see the subsequent section describing the &FILL keyword).

(3) In some cases the centers of concatenated blocks cannot be aligned exactly. For example

'X' % 'YZ'

cannot be a block in which the center of X is directly over the separation between the Y and Z. In such cases, the alignment will be off by 1/2 character. The algorithm used is as follows. The overall size of the resultant block of a concatenation is computed. If the difference between a component's size and the overall size is even, then the component can be centered exactly. If this difference is odd, the center is found and the component is placed one character closer to the left, the top, or the front, as appropriate.

(4) Integers and real (floating-point) numbers are automatically converted to string when they appear in a context which expects a block.

VACUOUS BLOCKS

The function reference

VER(n)

where "n" is any integer expression, will return a block of fill characters (blanks by default) whose height is "n" but whose width and depth are zero. A vertical line such as this is useful for specifying vertical separation between blocks. It can also serve as a convenient line skipper. For example

```
PRINT(VER(20))
```

will skip 20 lines on the output page.

The function reference

```
HOR(n)
```

will return a block of fill characters whose width is "n" but whose height and depth are zero. For example

```
PRINT(HOR(10) 'THIS IS INDENTED')
```

will print

```
THIS IS INDENTED
```

on the output page where 10 blanks precede the first character of 'THIS'.

The function reference

```
FRONT(h,w)
```

where "h" and "w" are integer expressions, will return a block of fill characters of height "h", of width "w", and of depth 0. For example,

```
PRINT(FRONT(60,130) # B)
```

will center block B in the middle of a 60 by 130 area which is roughly the size of a computer printout page.

The function reference

```
BOX(h,w,d)
```

will return a block of fill characters of height "h", of width "w", and of depth "d". The arguments to BOX should be non-negative integers. If one or more of the arguments to BOX is negative, the program will terminate in error.

The functions HOR, VER and FRONT are actually special cases of calls to BOX. Thus, HOR(n) is actually BOX(0,n,0). Using the function BOX, one may obtain the two planes and one line having different orientation than HOR, VER, and FRONT.

September 1975

REGISTRATION

When blocks of unequal size are joined together, it is sometimes desirable to align, not the centers of the blocks, but one of the edges. For example, in printing a column of figures with a heading, we may want the heading centered above the data but the data itself aligned on the right in the case of integers and on the left in the case of strings. Such a column is given below

```
COLUMN OF
ARBITRARY DATA

 13
216
  2
412
 66
300
```

In the case of the data above, the horizontal registration is on the right.

If a block "b" consists of a sequence of blocks joined vertically, or normally, then execution of the statement

```
HOR_REG(b) = 'RIGHT'
```

will cause alignment on the right. For example, the data above may appear on the input stream as

```
13
216
2
412
66
300
end-of-file
```

The following program will read in this data and print it aligned on the right (assume COL is initially null):

```
L    COL = COL % TRIM(INPUT)        :S(L)
     HOR_REG(COL) = 'RIGHT'
     PRINT('COLUMN OF' % 'ARBITRARY DATA' % VER(1) % COL)
```

In this program the first statement reads in the sequence of integers and joins them vertically. The statement is repeated until the end-of-file is encountered. Control then passes to the second statement, which sets the horizontal registration to the right. Note that all of the blocks that have been joined together so far are aligned on the right. The third statement appends the title to the blocks. The primitive function VER is called with

an argument of 1 so that 1 line will be skipped between the title and the data.

Other statements can set the vertical registration or the normal registration of a block and the effect is similar to that of setting the horizontal registration. The following is a complete list of registration-setting statements.

```
HOR_REG(b) = 'RIGHT'
HOR_REG(b) = 'LEFT'
HOR_REG(b) = ''

VER_REG(b) = 'TOP'
VER_REG(b) = 'BOTTOM'
VER_REG(b) = ''

NORM_REG(b) = 'FRONT'
NORM_REG(b) = 'REAR'
NORM_REG(b) = ''
```

Setting a registration to the null string indicates in an explicit way that the registration is to be centered. This is used to indicate that the blocks thus far joined together should be taken as a group in forming further joins. For example, if

```
B = 'A' % 'BCD'
B = B % 'E'
HOR_REG(B) = ''
```

then B has the value

```
 A
BCD
 E
```

Continuing, if

```
B = B % FGHIJKL'
HOR_REG(B) = 'LEFT'
```

then B has the value

```
 A
BCD
 E
FGHIJKL
```

If the first setting of the horizontal registration were not executed, then the last setting would have aligned the 'A' and the 'E' to the extreme left.

It is possible to set the registrations of the same block in two directions. For example:

```
B = FRONT(60,120) # '*'
HOR_REG(B) = 'RIGHT'
VER_REG(B) = 'BOTTOM'
```

places the asterisk in the lower right of the indicated field.

If the argument to the registration-setting statements is not a block (or if it will not automatically be converted to an integer or real block), then the statement causes the program to terminate in error. If the argument to the registration-setting statement is a non-settable block (VER(3), a string, etc.), then the statement fails.

BLOCK ORGANIZATIONS

A block that is not a string is a block structure. A block structure can be organized in one of several ways. A block formed by vertical concatenation has vertical organization. A block formed by horizontal concatenation has horizontal organization, and a block formed by concatenation in the normal direction has normal organization. These three types of organizations, as a group, are called contiguous organizations. By contrast, blocks returned by the functions VER, HOR, FRONT, and BOX have what is called physical organization.

Any block organized contiguously contains a sequence of two or more pointers to other blocks. These other blocks are called its daughters. It is this set of blocks, i.e., the daughters, which is aligned when the registration of a block is specified. The contiguous block is the mother of these daughters.

The block

```
'C' % VER(2) % 'A+B'
```

is organized vertically and has three daughters; the three daughters being the three operands shown. The statement

```
C = 'A' ('C' % VER(2) % 'A+B') ''
```

assigns to C a block, organized horizontally, with two daughters (the null string is ignored in concatenation). One of these daughters is a block, organized vertically, with three daughters.

If a block, organized contiguously in a certain direction, is concatenated in that direction, then the daughters of this block are incorporated as daughters of the resultant block. Thus, continuing from above, after executing

```
D = 'B' C HOR(7)
```

D is a block, organized horizontally, with four daughters. For this reason, each concatenation operator is associative. Hence

```
B1 % B2 % B3
B1 % (B2 % B3)
(B1 % B2) % B3
```

are equivalent, i.e., the resultant blocks have the same daughters in the same sequence.

A block organized contiguously is said to be set if the registration of the block has been set (by one of the 9 statements indicated in the previous section). Consider the application of a concatenation operator to an operand to produce a mother block. Normally, an operand becomes a daughter of the resulting mother. However, if the operand is organized in the same way as the mother, and if the operand has not been set, then the daughters of the operand (and not the operand itself) become daughters of the resulting mother. Two examples of this daughter-formation process are shown in Figures 2 and 3 below.

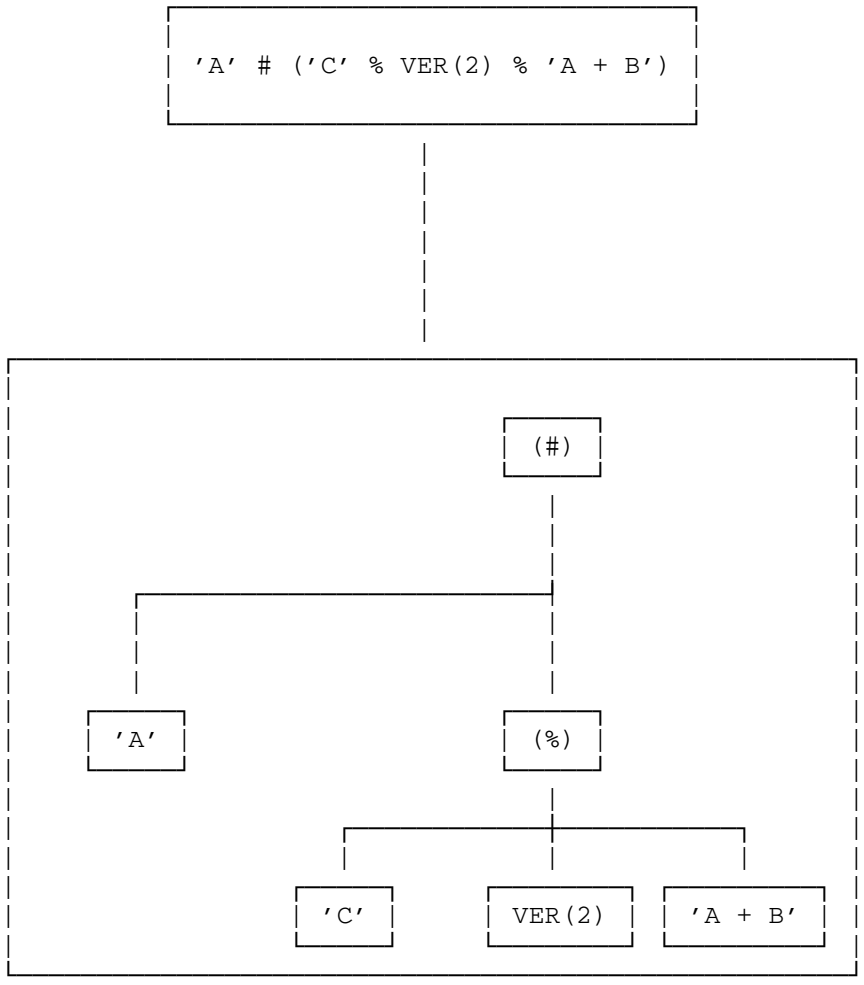


Figure 2: The tree structure resulting from the evaluation of the expression shown.

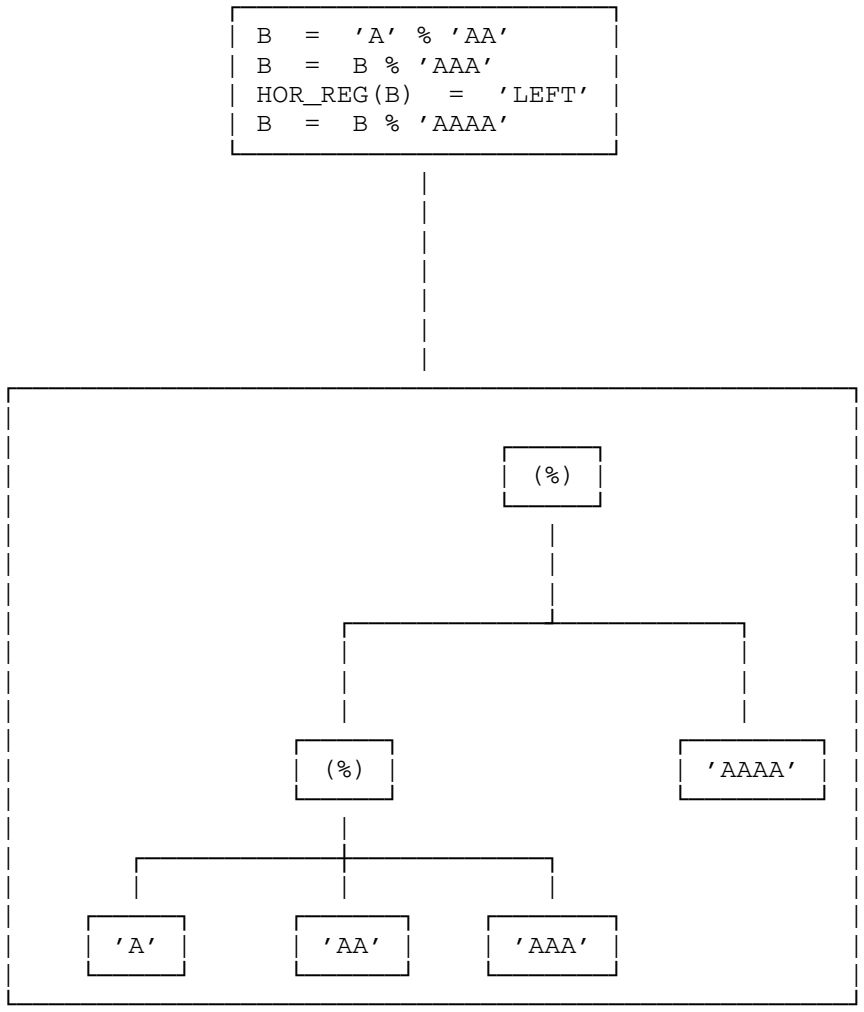


Figure 3: The organizational structure of B after executing the statements shown.

FIXING A BLOCK

Prior to printing a block, characters are moved, strings are concatenated, and blanks are added at appropriate places in order to develop completely the various parts of the block. This development of a physical counterpart to a given block can be done, without actually printing the block, by a call to the function FIX.

The function reference

```
FIX(b)
```

where "b" is any block expression, will return a block whose organization is physical. A physical block has no daughters and therefore contains no information as to how the block was formed. Thus,

```
FIX('AB' % 'CD')
```

and

```
FIX(('A' % 'C') ('B' % 'D'))
```

are indistinguishable. FIX can be used to increase efficiency by insuring that a block is built into a physical object just once even though it may be used several times. For example, in the following:

```
B = FIX(B)
A = B HOR(20) B
```

the block B is fixed prior to joining because otherwise it would be built into a physical object twice.

ADAPTIVE BLOCKS

An adaptive block is a block whose exact character configuration is dependent upon its environment, i.e., the set of neighboring blocks joined together with the adaptive block to form a larger block. There are two kinds of adaptive blocks, iterated and replicated.

Iterated Blocks

The function reference

```
IT(b)
```

where "b" is any expression for a block, returns a block whose organization is iterated; it has one daughter, i.e., the block "b". An iterated block will iterate its daughter a number of times in a direction orthogonal to the organization of its mother. The number of iterations will depend on and be limited by the size of the mother in the orthogonal directions. Several examples, shown in Figures 4 through 12, are intended to illustrate and clarify this definition.

```
A  =  'A'
C1  =  A % A % A
POLE  =  IT('|')
C  =  POLE  C1  POLE
PRINT(C)
```

```
|A|
|A|
|A|
```

Figure 4: The program in the box produces the output shown below the box. Here, C is the mother of 3 daughters, two of which are POLE, an iterated block. POLE iterates its daughter vertically 3 times to equal the size of C.

```
B  =  'B'
C2  =  B % B % B % B % B
C  =  C  C2  POLE
PRINT(C)
```

```
| |B|
|A|B|
|A|B|
|A|B|
| |B|
```

Figure 5: ...continuing from the previous figure. Two more daughters are joined to the mother C. The column C2 has a height of 5, giving a height of 5 to C. Each iterated block of C adapts to this height by iterating its daughter 5 times.

```
VER_REG(C)  =  'TOP'
PRINT(C)
```

```
|A|B|
|A|B|
|A|B|
| |B|
| |B|
```

Figure 6: ...continuing from the previous figure. The registration of a block having iterated daughters can be set as this figure illustrates.

```
POLE  =  IT('  ||  ')
C  =  POLE C1 POLE C2 POLE
PRINT(C)
```

```
||     ||  B  ||
||  A  ||  B  ||
||  A  ||  B  ||
||  A  ||  B  ||
||     ||  B  ||
```

Figure 7: ...continuing. The daughter of an iterated block need not be a single character as this example shows.

```
H  =  'COLUMN OF' % 'ARBITRARY DATA'  #  IT('_')
PRINT(H)
```

```
 COLUMN OF
ARBITRARY DATA
```

Figure 8: In this example, the iteration takes place in two dimensions. Since the mother of the iterated block is organized normally, the iteration takes place vertically (twice) and horizontally (14 times).

```
        U  =  IT('_')
        PRINT( ('COLUMN OF' # U) % ('ARBITRARY DATA' #U) )
```

COLUMN OF
ARBITRARY DATA

Figure 9: This figure demonstrates how the lines of a heading may be underlined separately.

```
        E  =  IT('*')
        PRINT(E % E ' ENCLOSED ' E % E)
```

```
************
* ENCLOSED *
************
```

Figure 10: A block may easily be surrounded by asterisks using an iterated block, as this example shows.

```
        E  =  IT(' *' % '* ')
        B  =  'DOUBLY' % ' ENCLOSED '
        PRINT(E % E B E % E)
```

```
 * * * * * * *
* * * * * * *
 *  DOUBLY   *
*  ENCLOSED *
 * * * * * * *
* * * * * * *
```

Figure 11: The example shown here illustrates that blocks other than strings may be iterated.

```
POLES = IT(HOR(5) % '|')
BARS  =  IT(VER(5) '-')
PRINT(FRONT(20,55) # POLES # BARS)
```

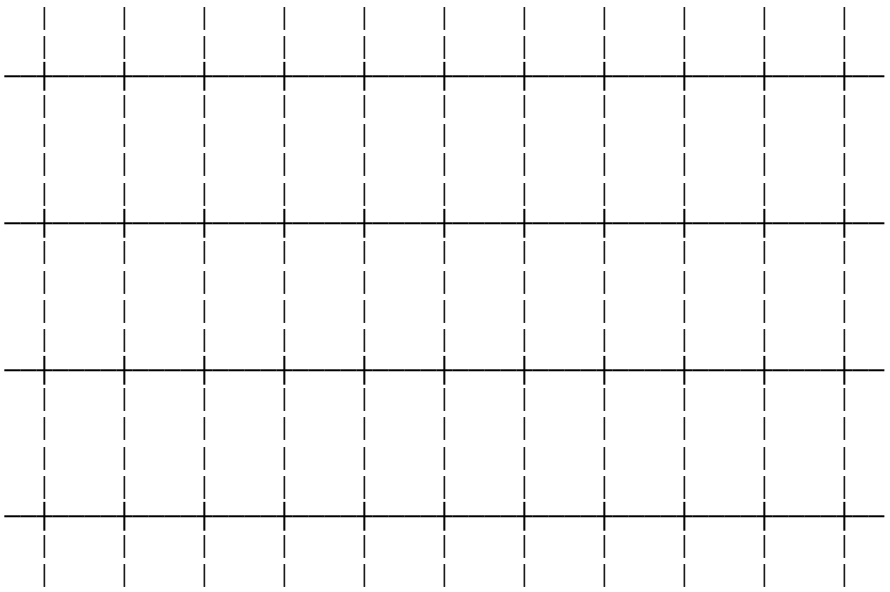


Figure 12: This example shows how a grid can be formed relatively easily.

Notes on iterated blocks:

(1) If an integral number of iterations does not completely fill the area in which the iterated block is to lie, then the block is placed within the aggregate just as any block is placed; that is, its alignment is subject to any registration commands placed upon the mother.

(2) An iterated block will always occupy a fixed amount of space in a direction colinear with the direction in which its parent is joined, even if there are zero iterations of the block.

(3) FIXing a block in which an iterated block is embedded sets the number of iterations of that block. Further joining will not affect the number of iterations.

(4) IT(IT(b)) is equivalent to IT(b).

Replicated Blocks

The function reference

REP(b)

where "b" is any expression denoting a block, returns a block whose organization is replicated. Like the iterated block, the replicated block adapts to its environment and consists of zero or more duplications of its daughter. A replicated block differs from an iterated block in that it causes a replication of its daughter in the direction of the joining of its mother, rather than in a direction orthogonal to the joining of its mother. The overall size of the mother, that is, the replicated block, is determined and limited by the size of the grandmother, i.e., the mother's mother. These last two statements assume that both mother and grandmother are contiguous and have different types of organization. Consideration of the general case is deferred to the section on surrogates.

Figures 13 through 19 illustrate the workings of replicated blocks and suggest some applications.

```
        LINE1  =  'INTRODUCTION'  REP('.') 1
        LINE2  =  'DEFINITIONS'  REP('.') 12
        PAGE  =  HOR(40) % LINE1 % LINE2
        PRINT(PAGE)
```

```
INTRODUCTION.........................1
DEFINITIONS.........................12
```

Figure 13: In the above example, LINE1 and LINE2 each have one replicated daughter. These expand in the direction of joining (horizontally) until the sizes of LINE1 and LINE2 are equal to the horizontal dimension of their mother, PAGE. The width of PAGE is 40 and there are 40 characters from the 'I' of 'INTRODUCTION' to the 1 inclusive.

September 1975

```
PAGE = 'CONTENTS OF' % VER(1) % PAGE
PRINT(PAGE)
```

```
                    CONTENTS OF

INTRODUCTION.............................1
DEFINITIONS..............................12
```

Figure 14: ...continuing. PAGE is a block and can be joined with other blocks as this example suggests.

```
A  =  'A'
B  =  'B'
C1  =  A % A % A
C2  =  B % B % B % B % B
C3  =  A % B % A % B
SP  =  REP(' ')
C  =  C1 SP C2 SP C3
PRINT(HOR(20) % C)
```

```
          B        A
A         B        B
A         B        A
A         B        B
          B
```

Figure 15: In this example, two replicated blocks SP are daughters of the same contiguous block C. These daughters expand equally (or as nearly equally as possible) in a horizontal direction until C is equal to the width (20) of its mother.

```
        VER_REG(C)  =  'TOP'
        PRINT(HOR(10) % C)
```

```
A    B   A
A    B   B
A    B   A
     B   B
     B
```

Figure 16: ...continuing. This figure illustrates that the registration of a block containing replicated blocks may be set, and also illustrates how the overall size of a block may be reset.

```
          DEFINE('OUTLINE(B,DH,DW)')
                              :(OUTLINE_END)
OUTLINE
          B  =  VER(DH) % HOR(DW) B HOR(DW) % VER(DH)
          B = '┌' REP('-') '┐' % IT('|') B IT('|') % '└' REP('-') '┘'
          OUTLINE  =  FIX(B)
                              :(RETURN)
OUTLINE_END
          PRINT(OUTLINE('ABC',1,2))
```

```
┌─────┐
│     │
│ ABC │
│     │
└─────┘
```

Figure 17: Given the special corner characters (┐┘┌└) available on the TN print train, it is relatively easy to surround textual matter with boxes. A function which OUTLINEs any given block is defined above. It has 3 arguments, a block B to be outlined, the vertical separation DH between the sides of the box and the block, and the horizontal separation DW. The first executable statement of the function adds on the white border. After this, two iterated blocks are used to form the sides of the box. The top and bottom are formed by means of replicated blocks. The block is fixed to keep the replicated blocks from expanding any further as they might if the block were thereafter joined vertically.

```
PRINT(OUTLINE(OUTLINE('ABC',1,2),,1))
```

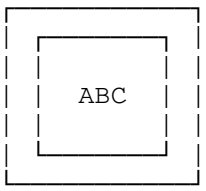


Figure 18: The function OUTLINE, defined in Figure 17, can outline any block, even an OUTLINEd one, as this figure illustrates.

```
SP  =  REP(' ')
S1  =  'A'  SP
S2  =  'AARDVARK'  SP
S3  =  'ABC'  SP
PRINT(S1 % S2 % S3)
```

```
A
AARDVARK
ABC
```

Figure 19: Appending a replication of blanks to a string as shown above has the effect of setting its horizontal registration to the left. This is less efficient than the registration setting statements. It is sometimes more convenient, however, to attach registration to the daughter rather than to the mother. This is especially true when not all daughters have the same registration.

Notes on replicated blocks:

(1) Replications of iterations and iterations of replications are permitted. In fact,

```
REP(IT(b))
```

is identical to

```
IT(REP(b))
```

(2) REP(REP(b)) is equivalent to REP(b).

DEFERRED BLOCKS

The function reference

DEF(name)

where "name" is the name of a variable (the name of the variable V is the string 'V'), will return a block whose organization is said to be deferred. When this block or any block containing this block is printed, the variable's value is printed. Examples are shown in Figures 20 and 21.

```
        B1  =  '---' % DEF('B2') % '---'
        B2  =  '*' % '*'
        PRINT(B1)
```

```
---
 *
 *
---
```

Figure 20: In this example, the deferred block is a daughter of block B1. The variable in question is B2 (whose name is the string 'B2'). The value that B2 has at the time of printing is embedded in the printed image.

```
LOOP    N  =  LT(N,5) N + 1            :F(DONE)
        TEST  =  TEST % 'PAGE NO.' N ' OF ' DEF('N')
                                       :(LOOP)
DONE    PRINT(TEST)
```

```
PAGE NO. 1 OF 5
PAGE NO. 2 OF 5
PAGE NO. 3 OF 5
PAGE NO. 4 OF 5
PAGE NO. 5 OF 5
```

Figure 21: In this example, control passes through the loop 5 times. Each time through, a new line is appended onto the growing block. The new line contains, in addition to some constant characters, the current value of N and the deferred value, i.e., the value that N will have when the PRINT is executed.

September 1975

Notes on deferred blocks:

(1) A deferred block is evaluated when the block in which it is contained is either PRINTed or FIXed.

(2) The effect of a deferred block is the same as if the deferred value had been originally used in place of the deferred block. If the deferred value is adaptive, then the block will adapt to whatever environment it finds at PRINTing or FIXing time.

NODES AND MERGERS

Although the operations of concatenation and registration and the adaptive and deferred blocks offer powerful tools for forming and manipulating printable output, there are yet a number of structures which escape a straightforward specification. Consider, for example, the block diagram shown in Figure 22. It is a relatively simple matter to construct the top, the bottom, or the sides of the diagram using concatenation alone. For example, if N1 is the block labeled N1, and N2 is the block labeled N2, then the upper part of the diagram can be specified as a horizontal concatenation of the form

```
N1 '----------' N2
```

However, it is not easy, by concatenation, to put together these four sides to form the indicated diagram. This is done by a process called merging. Merging blocks requires a means of specifying the subsections that are forced to coincide in the resulting block. These subsections form a kind of block called a node.

If "b" is any expression for a block, then

```
NODE(b)
```

will return a block whose organization is node. It has one daughter, i.e., its argument. Its size, shape, characters, and degree of adaptability are the same as its daughter. It differs from its daughter only in that it has a property of individuality, i.e., it is a node, which its daughter presumably does not have.

Two blocks containing the same node N can be merged so that the block N appears in the same position in the resulting block. For example,

```
N = NODE('O')
CROSS = 'CR' N 'SS'
WORD = 'W' % N % 'R' % 'D'
PRINT(MERGE(CROSS,WORD))
```

will print

W
CROSS
R
D

on the output page. Here the one node serves to bind the 2 blocks together at the O.

The value returned by MERGE is a block and can be used in any context that permits blocks, e.g., concatenation, iteration, etc. In general,

MERGE(b^1,b^2,...)

where b^1,b^2,... are block expressions, will return a block whose organization is merged (the block itself is sometimes referred to as a merger). The merging takes place in such a way that identical nodes are made to coincide. The resulting block, when formed physically, will be a parallelepiped just large enough to encompass all of the daughter blocks. There is no intrinsic limit to the number of arguments, or the number of nodes per argument.

We are almost, but not quite, ready to specify the construction of Figure 22. If we were to specify each side of the diagram rigidly and exactly, then we would have the responsibility of ensuring that the horizontal distances between nodes on the top of the diagram correspond in a precise way to the horizontal distance between nodes on the bottom of the diagram. A similar sort of difficulty would exist at the sides as well. To avoid this problem, it would be desirable to have an adaptive block which could automatically adjust to the spacing between nodes. This role is performed by the replicated block.

It was mentioned in the section on replicated blocks that the number of replications is determined by the grandmother. This is true even if the grandmother is formed by a merger. However, if this is the case, the size of the replicated block is determined, not by the overall size of the grandmother, but by node spacing of other daughters in the merger that formed the grandmother. This adaptability is used to advantage in constructing Figure 22. For example, the horizontal spacing between N3 and N4 is not specified explicitly but simply given as a replication of hyphens (REP('—')). The actual resulting distance is deduced by the merging process on the basis of horizontal node information contributed by the three other daughters of the merger.

```
        N1  =  NODE(OUTLINE('N1',2,4))
        N2  =  NODE(OUTLINE('N2',1,2))
        N3  =  NODE(OUTLINE('N3',3,1))
        N4  =  NODE(OUTLINE('N4',2,8))
        VERT  =  '|' % '|' % '|'
        B  =  MERGE(N1 '----------' N2, N1 % VERT % N3,
+               N2 % REP('|') % N4, N3 REP('-') N4)
        PRINT(B)
```

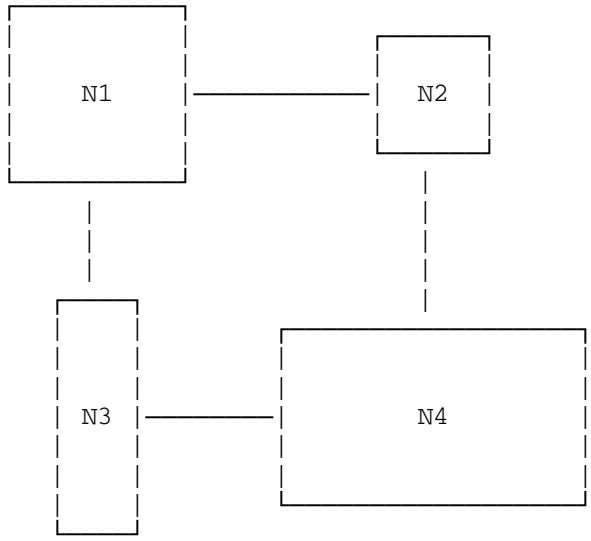


Figure 22: The formation of the block diagram shown is done by merging together 4 contiguous blocks. Nodes N1, N2, N3, and N4 are obtained by calls to the function OUTLINE (defined in Figure 17). These nodes serve to link the diagram together. The replicated blocks adapt to the space available between nodes.

Notes on merging:

(1) The function MERGE is associative and forms daughters in a manner analogous to the daughter formation process of contiguous blocks. Thus

```
B = MERGE(B1,B2)
B = MERGE(B,B3)
```

will (assuming B1,B2 and B3 are not themselves mergers) assign to B a merger having 3 daughters, B1, B2, and B3.

(2) Physical merging is deferred until the block is PRINTed or FIXed. For this reason intermediate merged blocks need not be well defined. If contradictory information is provided, some of the information is simply ignored.

(3) A node may be embedded within a block to an arbitrary depth and it will still be effective for the purpose of merging.

(4) If the information needed to bind nodes together is incomplete, a warning will be issued and a guess made.

(5) The arguments to MERGE cannot have iterated or replicated organizations.

(6) If a daughter of a contiguous block is a replicated block, then the contiguous block is not self-defined: it will adjust to its environment. By contrast, a merger is always self-defined.

(7) The physical merger of several blocks is a two-step process. On the first pass the overall size of the resulting block is determined as well as the position of every node in the block. An aggregate of blanks of the proper size is created. Then, in left-to-right sequence, each daughter is called on to embellish the growing image.

(8) If two daughters of a merger insert a character in the same location, the first insertion will be overwritten by the second unless the second happens to be the fill character (usually blank). For example, if block C has the characters

```
CCCCCCCCC
C       C
C   N   C
C       C
CCCCCCCCC
```

where 'N' is node N and if B is

```
  BBBBB
  B   B
  B   B
  B   B
  BBBBB
```

then MERGE(C, N % B) will result in

```
CCCCCCCCC
C       C
C   N   C
C BBBBB C
CCBCCCBCC
  B   B
  B   B
  BBBBB
```

(9) A replicated block will expand to fill the space between two blocks within a block diagram even if these blocks are not in the order indicated. For example, let D be a diagram containing nodes N1 and N2. Then,

```
MERGE(D, N1 % REP('|') % N2)
```

will draw a vertical line between N1 and N2 even if N2 is above N1. The only provision, of course, is that they be aligned vertically. If N1 and N2 are on the same plane, then a line of asterisks can be drawn from one to the other as follows:

```
PIN = NODE('*')
R = REP('*')
D = MERGE(D, N1 R PIN, PIN % R % N2)
```

Notes on nodes:

(1) Each call to the NODE function returns a unique block. This block can be passed from variable to variable. Thus, if the statements below were executed

```
N1 = NODE(B)
N2 = N1
```

then N1 and N2 would be identical. The above differs from

```
N1 = NODE(B)
N2 = NODE(B)
```

in which N1 and N2 have different but equivalent nodes as value.

(2) If two blocks containing the same node are concatenated, the location of the node in the resulting block is undefined.

(3) An iterated or a replicated block contains no nodes even though the argument may.

(4) A node may have an iterated or replicated block as an argument. Thus,

```
N = NODE(IT(b))
```

is possible. The iteration is done once per merger even though this node may appear in several contexts within the merger.

(5) FIXing blocks with node information - FIXing a block "b" such as

```
FIX(b)
```

where "b" has nodes embedded in it, will return a physical block in which information concerning the location of all of its nodes is retained. The returned block can then be merged with other blocks

on the basis of its node's positions. For reasons of storage efficiency, it may be desired to remove this node information; this can be done by giving a second argument to FIX equal to 1. Thus,

FIX(B,1)

will strip all node information from B.

Finally, it is possible to selectively retain some node information. This can be done by a call of the form

FIX(B,n^1,n^2,...)

where n^1,n^2,... are nodes. Here information pertaining to the location of nodes n^1,n^2,... is retained, whereas all other node information is removed.

THE &FILL KEYWORD

When blocks of unequal size are concatenated or when blocks are merged, the void areas are filled by a fill character. The fill character is specified by the keyword &FILL which is initially blank but which may be tested and set by the programmer. For example, the programmer may change the fill character to a period by executing the statement

&FILL = '.'

Such a specification may be made for the purpose of debugging where it is desired to distinguish between the white of the paper and the fill character of a block.

SURROGATES

The assumption made in the section on adaptive blocks was that the mother of an iterated block was contiguous and it controlled the size of the iterated block. It was also assumed that the mother and grandmother of a replicated block were contiguous, with the grandmother controlling the size and the mother controlling the direction of duplication of the replicated block. In the section on mergers, it was seen that the grandmother of a replicated block could also be a merged block, but there also it was assumed that the mother was contiguous. It is, however, not necessary that the mothers of iterated and replicated blocks be contiguous for these to adapt appropriately to ancestral blocks. In this section we will define the cases that are allowed and attempt to describe the adaptive procedures when the more general cases occur.

September 1975

By the nature of the daughter formation process, only contiguous blocks can have 2 or more daughters. We will refer to these blocks as a group as multi-daughtered blocks. The iterated, replicated, and node blocks are, by contrast, uni-daughtered. The surrogate of a block is defined as the nearest multi-daughtered ancestor. It is, in general, the surrogate of an iterated block that will control the number of iterations. The grandsurrogate is defined as the surrogates's surrogate. It is the grandsurrogate that will control the number of replications and the surrogate that will control the direction of replications for a replicated block.

For example, to draw a vertical line down the right-hand side of a block B, 3 characters in from the right-hand edge, the following statement can be executed:

```
B = B # IT(REP(' ') '|' HOR(3))
```

In the above line, the surrogate of the replicated block is the 3-daughtered contiguous block used as an argument to the IT function. Its grandsurrogate is the block formed by the concatenation in the normal direction. Hence, the number of replications is governed by the width of B, whereas the direction of replication is controlled by the horizontally organized block. The surrogate for the iterated block is the normally concatenated block. The number of iterations in the vertical direction is dictated by the height of B. The number of iterations in the horizontal direction will be 1. This is guaranteed by the adaptive behavior of the replicated block.

As another example, let NUM and DEN be the pictures of the numerator and denominator, respectively, of an algebraic expression. Then the following merged block Q will represent the picture of the negative of the quotient of these two expressions:

```
BAR = NODE(IT('-'))
Q = MERGE(NUM % BAR % DEN, '- ' BAR)
```

The above merger will ensure that the minus sign is placed directly to the left of the bar. Note that the iteration is carried out just once (the leftmost occurrence in the merger) and that the surrogate of the iterated block (for this occurrence) is a vertically concatenated block with the 3 daughters.

Note on surrogates:

(1) The grandsurrogate of a replicated block must be organized differently from the surrogate. Thus

```
A = B % REP(' ') % C
HOR_REG(A) = 'LEFT'
A = A % D
```

is not permitted.

SPECIAL BUILT-IN FUNCTIONS

EJECT() will eject a page and return the null string.

DUP(b,dir,n) where

b = any block
dir = an integer expression specifying a direction as
0 - vertical
1 - horizontal
2 - normal
n = an integer expression (non-negative),

will return a duplication of block "b", in the direction "dir", a number of times "n". For example,

```
DUP('|',0,60)
```

returns a vertical line of height 60. If n is 0, then BOX(0,0,0) is returned.

HEIGHT(b) will return an integer equal to the height of block "b".

WIDTH(b) will return an integer equal to the width of block "b".

DEPTH(b) will return an integer equal to the depth of block "b".

BLOCKSIZE(b,dir)

will return an integer equal to the size of a block "b" in the direction "dir".

Note: Determining the size of a block implies building it. Therefore, it is usually more efficient to FIX a block before determining its size.

SLAB(b,dir,offset,length)

where

b = any block expression
dir = a direction (as in DUP)
offset = an integer expression
length = an integer expression,

will return a physical block equal to a cross-sectional cut of block "b". The cross-section is taken orthogonally to the direction "dir". The length of the cross-section (the thickness) in this direction is "length" and its offset from the start of the block is given by "offset". Thus,

```
SLAB('STRING',1,2,3)
```

will return 'TRI'. If the offset and/or length are such that the resultant SLAB would extend beyond the boundaries of the original block, then only the intersection with the original block is returned. Thus,

```
SLAB('STRING',1,5,20)
```

returns 'NG'.

CHAR(b) where "b" is any block expression, returns a two-dimensional array of strings. Each element of the array corresponds to a horizontal sequence of characters in the block "b". The array is dimensioned D x H where D is the depth of the block and H is the height of the block. The (i,j)th element of the array is the string of characters in the ith plane, counting from the front, and in the jth plane, counting from the top. For example,

```
A = CHAR(FRONT(3,3) # 'ABC')
```

returns a 1 x 3 array. The element A<1,2> has the value 'ABC'. The elements A<1,1> and A<1,3> have the value ' '.

DATATYPE(e) will return the data type of its argument "e", where "e" is any expression. If "e" is an expression for a block (other than string), DATATYPE will return the string 'BLOCK'. Thus,

```
DATATYPE('A' 'B')
DATATYPE('A' % 'B')
```

return 'STRING' and 'BLOCK' respectively.

LOC(n,b,dir) where "n" is a node, "b" is a block, and "dir" is a direction (as in DUP), will return the location of node "n" and block "b" in the direction "dir", i.e., the number of characters that the node is indented from the top, left, or front. Thus,

```
N = NODE('AB')
I = LOC(N, '123' N , 1)
```

will set I equal to 3. Normally, the block should be FIXed prior to calling LOC in order to prevent building the block more than once. If the node "n" is not in the block "b", then the function fails.

An example, using several of these built-in functions, is described below.

Assume we are to print a block that is potentially larger than a printout page, which for the sake of this discussion is 66 x 132 print

positions. Assume that we would be willing to paste the separate printouts together when we were finished. Also assume that block heights of greater than 66 do not cause any special problem but block widths of greater than 132 do. The following function will print a large block.

```
                DEFINE('PRINTL(B)N,PW') :(PRINTL_END)
PRINTL
                B  =  FIX(B)
                PW  = 132
                N  =  1
PRINTL_1
                GT(N,WIDTH(B))          :S(RETURN)
                EJECT()
                PRINT(SLAB(B,1,N,PW))
                N  =  N  +  PW          :(PRINTL_1)
PRINTL_END
```

Broadcasting

Output can be broadcast to destinations other than the printer by an extended form of the PRINT function. The statement

PRINT(b,i^1,i^2,...)

where "b" is a block expression and i^1,i^2,... are integer expressions will, for each "i", do one of the following:

(1) if i > 0: print "b" on unit number "i"
(2) if i = 0 : print "b" on the printer (SPRINT)
(3) if i < 0 : do nothing

Similarly, EJECT has an extended form.

EJECT(i^1,i^2,...)

will broadcast page-eject characters to the indicated unit numbers. As in the PRINT function, a negative unit number is ignored; specification of the unit number 0 broadcasts page-eject characters to the printer.

Carriage Control

When a block is PRINTed, a carriage-control character is normally supplied to the printer (when unit 0 is specified) but not to other files. Thus, when

September 1975

```
PRINT('THIS' # '____', 0, 7)
```

is executed, the two strings ' THIS' and '+____' are normally sent to the printer, but the two strings 'THIS' and '____' are sent to unit number 7. It can be specified that the carriage-control character be supplied or withheld for any given unit number by a statement of the form

```
CC(unit) = bit
```

where "unit" is some unit number (possibly 0). If bit = 1, the carriage control is supplied; if bit = 0, the carriage control is withheld. If the unit number is negative, the statement will have no effect. For example,

```
CC(10) = 1
```

will supply carriage control for unit number 10. As another example,

```
CC(N) = 1 - CC(N)
```

will change the status of carriage control for unit number N.

EXAMPLES

The four examples depicted in Figures 23 through 31 illustrate the use of blocks in a variety of print-composition situations. Included are a table-of-contents generator, a function which returns the perspective view of a block, a bar-graph maker, and a function which transforms a FORTRAN expression into two-dimensional form. The program comments plus the figure captions are designed to fully explain the programs. Printouts depicting the result of running the indicated computations are also included.

```
        DEFINE('FLATTEN(B)D,N')
                                                    :(FLATTEN_END)
FLATTEN
        N  =  NODE()
        B  =  FIX(B)
        D  =  DEPTH(B)
*    If B has zero depth, return B.
        FLATTEN  =  EQ(D,0)  B                      :S(RETURN)
*  From  back to front successive layers of the block are merged with
*  preceding layers.
FLATTEN_1
        FLATTEN  =  MERGE(FLATTEN , N % SLAB(B,2,D,1))
        D  =  D - 1  GT(D,1)                        :S(FLATTEN_1)
        FLATTEN  =  FIX(FLATTEN)                    :(RETURN)
FLATTEN_END


        SP  =  REP(' ')
        P  =  HOR(40)
LOOP
        INPUT   LEN(39) . LINE  REM . N             :F(DONE)
        P  =  P % TRIM(LINE) REP(' ') TRIM(N)       :(LOOP)
DONE
        P  =  FLATTEN(P # IT(' .'))
        PRINT(P)
END
INTRODUCTION                            1
FUNCTIONS                               17
SYNTAX                                  48
SEMANTICS                               85
PRAGMATICS                              103
HISTRIONICS                             132
CONCLUSION                              149
```

Figure 23

The listing above constitutes a complete input deck to SNOBOL4B. It consists of a function (called FLATTEN) followed by the main program, followed by data. The result of running the program is the printout shown in Fig. 24. The function FLATTEN serves the purpose of reducing the depth of a block to 1. The character used in each position of the FLATTENed block is the first nonfill character as viewed from the front. FLATTEN utilizes the way in which MERGE overwrites characters, viz. that the merging is from left-to-right and that existing characters are overwritten by only nonfill characters.

Figure 24

The printout above resulted from the program in Fig. 23. The background for the table of contents is an iteration of the string ' .' and is placed behind the block. Different possibilities exist with respect to the choice of background (including overstruck characters), and several different alternatives can easily be tested. To 'turn off' the background completely (such as in a line skip) a nonprinting character other than blank should be used in the foreground.

```
*          Define special characters for upper left, lower left,
*  upper rt. and lower rt. corners and the horizontal line.
        UL = '┌' ; LL = '└' ; UR = '┐' ; LR = '┘' ; HL = '─'
*************************************************************************
*  PIPED will return what appears to be a perspective view of a         *
*  parallelepiped.  It will be labeled with a block given by the        *
*  first argument LA.  Its (apparent) height H, width W and depth D     *
*  are given by the 2nd through 4th arguments.                          *
*************************************************************************
        DEFINE('PIPED(LA,H,W,D)HH,WW,I,BL')
                                                    :(PIPED_END)
PIPED
        H = H - 1;   W = W - 1;   D = D - 1
*  Surround label with rectangle.  This will comprise the front plane
*  of the parallelepiped.
        HH  =  DUP('|',0,H)
        WW  =  DUP(HL,1,W)
        LA  =  FRONT(H,W) # LA
        BL  =  UL WW UR  %  HH LA HH  %  LL WW LR
*   Each time through the loop add on a layer of depth.
PIPED_2
        I  =  I + 1  LT(I,D)                        :F(PIPED_1)
        BL  =  HOR(I) '/' HOR(W) '/'   %   BL (VER(H) % '/' % VER(I))
        BL  =  FIX(BL)                              :(PIPED_2)
PIPED_1
*   Now add on the back plane.
        BL = HOR(D + 1) UL WW UR  %  BL (HH % LR % VER(D + 1))
        PIPED  =  FIX(BL)                           :(RETURN)
PIPED_END
```

Figure 25

The function shown above returns a block resembling a parallelepiped, of a labeling and size as indicated by its arguments. One use for such a parallelepiped is in drawing organization charts as shown in Fig. 26.

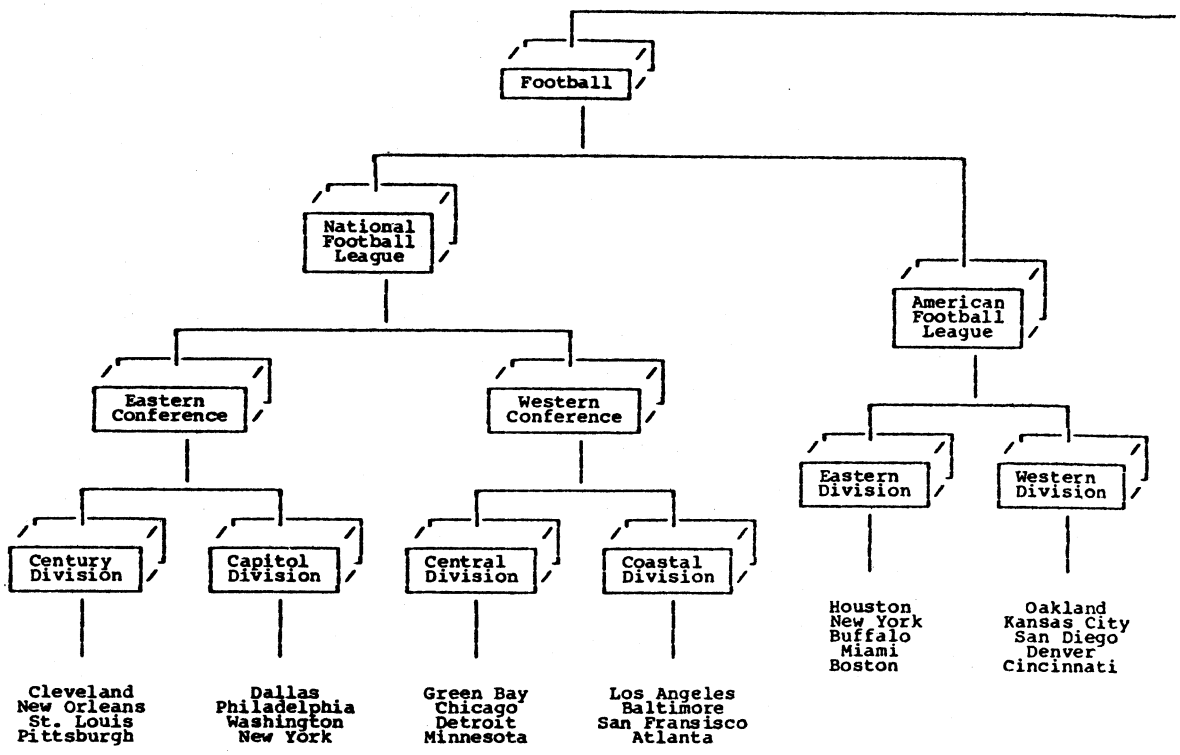


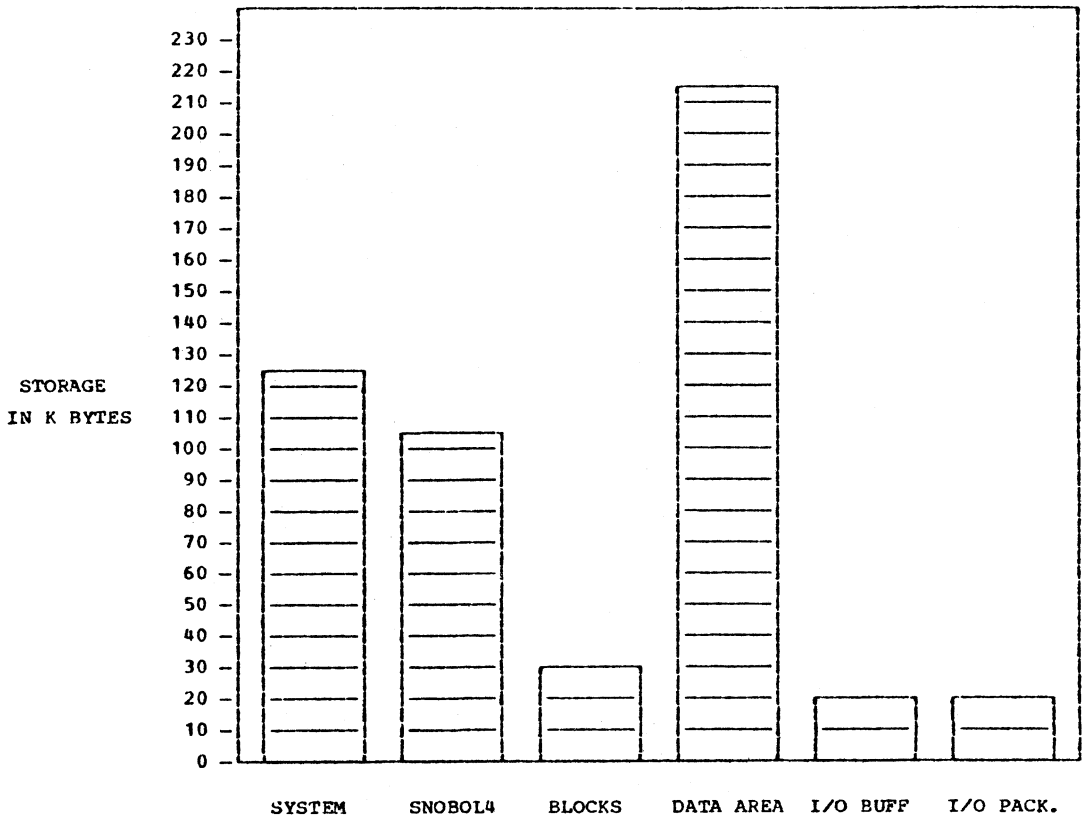
Figure 26

The data above illustrates the use of the subroutine PIPED, shown in Fig. 25. The chart above was printed on a line printer (at 8 lines per inch) using a special print train (TN train) with an extended character set. The program used to print organization charts was also written in SNOBOL4B.

```
*************************************************************************
*  BGRAPH  will return a bar graph.  A string of heights are provided   *
*  by the first argument.  A string of labels (for these heights) are   *
*  provided  by  the  second  argument.  The ordinate of the graph is   *
*  labeled with numbers every VLF spaces (vertical label  frequency).   *
*  The  graph  is  scaled  down by a factor of LPP (logical units per   *
*  physical unit).  Here the  logical  unit  refers  to  the  numbers   *
*  coming in whereas a physical unit is the height of a printer line.   *
*  The 5th and 6th arguments refer to horizontal spacing; BW  is  the   *
*  bar width and BS is the spacing between bars.                        *
*************************************************************************
        DEFINE('BGRAPH(HTS,LBS,VLF,LPP,BW,BS)HT,BAR,I,H,HT,LB,I.,BF')
                                                        :(BGRAPH_END)
BGRAPH
*   Set default values
        VLF  =  IDENT(VLF)  2
        LPP  =  IDENT(LPP)  1
        BW   =  IDENT(BW)   9
        BS   =  IDENT(BS)   2
*   BF is bar fill block
        BF   =  DUP(HL,1,BW - 2) % VER(VLF - 1)
        V    =  0
*     First character is break char.
        HTS   LEN(1) . HTS_B  =
        LBS    LEN(1) . LBS_B  =
BGRAPH_1
*   Get next height
        HTS    BREAK(HTS_B) . HT   LEN(1)  =        :F(BGRAPH_2)
*  compute height in terms of physical units (rounded)
        HT   =  (HT + (LPP / 2)) / LPP
*   get next label
        LBS    BREAK(LBS_B) . LB  LEN(1)  =
*   give label a height of at least 1
        LB   =  IDENT(LB)  VER(1)
*  make insides of bar
        I    =  DUP(BF,0,(HT / VLF) + 1)
*  use bottom HT - 1 rows of I
        I    =  SLAB(I,0,HEIGHT(I) - HT,HT - 1)
        BAR  =  OUTLINE(I)
*  but make bar null if HT is 0.
        BAR  =  LE(HT,0)
*  add label and attach to BGRAPH
        BAR  =  BAR % VER(2) % LB % HOR(BW + BS)
        BGRAPH  =  BGRAPH  BAR                       :(BGRAPH_1)
BGRAPH_2
        VER_REG(BGRAPH)  =  'BOTTOM'
*  surround  the  bars with a border. the border should be 3 units up
*  from the bottom of BGRAPH.
        BGRAPH  =  FIX(BGRAPH)
        H  =  HEIGHT(BGRAPH)
        BGRAPH  =  BGRAPH  #  OUTLINE(BOX(H,WIDTH(BGRAPH),1)) % VER(3)
        VER_REG(BGRAPH)  =  'BOTTOM'
*  now compute and form the ordinate.
        I  =  0
        BAR  =  0  ' '  HL
BGRAPH_3
        I  =  I + VLF
        GT(I,H)                                      :S(BGRAPH_4)
        BAR  =  (I * LPP)  ' ' HL  %  VER(VLF - 1)  %  BAR
                                                     :(BGRAPH_3)
BGRAPH_4
        HOR_REG(BAR)  =  'RIGHT'
        BGRAPH  =  (BAR % VER(3))  BGRAPH
        VER_REG(BGRAPH)  =  'BOTTOM'
                                                     :(RETURN)
BGRAPH_END
```

Figure 27

The function defined above, BGRAPH, produces a bargraph; one such is shown in Fig. 28. BGRAPH calls on the function OUTLINE previously defined in Fig. 17.



STORAGE ALLOCATION OF 512K CORE MEMORY WHEN RUNNING SNOBOL4B UNDER O.S. (PCP)

Figure 28

The bargraph shown above serves the dual purpose of describing storage allocation for SNOBOL4B (on an IBM 360/65) and furnishing an example of the use of the function BGRAPH defined in Fig. 27. The display shown above was produced by the following lines of program

```
        HTS  =  ',124,103,32,215,20,18,'
        LBLS  =  ',SYSTEM,SNOBOL4,BLOCKS,DATA AREA,I/O BUFF,I/O PACK.,'
        B  =  BGRAPH(HTS,LBLS,2,5)
        B  =  ('STORAGE' % ' ' % 'IN K BYTES') HOR(3) B
        B  =  B  %  VER(2)  %  'STORAGE ALLOCATION OF 512K CORE MEMORY'
+           ' WHEN RUNNING SNOBOL4B UNDER O.S. (PCP)'
        PRINT(B # FRONT(60,132))
```

The call to the function BGRAPH (defined in Fig. 27) requests a bargraph having heights and labels and vertical spacings as specified by arguments in the call. This graph is then combined with appropriate labels on the left and bottom.

```
*************************************************************************
*       POLISH converts an expression to polish notation in the         *
*  following way.  A+B  is converted to +,2,A,B,     A+B*-C   is        *
*  converted to      +,2,A,*,2,B,-,1,C,   and (A+B)  is converted to     *
*  (,1,+,2,A,B,.  Unary operators have higher precedence than binary     *
*  operators.    The precedence for binary operators is =,+-,*/,!        *
*  from lower to higher where +- have equal precedence as do */ and !    *
*  means exponentiation.  Also, the binary operators associate to the    *
*  left except exponentiation which associates to the right.             *
*************************************************************************
        DEFINE('POLISH(STR)E1,E2,TEMP,OPS,OP,T')
        UNARY  =  '+- '
        BINARY  =  '= +- */ ! '
        E1.OP.E2  =  POS(0)  BAL . E1  ANY(*OPS) . OP BAL . E2 RPOS(0)
        OP.E1  =  POS(0)  ANY(UNARY) . OP  BAL . E1  RPOS(0)
        PARENS  =  POS(0) '(' BAL . E1  ')'  RPOS(0)
*      REVERSE  is  an external function which will reverse a string.
*  It can be written in SNOBOL.
        LOAD('REVERSE(STRING)STRING')
*    Define  a  function  REV  which  will  reverse a string and also
*  invert parens
        DEFINE('REV(S)')                                :(REV_END)
REV     REV  =  REPLACE(REVERSE(S), '()', ')(')         :(RETURN)
REV_END
                                                        :(POLISH_END)
*    Entry point for the POLISH function
DEBRACKET_END
POLISH
        STR    ANY(UNARY BINARY  '()')                  :S(POLPAREN)
        POLISH  =  STR  ','                             :(RETURN)
POLPAREN
        STR  PARENS                                     :F(POLBIN)
        POLISH  =  '(,1,'  POLISH(E1)                   :(RETURN)
POLBIN  TEMP  =  BINARY
        STR  =  REV(STR)
POLOOP  TEMP  BREAK(' ')  . OPS  ' '  =
        IDENT(OPS,'!')                                  :S(POLEXP)
        STR  E1.OP.E2                                   :F(POLOOP)
        POLISH  =  OP  ',2,'  POLISH(REV(E2)) POLISH(REV(E1))
                                                        :(RETURN)
POLEXP  STR  =  REV(STR)
        STR    E1.OP.E2                                 :F(POLUNY)
        POLISH  =  OP  ',2,'  POLISH(E1)  POLISH(E2)
                                                        :(RETURN)
POLUNY  STR   OP.E1                                     :F(POLERROR)
        POLISH  =  OP  ',1,'  POLISH(E1)                :(RETURN)
POLERROR PRINT('UNDECIPHERABLE:' % STR)                 :(RETURN)
POLISH_END
*************************************************************************
*       DEBRACKET - remove unnecessary brackets.  For example, the      *
*  brackets in A/(B+C) are necessary when the equation is written in    *
*  linear form but are unnecessary and hence undesirable when the       *
*  equation is in 2 dimensional form.  It is also desirable to have a   *
*  way of forcing bracketing.  Double brackets such as A/((B+C)) will   *
*  always force a single bracket.                                       *
*************************************************************************
        DEFINE('DEBRACKET(POLISH)EXP,DIV,B,NL,T')
*  POLPAT  is  a  recursive  pattern  which  will  match  any  polish
*  expression.
        POLPAT  =  ANY(BINARY) ',2,' FENCE *POLPAT *POLPAT  |
+          ANY(UNARY '(')  ',1,' FENCE *POLPAT  |  BREAK(',') ','
                                                        :(DEBRACKET_END)
*     Entry  point  to  DEBRACKET.    Remove  any  single  brackets
*  surrounding 1) divisions, 2) exponentiations,  3)  numerators,  4)
*  denominators
DEBRACKET
        EXP  =  '!,2,'
        DIV  =  '/,2,'
        B  =  '(,1,'
        NL  =  ' ,1,'
DEB_DIV  POLISH   B  DIV        =   NL  DIV             :S(DEB_DIV)
DEB_EXP  POLISH   B  EXP        =   NL EXP              :S(DEB_EXP)
DEB_NUM  POLISH   DIV  B            =  DIV  NL          :S(DEB_NUM)
DEB_DEN  POLISH   (DIV POLPAT) . T  B  =  T  NL         :S(DEB_DEN)
*    Remove null operators and convert double brackets to single
DEB_NL   POLISH  NL  =                                  :S(DEB_NL)
DEB_BB   POLISH   B B  =  B                             :S(DEB_BB)
        DEBRACKET  =  POLISH                            :(RETURN)
```

Figure 29

The 2 functions defined above are used in forming 2-dimensional images of mathematical equations expressed in FORTRAN notation. POLISH converts the expression to polish prefix. DEBRACKET removes unnecessary brackets as described in the comments.

```
*************************************************************************
*  IMAGE  builds  a 2 diminsional image for an arithmetic expression.  *
*  The expression is given in polish notation in  the  global  string  *
*  POLISH  which  IMAGE  modifies.  IMAGE calls itself recursively to  *
*  obtain the image of each of the arguments to the  operator  it  is  *
*  currently  processing.  The image is a programmer-defined datatype  *
*  having 3 fields, PICT, LNUB and RNUB.  PICT is the  2  dimensional  *
*  representation  of  the  expression.    LNUB  (left nub) is a node  *
*  embedded in PICT indicating  the  left  connect  point,  and  RNUB  *
*  similarly indicates the right connect point.                        *
*************************************************************************
        DEFINE('IMAGE()OP,N,PICT,LNUB,RNUB,IMAGE1,IMAGE2,TEMP,LINE,'
+                                               'SIDE,H')
        DATA('CLUMP(PICT,LNUB,RNUB)')
        UL = '┌' ; LL = '└' ; UR = '┐' ; LR = '┘' ; HL = '─'
        OP.N = POS(0) ANY(UNARY BINARY '(') . OP ',' BREAK(',') . N ','
                                                        :(IMAGE_END)
IMAGE
*  first  we determine and remove the operator (OP) and the number of
*  arguments (N) from the polish string.
        POLISH    OP.N   =                              :F(IMAGE_SIMPLE)
*    Get the image of the arguments
        IMAGE1  =  IMAGE()
        IMAGE2  =  EQ(N,2)  IMAGE()
*     Now branch to the location appropriate to the operator
                                                        :($('IMAGE' OP N))
IMAGE=2  ;IMAGE+2  ;IMAGE-2  ;IMAGE*2
        OP  =  ' ' OP ' '
        OP  =  IDENT(OP,' * ')      ' '
        PICT = MERGE(PICT(IMAGE1),RNUB(IMAGE1) OP LNUB(IMAGE2),
+                                                       PICT(IMAGE2))
        LNUB  =  LNUB(IMAGE1)
        RNUB  =  RNUB(IMAGE2)                           :(IMAGE_RET)
IMAGE+1  ;IMAGE-1
        LNUB  =  NODE(OP)
        PICT  =  MERGE(LNUB ' ' LNUB(IMAGE1), PICT(IMAGE1))
        RNUB  =  RNUB(IMAGE1)                           :(IMAGE_RET)
IMAGE/2  TEMP  =  NODE(IT(HL))
        LNUB  =  NODE(HL)
        RNUB  =  NODE(HL)
        PICT = MERGE(PICT(IMAGE1) % TEMP % PICT(IMAGE2),LNUB TEMP RNUB)
                                                        :(IMAGE_RET)
IMAGE!2  PICT = FIX(PICT(IMAGE2),1)
        LINE  =  HOR(WIDTH(PICT))
        PICT = PICT % PICT(IMAGE1) LINE
        HOR_REG(PICT) = 'RIGHT'
        RNUB  =  NODE()
        LNUB  =  LNUB(IMAGE1)
        PICT = MERGE(PICT,RNUB(IMAGE1) LINE RNUB)       :(IMAGE_RET)
IMAGE(1
        PICT = FIX(PICT(IMAGE1),1)
        H  =  HEIGHT(PICT)
        PICT  =  EQ(H,1)    '(' PICT ')'                :S(IMAGE_SIM)
        SIDE  =  DUP('|',0,H)
        PICT  =  (UL % SIDE % LL)  ' ' PICT ' ' (UR % SIDE % LR)
                                                        :(IMAGE_SIM)
IMAGE_SIMPLE
        POLISH   BREAK(',') . PICT  ','  =
IMAGE_SIM
        LNUB  = NODE() ; RNUB  =  NODE()
        PICT  =  LNUB  PICT  RNUB                       :(IMAGE_RET)
*     Common return point
IMAGE_RET
        IMAGE  =  CLUMP(PICT,LNUB,RNUB)                 :(RETURN)
IMAGE_END

        DEFINE('EQU(S)')                                :(EQU_END)
*  Convert exponentiation symbol and remove blanks.
EQU     S  '**'  =  '!'                                 :S(EQU)
EQU_1   S  ' '  =                                       :S(EQU_1)
        POLISH  =  DEBRACKET(POLISH(S))
        EQU  =  PICT(IMAGE())                           :(RETURN)
EQU_END
```

Figure 30

There are 2 functions defined above, EQU and IMAGE. EQU accepts an equation written in FORTRAN notation (without function calls) and returns a block representing its two-dimensional form. EQU calls DEBRACKET and POLISH defined in Fig. 29. This converts the expression into polish prefix which can then be operated upon by IMAGE. IMAGE is a recursive procedure reflecting the recursive nature of the picture it is building. One crucial aspect of the function IMAGE is that the value returned is not simply a block but a block plus 2 nodes indicating the left and right connect points of the picture. Examples of converted expressions are shown in Fig. 31.

```
1/(1+T) = 1-T+T**2-T**3+...+(-1)**N*T**N+(-1)**(N+1)*T**(N+1)/(1+T)
```

$$\frac{1}{1+T} = 1 - T + T^2 - T^3 + \ldots + (-1)^N T^N + \frac{(-1)^{(N+1)} T^{(N+1)}}{1+T}$$

```
D*(F/G)/DZ = (G*(DF/DZ) - F*(DG/DZ)) / G**2
```

$$D\frac{\frac{F}{G}}{DZ} = \frac{G\frac{DF}{DZ} - F\frac{DG}{DZ}}{G^2}$$

Figure 31

The two equations in linear form shown above were converted to two-dimensional form by calling EQU defined in Fig. 30.

September 1975

APPENDIX: PUBLIC FILE DESCRIPTIONS

The following public file descriptions have been extracted from MTS Volume 2, Public File Descriptions. Each of these descriptions pertains to the use of SNOBOL4 in MTS.

MTS 9: SNOBOL4 in MTS

September 1975

September 1975 Page Revised June 1979

*CONVSNOBOL

Contents: A conversational SNOBOL4 program that reads and executes SNOBOL4 statements.

Use: The file contains commands to run the program. It should be invoked by the $SOURCE command, i.e.,

```
$SOURCE *CONVSNOBOL
```

Program Key: *EXEC

Logical I/O Units Referenced:

(Specified on a $RUN command in the file)

5 - Input for SNOBOL4 initial compilation.
6 - Output for SNOBOL4 error comments.
8 - Input of conversational SNOBOL4 statements.
9 - Output of conversational SNOBOL4 statements.

Description: The SNOBOL4 program in this file reads SNOBOL4 statements from *MSOURCE*. These statements are to be either executed immediately or stored away as part of a program. If a statement which is entered begins with a number, it is compiled and the code stored in the array PROG, using the number as a subscript to the array. Otherwise, the statement is executed immediately. Any compilation errors detected while compiling the statements entered cause the comment "COMPILATION ERROR" to be printed. The error message itself is not printed unless execution of *CONVSNOBOL is terminated (by entering an end-of-file).

To begin execution of a stored program, an immediate-execution unconditional transfer to one of its statements should be entered.

Fatal errors detected during the execution of an immediate-execution statement or during the execution of a stored program cause execution of *CONVSNOBOL to be terminated with the appropriate error comment and a string dump to be produced (if one was requested).

Example Session: (User input is in lower-case)

```
$source *convsnobol
$RUN *SNOBOL4 5=*SOURCE* 6=*MSINK* 8=*MSOURCE*@UC 9=*MSINK*
EXECUTION BEGINS
  .
  .
  .
ENTER SNOBOL STATEMENTS
```

```
  output = "hello"
HELLO
$endfile
NORMAL TERMINATION AT LEVEL 0
          .
          .
          .
```

*SNOBOL4

Contents: The object module of version 3 of the SNOBOL4 interpreter (without BLOCKS).

Purpose: To interpret SNOBOL4 source programs.

Use: The program is invoked by the $RUN command.

Program Key: *SNOBOL4

Logical I/O Units Referenced:

- 5 - source for the SNOBOL4 program to be translated, immediately followed by data read via the default "INPUT" string.
- 6 - Output from compilation of the SNOBOL4 program and output via the default "OUTPUT" string.
- 7 - Output via the default "PUNCH" string.

Parameters: The following parameters may be specified in the PAR field of the $RUN command. The parameters must be separated by a comma.

- DUMP - causes a SNOBOL4 string dump only if a fatal error occurs during execution.
- XREF - causes a cross-reference dictionary of the source program to be printed after compilation.
- NOEX - prevents execution of a SNOBOL4 program if any compilation errors are present.
- SIZE=xxxx - specifies the amount of storage to be used for the compiled SNOBOL4 program and its variables (the SNOBOL4 interpreter itself uses about 40 pages). "xxxx" may be any number specifying the number of pages to be reserved or the word MIN may be used to specify that 10 pages be reserved. If no SIZE parameter is present, the default value is 30 pages.

Description: The language accepted by *SNOBOL4 is described in the publication The SNOBOL4 Programming Language, by R. Griswold, J. Poage, and I. Polonsky (Prentice-Hall, 1971). See the descriptions of *SNOBOL4B and *SPITBOL in this volume for alternative language processors for the SNOBOL4 language.

Examples: $RUN *SNOBOL4 5=*SOURCE* 6=*SINK*

In the above example, the source program and data are read from *SOURCE* and the output is written to *SINK*.

```
$RUN *SNOBOL4 5=SNOPROG+*SOURCE* 6=*DUMMY*(1,20)+*SINK*
     7=*PUNCH*
```

In the above example, the source program is read from the file SNOPROG and the data is read from *SOURCE*. The first twenty lines of output are discarded with the remainder being written to *SINK*. Punch output is written to *PUNCH*.

*SNOBOL4B

Contents: The object module of version 3 of the SNOBOL4 interpreter (with BLOCKS).

Purpose: To interpret SNOBOL4 source programs.

Use: The program is invoked by the $RUN command.

Program Key: *SNOBOL4B

Logical I/O Units Referenced:

- 5 - source for the SNOBOL4 program to be translated, immediately followed by data to be read via the default INPUT string.
- 6 - output from compilation of the SNOBOL4 program and output via the default OUTPUT string.
- 7 - output via the default PUNCH string.

Parameters: The following parameters may be specified in the PAR field of the $RUN command. The parameters must be separated by a comma.

- DUMP - causes a SNOBOL4 string dump only if a fatal error occurs during program execution.
- XREF - causes a cross-reference dictionary of the source program to be printed after compilation.
- NOEX - prevents execution of a SNOBOL4 program if any compilation errors are present.
- SIZE=xxxx - specifies the amount of storage to be used for the compiled SNOBOL4 program and its variables (the SNOBOL4 interpreter itself uses about 45 pages). "xxxx" may be any number specifying the number of pages to be reserved or the word MIN may be used to specify that 10 pages be reserved. If no SIZE= parameter is present, the default value is 30 pages.

Description: For a description of the BLOCKS feature in SNOBOL4, see the section "SNOBOL4 Blocks" in this volume.

Examples:

```
$RUN *SNOBOL4B 5=*SOURCE* 6=*SINK*
```

In the above example, the source program and data are read from *SOURCE* and the output is written to *SINK*.

```
$RUN *SNOBOL4B 5=SNOPROG+*SOURCE* 6=*DUMMY*(1,20)+*SINK*
     7=*PUNCH*
```

In the above example, the source program is read from the file SNOPROG and the data is read from *SOURCE*. The first twenty lines of output are discarded with the remainder being written to *SINK*. Punch output is written to *PUNCH*.

*SPITBOL

Contents: The object module of version 2 of the SNOBOL4 compiler developed at the Illinois Institute of Technology.

Purpose: To compile and execute SNOBOL4 source programs.

Use: The compiler is invoked by the $RUN command.

Program Key: *SPITBOL

Logical I/O Units Referenced:

SCARDS - SNOBOL4 source program to be compiled followed by the data read via the default variable INPUT.
SPRINT - source listing, object listing, parameter listing, compiler diagnostics, execution-time diagnostics, and output via the default variable OUTPUT.
SPUNCH - object module if the DECK option was specified, and output from the default variable PUNCH.
0 - object module if NODECK and LOAD parameters are specified.
SERCOM - prompting if a terminal.
GUSER - user responses to prompting if a terminal.

Parameters: The following parameters may be specified in the PAR field of the $RUN command. The parameters must be separated by a comma or by one or more blanks. A semicolon may be used to terminate the PAR field; the text after the semicolon is not processed by the SPITBOL compiler but is available to the program via the SYSPAR external function. The parameters may be either keywords or free verbs. Free verbs may be negated by one of three prefixes: "NO", "¬", or "-". The underlined portion of the parameter may be used as a minimum abbreviation. The legal parameters are:

A[LIST] or [NO]ALIST

If ALIST is specified, an object-code listing will be produced on SPRINT. The default is NOALIST. This parameter replaces the OLIST parameter.

B[ATCH] or [NOB]ATCH

If BATCH is specified, the compiler will batch process input decks. The batch pseudo-end-of-file is "./*" in columns 1-3. The default is NOBATCH.

CSTAT or NOCSTAT

If CSTAT is specified, compilation statistics are printed on SPRINT. The default is NOCSTAT.

DECK or NODECK

If DECK is specified, an object module will be produced on SPUNCH. SPITBOL object modules must be run in concatenation with *SPITLIB. The default is NODECK.

DUMP=nnn

At termination of execution, the SPITBOL dump function is called with "nnn" as the argument. The default is DUMP=0 which produces no dump.

EDUMP=nnn

If execution terminates abnormally, the SPITBOL dump function will be called with "nnn" as the argument. If SPRINT is assigned to a terminal, the default is EDUMP=0 which produces no dump; if SPRINT is not assigned to a terminal, the default is EDUMP=1 which generates a dump of natural variables and keywords.

ERRXEQ or NOERRXEQ

If ERRXEQ is specified, the compiled program is executed even if errors were detected during compilation. The default is ERRXEQ.

ESTAT or NOESTAT

If ESTAT is specified, execution statistics are produced on SPRINT. The default is NOESTAT.

EXECUTE or NOEXECUTE

If EXECUTE is specified, the compiled program is executed. The default is EXECUTE.

FAILCHK or NOFAILCHK

If FAILCHK is specified, execution is terminated with an error if a statement is executed that fails and there is no conditional "goto" field. This is a useful debugging tool. The default is NOFAILCHK.

INMGN=nnn

"nnn" is the right-hand margin for the compiler when scanning input source programs. The default is set to the minimum of the input record length for SCARDS and 255. Programs that have sequential IDs in columns 73-80 should compile with INMGN=72.

LINECNT=nnn

"nnn" is the number of lines per page for printed output. This must be in the range of $3 \leq nnn \leq 32767$. The default is 58.

LIST or NOLIST

If LIST is specified, a source listing of the compiled program is printed on SPRINT. The default is LIST unless SPRINT is assigned to a terminal. This parameter replaces the SLIST parameter.

LOAD or NOLOAD

If both LOAD and NODECK are specified, an object module will be produced on logical I/O unit 0. The default is NOLOAD.

OPTIMIZE or NOOPTIMIZE

If OPTIMIZE is specified, the object code produced will be optimized. This can reduce the CPU time and storage required for program execution. The default is OPTIMIZE.

PLIST or NOPLIST

If PLIST is specified, a list of parameter values will be printed on SPRINT. The default is NOPLIST.

SDUMP or NOSDUMP

If SDUMP is specified, a storage dump of the SPITBOL work areas will be produced on SPRINT if an internal SPITBOL error is detected. This is useful only for system debugging. The default is NOSDUMP.

SEQCHK or NOSEQCHK

If SEQCHK is specified and INMGN=72, the sequential ID field (columns 73-80) is checked for ascending order. Out-of-order cards will be flagged. The default is NOSEQCHK.

SIZE=nnn

"nnn" is the number of pages allocated for dynamic storage within the compiler. This must be in the range of 4 ≤ nnn ≤ 256. The default is 20.

TIMECHK or NOTIMECHK

If TIMECHK is specified, SPITBOL will terminate execution of the user program 0.25 seconds before any specified time limit in order that a symbolic dump may be given. The default is TIMECHK.

XREF or NOXREF

A cross-reference listing of the symbols in the compiled program is printed on SPRINT. The default is NOXREF.

Description: *SPITBOL is a SNOBOL4 compiler which is considerably faster than *SNOBOL4, and can generate object modules if desired. However, the object modules must be run with *SPITLIB. For further details, see the section "SPITBOL" in this volume.

Examples: $RUN *SPITBOL SCARDS=PROG.S PAR=SIZE=50,BATCH

In the above example, each of the programs in the file PROG.S is compiled and executed with a dynamic storage size of 50 pages.

$RUN *SPITBOL SCARDS=PROG.S SPRINT=*PRINT* SPUNCH=PROG.O
PAR=DECK,NOEX

In the above example, the program in PROG.S is compiled without being executed. The object module is written into the file PROG.O; the source listing is produced on *PRINT*.

*SPITDEBUG

Purpose: To assist in the interactive debugging of SPITBOL programs.

Use: The following statement should be included in the SPITBOL source program at a place where it will be executed at the beginning of the program starting at column one:

```
-COPY *SPITDEBUG
```

If a copy of the SPITBOL-generated source listing is placed in the file -PRINT*, *SPITDEBUG will print the statement causing the error.

Program Key: *EXEC

Description: *SPITDEBUG contains a number of SPITBOL source statements that are invoked whenever an execution error occurs or the DEBUG function is called.

Errors are trapped using the SETEXIT function and setting &ERRLIMIT to 1. When SPITDEBUG gains control after an error, the error number and associated error comment are printed. If the statement can be found in the file -PRINT*, it is also printed. After these messages are printed, the user is placed in SPITDEBUG command mode where he may enter SPITDEBUG commands or SPITBOL statements.

If the DEBUG function is called, it may have a string as an argument which is printed at the time of the call. These calls may be placed in the source program in order to set a breakpoint and examine the status of variables using SPITDEBUG. Processing may be continued with the SPITDEBUG CONTINUE command.

The SPITDEBUG commands are given below. The underlined portion of each command is the minimum acceptable abbreviation that may be used.

BREAK label — Sets an execution breakpoint at "label". This breakpoint will be effective only on a goto to that label; it is not effective if execution of the previous statement falls into that label. Labels may be generated automatically on all statements via the SNOSTORM DEBUG option. The breakpoint feature is implemented with the SPITBOL TRACE function and the keyword &TRACE is incremented by 100 each time a BREAK command is issued.

	This inadvertently may enable other unrelated tracing in the program. When used with programs compiled with the SNOSTORM DEBUG option, it is possible to use the source-file line number in place of the label.
CONTINUE	Resumes program execution.
DISPLAY express	Displays the program value "express", e.g., DISPLAY x<2> x<2> = "HELLO"
DUMP {1|2}	Produces a SNOBOL dump (DUMP(1) or DUMP(2)).
EXPLAIN	Synonym for HELP.
GOTO label	Resumes program at the location "label"; this is the same as ":(label)" in the program. GOTO START
HELP	Provides a list of the legal SPITDEBUG commands with a brief description of each.
IGNORE ["msg"] n	Causes the next "n" calls on the DEBUG function with parameter "msg" to be ignored. If "msg" is omitted, DEBUG calls with a null string parameter will be ignored.
MTS [command]	Enters MTS command and optionally executes an MTS command, e.g., MTS $EDIT MYFILE
REMOVE label	Removes the breakpoint at "label".
RESTORE label	Synonym for REMOVE.
STOP	Terminates execution.
$command	Executes an MTS command, e.g., $DISPLAY COST

September 1975 Page Revised June 1979

*SPITERR

Contents: The SPITBOL error messages.

Program Key: *EXEC

Description: The file *SPITERR contains all of the SPITBOL error messages. Each message is located at an MTS line number that is the same as the SPITBOL error number for that message. The message may be obtained via the $COPY command.

Example: $COPY *SPITERR(11.009,11.009)

The above example copies SPITBOL error message 11.009 to *SINK*.

Page Revised June 1979 September 1975

*SPITLIB

Contents: The object module of the version 2 execution-time support routines for programs compiled by *SPITBOL.

Use: *SPITLIB should be concatenated with the object file to be executed on the $RUN command, i.e.,

```
$RUN object+*SPITLIB
```

Program Key: *EXEC

Logical I/O Units Referenced:

SCARDS - data to be read via the default variable INPUT.
SPRINT - execution-time diagnostics, and output via the default variable OUTPUT.
SPUNCH - output via the default variable PUNCH.
SERCOM - prompting if a terminal.
GUSER - user responses to prompting if a terminal.

Parameters: The following parameters may be specified in the PAR field of the $RUN command. The parameters must be separated by a comma or by one or more blanks. A semicolon may be used to terminate the PAR field; the text after the semicolon is not processed by the SPITBOL compiler but is available to the program via the SYSPAR external function. The parameters may be keywords or free verbs. Free verbs may be negated by one of three prefixes: "NO", "¬", or "-". The underlined portion of each parameter may be used as a minimum abbreviation. The legal parameters are:

DUMP=nnn

At termination of execution, the SPITBOL dump function is called with "nnn" as the argument. The default is DUMP=0 which produces no dump.

EDUMP=nnn

If execution terminates abnormally, the SPITBOL dump function will be called with "nnn" as the argument. If SPRINT is assigned to a terminal, the default is EDUMP=0 which produces no dump; if SPRINT is not assigned to a terminal, the default is EDUMP=1 which generates a dump of natural variables and keywords.

ESTAT or NOESTAT

If ESTAT is specified, execution statistics are generated on SPRINT. The default is NOESTAT.

LINECNT=nnn

"nnn" is the number of lines per page for printed output. This must be in the range of 3 ≤ nnn ≤ 32767. The default is 58.

PLIST or NOPLIST

If PLIST is specified, a list of parameter values is printed on SPRINT. The default is PLIST.

SDUMP or NOSDUMP

If SDUMP is specified, a storage dump of the SPITBOL work areas will be produced on SPRINT if an internal SPITBOL error is detected. This is useful only for system debugging. The default is NOSDUMP.

SIZE=nnn

"nnn" is the number of pages of dynamic storage allocated. This must be in the range of 4 ≤ nnn ≤ 256. The default is 20.

TIMECHK or NOTIMECHK

If TIMECHK is specified, SPITBOL will terminate execution of the user program 0.25 seconds before any specified time limit in order that a symbolic dump may be given. The default is TIMECHK.

Description: Object modules generated by *SPITBOL must be run with the execution-time support routines in *SPITLIB. For further details, see the section "SPITBOL" in this volume.

Examples: $RUN PROG.OBJ+*SPITLIB SCARDS=DATA1

In the above example, the program in the file PROG.OBJ is executed with the data being read from the file DATA1.

$RUN -OBJ+*SPITLIB 5=INFILE 6=OUTFILE PAR=SIZE=50

In the above example, the program in the file -OBJ is executed with input from the file INFILE and output being written to the file OUTFILE.

September 1975 Page Revised June 1979

*TRANSNOBOL

Contents: A SNOBOL3 to SNOBOL4 translator.

Purpose: To convert SNOBOL3 source programs into SNOBOL4 source programs.

Use: The file *SNOBOL4 should be invoked by the $RUN command with the file *TRANSNOBOL assigned to logical I/O unit 5.

Program Key: *EXEC

Logical I/O Units Referenced:

2 - translated SNOBOL4 program listing.
3 - translated SNOBOL4 source program.
4 - SNOBOL3 source program input.
5 - input of translator (*TRANSNOBOL).
6 - listing of translator (normally set to *DUMMY*).

Description: The SNOBOL4 program in *TRANSNOBOL accepts as data a SNOBOL3 program of any length and produces as output a SNOBOL4 program (version 2, release 1). Any number of programs can be run in succession with an end-of-file following the last program, and each will be translated separately.

Because SNOBOL3 uses tapes for all I/O operations and SNOBOL4, working in MTS, uses logical I/O units, slight changes will occur regarding the use of scratch tapes. A SNOBOL3 scratch tape can be simulated in SNOBOL4 using a temporary file attached to an I/O unit corresponding to the tape number. For example, if the assignment "3=-SCRATCH" is specified on the control card, the file -SCRATCH will be treated as scratch tape 3, and no changes need be made in the SNOBOL4 program regarding this tape. However, some SNOBOL3 functions operating on this tape may not work in SNOBOL4 (see below). Logical I/O units 1-9 are available in MTS.

(1) The translated program will use the following I/O units:

4 punched output
5 program input
6 printed output
7 data input

The remaining I/O units 1-3 and 8-9 can safely be used for scratch files. The following example illustrates a control card for a SNOBOL4 program:

```
$RUN *SNOBOL4 4=*PUNCH* 5=-OUTPUT 6=*SINK* 7=*SOURCE*
```

(2) The following functions have no translation:

BSR, BSF, COMPRS, DETACH, DUMP, EOF, MODE(INTEGER), MODE(TRUNCATION), SQUEEZ, TAPRD, TAPWR, and UNLOAD

(3) Control cards will not be interpreted but will be considered as SNOBOL3 statements.

(4) The following string names will be flagged during translation since they have special meanings in SNOBOL4:

ARB, BAL, INPUT, OUTPUT, PUNCH, REM, and FAIL

(5) The string name SYSLOK will be undefined.

(6) Variables used in back-referencing will not retain their original values if the pattern-match fails. They will instead have the value of the last unsuccessful attempt at matching.

Since *TRANSNOBOL will accept only decks punched on an 029 keypunch (EBCD code), decks punched on an 026 keypunch (BCD code) must be converted before translation. This can be done on the IBM 360 Model 20 computer using a "#26REPRODUCE" control card or it may be done immediately before translation with the following sequence:

```
$RUN *BCDEBCD SPUNCH=-DATA
          .
          .
          .
   deck (punched on 026 punch)
          .
          .
          .
$ENDFILE
$RUN *SNOBOL4 2=*SINK* 3=-OUTPUT 4=-DATA 5=*TRANSNOBOL
     6=*DUMMY*
```

Example:

```
$RUN *SNOBOL4 2=*SINK* 3=OUTPUT 4=*SOURCE* 5=*TRANSNOBOL
     6=*DUMMY*
```

In the above example, the SNOBOL3 program from the source stream (*SOURCE*) is translated to SNOBOL4 and written to the file OUTPUT.

INDEX

Reader's Comment Form

SNOBOL4 in MTS
Volume 9
September 1975
(May 1984 Reprint)

Errors noted in publication:

Suggestions for improvement:

Your comments will be much appreciated. The completed form may be sent to the Computing Center by Campus Mail or U.S. Mail, or dropped in the Suggestion Box at the Computing Center, NUBS, or UNYN.

Date _____

Name _____

Address _____

_____

_____

Publications
Computing Center
University of Michigan
Ann Arbor, Michigan 48109

Update Request Form

SNOBOL4 in MTS
Volume 9
September 1975
(May 1984 Reprint)

Updates to this manual will be issued periodically as errors are noted or as changes are made to MTS. If you desire to have these updates mailed to you, please submit this form.

Updates are also available in the memo files at the Computing Center, NUBS, and UNYN; there you may obtain any updates to this volume that may have been issued before the Computing Center receives your form. Please indicate below if you desire to have the Computing Center mail to you any previously issued updates.

Name _____

Address _____

_____

_____

Previous updates needed (if applicable):_____

The completed form may be sent to the Computing Center by Campus Mail or U.S. Mail, or dropped in the Suggestion Box at the Computing Center, NUBS, or UNYN. Campus Mail addresses should be given for local users.

Publications
Computing Center
The University of Michigan
Ann Arbor, Michigan 48109

Users associated with other MTS installations (except the University of British Columbia) should return this form to their respective installations. Addresses are given on the reverse side.

Addresses of other MTS installations:

Publications Clerk
352 General Services Bldg.
Computing Services
The University of Alberta
Edmonton, Alberta
Canada T6G 2H1

Information Officer, NUMAC
Computing Laboratory
The University of Newcastle upon Tyne
Newcastle upon Tyne
England NE1 7RU

Rensselaer Polytechnic Institute
Documentation Librarian
310 Voorhees Computing Center
Troy, New York 12181

Simon Fraser University
Computing Centre
User Services Information Group
Burnaby, British Columbia
Canada V5A 1S6

Wayne State University
Computing Services Center
Academic Services Documentation Librarian
5950 Cass Ave.
Detroit, Michigan 48202