P. McJones

INTERACTIVE SNOBOL4 SYSTEM FOR THE SDS 940

System Implemented By

Eric R. Anderson and Roger Sturgeon

University of California, Berkeley

Document No. R-34

Issued September 6, 1968

Contract No. SD-185

Office of Secretary of Defense

Advanced Research Projects Agency

Washington, D. C. 20325

TABLE OF CONTENTS

Introduction. 1
SNOBOL4 Program 2
Strings. 3
Names and Variables. 4
String Assignment 5
Concatenation 6
Simple Pattern Matching 7
Labels. 9
The Go-To Field 10
Simple Pattern Matching Continued 11
Fields of a Statement 12
Teletype Input and Output 13
Binary and Unary Operators. 14
Arithmetic. 15
Indirect Referencing. 16
Grouping. 17
Functions 18
User Functions. 19
Distinction Between Names 22
Order of Evaluation 23
Patterns. 24
Alternation ("OR"). 24
Concatenation 25
Arbitrary Strings 25
Balanced Strings. 25
Fixed Length Strings. 26
Fixed Positions In Strings. 26
Tabulation. 26
Remainder. 26
Alternative 27
Runs of Characters. 27
Repetitions 28
Signaling Failure 28

The Order of Pattern Matching. 29
Deferred Pattern Definition. 31
Value Assignment. 32
Immediate Value Assignment 33
Infinite Loops. 34
Additional Built-in Functions. 35
Additional Input-Output Facilities 36
The Editor, Compiler, and Runtime. 38
Special Operations. 40
Keywords. 41
Pseudo-Teletype Functions 43
LOGIN( ) or LOGIN(NAME,PASSWORD) 43
LOGOUT( ). 44
WAIT( ). 44
SEND(S). 44
ATSEND(S). 44
RECV(N). 45
RECVLINE( ). 45
ECHO(N). 46
Sample Pseudo-Teletype Programs. 47
Appendix A 48
Appendix B 52
References 53

Introduction

The SDS 940 SNOBOL4 system will accept programs written in a language which is basically compatible with a subset of Bell Labs' November 22, 1967 version of SNOBOL4. SNOBOL4 is not a superset of SNOBOL3 but is in most ways very similar to SNOBOL3. The major exception is in pattern matching and the pattern datatype. The SNOBOL4 system permits programs to be created, run and debugged interactively.

The principal data object in the SNOBOL language is a string of characters. The language permits building up longer strings from shorter strings through concatenations. In addition, through pattern matching, strings can have their contents tested and have the matched substrings assigned to string variables.

Other features of the language are arithmetic on integer strings, built-in functions for general use, and programmer defined functions which may have local variables and can be recursive to arbitrary depth. Input-output from files is provided as well as from the teletype.

SNOBOL4 Program

A SNOBOL4 program is a set of statements, each involving a rule. A set of rules provides the means for manipulating strings and other data objects. Each statement of a program is written only with printable characters, but the contents of the data strings can be any 8-bit characters. The & character is reserved as an escape character for entering non-printable characters literally into the source program. To enter an & in a source program use && (see section on special operations).
(Non-printing chars in source are ignored by the compiler.)

Characters in the language

blank ! " # $ % & ' ( ) * + , - . / Ø 1 2 3 4 5 6 7 8 9 : ; < = > ?

@ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [\ ] ↑ ←

Strings

A string is a sequence of 8 bit characters ordered from left to right (see special operations for entering non-printable characters). A string may be represented literally in the language by surrounding its contents by a pair of single quotes or double quotes. When one kind of quote is used, only the other kind may appear within the literal string. A fundamental property of a string is its length. In particular, the string of length 0 exists and is called the null string. It can appear literally as '' or "".

The string which contains the digits in order from ∅ to 9 can be literally written as

'∅123456789' or "∅123456789"

These are legal	"'"	These are illegal	'''
	'"'		""""
	"'/FILENAME'"		'CAN'T'

This is a string of length			
	∅	''	the null string
	1	'X'	the string containing X
	2	'PQ'	the string containing PQ
	3	'*1:'	the string containing *1:

These strings have different contents: 'AB', 'BA'

One contains AB, the other contains BA.

Names and Variables

Names in the SNOBOL4 language may be of any length (up to 4095 characters). The first character must be a letter or @. Each of the remaining characters must be a ., letter, or digit. The @ is intended for keyword names (see section on keywords). Variables in the language are those things which are given a name and have strings, patterns, or some other data object as their contents.

These are legal names	These are illegal names
X	%
STRING	1ABC
@ANCHOR	AT@
A.LONG.NAME	.NAME

String Assignment

In a string processing language it is necessary to store strings, to build up longer strings, to test strings for their contents, and to take strings apart. The storing of a string is specified by an assignment rule of one of the following forms:

```
STRINGNAME '=' LITERAL STRING
STRINGNAME '=' STRINGNAME
STRINGNAME '='
```

Blanks around the = are not necessary, but all other binary operators in the SNOBOL4 language require blanks on both sides. The third example is semantically equivalent to the first with a null string, '', on the right-hand side. Names which have not been assigned a value contain the null string. The above rules say to take the contents of the strings on the right-hand side and store them in the string variable whose name is given on the left hand side.

Examples:

```
STRING = 'THING'
ALPHABET = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
LETTERS = ALPHABET
NULL =
```

Concatenation

Building up longer strings can be specified by concatenation (or juxtaposition). Any number of strings may be concatenated to produce one long string. The operation is denoted by a space between each of the parts to be concatenated (sometimes parentheses are required to denote the range of the concatenation). Thus, to store the results of a concatenation into a string variable, simply use an assignment rule with the concatenation appearing on the right-hand side.

Assume the following are executed in order:

```
A = 'ALPHA'          A would contain the characters ALPHA
B = 'BETA'           B would contain the characters BETA
D = A B C            D would contain the characters ALPHABETA
                          since C is assumed to be the null string.
A = B                A would contain the characters BETA
B =                  B would contain the null string.
```

Simple Pattern Matching

It is often desirable to know if one string is contained somewhere within another string. A test of this type is denoted by a rule of the form: STRING ' ' STRING. That is, the string to be tested (the subject string), followed by a blank (or blanks), followed by the string to be searched for in the subject string (the object string). The possible confusion between pattern matching and concatenation is avoided by the fact that the subject string must be the first string in the statement and be immediately followed by another string, the object string, with a separating blank (or blanks). If the subject string is to be a concatenation of other strings, then the concatenation must be surrounded by parentheses. If the object string is to be a concatenation, it does not have to have surrounding parentheses. If the object string is found anywhere in the subject string, i.e., it is a substring of the subject string, then the pattern match succeeds, otherwise, the pattern match fails.

Each of the following statements indicate pattern matching is to be done.

```
NAME1 NAME2
NAME1 NAME2 NAME3 NAME4
NAME 'STRING'
'STRING' NAME
(NAME1 'STRING1') "STRING2" NAME2
```

Assume X = 'AB' Y = 'ABC' Z = 'ABCD'

The following pattern matches succeed.

```
X 'A'
X 'B'
X 'AB'
X  X
Y  X
Z  X  'CD'
Z  Y
(X Y)  "BAB"
'BABABCABCDC'  X Y Z
```

The following pattern matches fail

```
X     'X'
X      Y
X      Z
'A'    Y
(X Y)   'AB'   Z
```

Labels

Any statement in a SNOBOL4 program may be labeled. A statement is labeled if there is a character in the first character position (except '*'). The label is all the characters up to the first blank. If a statement is to be unlabeled, the first character position must be blank. The purpose of the label is to give a name to the statement so that it may be referred to easily. END, RETURN, FRETURN may not be used as labels since they are reserved for special purposes.

The following statements are labeled FIRST, LOOP and NAME1.

```
FIRST  ALPHANUMERIC = @ALPHABET @DIGITS
LOOP   X = INPUT
NAME1  NAME2 NAME3
```

The following statements are not labeled.

```
       A = 10
       NAME1 NAME2 NAME3
```

The Go-To Field

The last field of a statement is called the go-to field. If it is not present then, after the current statement is executed, the statement below it will be executed. The field starts with a colon, :, (followed by any number of blanks). Only exit commands may follow the colon. Below are the three kinds of exit commands.

'(' label name ')'	unconditional exit
'S(' label name ')'	success exit
'F(' label name ')'	failure exit

If none of the exit commands are given after the colon, then the statement is treated like no colon was present.

A statement fails (immediately) if any part of it fails, otherwise, it succeeds. If an unconditional exit is given, then the statement to be executed is given by the label name regardless of success or failure of the statement. Otherwise, the colon may be followed by a success exit, failure exit, or both (either order, blanks permitted between the commands). If a statement fails and it has a failure exit, then the next statement is given by that label name; similarly, if it succeeds and there is a success exit. Otherwise, the next statement to be executed is the following statement. An exit to END will terminate the execution of statements.

The following are legal go-to fields.

```
:
: (LOOP)
: S(LOOP) F(DONE)
: F(DONE) S(LOOP)
: S(HERE)
: F(NEXT)
```

The next statement will be . . . .

'ABC' 'B' :S(L1) F(L2)		L1
'AB' 'XY' :S(L1)		the following statement
'AB' 'AB' :S(L1)		L1
X = Y :(AGAIN)		AGAIN
'X' 'Y' :(TOP)		TOP

Simple Pattern Matching Continued

One method of statement failure is for a pattern match to fail. Using this fact, appropriate exit commands can be used to decide if one string is a substring of another. If the subject string is given by name (i.e., it is not a concatenation or literal) then a successful pattern match can be followed by a replacement. The rule has the form: STRINGNAME STRING '=' STRING, where either STRING can be an arbitrary concatenation of strings. The subject string is searched from left to right for the first occurrence of the object string. If it is found, the part of the subject string matched by the object string is replaced by the string on the right hand side. If the pattern fails, no replacement is done since the statement fails immediately. For statements which contain pattern matching but no replacement field (i.e., no =) the subject string is not affected even if matching is successful.

These statements result in TOPCARD = 'KING OF SPADES' and an exit to OUTIT.

```
TOPCARD = 'ACE OF SPADES'
TOPCARD 'ACE' = 'KING'   :S(OUTIT)F(TRYAGAIN)
```

These statements result in ST3 = 'ABBBXXXBBA'

```
ST1 = 'AB'
ST2 = 'BA'
ST3 = 'ABBBABBABBA'
ST3 ST1 ST2 = 'XXX'
```

These statements result in TEST = ',PQR,WXY,KLM,'

```
LIST = ',PQR,WXY,KLM,XYZ,'
ELEMENT = 'XYZ'
TEST = LIST
TEST ',' ELEMENT ',' = ','   :S(SUCCESS)F(FAIL)
```

Fields of a Statement

There are five fields to every statement.

LABEL REFERENCE PATTERN REPLACEMENT GO-TO

If the label field is missing, then the statement is unlabeled.

If the reference field is missing, the pattern and replacement field must be missing. Thus, the statement is at most a go to [illegible] statement that will succeed.

If the pattern field is missing, then the statement is at most an assignment statement.

If the replacement field is missing, it is at most a pattern match without replacement.

If the go-to field is missing, the following statement will be executed next.

The following statements contain:

1. Pattern match with replacement and an unconditional exit,
2. Simple assignment,
3. Unconditional branch,
4. Pattern match with exit depending on success or failure of the pattern match.

	Label Field	Reference Field	Pattern Field	Replacement Field	Go-To Field
1.	LABEL	REFER	PAT	= REPLACE	:(GOTONEXT)
2.		NAME		= 'ABC'	
3.	FINI				:(END)
4.		SUBJECT	OBJECT		:S(S) F(F)

Teletype Input and Output

INPUT, OUTPUT, INPUTC, OUTPUTC are special teletype input-output variables. Anytime the variable OUTPUT is assigned a string value its contents are printed. A carriage return and linefeed are supplied at the end of the string and after every 72nd character printed on the teletype. Anytime the variable INPUT is used, its value will be collected from the teletype up to a carriage return, which is deleted from the string.

INPUTC and OUTPUTC are used for character-oriented input and output rather than line input and output. INPUTC collects exactly one character from the teletype. OUTPUTC outputs its contents to the teletype when it is assigned a value. No carriage returns are supplied, that is, it outputs its contents literally (see special conventions concerning line input). Execution of the following will print the line "NOW IS" after the line "NOW" is typed in.

```
     OUTPUT = INPUT 'IS'
```

After the following program is run the teletype line will be "APPEND", "BREAK", or "CHANGE", or a character which is not A, B, or C followed by a ?.

```
     X = INPUTC
     X  'A'    :S(A)
     X  'B'    :S(B)
     X  'C'    :S(C)
     OUTPUT = '?'           :(END)
  A  OUTPUT = 'PPEND'       :(END)
  B  OUTPUT = 'REAK'        :(END)
  C  OUTPUT = 'HANGE'       :(END)
```

The following statements will print "PROBLEM NUMBER" on the teletype and will pick up a response terminated by carriage return on the same line.

```
     OUTPUTC = 'PROBLEM NUMBER'
     NO. = INPUT
```

Binary and Unary Operators

There are many operators in the SNOBOL4 language, e.g., +, -, *, /, **, $, =, ., !. A binary operator requires a space on both sides of it (except for the binary operator space, as in concatenation, and the = operator). A unary operator may not have a space between it and its operand. Parentheses are not required for multiple unary operations (see precedence table).

The following are legal statements:

```
X=A * B
A B = C
X = $$$Y
```

These are illegal:

```
X = A* B
A B .C
X = $$$ Y
```

Arithmetic

A string is an integer if it is the null string (value Ø) or it is a string of digits with or without a leading + or -, and its absolute value is less than $2^{23}-1$. A literal string of digits may be written with or without surrounding quote marks. Arithmetic on integers results in integers with leading + signs and Ø's suppressed. If the value of an arithmetic operation is Ø, the result will be the string 'Ø'. The binary operators +, -, *, /, ** are used for addition, subtraction, multiplication, division, and exponentiation, with the usual precedences prevailing. The unary operators +, - are used for plus and minus. Parentheses can be used as needed. An arithmetic operation will cause an error message if the resulting integer is too large, if division by Ø occurs, or if Ø is raised to a power ≤ Ø.

These are integers

```
'123'
 123
 +1
'-1'
```

These are legal statements

```
X = (Y + 2 + 1) ** W
OUTPUT = 5 1 '2'
```

These statements output the result of dividing X by Y. If Y is Ø, it will output "INFINITE". The function NE, not equal, will be explained later.

```
       ANSWER = NE(Y,Ø)  X / Y    :S(OUTIT)
INF    OUTPUT = 'INFINITE'   :(END)
OUTIT  OUTPUT = ANSWER    :(END)
```

Indirect Referencing

A program may construct names by using the unary operator $ applied to a string. The result is a name which is the same as the contents of the string. Indirect referencing may appear anywhere that a name is legal (except in the label field). In the go-to field the resulting name should be a label. Indirect referencing can become a remarkably powerful facility since it provides the ability to change the names that are used in a statement between executions of that statement. It is important to note that names obtained by indirect referencing do not have to conform to the @, letters, digits, and . rules for names appearing in the source language.

If NAME1 = 'ALPHA',NAME2 = 'BETA',NAME3 = 'GAMMA', and LABEL = 'OK' then the following two statements would accomplish the same thing.

```
ALPHA BETA = GAMMA       :S(OK)
$NAME1 $NAME2 = $NAME3 :S($LABEL)
```

If Y = 'A', A = 'B', B = 'C', C = 'D' then after the following statement is executed X will contain D.

```
X = $$$Y
```

Grouping

Parentheses are used for grouping parts of a statement together, e.g., in arithmetic operations. The subject string of a pattern match can be given by a grouping of a number of strings together, or the name of the subject string can be given by a $ applied to a grouping. In general, groupings can appear in any field of a statement except the label field.

This is a legal statement.

```
$(X Y Z) (A + B) / C = (A * B) ** 2  :S($(A1))
```

Functions

In most programming languages the idea of a function is perhaps the most powerful feature. A function will take some arguments and produce a result which depends on those arguments. A function appears in the SNOBOL4 language as a function name followed by a '(', followed by a list of arguments separated by commas, followed by a closing ')'. Null arguments are permissible and missing arguments are assumed to be null. SNOBOL4 functions are recursive and the arguments are transmitted to the function by value (to be explained later). Throughout the SNOBOL4 language there are a number of pre-defined functions, e.g., SIZE(S), LE(I,J), LT(I,J), GE(I,J), GT(I,J), EQ(I,J), NE(I,J). SIZE returns the length of the string argument. The others make comparisons between two integer string arguments. For example, LT(I,J) returns the null string if I < J; otherwise, it fails. SNOBOL4 functions may either succeed or fail. If a function succeeds, it will return a value (many times it is a null string). A function may appear in any field of a statement (not the label field). It is a fatal error for a function to fail if it is in the go-to field.

The following are legal statements

```
     X = SIZE(Y)
     X = GT(Y,X) Y  :S($('X' SIZE(Y)))
```

If X and Y are null strings, then the following function calls are equivalent.

```
     EQ(X,Y)
     EQ(,Y)
     EQ(X,)
     EQ(X)
     EQ(,)
     EQ( )
```

User Functions

The user is permitted to define his own functions. There are four parts to the use of a user defined SNOBOL function.

1. Defining the function, listing its formal arguments, its local variables, and the label of its starting statement.
2. Calling the function with actual parameters.
3. Executing the function.
4. Returning from the function with a value or a failure return from the function, and in either case restoring saved values.

A function is defined by executing a DEFINE function with appropriate arguments. The DEFINE function has two arguments. The first argument is a string which contains the name of the function, followed by '(', followed by a list of formal arguments (if any) separated by ','s, followed by ')', followed by a list of local variables (if any) separated by ','s. The second argument is a string which contains the label name of the first statement to be executed in the function. If it is null, the label name is assumed to be the same as the function name.

The body of a function can be any of the statements of the program. The termination of a function is by an exit to RETURN or FRETURN. (RETURN & FRETURN cannot be user-defined labels.)

The call of a function is done when the part of the statement containing the function is evaluated (see order of evaluation). It appears in the source statement as a function name followed by an argument list in parentheses. The execution of the function is as follows. The actual arguments have been evaluated, i.e., all operations and function calls in the arguments have been completed, yielding actual argument values (of any datatype) to be assigned to the formal arguments. Then the current contents of the variable whose name is the same as the function name is saved. Similarly, the values of the formal arguments and the local variables are saved in the order specified when the function was defined. Then for formal

arguments are given the values of the actual arguments. The assignments are done left to right; each actual argument is assigned to the formal argument in the corresponding position. Any missing actual arguments are assumed to have a null value. The variable whose name is the same as the function name is given a null value and the local variables are also assigned null values.

The function is terminated by an exit to either RETURN or FRETURN. If the exit is to RETURN, then the function's value is the contents of the variable whose name is the same as the function name. If the exit is to FRETURN, then the statement which calls the function fails. In either case the saved values of the variable whose name is the same as the function name, the formal arguments, and the local variables are restored. It is quite permissible for a function to call other functions (before returning) including itself. Any exit to RETURN or FRETURN is a return from the most recent function call. The number of functions called which have not yet returned is called the level of recursion. Every call of a function increases the level by one. Before any function has been called the level is Ø. A function which is called at level n changes the level to n+1 and the return from the function is when the level changes from n+1 to n by a RETURN or FRETURN. An exit to RETURN or FRETURN at level Ø is an error.

The following is the renowned factorial function.

```
FACT    FACTORIAL = GT(N,Ø) N * FACTORIAL(N - 1)  :S(RETURN)
        FACTORIAL = 1  :(RETURN)
```

A program which takes a number N ≥ Ø from the teletype and outputs N! is the following.

```
START DEFINE ('FACTORIAL(N)','FACT')
      OUTPUT = FACTORIAL(INPUT)  :(END)
FACT FACTORIAL = GT(N,Ø) N * FACTORIAL(N - 1)  :S(RETURN)
     FACTORIAL = 1  :(RETURN)
```

The maximum function of two integer arguments can be defined by the statement

```
DEFINE ('MAX(X,Y)')
```

and the function body can be

```
MAX  MAX = GT(X,Y) X  :S(RETURN)
     MAX = Y  :(RETURN)
```

Distinction Between Names

The names of a variable, a function, and a label are distinct even when they are spelled the same. But there is the definite connection between the value of a function and the contents of the variable whose name is the same as the function's name. Also, it is common for the label of the first statement in the function to be the same as the function name.

Order of Evaluation

The order of evaluation of a statement is extremely important in determining the effect of the statement's execution. The ordering is as given below and is left to right in all fields, except as modified by the precedence of operators.

1. The reference field is evaluated. If it fails, the statement fails.
2. If there is a pattern field, it is evaluated. If it fails, the statement fails.

   Pattern matching is attempted. All immediate assignments (to be explained later) are done regardless of eventual success or failure of the pattern match. If the match fails, the statement fails. If the match succeeds, all assignments (to be explained later) are done regardless of eventual success or failure of the statement.
3. If there is a replacement field, it is evaluated. If it fails, the statement fails; otherwise, the replacement or assignment is made.

   If any of the above fails, the statement fails; otherwise, the statement succeeds.
4. The appropriate go-to field is evaluated. All function calls within the go-to field must succeed.

Patterns

So far the only data object discussed has been a string (although some strings are integers). In this section we will introduce a new object called a pattern. Since a pattern is a data object, it may be stored in a variable. That is, assignment statements with a pattern in the right-hand side store the pattern in the variable on the left-hand side. So far we have met just one kind of pattern matching, i.e., a test of whether or not one string is a substring of another. For the purposes of this section, a string can sometimes be thought of as a pattern (although it is a data object of type string, not pattern). The general idea behind a pattern is that the pattern matches a number of different strings. It tries each of the possible matches against the subject string in some specified order. The first match is taken as the successful pattern match; the matched substring is replaced if there is a replacement field. If none of the possible substrings match, then pattern matching fails. Below are listed the available pattern elements and rules for combining them.

Alternation ("OR")

A pattern which can match whatever any one of a number of alternative patterns will match may be formed by using the ! binary operator. The operands are patterns (or strings). The ! operator has lowest precedence of all operators. If P1, P2, and P3 are patterns, then the pattern which will match whatever P1, or P2, or P3 matches can be written P1 ! P2 ! P3. First, P1 is tried for a match; if it fails, then P2; if it fails, then P3.

If P = 'AA' ! 'AB' ! 'AC',then P can match any of the substrings 'AA', 'AB', or 'AC'.

Concatenation

A pattern may be formed by the concatenation operator (space) which can match the concatenation of strings matched by each of a number of patterns. If P1 matches some part of the subject string and P2 matches some other part of the subject string such that the two parts are adjacent in the subject string, then P1 P2 matches the concatenation of the two parts.

If P = 'A' ('A' ! 'B' ! 'C'), then P can match an A followed by an A, B, or C.

Arbitrary Strings

The variable name ARB contains the primitive pattern which can match any number of characters. It first matches the null string (Ø characters). If that fails, it will try one more character, etc.

```
'A' ARB ('B' ! 'C')
```

can match substrings of the form A followed by any number of characters up to a B or a C.
ARB ',' can match any substring ending in a ,.

Balanced Strings

The variable name BAL contains the primitive pattern which can match any non-null string of characters which is balanced with respect to the number of left and right parentheses. That is, it matches at least one character, and left and right parenthesis can be paired up such that every left parenthesis comes before the corresponding right parenthesis.

Thus, BAL can match any of the following substrings 'ABC', '(XYZ)', '( )', '((AB)CD)' and not any of these ') (', 'ABC)', '((X)' . The following pattern match will succeed with ARB matching ')))(' and BAL matching '(( ))' .

```
')))((( ))' ARB BAL.
```

There are several primitive functions which will return patterns as their value.

Fixed Length Strings

The function LEN(N) requires an integer argument and returns as its value a pattern which can match any string of exactly N characters.

```
'ABCDEFGH' LEN(3) 'G'
```

Here LEN(3) matches 'DEF' and 'G' matches 'G'.

Fixed Positions In Strings

The function POS(N) requires an integer argument and returns as its value a pattern which will match the null string immediately after the Nth character of the subject string. That is, it checks for the proper position in the subject string, in particular, POS(Ø) will only match at the start of the subject string.

Similarly, RPOS(N) will match the null string N characters from the end of the subject string. In particular, RPOS(Ø) will only match at the end of the subject string.

```
SUBJECT POS(Ø) BAL RPOS(Ø)
```

This will succeed if the subject string is balanced with respect to parentheses since BAL is forced to match the whole string.

Tabulation

The function TAB(N) requires an integer argument and returns as its value a pattern which will match all characters up to and including the Nth character of the object string. Similarly, RTAB(N) will match up to the last N characters. In particular, RTAB(Ø) will match to the end of the subject string.

Remainder

The variable name REM contains the primitive pattern which will match the remainder of the subject string. It is equivalent to RTAB(Ø).

The following pattern match will succeed with TAB(4) matching 'CD' and RTAB(2) matching 'EF'

```
'ABCDEFGH' 'B' TAB(4) RTAB(2)
```

In the following REM matches 'BABCBA'

```
'ABCBABCBA' 'C' REM
```

Alternative Characters

The function ANY(S) requires a string argument and returns as its value a pattern which will match any character which is in the string S. Conversely, NOTANY(S) will match any character which is not in S.

Runs of Characters

The function SPAN(S) requires a string argument and returns as its value a pattern which will match a string composed of characters which are in the string S. It *will not* match the null string, i.e., it must match at least one character. It *will not* match a string of characters if the run of characters from S can be lengthened, i.e., it matches up to the first character not in S or else the end of the subject string. Conversely, BREAK(S) will match characters which are not in S up to the first character which is in S. It *can* match the null string and *will not* match if a break character cannot be found.

Let X = 'ABCDEFGHIJKLMNOPQRSTUVWXYZØ123456789' then the pattern P = (POS(Ø) ! NOTANY(X)) NAME (NOTANY(X) ! RPOS(Ø)) will match successfully if there is an occurrence of the string NAME in the subject string which is not preceded or followed by an alphanumeric character.

```
'123ABCD456' SPAN('ABCDEFGHIJKLMNOPQRSTUVWXYZ')
```

Here the SPAN matches 'ABCD' .

Repetitions

The function ARBNO(P) has a pattern argument and returns as its value a pattern which matches any string that would be matched by an arbitrary number of consecutive occurrences of the pattern P. It first matches the null string. It is equivalent to the pattern X where X = '' ! P *X (the * operator will be defined later). That is, if it ever matches n P's, then it will try n+1 P's next. If the n+1st P fails to match, it will try more cases of n P's, if any.

```
'ABCDEFGHIJKL' POS(0) ARBNO(LEN(3)) RPOS(0)
```

will match the complete subject string since it is of length 12 = 3*4.

Signaling Failure

The variable name FAIL contains the primitive pattern which will always fail to match. The variable name FENCE contains the primitive pattern which will match the null string, but if tried for alternatives (rematch), it will cause pattern matching to completely fail. The variable name ABORT contains the primitive pattern which will cause pattern matching to completely fail.

No matter what pattern P is, the following will always fail:

```
    SUBJECT P ABORT
```

This succeeds

```
    'AB' 'A' FENCE 'B'
```

This fails

```
    'ACAB' 'A' FENCE 'B'
```

The FAIL alternative in the following is superfluous

```
    'ABC' 'A' (FAIL ! 'B' ! 'C')
```

The Order of Pattern Matching

A pattern is made up of subpatterns which are combined by concatenation and alternations. The primitive patterns are the contents or else the returned values of: strings, ARB, BAL, LEN(N), POS(N), RPOS(N), TAB(N), RTAB(N), REM, ANY(S), NOTANY(S), SPAN(S), BREAK(S), ARBNO(P), FAIL, FENCE, ABORT. There are four states of the pattern matching process that are of interest: match, success, fail, rematch. Success and failure here have little to do with success and failure of the statement. These are local states of the pattern matching process. The previous sections state what each of the primitive patterns will first match. If for some reason a match of an element does not work out later on, it is tried for a rematch. Most of the primitive elements fail to rematch. ARB, BAL, ARBNO(P) can be tried to rematch. (FENCE aborts all matching on rematch.) ARB and ARBNO(P) first match the null string. BAL first matches a substring of one character, or else more, if the first character was a '(' which needs to be balanced. On rematch whatever ARB has matched it extends that by one character. If no characters remain in the subject string, then ARB fails to rematch. What BAL has matched, on rematch it will try to extend that by another balanced substring of one or more characters. Failing that, rematch fails. ARBNO(P) on rematch tries to extend whatever it has matched by whatever another P will match. Failing that, it will rematch the previous P's.

A match of a concatenation is attempted by trying to match its first operand. If that succeeds, it will try its next operand. If all operands eventually succeed, then the concatenation succeeds. If any operand fails to match, then the previous operand is tried for a rematch. If the first operand fails, then the concatenation fails to match. If a concatenation must be rematched, then the last operand is rematched, etc.

A match of an alternative is attempted by trying to match the first operand. If that succeeds, the alternation succeeds. If it fails, then the next operand is tried. If all operands

fail to match, then the alternation fails to match. If an alternation must be rematched, the operand that was matched last is rematched. If this fails, then the next operand is tried for a match, etc.

The matching process begins with the first character of the subject string. Each primitive pattern element that matches extends the substring that has been matched. If pattern matching fails using the first character of the subject string, a pattern match is attempted starting with the next character in the subject string, and so on, until there are no more characters in the subject string at which to try to start a match. If pattern matching succeeds, it will have matched some substring of the subject string which can be replaced if the statement contains a replacement field. If pattern matching fails, then all possible substring matches of the subject string have failed to match. Complete failure of the pattern matching process causes the statement to fail. It is possible to set a mode where only matches which include the first character of the subject string (or no characters at all) are attempted. This mode can be set by assigning a negative integer to the keyword @ANCHOR (see keywords).

Deferred Pattern Definition

Patterns can be stored into variables. When the variables is used, it is just like using the pattern that was stored in the variables. In particular, when a pattern is defined, it may be defined in terms of other patterns. When a pattern is constructed (defined), the current values of its components are used. Consider the following statements:

```
P = RPOS(Ø)
Q = POS(Ø) ARB P
P = RPOS(1)
SUBJECT Q
```

What is the pattern Q that is used? When Q was defined, it became the pattern POS(Ø) ARB RPOS(Ø) and it has not been redefined.

At times it is desirable to define a pattern in terms of another pattern without the value of the other pattern being defined yet. The unary operator *, when applied to a name (the name is evaluated at definition time), says to use the pattern given by the name whenever a match of this pattern is attempted. Because * operates on a name to yield (eventually) a pattern, the * operator may only appear where a pattern is allowed. In particular, it cannot be used where an integer or string argument is expected, i.e., SPAN(*S) and POS (*N) are illegal.

Altering the previous example a bit, now what is the pattern Q that is used?

```
P = RPOS(Ø)
Q = POS (Ø) ARB *P
P = RPOS(1)
SUBJECT Q
```

Q is POS(Ø) ARB *P which here is equivalent to POS(Ø) ARB RPOS(1).

The * operator can be used to yield recursive pattern definitions.

P = 'B' ! *P 'C' can match any of the following substrings 'B', 'BC', 'BCC', 'BCCC', etc.

Value Assignment

When a pattern successfully matches, it is possible to assign the substring matched by any component (subpattern) of the pattern to a variable. The binary operator . is used to indicate value assignment in case the pattern successfully matches. Its left operand is a pattern (or subpattern) and its right operand is a name.

If pattern matching is successful and the subpattern was part of the successful match, then the substring that the subpattern matched will be assigned to the variable with the given name. If the subpattern was not part of the successful match, then no assignment is made. It is possible to do multiple assignments like ARB . X . Y. If the ARB was part of a successful match, then X and Y would receive the same value. Assignments are made left to right; thus, if two assignments are made to the same name, the last assignment would be the right most assignment.

Consider the following pattern matches.

```
'ABCDEFGH'  (ARB . X 'F') . Y
```

The pattern match succeeds resulting in the assignments

X = 'ABCDE', Y = 'ABCDEF' .

```
'123456789' (1 . X ! '2' . Y) LEN(3) . Z
```

The pattern match succeeds resulting in the assignments

X = '1', Z = '234' .   Y will retain its previous value.

Suppose STRING = 'AB,CD,EF' .

```
STRING ',' ARB . X ',' = ';' X ';'
```

The pattern match will succeed with X = 'CD' and will result in STRING = 'AB;CD;EF' .

Immediate Value Assignment

The value assignment described in the previous section occurs only on successful completion of pattern matching. It is also possible to assign a substring matched by any component (subpattern) of a pattern whenever that component successfully matches during the pattern matching process, regardless of the eventual success or failure of pattern matching. The binary operator $ is used in the same way . is used, except that assignments are immediate. Immediate assignment can be combined with deferred pattern definition, so that whenever a variable is assigned a new string value by immediate assignment the deferred pattern of the same name becomes a pattern which matches the new string value. Due to the fact that the pattern matcher will signal failure as soon as it knows it is no use trying any more possible matches, immediate assignments may not always have their expected final value when the subpattern is not part of a successful pattern match. In order to assure that the pattern matches will try all possible matches, the keyword @FULLSCAN can be set (to -1).

Consider the following pattern matches.

```
'BABCABCD' BAL $ Z *Z
```

The pattern match succeeds with Z = 'ABC' . The pattern BAL $ Z *Z matches only substrings of length 2 or larger in which the first half of the substring is identical to the second half and is balanced with respect to parentheses.

```
'ABC' ('A' $ X ! 'B' $ Y) 'D'
```

The pattern match fails but X and Y are assigned new values. X = 'A', Y = 'B' since an A and a B occur in the string.

Infinite Loops

The pattern matcher is sophisticated enough to prevent all infinite loops (due to recursive pattern definitions). When an infinite loop is detected, the matcher will know that it is useless to try to match some deferred pattern and will signal that the match of that deferred pattern fails, thus seeking alternative rematches. Suppose X = 'A' ! *X 'B' then, taken literally, the following pattern match would go into an infinite loop.

superfluous

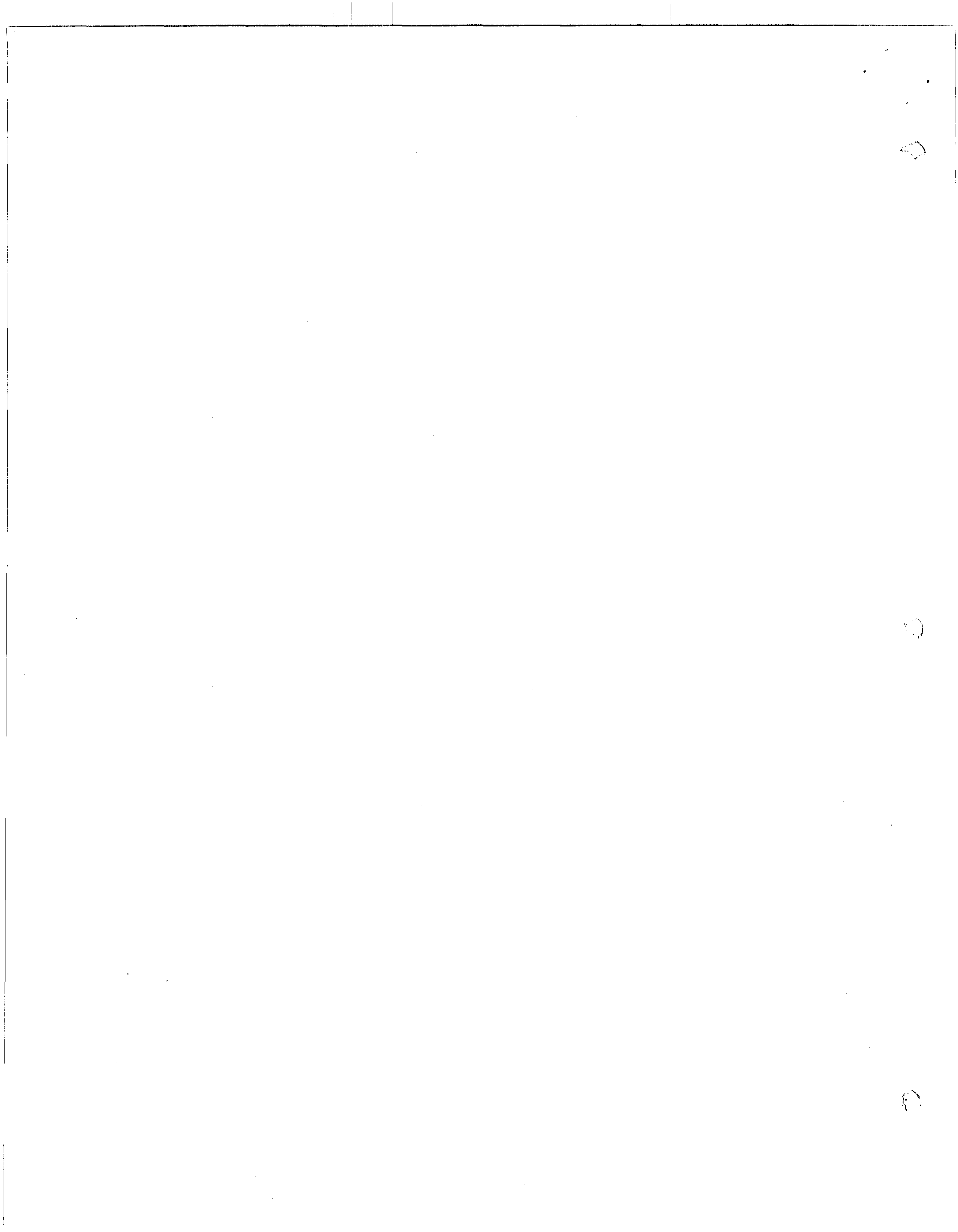
'C' (*X ! 'C')

That is, X is first tried for a match. 'A' fails to match, therefore, the alternative is tried. The first thing in the alternative is a match of the current value of X. Thus, a second attempt to match X is made. 'A' fails to match; therefore, the alternative is tried. The first thing in the alternative is a match of the current value of X. Thus, a second attempt to match X is made. 'A' fails to match; therefore, the alternative (to this instance of X) is tried. The first thing in the alternative is to match another X. And so it goes. The fact is that the pattern matcher catches this loop quite easily, signaling failure at the second instance of X. Thus, the first instance of X also fails. The alternative 'C' is now tried and the pattern match succeeds. In particular, such patterns as X = *X will always fail.

Efficiency in Patterns

The following eight ideas on more efficient patterns are not exhaustive but cover many of the most common or most costly cases of inefficiency.

1. A pattern that could be anchored should be anchored if it can possibly fail and thus try many extra unanchored matches. Use FENCE or POS(Ø) to anchor the pattern.
2. ARBNO is relatively slow. It is much preferable to find another construction if possible (without resorting to deferred patterns). For example, in most cases, ARBNO(' ') is best replaced by SPAN(' ') ! ''
3. In many patterns a BREAK or SPAN can be used instead of ARB. In such places it is usually preferable to use such a construction since BREAK and SPAN are extremely efficient.
4. Such constructions as '.' ! ',' are best replaced by ANY('.,')
5. Immediate value assignment should be avoided if possible; otherwise, many superfluous assignments may be made during pattern matching.
6. If possible, do not use FULLSCAN mode.
7. When using deferred patterns, avoid left recursion and other associated inefficiencies. The pattern matcher can catch infinite recursion but it can be extremely expensive in time.
8. It is very important that patterns be constructed once instead of everytime the pattern is to be used. Constant patterns are best defined once and for all in the beginning of the program outside of program loops. This is done by assigning the pattern to a variable and using the variable wherever that pattern was to be used.



Additional Built-in Functions

INTEGER(X) - Returns a null string if X is an integer string, otherwise, it fails.

TRIM(S) - Takes a string argument and returns the same string with trailing blanks removed.

DATE() - Takes no argument and returns an 8 character string which is the current date.
Format MM/DD/YY

TIME() - Takes no argument and returns a 7 character string which is the current time according to a 24-hour clock.
Format HHMM:SS

CLOCK(I) - Takes an integer string argument. If the argument is Ø, it returns the elapsed time counter (BRS 88). If the argument is not Ø, it returns the real time counter (BRS 42). Both counters are in units of 1/6Ø of a second.

IDENT(S,S) - Compares two string arguments and returns the null string if they are identical, otherwise, it fails.

DIFFER(S,S)- Compares two string arguments and returns the null string if they are not identical, otherwise, it fails.

UNSTACK(K) - Removes level(s) from runtime stack:

if $K \geq 0$ and $@LEVEL \geq K$ then unstack to level K
if $K < 0$ and $@LEVEL \geq |K|$ then unstack K levels
else FAIL

Additional Input-Output Facilities

INPUT, OUTPUT, INPUTC, OUTPUTC have been introduced for teletype input-output by line or characters. The following explains how to associate other string names with file input-output.

To communicate with a file it must be opened. A file is opened by calling either OPENIN or OPENOUT depending on whether it is to be an input or an output file. These functions require a single argument which is a string containing a complete file name. The returned value is a file number which is used to make references to that file.

Changed: Now use "INFILE" for input files and "OUTFILE" for output files.

To facilitate obtaining file names, the string name FILENAME when used like INPUT prints "FILE NAME" on the teletype and collects a file name. Thus, the value of FILENAME will be a complete file name.

Associations between string names and files is done by the ASSIGN function, which requires two arguments. The first argument contains the string name, the second contains the file number. When a string name is assigned to an input file, all other assignments are voided. When it is assigned to an output file, other output assignments remain.

Input and output can be by line or character. Initially, a string name which is assigned to a file is of type line. It can be changed to type character by calling the function CHAR which requires one argument, the string name. Similarly, it can be changed back to type line by calling the function LINE. New assignments to the string name will not change the mode.

Character input is just one character. Line input from a file reads everything up to a carriage return, linefeed which is discarded. Line input from the teletype is in the same format as source statements. The resulting string has every $\leftarrow^c$ deleted and has all & codes translated. End of file (D^c as first character of teletype line) causes failure of the statement.

Character output is literal output of the contents of the string. Line output supplies needed carriage returns and line-feeds and recognizes the line length of the output file.

The line length of an output file is initially set to 72 when the file is opened. It can be changed by calling the function, LENGTH, which requires two arguments. The first argument contains the file number, the second contains the new line length of the file. (If the second argument is Ø or null, the new line length will be $2^{23}-1$.)

To release all input or output assignments associated with a string name, call the function RELEASE with the argument containing the string name.

To close a file call CLOSE with the file number as the argument. A negative argument will close all files.

Examples:

```
N = OPENIN(INFILE)
ASSIGN('IN',N)
CHAR('IN')
M = OPENOUT("'/NEWFILE'")
ASSIGN('OUT',M)
ASSIGN('OUT',1)
CHAR('OUT') LINE('OUT')
LENGTH(M,128)
OUTPUT = INPUT :F(EOF)
RELEASE('OUT')
CLOSE(N)
CLOSE(-1)
```

CLOSE
ASSIGN
LENGTH
} Fail if file not open.

The Editor, Compiler, and Runtime

The SDS 940 SNOBOL4 system is divided into two distinct parts: The editor-compiler and the runtime. The editor-compiler is used to write, modify, and compile source statements. The runtime is responsible for the execution of statements.

The editor types $ when it is ready for commands. The editor is in most ways like QED. Familiarity with QED is required to use the capabilities of the editor. The editor commands which are similar to QED commands are /, =, ←, APPEND, CHANGE, DELETE, EDIT, FINISHED, INSERT, MODIFY, QUICK, READ FROM, SUBSTITUTE, TABS, VERBOSE, WRITE ON. Additional commands are BREAK, GO, HELP,^JUMP KILL, LIST,^NEXT PROCEED, and space followed by a SNOBOL statement (which cannot be a comment nor be labeled) to be immediately executed. Source statements can be read from a file, or using APPEND may be typed in directly. As each statement is read or typed, it is compiled. If there is an error, one edits the statement immediately. All standard QED addressing can be used; however, buffer operations are not available for addressing and editing. One other difference between QED and the editor is that every line typed in is an edit of the previous line typed or deleted. One consequence is that control D is a terminator only when no characters are in the new line. The QED commands will not be explained (see the QED manual); the editing control characters are summarized in Appendix B.

GO - begins execution of the SNOBOL statements after closing all open files and clearing all variables and resetting preset variables and functions. "--OK" is printed out as a warning; respond with "." . The first statement executed is given by the address of the GO. If no address is given, then execution begins at the first statement.

BREAK - sets up breakpoints at all statements in the interval addressed. A break at a statement is made before executing that statement and returns control to the editor.

JUMP - just like GO, but without initialization side effects.

KILL - releases all breakpoints in the interval addressed.

LIST - prints all breakpoints in the interval addressed.

PROCEED - continues execution after a breakpoint.

NEXT - executes N statements (default = 1)

A single rubout during execution will cause a break at the start of the next statement. (Remember that to complete the current statement all teletype input must be completed. You may also have to wait until the teletype output buffer is empty before seeing where the break was done; this buffer may have as much as 15 seconds worth of typing in it.)

A second rubout will return to the editor immediately (before finishing the statement). It is not possible to proceed in this case.

An unlabeled statement may be executed while in the editor by typing it in. Of course, it must start with a space. In particular, branches (goto's) are legal. This is the way to begin execution without the side effects of the GO command.

The following are equivalent ($ printed by the editor)

$GO.

$1GO.

To set a breakpoint at every statement type

$@BREAK.

Or to kill all breakpoints type

$@KILL.

PROCEED and NEXT use integer addresses.

$PROCEED.

To list all breakpoints, type:

$LIST.

This is an example of a SNOBOL statement line.

$ OUTPUT = INPUT

Special Operations

The source statements must be written in printable characters. To enter non-printable characters, e.g., into a string, type & followed by 3 octal digits, e.g., &155, or else & followed by a non-octal character, e.g., &A; however, in the latter case, characters Ø to 37 will remain unchanged, the others (40-77) will become the corresponding control characters. Note that & may only be entered by typing && or &006. To aid the above, the K^c editing character produces four characters, &XXX, where XXX is the octal code of the next character typed.

Continuation of a statement is possible by typing $\leftarrow^c$ at the end of the line to be continued. This character is entered into the source string and is treated like a blank when W^c is used, but is ignored by the compiler. Therefore, be sure to type any needed blanks in the source statement. In the editor typing linefeed is equivalent to typing $\leftarrow^c$.

All teletype line input is edited and is subject to the same rules as a source statement except that $\leftarrow^c$ is deleted and & codes are translated into internal form. D^c in the first character position causes failure of the statement (due to end of transmission).

Statements which begin with an * are treated as comments. Comments have no affect on the execution of the program. Execution is done the same as if the comments were not present.

Keywords

Keywords provide an interface between the SNOBOL4 program and certain internal symbols in the SNOBOL4 system. It is expected that additional keywords beyond those listed below will be implemented.

Read-only keywords:

@STCOUNT contains the number of statements that have been entered since execution began.

@STFCOUNT contains the number of statements that have failed.

@LEVEL contains the current level of recursion.

User changeable flags (a flag which is non-negative is off, negative is on):

@ANCHOR if on, sets the mode of pattern matching to anchored, that is, all patterns must match beginning with the first character of the subject string.

@FULLSCAN if on sets the mode of pattern matching to try all possible matches regardless of the impossibility of ever matching (i.e., no heuristics to speed up pattern matching).

Changeable limits:

@MAXLNGTH is the limit on the length of strings that can be formed. It is preset to 32000 which is the largest it can be set.

@STLIMIT is the limit on the number of statements that can be executed. It is preset to $2^{23}-1$ (its maximum)

@INTLIMIT is the limit on the maximum absolute value an integer can be. It is preset to (and can have a maximum of) $2^{23}-1$.

Literals (these keywords have predefined values and are unchangeable):

@ALPHABET contains 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

@DIGITS contains 'Ø123456789'

The following have the same values that the corresponding predefined pattern variables initially have.

@ARB

@BAL

@REM

@FAIL

@FENCE

@ABORT

Pseudo-Teletype Functions

There are a number of predefined functions which enable communication with a pseudo-teletype. The list includes LOGIN( ), LOGOUT( ), WAIT( ), SEND(S), ATSEND(S), RECV(N), and RECVLINE( ). Additionally, there is the ECHO(N) function which can be used independently of the pseudo-teletype functions. The conditional command processing capability produced by the combination of the pseudo-teletype functions and the SNOBOL language has been inspired by the CCP subsystem.

LOGIN( ) or LOGIN(NAME,PASSWORD)

The LOGIN function may be called with either two arguments or no arguments. The LOGIN function attempts to log in (ENTER) the user at a pseudo-teletype either under his name or under another name. If two arguments are given, the first is a user name and the second is the corresponding password to be used in entering at the pseudo-teletype. If no arguments are given, the user is logged in under his own name and password. If LOGIN is successful, it returns a null string as its value and places the pseudo-teletype in BEGINNER mode at the EXECUTIVE level with the input and output buffers clear. If LOGIN fails, it is due to one of the following reasons:

1. No pseudo-teletype is free.
2. No room on the time-sharing system.
3. The user name or password is incorrect.

It is an error to try to LOGIN if a previous LOGIN has not been logged out. It is also an error to call WAIT( ), SEND(S), ATSEND(S), RECV(N), or RECVLINE( ) if LOGIN has not been successfully called.

LOGOUT( )

The LOGOUT function is used to log out the pseudo-teletype. The user is automatically logged out (if logged in) whenever the GO or FINISHED command is used in the SNOBOL editor. If the LOGOUT function succeeds, it will return a null value. It will fail if the user is not logged in at a pseudo-teletype.

WAIT( )

The WAIT function always succeeds and returns the null string as its value, but before returning it waits until the pseudo-teletype is waiting for the teletype input with an empty input buffer. While it is waiting, it throws away all output from the last SEND or ATSEND function call. Also, before returning it clears the pseudo-teletype output buffer.

SEND(S)

The SEND function requires one argument which must be a string. SEND first does a WAIT, then sends the characters of the argument string to the pseudo-teletype. An error results if the internal collection buffer (about 6ØØØ characters long) of characters from the pseudo-teletype overflows before all the characters are sent. The SEND function succeeds with a null string as its value.

ATSEND(S)

The difference between ATSEND and SEND is that ATSEND does not do a WAIT; instead it sends rubouts to the pseudo-teletype to get it back to the EXECUTIVE level. Many times the user may want to wait until the pseudo-teletype is waiting for input or to receive all the output from the pseudo-teletype before preceding with an ATSEND. To do this either do a WAIT( ) or

enough RECV's or RECVLINE's to collect all the output. Both SEND and ATSEND initialize the collection machinery, that is, previous output from the pseudo-teletype is discarded.

RECV(N)

RECV(N) takes an integer argument (N > ∅). It collects literally the next X characters (X not greater than N but otherwise as large as possible) from the pseudo-teletype output resulting from the last SEND or ATSEND. If X is ∅, RECV fails, indicating that all output from the last SEND or ATSEND has already been collected. If X is greater than ∅, RECV succeeds in returning the X characters as its value. It should be noted that after any SEND or ATSEND, at most one call of RECV(N) can successfully return with less than N characters. Also, the function fails only if the pseudo-teletype is waiting for input.

RECVLINE( )

The RECVLINE function is used for receiving the output from the pseudo-teletype by line. The algorithm is that the first character is ignored if it is a linefeed, then all characters up to a carriage return are collected and returned as the value of RECVLINE; the carriage return is discarded. If the output from the pseudo-teletype does not contain a carriage return, then all the remaining characters are returned, unless the returned string would be the null string, in which case RECVLINE fails. RECV and RECVLINE can be intermixed.

The following will print the same thing as what would appear on the pseudo-teletype except it outputs an extra carriage return, linefeed in the case where the last line from the pseudo-teletype does not terminate with a carriage return, linefeed.

```
OUTLOOP OUTPUT = RECVLINE( )    :S(OUTLOOP)
```

ECHO(N)

The ECHO function requires an integer string argument. It succeeds and returns the null string as its value. ECHO is used for turning on and off the echoing of characters typed during teletype input in the running of a SNOBOL program. If the argument is negative, the echoing of characters is turned off, if non-negative, it is turned on. Turning off the echo may be of use in collecting passwords. Also, it is of use in preventing a double echoing effect that would exist in the first sample program if the echo was not turned off.

Sample Pseudo-Teletype Programs

```
*  This program implements a direct interaction with the
*  pseudo-teletype.
*  If a control ← is typed, a rubout is sent to the
*  pseudo-teletype.
*  If a control ↑ is typed, the pseudo-teletype is logged out.

BEGIN  LOGIN( )
       ECHO(-1)
LOOPA  A = INPUTC
       IDENT(A,'&↑') LOGOUT( )  :S(END)
       IDENT(A,'&←)  SEND('&137')  :S(LOOPB)
       SEND(A)
LOOPB  OUTPUTC = RECV(@MAXLNGTH)   :S(LOOPB) F(LOOPA)
*  This program does a fixed assembly, load and a dump on a
*  specified file (second OUTFILE).
*  Output from the pseudo-teletype also goes to a specified
*  file (first OUTFILE).

BEGIN DEFINE('XMIT(X)')
      N = OPENOUT(OUTFILE)
      ASSIGN('OUT',N)
      LOGIN( )
      XMIT("KDF.RS1.'/S1'.RS2.'/S2'.F.")
      XMIT("NARP./S1.'/B1'.")
      XMIT("NARP./S2.'/B2'.")
      XMIT("DDT.;T/B1.; T/B2.%FDUMP ON FILE" OUTFILE ".")

FINI  CLOSE(N) LOGOUT( )   :(END)
XMIT  SEND(X)
XMIT1 OUT = RECVLINE( )    :F(RETURN)
      OUT '?'   :F(XMIT1)
      OUTPUT = '?ERROR'   :(FINI)
```

APPENDIX A

Primitive Functions

ANY(S)
NOTANY(S)
SPAN(S)
BREAK(S)
POS(I)
RPOS(I)
TAB(I)
RTAB(I)
LEN(I)
ARBNO(P)
SIZE(S)
LE(I,J)
LT(I,J)
GE(I,J)
GT(I,J)
EQ(I,J)
NE(I,J)
CHAR(S)
LINE(S)
OPENIN(S)
OPENOUT(S)
CLOSE(I)
ASSIGN(S,I)
RELEASE(S)
LENGTH(I,J)
DEFINE(S,S)
INTEGER(X)
TRIM(S)
DATE()
TIME()
CLOCK(I)
IDENT(S,S)
DIFFER(S,S)
UNSTACK(K)

(Appendix A Continued)

Variables With Preset Values

ARB

BAL

REM

FAIL

FENCE

ABORT

Special Input-Output Variables

INPUT

OUTPUT

INPUTC

OUTPUTC

[illegible]

INFILE

OUTFILE

Reserved Labels

RETURN

FRETURN

END

(Appendix A Continued)

Operator Precedence Table

binary	!	(lowest)
		(space)
	.,$	
	+,-	
	*,/	
	**	
unary	+,-,*,$	(highest)

SNOBOL4 Syntax

```
statement=[label] [' '[semi] [' '][':' ! ':' goto]][' ']
semi=anam[[' '] '=' [' '] pexp] ! anam ' ' [patt [' ']['=' [' '] pexp]]!
     aatm[' '][patt]
patt=non-null pexp
goto=[' ']'F' '(' gexp ')' [' ']['S' '(' gexp ')'] !
     [' ']'S' '(' gexp ')' [' ']['F' '(' gexp ')'] !
     [' '] '(' gexp ')'
gexp=[' ']('$' aatm ! label)[' ']
pexp=ptrm{' '  '!'  ' ' ptrm}
ptrm=pprm{' ' pprm}  !  null
pprm=patm{' ' ('.' ! '$') ' ' anam}
patm='*' anam ! bexp ! '(' [' '] pexp [' '] ')'
bexp=atrm{' ' ('+' ! '-') ' ' atrm}
atrm=aprm{' ' ('*' ! '/') ' ' aprm}
aprm=aun[' ' '*' '*' ' ' aprm]
aun='+' aatm ! '-' aatm ! aatm
aatm='$' aatm ! lit ! name ! fncl !
     '(' [' '] aexp [' '] ')'
fncl=fnam '(' [' '][pexp[' ']{','[' '] pexp [' ']}] ')'
aexp=bexp{' ' bexp}
anam='$' aatm ! name
```

(Appendix A Continued)

Keywords

@ABORT

@ALPHABET

@ANCHOR

@ARB

@BAL

@DIGITS

@FAIL

@FENCE

@FULLSCAN

@INTLIMIT

@LEVEL

@MAXLNGTH

@REM

@STCOUNT

@STFCOUNT

@STLIMIT

Pseudo-Teletype Functions

LOGIN(NAME,PASSWORD)

LOGOUT( )

WAIT( )

SEND(S)

ATSEND(S)

RECV(N)

RECVLINE( )

ECHO(N)

APPENDIX B

EDITING CONTROL CHARACTERS

Control Character	Result
A	Delete last character typed
B	No action
C	Copy character from old line
D	Terminate or copy rest of line
E	Change insert-replace mode
F	Copy rest of old line, no typing
G	No action
H	Copy to end of old line
I	Tab
J,LINEFEED	Continuation
K	Give code for next char typed.
L	Delete line
M,CARRIAGE RETURN	Terminate statement
N	Character delete, restorative
O	Copy up to next character typed
P	Skip up to next character typed
Q	Delete statement, restorative
R	Retype, fast
S	Skip character
T	Retype, aligned
U	Copy up to next tab stop
V	Take next char literally
W	Delete word
X	Skip through next character
Y	Concatenate and re-edit
Z	Copy through next character
←	Continuation

References

[1] D.C. Angluin, and L. P. Deutsch, "Reference Manual, Q.E.D., Time-Sharing Editor," Document No. R-15, Project Genie, Advanced Research Projects Agency, University of Calif., Berkeley, California (March 26, 1968)

[2] E.J. Desautels, and Douglas K. Smith, "An Introduction To The String Processing Language SNOBOL4" Programming Systems and Languages, (1967) pp. 419-454

[3] L. Peter Deutsch, Larry Durham, and Butler W. Lampson, "Reference Manual, Time-Sharing System," Document No. R-21, Project Genie, Advanced Research Projects Agency, University of Calif., Berkeley, California (November 13, 1967)

[4] D. J. Farber, R. E. Griswold, and I. P. Polonsky, "The SNOBOL3 Programming Language," Bell System Technical Journal (July-August, 1966) pp. 895-943

[5] Allen Forte, "SNOBOL3 Primer," Massachusetts Institute of Technology, Cambridge, Massachusetts, and London, England, (1967)

[6] C. A. Grant, "Reference Manual, CCP, Conditional Command Processor," Document No. R-29, Project Genie, Advanced Research Projects Agency, University of Calif., Berkeley, California (July 14, 1967)

[7] R. E. Griswold, J. F. Poage, and I. P. Polonsky, "Preliminary Description of the SNOBOL4 Programming Language," Bell Telephone Laboratories, Inc., Holmdel, New Jersey, S4D1

[8] R. E. Griswold, J. F. Poage, and I. P. Polonsky, "Preliminary Report On The SNOBOL4 Programming Language," Bell Telephone Laboratories, Inc., Holmdel, New Jersey (November 22, 1967) S4D4

[9] R. E. Griswold, J. F. Poage, and I. P. Polonsky, "Preliminary Report On The SNOBOL4 Programming Language," Bell Telephone Labortories, Inc., Holmdel, New Jersey (March 20, 1968) S4D4b

[10] Butler W. Lampson, "930 SNOBOL System," Document No. 30.50.70, Project Genie, Advanced Research Projects Agency, University of Calif., Berkeley, California (April 18, 1966)

$P = 4242$