Preliminary Report on the
SNOBOL4 Programming Language

R. E. Griswold, J. F. Poage, and
I. P. Polonsky

Bell Telephone Laboratories, Inc.
Holmdel, New Jersey

November 22, 1967
The SNOBOL4 Programming Language

November 22, 1967

1. INTRODUCTION

This memorandum is a revision and elaboration of a previous report on SNOBOL4 [1]. A number of new language features are described, and some material is covered in more detail. A knowledge of SNOBOL3 is assumed.

The SNOBOL4 language is still under development. While the basic structure of the language will remain as described in this paper, further additions and changes may be expected from time to time.

While the specification of the language is as independent of a particular implementation as possible, certain dependencies are inevitable. The particular characters used in the syntax, and input-output specifications are examples. In such areas, this paper describes the implementation of SNOBOL4 for the IBM System/360 operating under OS.
2. DIFFERENCES BETWEEN SNOROL3 AND SNOROL4

The basic structure of SNOROL4 is much the same as that of SNOROL3 [2]. The pattern matching facility has been revised and extended, and several new features have been added. A number of minor changes have been made both in the syntax and in names and functions. These minor changes are described in the following two sections.

2.1 Changes in Syntax

The principal changes are:

1. Identifiers (string names or function names) must begin with a letter, and may not contain colons.

2. A colon rather than a slash separates the rule portion of a statement from the goto. Gotos may be separated from the colon by blanks, or the colon may stand alone. No colon is required unless there is a goto.

3. Complicated expressions do not require parentheses. Arithmetic operators have the usual precedences and associate in the usual way. Thus

\[ X = \frac{N}{M * Q} - P ** A ** B \]

is equivalent to

\[ X = \left( \frac{N}{(M * Q)} \right) - (P ** (A ** B)) \]

Other binary operators are described elsewhere in this paper. A table of operator precedence is given in Appendix A.

4. Several unary operators may be written consecutively without parentheses. Thus

\[ X = $$$N \]

is acceptable. The unary arithmetic operators + and - are included in the language. Other unary operators are described elsewhere in this paper.
Warning: Binary operators must be surrounded by blanks to distinguish them from unary operators. For example

\[ M = N - P \]

is a difference, while

\[ M = N - P \]

is a concatenation.

5. Either double or single quotation marks may be used for literals. Either

\[ \text{EXP} = 'A-(B+C)' \]

or

\[ \text{EXP} = "A-(B+C)" \]

may be written. Quotation marks must be used in like pairs. Single quotation marks may occur in literals surrounded by double quotation marks, and conversely. Thus

\[ \text{QUOTE} = "'" \]

assigns a single quotation mark as the value of \text{QUOTE}.

6. Integers are literals and need not be enclosed in quotation marks. For example

\[
\begin{align*}
\text{NEXTN} &= N + 1 \\
\text{SQUARE} &= M ** 2 \\
\text{RESULT} &= \text{PROD} * -3
\end{align*}
\]

are acceptable. The enclosing quotation marks may be used if desired.

7. Many expressions which are illegal in SNOBOL3 are syntactically correct in SNOBOL4. For example

\[ \text{LSON(N1)} = N2 \]
is syntactically correct in SNOBOL4. Depending on the definition of LSON, this expression may or may not be semantically correct when executed.

8. Pattern matching is significantly changed in SNOBOL4. The string variables of SNOBOL3 are no longer used. In fact

```
TEXT *WORD*
```

is syntactically incorrect in SNOBOL4. See Section 3 which describes pattern matching.

9. The semicolon may be used to terminate statements in SNOBOL4. For example

```
LOOP TEXT '=' = :S(LOOP); OUTPUT = TEXT
```

is a line consisting of two statements. As in SNOBOL3, the end of a card terminates a statement unless the next card begins with the continuation mark period.

2.2 Changes in Names and Functions

The principal changes are:

1. INPUT, OUTPUT and PUNCH replace SYSPIT, SYSPOT and SYSPPT as the names associated with input, output, and punch respectively. The I/O association functions READ and PRINT have been somewhat modified. See Section 11.

2. QUOTE does not have a preassigned value. See Section 2.1.

3. LT, LE, EQ, NE, GE, GT and INTEGER replace the numerical predicates .LT, .LE, .EQ, .NE, .GE, .GT, and .NUM of SNOBOL3.

4. IDENT and DIFFER replace the comparison predicates EQUALS and UNEQL of SNOBOL3.

5. The format of the function DEFINE is slightly modified. Local arguments are listed after the function prototype, rather than in a third argument. The entry label may be omitted (i.e. null) in which case the label is taken to be the same as the name of the function. For example
DEFINE('F (X,Y) N,M', 'FENTR')

defines a function F with formal arguments X and Y and two local
variables N and M. The entry label is FENTR. On the other hand

DEFINE('CPY (Z)')

defines a function CPY with formal argument Z and entry label CPY.

There is no intrinsic limit on the number of formal arguments
or local variables which may be specified for a defined function.

6. In addition to the functions described above, several new
functions are described elsewhere in this paper. The functions
OPSYN, SIZE and TRIM of SNOBOL3 are also available. The
functions MODE, ANCHOR and UNANCHOR are not included in SNOBOL4.
3. PATTERN MATCHING

SNOBOL4 departs radically from SNOBOL3 in the area of pattern matching. While the same rule formats are used to perform pattern matching, the facilities for creating patterns are quite different. In SNOBOL3, the structure of a pattern can only be specified by a particular syntactic configuration and remains fixed throughout program execution. In SNOBOL4, a pattern is a data object which may be constructed and changed during program execution.

3.1 Pattern Construction

There are several facilities for constructing patterns. These may be used in combination to construct very elaborate patterns. The following sections describe the fundamentals of pattern construction.

3.1.1 Alternation

A pattern which will match any one of a number of alternatives may be formed by use of the binary operator |. Thus

\[ \text{VOWEL} = 'A' | 'E' | 'I' | 'O' | 'U' \]

assigns to VOWEL a pattern which will match any single vowel.

Operands may be other patterns as well as strings:

\[ \text{EVOWEL} = \text{VOWEL} | 'Y' \]

creates a pattern which will match a \( Y \) as well as the other vowels.

The operator | has the lowest precedence of all operators so that

\[ A B | C D \]
is equivalent to

\[(A \ B) \ | \ (C \ D)\]

See Appendix A.

3.1.2 Concatenation

Concatenation in SNOBOL4 is the same as in SNOBOL3, but patterns may be concatenated as well. Thus

\[VPAIR = VOWEL \ VOWEL\]

creates a pattern which will match any two vowels in succession.

3.1.3 Arbitrary Strings

In SNOBOL4 there is a primitive pattern which will match any string of characters. This pattern corresponds to the 'arbitrary string variable' of SNOBOL3. The variable \(ARB\) has this primitive pattern as value at the beginning of program execution. The SNOBOL4 pattern

\[ITEM = ARB \ "\ ,\"\]

has the SNOBOL3 equivalent

\[** \ "\ ,\"\]

When \(ARB\) is the last element in a pattern, it does not automatically match the rest of the string. In this respect it differs from the arbitrary string variable of SNOBOL3. REM, described in Section 3.1.8, is used for this purpose.

3.1.4 Balanced Strings
The variable BAL has as its initial value a primitive pattern which will match any nonnull string of characters which is balanced with respect to parentheses. BAL is equivalent to the balanced string variable of SNOBOL3. Thus

\[ \text{NEST} = "( \text{BAL} \)" \]

is equivalent to the SNOBOL3 pattern

\[ "( * (* *) )" \]

### 3.1.5 Fixed-length Strings

There are several primitive functions in SNOBOL4 which return patterns as value. One of these, LEN, corresponds to the fixed-length string variable of SNOBOL3. The value of \( \text{LEN}(N) \) is a pattern which will match any string which is \( N \) characters long.

\[ \text{CARDLENGTH} = \text{LEN}(72) \]

is equivalent to the SNOBOL3 pattern

\[ */"72"* \]

### 3.1.6 Fixed Positions in Strings

Two functions, POS and RPOS, have patterns as value which specify fixed positions within strings.

The value of \( \text{POS}(N) \) is a pattern which will match a null string immediately after the \( N \)th character of a string. For example

\[ \text{OPFIELD} = \text{POS}(7) \text{LEN}(8) \]

will match eight characters following the seventh character of a string.
The value of RPOS(N) is a pattern which will match a null string N characters from the end of a string. For example

\[ \text{LASTCHAR} = \text{RPOS}(1) \text{ LEN}(1) \]

will match the last character of a string.

3.1.7 Tabulation

Two functions, TAB and RTAB, have patterns as value which specify tabulation to fixed positions within a string.

The value of TAB(N) is a pattern which will match up to \( \text{AND} \) including the Nth character. For example

\[ \text{OPFIELD1} = \text{POS}(7) \text{ TAB}(15) \]

is equivalent to the pattern OPFIELD given in Section 3.1.6.

The value of RTAB(N) is a pattern which will match up to the last N characters of a string. For example

\[ \text{LAST5} = \text{RPOS}(5) \text{ RTAB}(0) \]

will match the last five characters of a string.

3.1.8 Remainder

The variable REM has as its initial value a primitive pattern which will match the remainder of any string.

\[ \text{LAST5R} = \text{RPOS}(5) \text{ REM} \]

is equivalent to the pattern LAST5 in the previous section.

3.1.9 Alternative Characters
Two primitive functions, ANY and NOTANY, have pattern values which permit the specification of alternative characters.

The value of ANY(CS) is a pattern which will match any character in the string CS.

\[ \text{AVOWEL} = \text{ANY('AEIOU')} \]

is equivalent to the pattern VOWEL in Section 3.1.1. However, ANY is more efficient and compact than the explicit alternation of the individual characters.

The value of NOTANY(CS) is a pattern which will match any single character which does not occur in the string CS.

\[ \text{NOTVOWEL} = \text{NOTANY('AEIOU')} \]

will match any character which is not a VOWEL.

3.1.10 Runs of Characters

Two primitive functions, SPAN and BREAK, have patterns as value which permit the specification of runs of characters.

The value of SPAN(CS) is a pattern which will match a string composed of characters which appear in CS. SPAN(CS) will not match the null string.

\[ \text{INTEGER} = \text{SPAN('0123456789')} \]

will match any string consisting of digits.

Patterns resulting from SPAN(CS) match the longest possible run of characters from CS, and do not back up to match shorter strings. Thus the pattern

\[ \text{M10} = \text{INTEGER '0'} \]

will always fail to match.

The value of BREAK(CS) is a pattern which will match any string up to, but not including any character in CS. BREAK(CS) will match a null string.
The SNOBOL4 Programming Language

November 22, 1967

WORD = BREAK(',,;')

will match any string of symbols followed by a blank, comma, period, colon, or semicolon. The match fails if none of these symbols occurs.

3.1.11 Repetitions

The primitive function ARBNO permits the specification of an arbitrary number of repetitions of a pattern. The value of ARBNO(P) is a pattern which will match any string that would be matched by an arbitrary number of consecutive occurrences of the pattern P. For example

MUL5 = ARBNO(LEN(5))

will match any string whose length is a multiple of five (including the null string).

Patterns resulting from ARBNO(P) first match the null string (corresponding to zero occurrences of P). If an alternative match is requested as the result of backup, a match for P is attempted. Each subsequent request as a result of backup results in an attempt to match one more occurrence of P. Thus

M10A = ARBNO(ANY('0123456789')) '0'

will match the shortest string which consists of digits followed by a zero, including a string consisting of a single zero.

ARBNO(P) may be thought of as an alternation of the form

NULL | P | P P | P P P | P P P P | ...

3.1.12 Signalling Failure

Three primitive patterns permit the specification of various kinds of failure during pattern matching.

Pattern Matching 11
The variable FAIL has as its initial value a primitive pattern which will always signal failure to match. FAIL, when encountered, does not necessarily cause the pattern match to fail, but requests the pattern matching algorithm to seek an alternative. FAIL can be used to initialize a pattern to be constructed from many alternatives, much as the null string can be used to initialize the concatenation of many patterns. For example, if INPUT is attached to an input stream

```
ALTS = FAIL
LOOP
   ALTS = ALTS | TRIM(INPUT) :S(LOOP)
```

builds a pattern consisting of alternatives taken from INPUT.

During pattern matching, the first alternative, FAIL, will request an alternative immediately.

The variable FENCE has as its initial value a primitive pattern which always matches a null string, but causes the entire pattern match to fail if an alternative for it is requested in backup. Thus in

```
CARD = LABEL FENCE OP
```

if LABEL successfully matches, but OP fails to match, the pattern match will fail without seeking alternatives for LABEL. The main use of FENCE is to improve the efficiency of pattern matching by avoiding attempts to match which would be futile.

The variable ABORT has as its initial value a primitive pattern which, if encountered, causes the entire pattern match to fail. Thus

```
CARDFORM = "*" ABORT | CARD
```

will fail if matched against a string which starts with an asterisk, and otherwise will match for the pattern CARD.

### 3.2 The Order of Pattern Matching

The actual process of pattern matching is performed in much the same fashion as in SNOBOL3. In SNOBOL4 the order of pattern matching is much more important because alternatives can be specified.
The two structural aspects of patterns result from alternation and concatenation. Matches for alternatives are attempted from left to right. Thus in the pattern

\[
\text{DOTS} = '....' | '..'
\]

an attempt is first made to match for four dots. If this fails, a match for two dots is tried. Thus if DOTS is to be used to compress multiple occurrences of dots, as in

\[
\text{RDOTS TEXT DOTS} = '.' : S(\text{RDOTS})
\]

the order in which the alternatives are specified is important for efficiency. In fact, if the alternatives were specified in the other order

\[
\text{SDOTS} = '..' | '....'
\]

the second alternative would never match.

Matches for successive components in a concatenation proceed from left to right. If a component fails to match, the matching process backs up to the preceding component and tries a new match for it. Thus

\[
\text{BALPER} = \text{BAL '..'}
\]

will first match a balanced string. If the next character is a period, the match will succeed. If not, another attempt for BAL will be made, extending the string previously matched if possible.

### 3.3 Deferred Pattern Definition

The creation of a pattern is like the creation of a string in the sense that the current values of its components are used when the pattern is constructed. As a result of executing

\[
X = 'A+B'
X = '(', X, '*', X, ')
\]
The final value of $X$ is $(A+B\times A+B)$. The same principle applies to pattern construction. As a result of executing

$$P = 'X'$$
$$P = 'Y' | P 'Z'$$

the final value of $P$ is the same as it would be if

$$P = 'Y' | 'X' 'Z'$$

were executed.

On the other hand, it is sometimes desirable to defer the evaluation of a component of a pattern until pattern matching takes place. Consider the following section of program.

```
N = 0
DELETE LIST '( N )' = :F(OUT)
N = N + 1 : (DELETE)
OUT
```

Since the value of $N$ changes through a series of pattern matches, the pattern

```
'( N )'
```

must appear explicitly and be reconstructed for each new value of $N$.

Deferred pattern definition, implemented by the unary operator * overcomes this difficulty. When applied to a variable which appears in a pattern, * causes the evaluation of this component to be deferred until pattern matching actually occurs. Thus

```
NEST = '( *N )'
```

can be used as in the previous example:

```
N = 0
DELETE LIST NEST = :F(OUT)
N = N + 1 : (DELETE)
OUT
```
In this case, a new value of N is used in each pattern match.

Deferred pattern definition may also be used to achieve recursive patterns. If the example at the beginning of this section is revised so that

\[ P = 'Y' | \star 'Z' \]

then P will match strings of the form

\[ Y \]
\[ YZ \]
\[ YZZ \]
\[ YZZZ \]
... 

Similarly, the following statements create patterns which will match a simple class of arithmetic expressions.

\[
\begin{align*}
\text{VARIABLE} & = \text{ANY('XYZ')} \\
\text{ADDOP} & = \text{ANY('+-')} \\
\text{MULOP} & = \text{ANY('*/')} \\
\text{FACTOR} & = \text{VARIABLE} | '(' \star \text{EXP} ')' \\
\text{TERM} & = \star \text{FACTOR} | \star \text{TERM} \star \text{MULOP} \star \text{FACTOR} \\
\text{EXP} & = \star \text{ADDOP} \star \text{TERM} | \star \text{TERM} | \star \text{EXP} \star \text{ADDOP} \star \text{TERM}
\end{align*}
\]

Warning: \( \star N \) is a pattern whose value is determined when pattern matching is performed. As a consequence,

\[ P = \text{POS}(\star N) \]

is illegal, since the argument of POS must be a number.

3.4 Value Assignment

In SNOBOL3, a string name can be associated with a string variable so that if a pattern match is successful, the name is given a new value corresponding to the string matched by the variable. SNOBOL4 has two constructions which permit such a value assignment as a result of pattern matching.
3.4.1 Post-matching Value Assignment

In SNOBOL3, value assignment takes place only after the entire pattern has successfully matched. In SNOBOL4, the binary operator \( \cdot \) is used to associate a variable with a pattern component for this type of value assignment. Such an association has the form

\[
P \cdot V
\]

where \( P \) is a pattern and \( V \) is a variable to be associated. For example

\[
P = '( ' BAL \cdot B ' )'
\]

is equivalent to the SNOBOL3 pattern

\[
'( ' \ast ( B ) \ast ' )'
\]

A variable may be associated with any component of a pattern (including a literal). Value assignment is made only to the components of a pattern which match. Thus

\[
\text{LETTER} = ( \text{VOWEL} \cdot V | \text{LEN}(1) \cdot C ) \cdot L
\]

creates a pattern which will match any single character. If the character is a vowel it will become the new value of \( V \). If the character is not a vowel, it will become the new value of \( C \) instead. In either case the character will be the new value of \( L \).

The operator \( \cdot \) has the highest precedence of all operators and associates to the left. See Appendix A.

\[
P = '( ' BAL \cdot B \cdot \text{OUTPUT } ' )'
\]

would assign the value matched by \( BAL \) to both \( B \) and \( OUTPUT \). Assignment to an output-associated variable results in output just as if an explicit assignment had been made.

The same variable may appear in value associations more than once in a pattern. Value assignment is done from left to right and from the inside to outside in nestings. A value may be assigned to a variable more than once in this process, and the
The SNOBOL4 Programming Language

November 22, 1967

final value is determined by the order in which the assignments are done. For example, the pattern

\[
\text{NEST} = \text{'}(' \text{NEST} \cdot \text{OUTPUT }\text{'')} | \text{'}(' \text{BAL} \cdot \text{OUTPUT }\text{'})\text{'}
\]

when used in

\[
\text{'}((\text{X}))\text{' NEST}
\]

would print

\[
\text{X} \\
\text{(X)} \\
\text{((X))} \\
\text{(((X))))}
\]

3.4.2 Dynamic Value Assignment

Whereas the value assignment described in the previous section occurs only on successful completion of pattern matching, there is another type of association which results in value assignment whenever a component of a pattern successfully matches. The binary operator S associates a variable with a component of a pattern for this dynamic value assignment. The pattern

\[
\text{FULLBAL} = \text{BAL S OUTPUT RPOS(0)}
\]

when used in

\[
\text{'}A+ (B+C) \cdot (D/E)\text{' FULLBAL}
\]

would print

\[
A \\
A+ \\
A+ (B+C) \\
A+ (B+C) \cdot \\
A+ (B+C) \cdot (D/E)
\]

Since dynamic value assignment occurs whenever an associated component matches, values of associated variables may be changed even if the pattern match eventually fails. Thus the pattern
The SNOBOL4 Programming Language

S4D4a November 22, 1967

BALPAIR = BAL $ B1 FENCE BAL $ B2

would fail when used in

'(A+B))' BALPAIR

but (A+B) would be assigned as a new value of B1. The value of B2 would remain unchanged since no successful match would be made for the component with which it is associated.

Dynamic naming, together with deferred pattern definition, can also be used to achieve the effect of SNOBOL3 backreferencing. The pattern

L3PAIR = LEN(3) $ V ARB *V

will successfully match any string having at least two nonoverlapping identical substrings of length 3. If this pattern matches successfully, the desired substring will be the new value of V. If the pattern does not match, the new value of V will be the last three characters of the subject string.

The pattern FAIL can be used to force a pattern through all possible matches. Used in conjunction with dynamic value assignment, a listing of all the alternatives may be obtained. The pattern

ALLBAL = BAL $ OUTPUT FAIL

when used in

'((A+(B*C)) D)' ALLBAL

would ultimately fail, but print

((A+(B*C)) D)
(A+ (B*C))
(A+ (B*C)) +
(A+ (B*C)) +D
A
A+
A+ (B+C)
+
+ (B*C)
This device may be used to explicate the exact order in which pattern matching is attempted for any pattern.
4. ARRAYS

Arrays may be created by execution of the primitive function ARRAY. ARRAY(P, V) creates an array described by the prototype P and gives each element of the array the value V.

The prototype P describes the indexing and dimensionality of the array. For example

VECTOR = ARRAY('10')

assigns a one-dimensional array of length 10 to VECTOR. Since the second argument is omitted, all elements of the array have null strings as value. Indexing ordinarily starts at 1. Other lower bounds may be specified by using a colon to separate the upper and lower limits:

LINE = ARRAY('-5:5')

creates an array with a lower bound of -5 and an upper bound of 5.

Additional dimensions in a prototype are separated by commas.

BOARD = ARRAY('8,8','X')

defines an eight by eight array where all elements have the value 'X'.

There is no intrinsic limit on the size or dimensionality of an array.

Warning: The first argument of ARRAY is the prototype and the second is a value which is given to each element of the resulting array. Thus,

A = ARRAY('8,8')

creates a two-dimensional array with each element having the null string as value. On the other hand
A = ARRAY(8,8)

creates a one-dimensional array with each element having the value 8.

It is also important to realize that each element of an array is given the same object as value. Consequently

A1 = ARRAY(10)
A2 = ARRAY(10, A1)

creates only two arrays. Each element of A2 has the same array, A1, as value.

If the value of a variable is an array, an element in the array may be referred to through the variable. Angular brackets following any array-valued variable are used to specify the element.

VECTOR<2> = EXP

assigns the value of EXP to the second element of VECTOR. There is no requirement that all values of an array be the same kind of object.

If an index referring to an element of an array falls outside the range of the array, the array reference fails. Thus

OUTPUT = VECTOR<12>

would fail. This failure may be used to iterate through the elements of an array without knowing the size of the array. A function SUM, whose value is the sum of all the elements of an array, could have the defining statement

DEFINE('SUM (ARRAY) N')

with the definition

SUM N = N + 1
SUM = SUM + ARRAY<N> :S(SUM) F(RETURN)
The summation loop continues until \( N \) exceeds the extent of \( \text{ARRAY} \). This function does not need to know the size of \( \text{ARRAY} \), but only that it is a one-dimensional array with lower bound one.

The primitive function \( \text{PROTOTYPE} \) may be used to get an explicit representation of structure of an array. The value of \( \text{PROTOTYPE}(\text{A}) \) is the prototype of the array \( \text{A} \). Thus

\[
\text{STRUCTURE} = \text{PROTOTYPE}(\text{BOARD})
\]

assigns the value \( 8,8 \) to \( \text{STRUCTURE} \).

In some cases an array may not be the value of an explicitly known name. The primitive function \( \text{ITEM} \) permits reference to elements of such an array. The value of \( \text{ITEM}(\text{A}, I_1, \ldots, I_n) \) is the \((I_1, \ldots, I_n)\)th element of the array \( \text{A} \). For example

\[
\begin{align*}
X &= \text{VECTOR}(5) \\
\text{and} \\
X &= \text{ITEM}(\text{VECTOR},5)
\end{align*}
\]

are equivalent.

It is important to realize that an array is a data object. The same array may be the value of more than one variable. In this case, a change in the value of an element affects both variables which have this array as value. For example, as a result of executing the statements

\[
\begin{align*}
A &= \text{ARRAY}(5) \\
A<2> &= \text{"TWO"} \\
B &= A \\
B<2> &= \text{"W"}
\end{align*}
\]

the value of \( A<2> \) will be \( \text{TO} \).

A copy of an array may be made using the function \( \text{COPY} \). The copy is not changed by changing elements in the original, and conversely. Consequently, as a result of

\[
\begin{align*}
A &= \text{ARRAY}(5) \\
A<2> &= \text{"TWO"}
\end{align*}
\]
B = COPY(A)
B<2> 'w' =

the value of B<2> will be TO and the value of A<2> will be TWO.
5. REAL NUMBERS

SNOBOL4 provides a limited facility for real (floating point) arithmetic. Real numbers may appear in the program as literals. For example

\[ X = 72.1527 \]

Such literals must begin with a digit and contain a decimal point. Thus

\[ N = 0.01 \]

is acceptable, while

\[ N = .01 \]

is erroneous (see Section 12.2).

Addition, subtraction, multiplication, and division (but not exponentiation) may be performed on real numbers. Thus, as a result of

\[ M = X \times 3.24561 \]

the value of \( M \) becomes 234.1794.

An attempt to perform mixed arithmetic between real numbers and integer strings will result in error termination. Explicit conversion may be made using the function CONVERT. The value of \( \text{CONVERT}(V,T) \) is the result of converting \( V \) to type \( T \). Thus

\[ \text{OUTPUT} = \text{CONVERT}(M, 'STRING') \]

prints 234.1794. Similarly, strings can be converted to real numbers.

\[ \text{SUM} = N + \text{CONVERT}(5, 'REAL') \]
assigns the value 5.01 to SUM. In converting strings to real numbers, either real literals, such as 57.42, or integers such as 5 may be specified.
6. DATA TYPES

In SNOBOL4 there is only one kind of data object, the string. SNOBOL4 has many data types. Four types have already been described: STRING, PATTERN, ARRAY, and REAL. Others are described in the following sections.

6.1 Data Types in Operations

The data type of an object is used by the SNOBOL4 system to verify that appropriate types are given to procedures, or to select an appropriate procedure, depending on type. For example, the argument of the SIZE function must be a string.

\[
\text{VALUE} = \text{SIZE(ANY('0123456789'))}
\]

results in error termination since the argument of SIZE is a pattern. Similarly

\[
\text{SUM} = 3 + \text{ARRAY(7)}
\]

is erroneous. In other cases, different procedures are required for different data types. For example

\[
\text{SELECT} = \text{'BIN' SIZE(TRIM(INPUT))}
\]

is the concatenation of two strings and the result is a string. On the other hand

\[
\text{BINO} = \text{'BIN' (3 | 4)}
\]

is the concatenation of a string with a pattern. A different procedure is used and the result is a pattern.

6.2 Concatenation with the Null String
The SNOBOL 4 Programming Language

November 22, 1967

In concatenation, the null string is handled as a special case. If one of the two operands in concatenation is the null string, no concatenation is made. Thus while

\[
\text{POINTER} = \text{'X'} \text{ ARRAY}(10)
\]

is erroneous since a string cannot be concatenated with an array,

\[
\text{POINTER} = \text{IDENT(MARK)} \text{ ARRAY}(10)
\]

is acceptable, since IDENT returns a null string if it succeeds. Thus predicates may be used to achieve conditional expressions without interfering with the results of computation.

6.3 Data Type Determination

The programmer usually knows the data types of the objects which occur in his program. Sometimes, however, it is necessary to make an explicit determination. The function DATATYPE serves this purpose. The value of DATATYPE(V) is the data type of V. Thus

\[
\text{OUTPUT} = \text{DATATYPE(Span('01'))}
\]

would print PATTERN.
7. PROGRAMMER-DEFINED DATA TYPES

The programmer may define new data types by means of the function DATA. The result of executing DATA(P) is to create a data type and define field functions as given in the prototype P. For example

```
DATA('NODE(FATHER,LSON,RSIB,VALUE)')
```

creates a new data type NODE with four fields: FATHER, LSON, RSIB and VALUE. A NODE may be visualized as shown in Figure 7.1.

```
FATHER
LSON
RSIB
VALUE
```

Figure 7.1  Structure of a NODE

Execution of this DATA function defines a function NODE which creates objects of data type NODE. Hence

```
N1 = NODE()
```

creates a NODE which becomes the value of N1. The NODE function has four arguments corresponding to the fields FATHER, LSON, RSIB and VALUE. These fields may be assigned value when a NODE is created.

```
N2 = NODE(N1,"X")
```

creates a node with the node N1 as the value of its FATHER field and X as the value of its VALUE field. The LSON and RSIB fields are null.
Execution of the DATA function also creates field functions FATHER, LSON, RSIB and VALUE which refer to the fields of a NODE. Thus:

\[
\text{LSON}(N1) = N2
\]

assigns the node N2 to the LSON field of N1.

Using these functions, nodes can be created and trees constructed from them. The fields FATHER, LSON and RSIB permit representation of the structure of the trees. The VALUE field permits the assignment of contents of the nodes. For example:

\[
\begin{align*}
N1 &= \text{NODE}(,,**,*) \\
N2 &= \text{NODE}(N1,,**,y*) \\
N3 &= \text{NODE}(N1,,N2,**-) \\
N4 &= \text{NODE}(N3,,**,x*) \\
\text{LSON}(N1) &= N3 \\
\text{LSON}(N3) &= N4
\end{align*}
\]

creates a tree as illustrated in Figure 7.2.

![Figure 7.2 Representation of a Tree](image-url)
Subsequently, executing

\[
N5 = \text{NODE}(N3,\ldots,\text{VALUE}(N2)) \\
\text{RSIB}(N4) = N5 \\
\text{VALUE}(\text{RSIB}(\text{FATHER}(N4))) = \text{VALUE}(N4)
\]

produces the tree illustrated in Figure 7.3

![Figure 7.3 The Modified Tree](image)

This facility may be used to implement elaborate data structures. An example given in Appendix C illustrates how a set of tree functions [4] may be implemented.

There is no intrinsic limit on the number of data types which may be defined. The same field function may be defined on several data types. Hence

\[
\text{DATA('ITEM(FLINK,BLINK,VALUE)')}
\]

creates a data type ITEM which has the same field, VALUE, as the data type NODE.
As with arrays, a programmer-defined data object may be the value of more than one variable. A change in the value of a field through one variable will change the field value for the other variable. The function COPY may be used for programmer-defined data objects as well as arrays. See Section 4.
8. COMPILATION DURING EXECUTION

The first phase of a SNOBOL4 run is 'compilation' in which the source program is converted into intermediate object code which is then interpreted in an execution phase.

8.1 Creating Object Code

A program can compile more object code during execution and then execute this new code. Compilation is accomplished by using the CONVERT function to convert a string to data type CODE. The string to be converted should consist of SNOBOL4 statements terminated by semicolons. For example

```
NEWS = 'NEW OUTPUT = SUM; SUM = SUM + 1 : (OLD);'
NEWWCODE = CONVERT(NEWS,'CODE') : (NEW)
```

creates two new statements. One of these statements contains a label NEW. The goto then transfers to this new block of CODE. The two new statements are then executed and transfer made to OLD.

Blanks are as important in strings to be converted to code as they are in the program itself. A statement without a label must begin with a blank. The string to be converted must end with a semicolon.

8.2 Direct Gotos

The goto field specifies a variable which occurs as a program label. The result of converting a string to object code is a data object. In the previous example, this data object became the value of the variable NEWCODE. A special type of goto permits transfer directly to a block of object code, rather than through a label. This type of goto uses enclosing angular brackets rather than parentheses. The previous example could have been


```
NEWS = ' OUTPUT = SUM; SUM = SUM + 1 : (OLD);'
NEWCODE = CONVERT (NEWS, 'CODE') : <NEWCODE>
OLD

In this case transfer is made directly to the value of NEWCODE, and the label NEW is not needed.

Execution-time compilation permits a programmer-defined function CALL similar to the primitive function CALL of SNOBOL3. The defining statement can be

```
DEFINE ('CALL (FORM) S')
```

with the definition

```
CALL S = ' CALL = ' FORM ' : S (RETURN) F (RETURN);'
S = CONVERT (S, 'CODE') : <S>
```

The first statement creates the string which will become the code to be executed. The statement is then converted to code and executed. When executed, it evaluates FORM and assigns the result to CALL, through which the value is returned.

Compilation during Execution 33
9. KEYWORDS

Keywords provide an interface between the SNOBOL4 program and certain internal symbols in the SNOBOL4 system. Using keywords the program may determine, for example, how many statements have been executed. Keywords also permit the program to change the value of certain internal symbols, such as the limit on the number of statements which may be executed.

Keywords begin with an ampersand (&) which distinguishes them from program identifiers which may not contain ampersands.

An example is &STCOUNT whose value is the number of statements which have been executed. Similarly, the value of &STLIMIT is the limit on the number of statements which may be executed. Using these two keywords, the program may at some point limit further execution to 100 statements by executing

\[ &STLIMIT = &STCOUNT + 100 \]

&STLIMIT, whose value may be changed, is called an unprotected keyword. The value of &STCOUNT may not be changed, however, and is called a protected keyword. An attempt to change the value of a protected keyword results in error termination. The following sections describe the available keywords.

9.1 Protected Keywords

There are two types of protected keywords. The first type includes values internal to the SNOBOL4 system. The second type includes strings and primitive patterns which are predefined in the SNOBOL4 language.

9.1.1 Internal Values

1. &STCOUNT. The value of &STCOUNT is the number of statements which have been entered during program execution.

2. &STFCOUNT. The value of &STFCOUNT is the number of statements which have failed.
9.1.2 Predefined Values

Certain keywords have values which are predefined in the SNOBOL4 language. These include the alphabet for the machine, and the primitive patterns. These protected keywords are provided so that their values will always be available.

1. &ALPHABET. The value of &ALPHABET is a string containing all the characters of the machine on which SNOBOL4 is implemented. The characters are ordered according to their internal coding.

2. &ARB. The value of &ARB is the primitive pattern which matches any string of characters. &ARB and ARB have the same value at the beginning of program execution. The value of ARB may be changed, however, while the value of &ARB is protected.

3. &ABORT. &ABORT has the same value as ABORT at the beginning of program execution. See &ARB.

4. &BAL. As above.

5. &FAIL. As above.

6. &FENCE. As above.

7. &REM. As above.

9.2 Unprotected Keywords

There are two types of unprotected keywords. The first type includes internal switches. The second type includes internal parameters which may be varied by the program.

9.2.1 Internal Switches

Keyword switches controlling the anchored pattern matching mode and the post-mortem dump replace the MODE function of SNOBOL3 [2]. Switches are off if their value is 0 or the null string, and on otherwise. All switches are off at the beginning of program execution.
1. &ANCHOR. If &ANCHOR is on, the pattern matching is anchored. That is, all patterns must match beginning with the first character of the subject string. Thus, e.g.

    &ANCHOR = 'ON'

sets the anchored mode.

2. &DUMP. If &DUMP is on, a post-mortem dump of variable storage will be given.

9.2.2 Internal Parameters

1. &MAXLENGTH. The value of &MAXLENGTH is the limit on the length of strings that may be formed. The initial value of &MAXLENGTH is 5000, but this may be changed. Thus

    &MAXLENGTH = 1000

limits the maximum length of subsequent strings to 1000 characters. An attempt to form a string longer than the limit results in error termination of the program. All types of string formation are included in this limit: concatenation, value assignment as a result of pattern matching, and string input.

2. &STLIMIT. The value of &STLIMIT is the limit on the number of statements which may be executed (see &STCOUNT). The initial value of &STLIMIT is 50000. Exceeding the limit on statement execution results in error termination.
10. TRUTH PREDICATES

Two predicates, implemented by unary operators, are available for testing the success or failure which may result from evaluating expressions.

10.1 Negation

The unary operator \(~\) fails if its operand succeeds and succeeds if its operand fails. A null string value is returned on success facilitating its use among other constructions. Thus

\[ M<0> = \neg M<N> \neg M<-N> 0 \]

assigns 0 to \(M<0>\) only if both \(N\) and \(-N\) are out of range of the array \(M\).

Similarly

\[ N = \neg F(N) N + 1 \]

increments \(N\) only if \(F(N)\) fails.

10.2 Affirmation

The unary operator \(?\) is the converse of \(~\). It succeeds if its operand succeeds and fails if its operand fails. A null string is returned on success permitting its insertion in other constructions without affecting their values. Thus

\[ M<0> = ?M<N> \neg M<-N> 1 \]

assigns 1 to \(M<0>\) only if \(N\) is in range and \(-N\) is out of range of the array \(M\).
11. INPUT AND OUTPUT

As in SNOBOL3, input and output are accomplished by associating variables with data sets. Three variables have standard associations:

1. INPUT is associated with the normal input data set.
2. OUTPUT is associated with the normal print data set.
3. PUNCH is associated with the normal punch data set.

Input occurs whenever the value of the associated variable is used. Thus

\[ \text{CARD} = \text{INPUT} \]

results in reading from the normal input data set. The resulting string becomes the value of CARD.

Similarly, output occurs whenever the value of the associated variable is changed. Thus

\[ \text{OUTPUT} = \text{CARD} \]

causes the value of CARD to be printed.

11.1 I/O Association Functions

Other variables may be associated with other data sets using the primitive functions PRINT and READ. These functions have the form

\[ \text{READ(V,N,L)} \]
\[ \text{PRINT(V,N,F)} \]

where

1. V is the variable to be associated.
2. N is the data set reference number (symbolic unit number) with which the association is to be made.
3. L is the length of a string to be read on input.
4. F is a format to be used for output.
The three standard I/O variables have associations corresponding to

```
READ('INPUT',5,80)
PRINT('OUTPUT',6,'(1X,131A1)')
PRINT('PUNCH',7,'(80A1)')
```

FORTRAN IV conventions for data set reference numbers apply. Data set reference numbers from 1 through 99 are usually available. The data set reference number may not be omitted.

11.2 Output

Valid FORTRAN IV formats must be used for output. A format may specify literals and the output of a string by A-conversion (using n A1 to output a string of n characters). Numbers are strings in SNOBOL4 and must be put out by A-conversion. For example

```
PRINT('CONTROL',6,'(132A1)')
```
associates control with the normal print data set. With the specified format, the first character of CONTROL is used for carriage control. Thus

```
CONTROL = 1
```
results in a page eject.

A literal may be included to provide other desired information: to identify the particular variable being printed, for example. The association

```
PRINT('SUM',6,'(5H SUM=',120A1)')
```
would result in the variable SUM being printed with its name prefixed as given in the literal part of the format. Similarly

```
PRINT('TITLE',6,'(1H1,131A1/(1X,131A1))')
```
associates TITLE so that when a value is assigned to TITLE, a page is ejected and the value titles the next page of output.
If the format is omitted, and the data set reference number is 6, a default format of \((1X,131A1)\) is used. For all other data set reference numbers, the default format is \((80A1)\).

If output is requested for a data object which is not a string, the name of the data type is printed. Thus

\[
\text{OUTPUT} = 3.5
\]

would print \text{REAL}.

### 11.3 Input

Any positive number up to the maximum allowed string length may be used to specify input length. Thus

\[
\text{READ('CARD',5,72)}
\]

associates \text{CARD} with the normal input data set. Subsequently

\[
\text{IMAGE} = \text{CARD}
\]

reads in a string of 72 characters which becomes the value of \text{IMAGE}.

If the length specified is shorter than the record length on the input data set, the remainder of the record is lost. If the length is longer than the record length, enough records are read to satisfy the input request.

If an end of file (end of data set) is encountered on input, the statement which requested the input fails.

If the length is omitted, the default length is 80.

### 11.4 Rewind

The primitive function \text{REWIND} rewinds a file. \text{REWIND}(N) rewinds the data sets associated with the data set reference number \(N\). The next input request will result in reading from the first data set (file) associated with \(N\).
11.5 Back Space

The primitive function BACKSPACE backspaces a file. BACKSPACE(N) backspaces one record on the data set currently associated with the data set reference number N. If the data set is positioned at its first record, BACKSPACE(N) has no effect.

11.6 End of File

The primitive function ENDFILE ends a file. ENDFILE(N) writes an end of file on (closes) the data set associated with the data set reference number N. The next output request is written on a new data set (file).
12. NAMES

There are several circumstances in which explicit handling of names is useful. A name is any object which can have a value. In

\[ \text{WORD} = 'MAY' \]

WORD is a name which is given the value MAY. The basic relation between names and values is exhibited by the indirectness operator \$\$. For example, in

\[ \$\text{WORD} = 2 \]

the name MAY is given the value 2. Through indirectness, any string can be used as a name.

Objects other than strings may be used as names. Individual array items and fields of programmer-defined data objects are examples.

\[ \text{BOARD}(-1, 1) = 'X' \]

and

\[ \text{LSON}() = \text{HTREE} \]

are examples of computed names to which values are assigned.

12.1 Passing Names

A number of functions interpret the values of their arguments as names. For example

\[ \text{PRINT('SUM',N,F)} \]

associates the name SUM in the output sense (see Section 11). Subsequently, whenever the value of SUM is changed, output is performed.
String names are typically passed in this manner as literals. Computed names, such as BOARD<-1,1> cannot be passed as literals. Thus

PRINT(BOARD<-1,1>,N,F)

associates the value of BOARD<-1,1>, but does not associate the array item BOARD<-1,1>. On the other hand

PRINT('BOARD<-1,1>',N,F)

just associates the string of symbols BOARD<-1,1> and not the array item, just as the statements

X = 'BOARD<-1,1>'
$X = 5

have no connection with the array BOARD.

12.2 The Name Operator

To overcome this difficulty and put computed names on a par with string names, the unary name operator . may be used. The value of

.BOARD<-1,1>

is the name of BOARD<-1,1>. Thus

PRINT(.BOARD<-1,1>,N,F)

associates the array item in the output sense, and output is performed whenever this array item gets a new value. Similarly

READ(.LSON(ROOT),N,F)

forms an input association with LSON(ROOT), so that whenever a value for LSON(ROOT) is requested, a new value is obtained by input.
The name operator serves much the same purpose for computed names as quotation marks do for string names. The name operator applied to a string name behaves the same as quotation marks. Thus

\[
\text{WORD} = \text{.MAY} \\
\$\text{WORD} = 2
\]

produces the same result as the example above. Indirectness may be applied to any value obtained by the name operator. Hence

\[
\text{MARKER} = \text{.LSON(\text{ROOT})} \\
\$\text{MARKER} = \text{HTREE}
\]

is equivalent to

\[
\text{LSON(\text{ROOT})} = \text{HTREE}
\]

If the argument of the name operator is a string, the value returned by the name operator has data type STRING. If the argument of the name operator is a computed name, the value returned has data type NAME. If the argument of the name operator is not a name, error termination occurs. For example

\[
\text{.SIZE(\text{WORD})}
\]

is erroneous.

12.3 Returning by Name

A programmer-defined function may return a computed name (rather than value) by transferring to the label NRETURN which signals return by name.

An example of this feature exists in the programming of tree functions, where a NODE may be defined by

\[
\text{DATA('NODE(FATHER,LSON,RSIB,VALUE)')}
\]

The field functions FATHER, LSON, RSIB, and VALUE are automatically defined. Additional functions may be desired, however. For example, a function ROOTFATHER, which is the father field of a
The SNOBOL4 Programming Language

November 22, 1967

S4D4a

tree's root node might be defined. The defining statement could be

```
DEFINE('ROOTFATHER(NODE)', 'RTF')
```

with the definition

```
RTF IDENT(FATHER(NODE)) : S(RTF1)
NODE = FATHER(NODE) : (RTF)
RTF1 ROOTFATHER = NODE : (RETURN)
```

This function merely returns the node which is the root. The father field could be returned with the following definition:

```
RTF IDENT(FATHER(NODE)) : S(RTF1)
NODE = FATHER(NODE) : (RTF)
RTF1 ROOTFATHER = .FATHER(NODE) : (RETURN)
```

The naming operator assigns the computed field name to ROOTFATHER. The transfer to NRETURN indicates the value of ROOTFATHER is to be returned as a name. Thus

```
ROOTFATHER(TREE) = NEWNODE
```

assigns the value of NEWNODE to the father field of the root of TREE.

NRETURN can always be avoided by resorting to other constructions. If the definition were

```
RTF IDENT(FATHER(NODE)) : S(RTF1)
NODE = FATHER(NODE) : (RTF)
RTF1 ROOTFATHER = .FATHER(NODE) : (RETURN)
```

the corresponding assignment statement would be

```
$ROOTFATHER(TREE) = NEWNODE
```

NRETURN permits ROOTFATHER to be used on a par with FATHER, LSON, RSIB, and VALUE without the need for indirectness.
13. ADDITIONAL FUNCTIONS

In addition to the functions described earlier in this paper, there are two functions derived from supplementary functions developed for SNOBOL3. These are REPLACE, corresponding to the SNOBOL3 function RPLACE [3], and LGT, corresponding to LEXGT [5].

13.1 Character Replacement

One-to-one character replacement in a string may be accomplished using the function REPLACE. The value of REPLACE(S,CS1, CS2) is the result of replacing in S characters in CS1 by corresponding characters in CS2. For example, as a result of

```
TEXT = REPLACE(TEXT, ',', '.')
```

all commas in TEXT are replaced by periods, and conversely.

13.2 Lexicographical Comparison

Two strings may be compared according to their lexicographic (alphabetical) position by using the function LGT ('lexicographically greater than'). LGT(A,B) succeeds and returns a null value if A follows B in alphabetic order, and fails otherwise. Thus

```
LGT('ARMY','AIR FORCE')
succeeds, while

LGT('ARMY','NAVY')
fails.
```

In the case that a string is an initial substring of another, the longer string is lexicographically greater. consequently

```
LGT('AIR FORCES','AIR FORCE')
```
succeeds.

The order of the characters in lexicographical ordering is given in the keyword $\text{SALPHABET}$. See Section 9.1.2.
ACKNOWLEDGEMENT

The SNOBOL4 language was developed over a period of time, and the authors are indebted to many people for their suggestions. The contributions of Messrs. B. N. Dickman, D. J. Farber, P. D. Jensen, M. D. McIlroy and R. F. Rosin have been particularly significant.

Contributions to the implementation have been made by Messrs. I. Benyacar, A. R. Breithaupt, B. N. Dickman, R. S. Gaines, Mrs. M. R. Hawkins, Messrs. P. D. Jensen, A. M. Jones, Mrs. D. F. Teitelbaum, and Mr. L. C. Varian. The authors wish to thank Mr. Varian in particular for his valuable assistance in preparing the IBM System/360 implementation.

The authors gratefully acknowledge the assistance of Mr. J. F. Gimpel in the preparation of this paper. In addition, Mr. Gimpel contributed four of the programs in Appendix C: character conversion, random number generation, 'Typeset', and 'The Towers of Hanoi'.

The SNOBOL4 Programming Language

S4D4a

November 22, 1967
REFERENCES


5. Griswold, R. E., Special Purpose SNOBOL3 Functions - II. Unpublished, April 18, 1966.
APPENDIX A

Operator Precedence

The relative precedence of the binary operators is listed below in order of decreasing precedence. Operators with the same precedence are listed on the same line. Exponentiation associates to the right. All other operators associate to the left.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>Value Assignment</td>
</tr>
<tr>
<td>**</td>
<td>Exponentiation</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>+ -</td>
<td>Addition and Subtraction</td>
</tr>
<tr>
<td>[</td>
<td>Concatenation</td>
</tr>
<tr>
<td>\</td>
<td>Alternation</td>
</tr>
</tbody>
</table>
## APPENDIX B

List of Functions with Section References

<table>
<thead>
<tr>
<th>Function</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANY(CS)</td>
<td>3.1.9</td>
</tr>
<tr>
<td>ARRNO(P)</td>
<td>3.1.11</td>
</tr>
<tr>
<td>ARRAY(P,V)</td>
<td>4</td>
</tr>
<tr>
<td>BACKSPACE(N)</td>
<td>11.5</td>
</tr>
<tr>
<td>BREAK(CS)</td>
<td>3.1.10</td>
</tr>
<tr>
<td>CONVERT(V,T)</td>
<td>5, 8.1</td>
</tr>
<tr>
<td>COPY(S)</td>
<td>4, 7</td>
</tr>
<tr>
<td>DATA(P)</td>
<td>7</td>
</tr>
<tr>
<td>DATATYPE(X)</td>
<td>6.3</td>
</tr>
<tr>
<td>DEFINE(P,L)</td>
<td>2.2</td>
</tr>
<tr>
<td>DIFFER(X,Y)</td>
<td>2.2</td>
</tr>
<tr>
<td>ENDFILE(N)</td>
<td>11.6</td>
</tr>
<tr>
<td>EQ(N,M)</td>
<td>2.2</td>
</tr>
<tr>
<td>GE(N,M)</td>
<td>2.2</td>
</tr>
<tr>
<td>GT(N,M)</td>
<td>2.2</td>
</tr>
<tr>
<td>IDENT(X,Y)</td>
<td>2.2</td>
</tr>
<tr>
<td>INTEGER(X)</td>
<td>2.2</td>
</tr>
<tr>
<td>ITEM(A,I,...,In)</td>
<td>4</td>
</tr>
<tr>
<td>LEN(N)</td>
<td>3.1.5</td>
</tr>
<tr>
<td>LE(N,M)</td>
<td>2.2</td>
</tr>
<tr>
<td>LGT(A,B)</td>
<td>2.2</td>
</tr>
<tr>
<td>LT(N,M)</td>
<td>2.2</td>
</tr>
<tr>
<td>NE(N,M)</td>
<td>2.2</td>
</tr>
<tr>
<td>NOTANY(CS)</td>
<td>3.1.9</td>
</tr>
<tr>
<td>OPSYN(F,G)</td>
<td>2.2</td>
</tr>
<tr>
<td>POS(N)</td>
<td>3.1.6</td>
</tr>
<tr>
<td>PRINT(V,N,F)</td>
<td>11.1, 11.2</td>
</tr>
<tr>
<td>PROTOTYPE(A)</td>
<td>4</td>
</tr>
<tr>
<td>READ(V,N,L)</td>
<td>11.1, 11.3</td>
</tr>
<tr>
<td>REWIND(N)</td>
<td>11.4</td>
</tr>
<tr>
<td>REPLACE(S,CS1,CS2)</td>
<td>2.2</td>
</tr>
<tr>
<td>RPOS(N)</td>
<td>3.1.6</td>
</tr>
<tr>
<td>RTAB(N)</td>
<td>3.1.7</td>
</tr>
<tr>
<td>SIZE(S)</td>
<td>2.2</td>
</tr>
<tr>
<td>SPAN(CS)</td>
<td>3.1.10</td>
</tr>
<tr>
<td>TAB(N)</td>
<td>3.1.7</td>
</tr>
<tr>
<td>TRIM(S)</td>
<td>2.2</td>
</tr>
</tbody>
</table>
APPENDIX C

Sample Programs

This appendix contains twelve sample SNOBOL4 programs and their printed output. These programs illustrate various features and uses of the language from the simplest character manipulation through the most complicated recursive pattern matching.

The sample programs are:

1. Character Conversion
2. Word Counting
3. Bubble Sort
4. Random Number Generation
5. "Typeset"
6. Column Justification
7. "The Towers of Hanoi"
8. Theorem Proving
9. Magic Square Generation
10. Regular Expression Recognition
11. Phrase Structure Grammar Recognition
12. Tree Functions
SAMPLE PROGRAM 1: CHARACTER CONVERSION

SNOBOL4 (PRELIMINARY VERSION, 9.18.67)

******************************************************************************
* BCD TO EBCDIC
* THE FOLLOWING ONE-LINE PROGRAM CAN BE USED BY
* INSTALLATIONS UNDERGOING A CHANGE FROM IBM'S SECOND GENERATION
* TO THIRD GENERATION HARDWARE. THE PROGRAM CONVERTS FROM THE OLD
* BCD CODE FOR SCIENTIFIC CHARACTERS TO THE NEW EBCDIC CODE. IN
* PARTICULAR, IF INPUT IS THE CARD READER AND IF PUNCH IS THE CARD
* PUNCH, AS IS USUALLY THE CASE, THEN THE PROGRAM CONVERTS A DECK
* OF CARDS FROM 026 KEY PUNCH CODE TO 029 KEY PUNCH CODE.
******************************************************************************

L PUNCH = REPLACE(INPUT, "@3<%", "=(+)" ) :S(L)
END

SUCCESSFUL COMPILATION
SAMPLE PROGRAM 2: WORD COUNTING

SNOBOL4 (PRELIMINARY VERSION, 9.18.67)

* THIS PROGRAM COMPUTES THE NUMBER OF USES OF
* EACH WORD IN A BODY OF TEXT. LIMITED PUNCTUATION AND
* INDENTING OF PARAGRAPHS IS RECOGNIZED.
* SEPARATOR = ' | ' | ' | ' | ','
READ OUTPUT = TRIM(INPUT) ' '
TEXT = TEXT OUTPUT :F(NEXT)

NEXT TEXT ARB * WORD SEPARATOR = :F(PRINT)
IDENT (WORD,NUL) :S(NEXT)
$ (WORD ' : ' ) = :F(PRINT)
$ (WORD ' : ' ) :F(PRINT)
NE($ (WORD ' : ' ),1) :S(NEXT)
LIST = LIST WORD ' , '

PRINT OUTPUT = :S(NEXT)
OUTPUT = 'COUNT WORD'
OUTPUT = :F(END)
MORE LIST BREAK(' ', WORD ' : ' ) = :F(END)
OUTPUT = :F(END)
OUTPUT = ' :F(END)
MORE WORD = :F(END)
END

SUCCESSFUL-compilation
The ideal computing machine must then have all its data inserted at the beginning, and must be as free as possible from human interference to the very end. This means that not only must the numerical data be inserted at the beginning, but also the rules for combining them, in the form of instructions covering every situation which may arise in the course of the computation.

COUNT

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>THE</td>
</tr>
<tr>
<td>1</td>
<td>IDEAL</td>
</tr>
<tr>
<td>1</td>
<td>COMPUTING</td>
</tr>
<tr>
<td>1</td>
<td>MACHINE</td>
</tr>
<tr>
<td>3</td>
<td>MUST</td>
</tr>
<tr>
<td>1</td>
<td>THEN</td>
</tr>
<tr>
<td>1</td>
<td>HAVE</td>
</tr>
<tr>
<td>1</td>
<td>ALL</td>
</tr>
<tr>
<td>1</td>
<td>ITS</td>
</tr>
<tr>
<td>2</td>
<td>DATA</td>
</tr>
<tr>
<td>2</td>
<td>INSERTED</td>
</tr>
<tr>
<td>2</td>
<td>AT</td>
</tr>
<tr>
<td>2</td>
<td>BEGINNING</td>
</tr>
<tr>
<td>1</td>
<td>AND</td>
</tr>
<tr>
<td>2</td>
<td>BE</td>
</tr>
<tr>
<td>2</td>
<td>AS</td>
</tr>
<tr>
<td>1</td>
<td>FREE</td>
</tr>
<tr>
<td>1</td>
<td>POSSIBLE</td>
</tr>
<tr>
<td>1</td>
<td>FROM</td>
</tr>
<tr>
<td>1</td>
<td>HUMAN</td>
</tr>
<tr>
<td>1</td>
<td>INTERFERENCE</td>
</tr>
<tr>
<td>1</td>
<td>TO</td>
</tr>
<tr>
<td>1</td>
<td>VERY</td>
</tr>
<tr>
<td>1</td>
<td>END</td>
</tr>
<tr>
<td>1</td>
<td>THIS</td>
</tr>
<tr>
<td>1</td>
<td>MEANS</td>
</tr>
<tr>
<td>1</td>
<td>THAT</td>
</tr>
<tr>
<td>1</td>
<td>NOT</td>
</tr>
<tr>
<td>1</td>
<td>ONLY</td>
</tr>
<tr>
<td>1</td>
<td>NUMERICAL</td>
</tr>
<tr>
<td>1</td>
<td>BUT</td>
</tr>
<tr>
<td>1</td>
<td>ALSO</td>
</tr>
<tr>
<td>1</td>
<td>RULES</td>
</tr>
<tr>
<td>1</td>
<td>FOR</td>
</tr>
<tr>
<td>1</td>
<td>COMBINING</td>
</tr>
<tr>
<td>1</td>
<td>THEM</td>
</tr>
<tr>
<td>2</td>
<td>IN</td>
</tr>
<tr>
<td>1</td>
<td>FORM</td>
</tr>
<tr>
<td>2</td>
<td>OF</td>
</tr>
</tbody>
</table>
INSTRUCTIONS
COVERING
EVERY
SITUATION
WHICH
MAY
ARISE
COURSE
COMPUTATION
SAMPLE PROGRAM 3: BUBBLE SORT

SNOBOL4 (PRELIMINARY VERSION, 9.18.67)

*  BUBBLE SORT PROGRAM  *

DEFINE('SORT(N) I') 1
DEFINE('SWITCH(I) TEMP') 2
DEFINE('BUBBLE(I)') 3

*  GET THE NUMBER OF ITEMS TO BE SORTED.  *

*  N = TRIM(INPUT) 4
A = ARRAY(N) 5

*  READ IN ITEMS.  *

READ I = I + 1 6
A<I> = TRIM(INPUT) : F(GO) 7
OUTPUT = A<I> : (READ) 8

GO SORT(N) 9
OUTPUT = 10
OUTPUT = 11
OUTPUT = 'SORTED LIST' 12
OUTPUT = 13
I = 1 14
PRINT OUTPUT = A<I> : F(END) 15
I = I + 1 : (PRINT) 16

*  FUNCTIONS  *

*  SORT I = LT(I,N - 1) I + 1 : F(RETURN) 17
LGT(A<I>,A<I + 1>) : F(SORT) 18
SWITCH(I) 19
BUBBLE(I) : (SORT) 20

*  SWITCH TEMP = A<I> 21
A<I> = A<I + 1> 22
A<I + 1> = TEMP : (RETURN) 23

Appendix C 57
The SNOBOL4 Programming Language

S4D4a

November 22, 1967

* BUBBLE I = GT(I, 1) I - 1 ; F(RETURN) 24
  LGT(A<I>,A<I + 1>) ; F(RETURN) 25
  SWITCH(I) ; (BUBBLE) 26
  * END 27

SUCCESSFUL COMPILATION
GETLTH
EQUAL
GENVAR
ADJUST
DVREAL
END
ETIME
EXPIRT
ENDJOB
FSHRIN
ADJITL
BKSPCE
GETBAL
BUKINT
CKVAL
CLERTB
COMPAR
BRANCH
COMPLG
COPPLX
FATHER
GETOT
GETLG
ADREAL
GETBLK
EQDT
GETCL
GETCLI
BUFFER
ARRAY
BKSIZE
DESCR
EQU
FETCH
APDSP
EQUIV
COPY
CPEQV
DIVIDE
FNDRES
DECRC
DECR
BUCKET
FORMAT
ALTERN
DIVINT
ADDLG
ADDSIB
ADDSOR
SORTED LIST

ADDLG
ADDSIB
ADDSOmnop
ADJTTL
ADJUST
ADREAL
ALTERN
APDSP
ARRAY
BKSIZEx
BKSPACE
BRANCH
BUCKET
BUFFER
BUKINT
CHKVAL
CLRTB
COMPAR
COMPLG
COPPLX
COPY
CP'EQV
DECR
DECRG
DESCR
DIVIDE
DIVINT
DIVREAL
END
ENDJOB
EQLT
EQU
EQUAL
EQUIV
ETIME
EXPINT
FATHER
FETCH
FINDRES
FORMAT
FSHRTN
GENVAR
GETBAX
GETBLK
GETCL
GETCLI
**SAMPLE PROGRAM 4: RANDOM NUMBER GENERATION**

**SNOBOL4 (PRELIMINARY VERSION, 9.18.67)**

### A RANDOM NUMBER GENERATOR

**RANDOM (N)** RETURNS A VALUE UNIFORMLY DISTRIBUTED OVER THE INTEGERS \(0, 1, 2, \ldots, N-1\)

THE PSEUDO-RANDOM NUMBER GENERATION IS ACCOMPLISHED BY THE SO-CALLED POWER RESIDUE METHOD. THE VARIABLE RAN.VAR CYCLES THROUGH ALL NONNEGATIVE INTEGERS BELOW 100,000. THE INITIAL VALUE OF RAN.VAR WILL DETERMINE THE SEQUENCE OBTAINED AND IS CALLED THE WARM-UP CONSTANT.

**REFERENCE:**

```sdb
DEFINE('RANDOM (N)') : (RANDOM.END)

RANDOM
RAN.VAR = RAN.VAR * 1061 + 3251
RAN.VAR ARB RPPOS(5) = RANDOM = (RAN.VAR + N) / 100000 : (RETURN)

RANDOM.END
```

**TO ILLUSTRATE ITS USE WE WILL GENERATE AND PRINT A FEW 'RANDOM' NUMBERS.**

**N = 50**

**RAN.VAR = 0**

**RANGE = 100**

```
OUTPUT = ' THE FIRST ' N ' RANDOM NOS. '
```

```
OUTPUT = ' WITH WARM-UP CONSTANT ' RAN.VAR
```

```
OUTPUT = ' UNIFORMLY DISTRIBUTED BETWEEN 0 AND. '
```

```
(RANGE - 1) ' ARE:
```

```
OUTPUT =
```

DEMO

62
November 22, 1967

The SNOROL4 Programming Language

OUTPUT = A
N = GT(N,1) N - 1
END

SUCCESSFUL COMPILATION
THE FIRST 50 RANDOM NOS.
WITH WARM-UP CONSTANT 0
UNIFORMLY DISTRIBUTED BETWEEN 0 AND 99 ARE:

3
52
71
99
52
82
36
17
45
17
63
10
44
63
43
95
27
3
2
80
28
35
43
12
79
14
84
15
45
89
0
25
68
48
61
38
8
53
73
44
78
82
19
70
89
The SNOBOL4 Programming Language

SAMPLE PROGRAM 5: 'TYPESET'

SNOBOL4 (PRELIMINARY VERSION, 9.18.67)

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * PARAGRAPH() IS A FUNCTION WHICH SCANS THE INPUT TEXT UP TO THE FIRST LINE OF A NEW PARAGRAPH (INDICATED BY INDENTATION I.E. A BLANK IN COLUMN 1). IT STRINGS ALL OF THE WORDS OF THE PARAGRAPH INTO ONE LONG STRING. BLANKS ARE INSERTED BETWEEN LINES (ONE BLANK NORMALLY AND 2 BLANKS IF THE FIRST LINE ENDS IN A PERIOD). WHEN NO MORE PARAGRAPHS REMAIN, PARAGRAPH FAILS. * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

DEFINE('PARAGRAPH(X)') : (PARA.END) 1
PARAGRAPH 2
* THIS IS THE ENTRY POINT FOR THE FIRST TIME PARAGRAPH IS CALLED.
* SUBSEQUENT CALLS ENTER AT PARA.1
DEFINE('PARAGRAPH(X)', 'PARA.1') 3
PARA.LINE = TRIM(INPUT) 4
PARA.1 5
OUTPUT = 6
OUTPUT = PARA.LINE 7
PARAGRAPH = PARA.LINE 8

PARA.2 9
PARA.LINE = TRIM(INPUT) : F (PARA.3) 10
* CHECK FOR LEADING BLANK
PARA.LINE POS(0) ' ' 11
PARAGRAPH '.' RPOS(0) = '.' 12
OUTPUT = PARA.LINE 13
PARAGRAPH = PARAGRAPH ' ' PARA.LINE : (PARA.2) 14

PARA.3 15
DEFINE('PARAGRAPH(X)', 'PARA.4') : (RETURN) 16
PARA.4 17
: (RETURN) 18
PARA.END 19

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * THIS IS THE MAIN PROGRAM. ITS NAME IS TYPESET AND ITS MAIN PURPOSE IS TO PRINT OUT A PARAGRAPH WHICH IT HAS READ IN SUCH THAT BOTH LEFT AND RIGHT COLUMNS ARE ADJUSTED (SEE EXAMPLE BELOW). IT DOES THIS BY PADDING OUT BLANK AREAS WITHIN A LINE IF NO SUCH HOLES ALREADY EXIST WITHIN THE LINE, THEN THE PROGRAM BEGINS TO SEPARATE THE LETTERS OF INDIVIDUAL WORDS * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
The SNOBOL4 Programming Language

November 22, 1967

INDENTATION = ' '
LINE_WIDTH = 60
NB = NOTANY(' ')
HOLE = NB * BLANK * NB * C
UNPADDED.LINE = ARB * LINE * ARBNO(' ') ARBNO(NB) * Y
LONG.WORD = (NB ARR) * LINE SPAN(' ') NULL * Y
TYPSET
P = PARAGRAPH()
OUTPUT =
TS.0
LE(SIZE(P),LINE_WIDTH) : F(TS.1)
OUTPUT = INDENTATION P : (TYPSET)
TS.1
P UNPADDED.LINE | LONG.WORD = : F(ERROR)
P = Y P
BLANK =
LINE NB SPAN(' ') NR : F(TS.3)
TS.2
BLANK = LT(SIZE(BLANK),LINE_WIDTH) BLANK ' ' : P(NEXT)
TS.3
GE(SIZE(LINE),LINE_WIDTH) : S(NEXT)
LINE HOLE = A B ' ' C : F(TS.2) S(TS.3)
NEXT
OUTPUT = INDENTATION LINE : (TS.0)
END
SUCCESSFUL COMPILATION
"Scientific men must often experience a feeling not far removed from alarm, when we contemplate the flood of new knowledge which each year brings with it. New societies spring into existence, with their proceedings and transactions, laden with the latest discoveries, and new journals continually appear in response to the growing demand for popular science. Every year the additions to the common stock of knowledge become more bulky, if not more valuable; and one is impelled to ask, where is this to end?"

Lord John William Strutt Rayleigh, 1874
SAMPLE PROGRAM 6: COLUMN JUSTIFICATION

SNOBOL4 (PRELIMINARY VERSION, 9.18.67)

KEYPUNCH OPERATORS FREQUENTLY HAVE DIFFICULTY RIGHT
ADJUSTING VARIABLE LENGTH DATA WITHIN FIELDS ON DATA CARDS.
KEYPUNCH ERRORS MAY BE REDUCED IF THE DATA ARE KEYED ONTO
THE CARDS WITH AN ARBITRARY NUMBER OF BLANKS BETWEEN DATA
ITEMS. THE FOLLOWING PROGRAM WILL THEN RIGHT JUSTIFY THE
DATA IN SPECIFIED FIELDS AND FLAG CARDS THAT HAVE TOO MANY
FIELDS, TOO FEW FIELDS, OR FIELDS THAT ARE TOO LONG FOR THE
SPACE PROVIDED.

THE FIRST CARD OF THE INPUT DECK LISTS THE RIGHT-
MOST COLUMNS OF ALL FIELDS ON THE OUTPUT CARDS. THE NUMBER OF
FIELDS WHICH MAY BE SPECIFIED IS LIMITED BY A TOTAL OF
80 CHARACTERS. Thus,

8, 16, 24, 48, 72

LEFT JUSTIFIED ON THE FIRST CARD SPECIFIES 5 OUTPUT FIELDS
OF SIZE 8, 8, 8, 24, AND 24 COLUMNS RESPECTIVELY.

M = ARRAY (30)
BLANKS = '
PAT = POS (0) BREAK (') . FIELD SPAN (')

INITIALIZE MATRIX WITH SIZE OF FIELDS

COLS = TRIM (INPUT) ', '
INIT
COLS ARB = COL ' , ' = :F (READ)
L = L + 1
M<CL> = COL - LINE
LINE = COL : (INIT)

READ AND REFORMAT LINE

READ N = 1
LINE =
CARD = TRIM (INPUT) '
BADCARD = CARD
CARD = POS (0) SPAN (' ') =
LOOP CARD = PAT = :F (BAD)
BLANKS GE (M<CL> , SIZE (FIELD)) . LEN (M<CL> - SIZE (FIELD)) . BL

Appendix C 69
The SNOBOL4 Programming Language

November 22, 1967

LINE = LINE BL FIELD
N = L'T (N, L) N + 1
IDENT (CARD)
OUTPUT = LINE

* * FLAG AND PRINT ORIGINAL OF BAD LINES *
BAD
OUTPUT = '******' BADCARD '******'
END

: F(BAD)
: S(LOOP)
: F(BAD)
: (READ)
: (READ)
### The SNOBOL4 Programming Language

**November 22, 1967**

<table>
<thead>
<tr>
<th>DECIMAL</th>
<th>EXP</th>
<th>HEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2**0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2**1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2**2</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>2**3</td>
<td>8</td>
</tr>
<tr>
<td>16</td>
<td>2**4</td>
<td>10</td>
</tr>
<tr>
<td>32</td>
<td>2**5</td>
<td>20</td>
</tr>
<tr>
<td>64</td>
<td>2**6</td>
<td>40</td>
</tr>
<tr>
<td>128</td>
<td>2**7</td>
<td>80</td>
</tr>
<tr>
<td>256</td>
<td>2**8</td>
<td>100</td>
</tr>
<tr>
<td>512</td>
<td>2**9</td>
<td>200</td>
</tr>
<tr>
<td>1024</td>
<td>2**10</td>
<td>400</td>
</tr>
<tr>
<td>2048</td>
<td>2**11</td>
<td>800</td>
</tr>
<tr>
<td>4096</td>
<td>2**12</td>
<td>1000</td>
</tr>
<tr>
<td>8192</td>
<td>2**13</td>
<td>2000</td>
</tr>
<tr>
<td>16384</td>
<td>2**14</td>
<td>4000</td>
</tr>
<tr>
<td>32768</td>
<td>2**15</td>
<td>8000</td>
</tr>
<tr>
<td>65536</td>
<td>2**16</td>
<td>10000</td>
</tr>
<tr>
<td>131072</td>
<td>2**17</td>
<td>20000</td>
</tr>
</tbody>
</table>

***** 2621442**18 400000  *****

***** 5242882**19 800000 / *****
SAMPLE PROGRAM 7: "THE TOWERS OF HANOI"

THE TOWERS OF HANOI

POLE A

POLE B

POLE C

THIS EXAMPLE ILLUSTRATES THE USE (AND POWER) OF RECURSIVE PROGRAMMING.

'THE TOWERS OF HANOI' IS AN ANCIENT GAME CONSISTING OF 3 POLES AND A NUMBER OF DIFFERENT Sized RINGS, AS INDICATED ABOVE. THE OBJECT OF THE GAME IS TO MOVE THE RINGS FROM POLE A TO POLE C SUCH THAT

(1) ONLY ONE RING IS MOVED AT A TIME AND
(2) A LARGER RING IS NEVER ALLOWED TO LAY UPON A SMALLER RING.

IN THE PROGRAM BELOW, A FUNCTION IS DEFINED WHICH MOVES N RINGS FROM POLE P1 TO POLE P2 USING POLE P3 AS TEMPORARY STORAGE. THE FUNCTION CAN BE DEFINED IN TERMS OF MOVING N-1 RINGS BETWEEN APPROPRIATE POLES AND A SINGLE MOVE OF THE NTH RING. FINALLY, THE MAIN ROUTINE CONSISTS MERELY OF A SINGLE CALL TO THE FUNCTION ALREADY DEFINED.

DEFINE ('HANOI (N,P1,P2,P3)')

HANOI

EQ(N,0)
HANOI(N - 1,P1,P3,P2)
OUTPUT = 'MOVE RING ' N ' FROM ' P1 ' TO ' P2
HANOI(N - 1,P3,P2,P1)
The SNOROL Programming Language

November 22, 1967

HANOI.END
HANOI(5,'POLE A','POLE C','POLE B')
END

SUCCESSFUL COMPILATION
The SNOBOL4 Programming Language

MOVE RING 1 FROM POLE A TO POLE C
MOVE RING 2 FROM POLE A TO POLE B
MOVE RING 1 FROM POLE C TO POLE B
MOVE RING 3 FROM POLE A TO POLE C
MOVE RING 1 FROM POLE B TO POLE A
MOVE RING 2 FROM POLE B TO POLE C
MOVE RING 1 FROM POLE A TO POLE C
MOVE RING 4 FROM POLE A TO POLE B
MOVE RING 1 FROM POLE C TO POLE B
MOVE RING 2 FROM POLE C TO POLE A
MOVE RING 1 FROM POLE B TO POLE A
MOVE RING 3 FROM POLE C TO POLE B
MOVE RING 2 FROM POLE C TO POLE A
MOVE RING 1 FROM POLE B TO POLE A
MOVE RING 3 FROM POLE B TO POLE C
MOVE RING 1 FROM POLE C TO POLE A
MOVE RING 2 FROM POLE C TO POLE B
MOVE RING 1 FROM POLE B TO POLE A
MOVE RING 4 FROM POLE B TO POLE C
MOVE RING 1 FROM POLE C TO POLE B
MOVE RING 2 FROM POLE C TO POLE A
MOVE RING 1 FROM POLE B TO POLE A
MOVE RING 3 FROM POLE A TO POLE C
MOVE RING 1 FROM POLE B TO POLE A
MOVE RING 2 FROM POLE B TO POLE C
MOVE RING 1 FROM POLE A TO POLE C
November 22, 1967
SAMPLE PROGRAM 8: THEOREM PROVING

SNOBOL4 (PRELIMINARY VERSION, 9.18.67)

THIS PROGRAM IS THE ALGORITHM BY HAO WANG (CF. TOWARD
MECHANICAL MATHEMATICS', IBM JOURNAL OF RESEARCH AND
DEVELOPMENT 4 (1) JAN 1960 PP. 2-22.) FOR A PROOF-DECISION
PROCEDURE FOR THE PROPOSITIONAL CALCULUS. IT PRINTS OUT A
PROOF OR DISPROOF ACCORDING AS A GIVEN FORMULA IS A THEOREM
OR NOT. THE ALGORITHM USES SEQUENTS WHICH CONSIST OF TWO
LISTS OF FORMULAS SEPARATED BY AN ARROW (--). INITIALLY, FOR
A GIVEN FORMULA F THE SEQUENT

--> F

IS FORMED. WANG HAS DEFINED RULES FOR SIMPLIFYING A FORMULA
IN A SEQUENT BY REMOVING THE MAIN CONNECTIVE AND THEN
GENERATING A NEW SEQUENT OR SEQUENTS. THERE IS A TERMINAL
TEST FOR A SEQUENT CONSISTING OF ONLY ATOMIC FORMULAS:

A SEQUENT CONSISTING OF ONLY ATOMIC FORMULAS IS VALID IF
THE TWO LISTS OF FORMULAS HAVE A FORMULA IN COMMON.

BY REPEATED APPLICATION OF THE RULES, ONE IS LED TO A SET OF
SEQUENTS CONSISTING OF ATOMIC FORMULAS. IF EACH ONE OF THESE
SEQUENTS IS VALID THEN SO IS THE ORIGINAL FORMULA.

UNOP = 'NOT' 1
BINOP = 'AND' | 'IMP' | 'OR' | 'EQU' 2
FORMULA = ' ' UNOP . OP '(' BAL . PHI ') ' | 3
        ' ' BINOP . OP '(' BAL . PHI ',' BAL . PSI ')' 3
ATOM = ( ' ' BAL ' ' ) . A 4

DEFINE ('WANG (ANTE, CONSEQ) PHI, PSI') 5

READ EXP = TRIM (INPUT) : P (END) 6
       OUTPUT = 7
       OUTPUT = 'FORMULA: ' EXP 8
       OUTPUT = 9

WANG (' ' EXP) : P (INVALID) 10
       OUTPUT = 'VALID' : (READ) 11
INVALID OUTPUT = 'NOT VALID' : (READ) 12
The SNOBOL4 Programming Language

S4D4a

November 22, 1967

* WANG OUTPUT = ANTE ' --- ' CONSEQ
ANTE FORMULA = S('A' OP )
CONSEQ FORMULA = S('C' OP )
ANTE = ANTE 
CONSEQ = ' ' CONSEQ 
TEST ANTE ATOM = ' ' CONSEQ A
ANTE CONSEQ : F (FRETURN)
ANTE : S (RETURN) F (TEST)

* ANOT WANG (ANTE, CONSEQ ' ' PHI)
* AAND WANG (ANTE ' ' PHI ' ' PSI, CONSEQ)
* AOR WANG (ANTE ' ' PHI, CONSEQ)
WANG (ANTE ' ' PSI, CONSEQ)

* AIMP WANG (ANTE ' ' PSI, CONSEQ)
WANG (ANTE, CONSEQ ' ' PHI)
* AEQU WANG (ANTE ' ' PHI ' ' PSI, CONSEQ)
WANG (ANTE, CONSEQ ' ' PHI ' ' PSI)
* CNOT WANG (ANTE ' ' PHI, CONSEQ)
* CAND WANG (ANTE, CONSEQ ' ' PHI)
WANG (ANTE, CONSEQ ' ' PSI)
* COR WANG (ANTE, CONSEQ ' ' PHI ' ' PSI)
* CIMP WANG (ANTE ' ' PHI, CONSEQ ' ' PSI)
* CEQU WANG (ANTE ' ' PHI, CONSEQ ' ' PSI)
WANG (ANTE ' ' PSI, CONSEQ ' ' PHI)
END

SUCCESSFUL COMPILATION
The SNOBOL4 Programming Language

November 22, 1967

FORMULA: IMP (NOT (OR (P, Q)) , NOT (P))

---* IMP (NOT (OR (P, Q)) , NOT (P))
NOT (OR (P, Q)) ---* NOT (P)
---* NOT (P) OR (P, Q)
P ---* OR (P, Q)
P ---* P Q
VALID

FORMULA: NOT (IMP (NOT (OR (P, Q)) , NOT (P)))

---* NOT (IMP (NOT (OR (P, Q)) , NOT (P)))
IMP (NOT (OR (P, Q)) , NOT (P)) ---*
NOT (P) ---*
---* P
NOT VALID

FORMULA: IMP (AND (NOT (P)), NOT (Q)), EQU (P, Q) ((\neg P \land \neg Q) \rightarrow (P \leftrightarrow Q))

---* IMP (AND (NOT (P)), NOT (Q)), EQU (P, Q)
AND (NOT (P)), NOT (Q)) ---* EQU (P, Q)
NOT (P) NOT (Q) ---* EQU (P, Q)
NOT (Q) ---* EQU (P, Q) P
---* EQU (P, Q) P Q
P ---* P Q Q
Q ---* P Q P
VALID

FORMULA: IMP (IMP (OR (P, Q), OR (P, R)), OR (P, IMP (Q, R))) ((P \lor Q) \rightarrow (P \lor R)) \rightarrow (P \lor (Q \lor R))

---* IMP (IMP (OR (P, Q), OR (P, R)), OR (P, IMP (Q, R)))
IMP (OR (P, Q), OR (P, R)) ---* OR (P, IMP (Q, R))
OR (P, R) ---* OR (P, IMP (Q, R))
P ---* OR (P, IMP (Q, R))
P ---* P IMP (Q, R)
P Q ---* P R
R ---* OR (P, IMP (Q, R))
R ---* P IMP (Q, R)
R Q ---* R P
---* OR (P, IMP (Q, R)) OR (P, Q)
---* OR (P, Q) P IMP (Q, R)
---* P IMP (Q, R) P Q
Q ---* P P Q R
VALID
SAMPLE PROGRAM 9: MAGIC SQUARE GENERATION

**SNOBOL4 (PRELIMINARY VERSION, 9.18.67)**

* THIS PROGRAM GENERATES A MAGIC SQUARE OF ODD ORDER. THE SIZE
  OF THE SQUARE IS READ IN AS N. FOR DETAILS OF THE ALGORITHM SEE
  JACM, AUGUST 1962, ALGORITHM 118.

* READ IN SIZE OF SQUARE AND DEFINE ARRAY.

```
N = TRIM(INPUT)
MAGIC = ARRAY(N * N)
```

* I AND J ARE ROW AND COLUMN COORDINATES. K IS THE NUMBER CURR-
  ENTLY BEING PLACED INTO THE SQUARE. LIM IS THE UPPER BOUND
  ON K.

```
I = (N + 1) / 2
J = N
LIM = N * N
K = 1
```

* MAIN PROGRAM LOOP WHICH MAKES A SINGLE ENTRY INTO THE ARRAY.

```
KLOOP IDENT (MAGIC[I,J], NULL)
I = I - 1
J = J - 2
I = LE(I, 0) I + N
J = LE(J, 0) J + N
ASSIGN MAGIC[I,J] = K
I = I + 1
I = GT(I, N) -I - N
J = J + 1
J = GT(J, N) J - N
K = LT(K, LIM) K + 1
```

* OUTPUT ROUTINE.

```
OUTPUT = 'MAGIC SQUARE OF SIZE '
OUTPUT =
I = 1
OUT
ROW =
OUT1 J = 1
OUT2 ' ' LEN(SIZE(MAGIC[I,J])) RTAB(0) . . . REST
```

78
The SNOBOL4 Programming Language

November 22, 1967

ROW = ROW REST MAGIC<1,J>
J = LT(J,N) J + 1
OUTPUT =
OUTPUT = ROW
I = LT(I,N) I + 1
END

SUCCESSFUL COMPILATION

Appendix C 79
MAGIC SQUARE OF SIZE 5

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>10</td>
<td>4</td>
<td>23</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>6</td>
<td>5</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>13</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>21</td>
<td>20</td>
<td>14</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>22</td>
<td>16</td>
<td>15</td>
</tr>
</tbody>
</table>
SAMPLE PROGRAM 10: REGULAR EXPRESSION RECOGNITION

SNOBOL4 (PRELIMINARY VERSION, 9.18.67)

THE FOLLOWING PROGRAM DETERMINES WHETHER A GIVEN STRING IS A MEMBER OF A SPECIFIED REGULAR SET OF STRINGS. SINCE LAMBDA IS UNAVAILABLE TO DENOTE THE NULL STRING, () CAN BE USED.

THE PROGRAM BEGINS BY DEFINING FOUR PATTERNS: SUM, ARBNO1, ARBNO2, AND TERM. AT NEXTSET, A REGULAR EXPRESSION IS READ IN AND PRINTED. NEXT, KLEENE IS CALLED TO CONVERT THE REGULAR EXPRESSION INTO A SNOBOL4 PATTERN USING THE FOUR PREVIOUSLY CONSTRUCTED PATTERNS. FINALLY, THE PATTERN RETURNED AS THE VALUE OF KLEENE IS USED TO TEST IF SENTENCES ARE VALID.

$ANCHOR = 'ON'
DEFINE('KLEENE(SPEC)EXP1,EXP2,RUN,REP')

DEFINE BASIC PATTERNS USED IN THE KLEENE FUNCTION

SUM = BAL . EXP1 ' V ' RTAB(0) . EXP2
ARBNO1 = ARB . RUN (LEN(1) . REP ' ' | ' (' ABORT)
ARBNO2 = ARB . RUN '{ ' FENCE BAL . REP ' ')
TERM = ARB . RUN '{ ' FENCE BAL . REP ' ')

READ IN THE SPECIFICATION OF THE REGULAR SET.

NEXTSET SETSPEC = TRIM(INPUT):F(END)
OUTPUT = ' SET: ' SETSPEC

CONSTRUCT A PATTERN CORRESPONDING TO THE SET.

SET = KLEENE SETSPEC RPOS(0)

READ IN STRINGS AND TEST THEM.

TEST TEST = TRIM(INPUT):F(END)
IDENT (TEST) : S(NEXTSET)
TEST SET : F(FAIL)
OUTPUT = TEST ' IS A MEMBER.' : (TEST)
FAIL OUTPUT = TEST ' IS NOT A MEMBER.' : (TEST)

KLEENE SPEC SUM : F(KI)

November 22, 1967
The SNOBOL4 Programming Language

SNOBOL4a

KLEENE = KLEENE(EXP1) | KLEENE(EXP2) : (RETURN) 17
K1 SPEC ARBNO1 = 18
KLEENE = KLEENE RUN ARBNO(REP) : (K1) 19
K2 SPEC ARBNO2 = 20
KLEENE = KLEENE RUN ARBNO(KLEENE(REP)) : (K1) 21
K3 SPEC TERM = 22
KLEENE = KLEENE RUN KLEENE(REP) : (K1) 23
K4 SPEC = IDENT(SPEC,'()') 24
KLEENE = KLEENE SPEC : (RETURN) 25
END

SUCCESSFUL COMPILATION
SET: (0 1 V 10) (00 V 11) *01
0101 IS A MEMBER.
1001 IS A MEMBER.
01000001 IS A MEMBER.
00001101 IS A MEMBER.
101001 IS NOT A MEMBER.
0001 IS NOT A MEMBER.
011100 IS NOT A MEMBER.

SET: 1*(0 V 000) (10 V 11)*00 V 0101(111)*0
1000 IS A MEMBER.
00000 IS A MEMBER.
1010111000 IS A MEMBER.
01010 IS A MEMBER.
0101111110 IS A MEMBER.
0101111110 IS NOT A MEMBER.
01101 IS NOT A MEMBER.
01100 IS NOT A MEMBER.

SET: (()) V 11 (()) V 00) 1
1 IS A MEMBER.
111 IS A MEMBER.
001 IS A MEMBER.
11001 IS A MEMBER.
101 IS NOT A MEMBER.

SET: (()) V 11 *1
1 IS A MEMBER.
111 IS A MEMBER.
1111 IS A MEMBER.
10 IS NOT A MEMBER.
111111 IS NOT A MEMBER.
SAMPLE PROGRAM 11: PHRASE STRUCTURE GRAMMAR RECOGNITION

SNOBOL4 (PRELIMINARY VERSION, 9.18.67)

* * *
THIS PROGRAM RECOGNIZES CONTEXT FREE PHRASE STRUCTURE GRAMMARS
* * *
GRAMMARS ARE SPECIFIED AS SHOWN IN THE OUTPUT. A CARD
WITH 'EOF' TERMINATES THE GRAMMAR. THE NEXT INPUT SPECIFIES THE
SYNTACTIC TYPE WHICH IS TO BE RECOGNIZED. NEXT ARE THE SENTEN­
CES TO BE TESTED, ONE TO A CARD. IF MORE GRAMMARS ARE TO BE
TRIED, AN 'EOF' SEPARATES THE SENTENCES FROM THE NEXT GRAMMAR.
* * *
DEFINE('XLATE(S)TYPE','XLATE')
* *
DEFINITION OF CONSTANTS AND BASIC PATTERNS
*
RP = ')'
LP = '('
SL = '/'
ALTPAT = ARB . LIT ' (' BAL . TYPE ')'
CLAUSEPAT = ARB . CLAUSE '/'
TYPEPAT = '(' BAL . TYPE ')'
*
CONSTRUCTION OF PATTERNS FOR SYNTACTIC TYPES
*
READ OUTPUT = 'GRAMMAR:
READ1 CARD = TRIM(INPUT)
IDENT(CARD,'EOF')
OUTPUT = CARD
CARD TYPEPAT =
$TYPE = FAIL
NEXTC CARD CLAUSEPAT =
$TYPE = $TYPE | XLATE(CLAUSE)
ENDC $TYPE = $TYPE | XLATE(CARD)
*
XLATE FUNCTION TO CONSTRUCT PATTERN FOR A CLAUSE
*
XL1 S ALTPAT =
DIFFER(LIT)
XLATE = XLATE *$TYPE
XL2 XLATE = XLATE LIT *$TYPE
XL3 IDENT(S)
XLATE = XLATE S
RECOGNIZER TO READ AND TEST SENTENCES

RECOG TYPE = TRIM(INPUT)
OUTPUT = F(Err)
OUTPUT = 'TEST FOR SENTENCES OF TYPE' TYPE
OUTPUT = 28
PAT = POS(0) $TYPE RPOS(0)
RCG1 CARD = TRIM(INPUT)
IDENT(CARD,'EOF')
CARD PAT = S(RCG4)
RCG2 OUTPUT = CARD 'IS OF TYPE' TYPE
RCG3 OUTPUT = CARD 'IS NOT OF TYPE' TYPE
RCG4 OUTPUT =
END

SUCCESSFUL COMPILATION
The SNOBOL4 Programming Language

S4D4a

November 22, 1967

GRAMMAR:
(A) = (B) / (C)
(C) = A / A (C)
(B) = A / A (C)

TEST FOR SENTENCES OF TYPE A
A IS OF TYPE A
ABA IS OF TYPE A
ABAA IS OF TYPE A
ABAAA IS OF TYPE A
ABBA IS NOT OF TYPE A
AB IS NOT OF TYPE A

GRAMMAR:
(S) = (S) A / C
(T) = (S) T

TEST FOR SENTENCES OF TYPE T
CT IS OF TYPE T
CAT IS OF TYPE T
CAAT IS OF TYPE T
CAAAAT IS OF TYPE T
CCT IS NOT OF TYPE T
CATT IS NOT OF TYPE T

GRAMMAR:
(IDENT) = X / Y / Z
(AREX) = (ADOP) (TERM) / (TERM) / (AREX) (ADOP) (TERM)
(TERM) = (FACTOR) / (TERM) (MULOP) (FACTOR)
(FACTOR) = (IDENT) / (LP) (AREX) (RP)
(ADOP) = + / -
(MULOP) = * / (SL)

TEST FOR SENTENCES OF TYPE AREX
X*Y*(Z+X) IS OF TYPE AREX
X+Y+Z IS OF TYPE AREX
XY (Z+X) IS NOT OF TYPE AREX

86
SAMPLE PROGRAM 12: TREE FUNCTIONS

SNOBOL4 (PRELIMINARY VERSION, 9.18.67)

This program is designed to illustrate the use of programmer-defined data types to add tree functions to the SNOBOL4 language. The functions given here are a subset of those described in 'Tree Functions for SNOBOL4'. The functions included were chosen to indicate how typical functions might be programmed, and not all are used in the example given.

This program consists of three parts:

1. The definition and coding of the tree functions.
2. The coding of a function to convert algebraic expression into trees.
3. A test program which converts sample algebraic expressions into canonical form.

PART 1: 6ANCHOR = 'ON'

Part 1: Definition of a node and the coding of tree functions which manipulate nodes to form trees and operate on these trees.

(The value of a node may be set by use of the 'value' field defined on a node, rather than by using indirectness as required by the SNOBOL3 tree functions.)

DATA('NODE(VALUE,L,R,F)')

DEFINE('ADDIB(NODE1, NODE2)')
DEFINE('ADDSO(NODE1, NODE2)')
DEFINE('FATHER(NODE)')
DEFINE('LCTR(NODE)')
DEFINE('LSIB(NODE)')
DEFINE('LSON(NODE)')
DEFINE('LTREE(CTR, FATHER) Y, Z')
DEFINE('MNODE(NODE)')
DEFINE('OHUNT(NODE, STRING)')
DEFINE('PRUNE(NODE X)')
DEFINE('RSIB(NODE)')
DEFINE('TCOPY(NODE, FATHER)')
The SNOBOL4 Programming Language

ADDsIB PRUNE (NODE1)
R (NODE1) = R (NODE2)
R (NODE2) = NODE1
F (NODE1) = F (NODE2)

ADDSON PRUNE (NODE1)
R (NODE1) = L (NODE2)
F (NODE1) = NODE2
L (NODE2) = NODE1

FATHER FATHER = F (NODE)
DIFFER (FATHER)

LCTR LCTR = VALUE (NODE)
LCTR = LCTR ' ( LCTR (LSON (NODE)) ' ) ' 
LCTR = LCTR ' , , ' LCTR (RSIB (NODE))

LSIB LSIB = L (FATHER (NODE))
IDENT (LSIB, NODE)
LSIB1 IDENT (R (LSIB), NODE)
LSIB = R (LSIB)

LSON LSON = L (NODE)
DIFFER (LSON)

LTREE CTR BAL = CTR ' ( ' BAL . Z ' ) ' 
CTR ARB = CTR ' ( ' BAL . Z ' ) ' 
LTREE = NODE (CTR, , FATHER)
L (LTREE) = LTREE (DIFFER (Z, Y), LTREE)
R (LTREE) = LTREE (DIFFER (Y, Z), FATHER)

NXNODE NXNODE = LSON (NODE)
NX1 NXNODE = RSIB (NODE)
NODE = FATHER (NODE)

OHUNT OHUNT = LSON (NODE)
OHUNT1 IDENT (VALUE (OHUNT), STRING)
OHUNT = RSIB (OHUNT)

PRUNE X = LSIB (NODE)
L (FATHER (NODE)) = R (NODE)
PRUNE2 F (NODE) =
R (NODE) =
PRUNE1 R (X) = R (NODE)
**PART 2:** FUNCTION which converts fully-parenthesized algebraic expressions into trees. Such a tree representation is useful in code optimization. The restriction that the expressions be fully parenthesized was included to simplify the program since the purpose was to illustrate the tree functions.

```
OP = '+', '-', '*', '/', '\n DEFINE('TREE(\text{EXP}) T1, T2, E1, E2, 0') : (PART3)
```

```
TREE EXP '(' BAL EXP ')' RPOS(0) : S(TREE) EXP '(' BAL E1 ')' BAL E1 OP E2 RTAB(0) E2 : F(TREE1)
```

```
TREE = LTREE(0) T1 = TREE(E1) T2 = TREE(E2) ADDSON(T1, TREE) ADDSIB(T2, T1) : (RETURN)
```

```
TREE1 TREE = LTREE(\text{EXP}) : (RETURN)
```

**PART 3:** TEST program which reads in algebraic expressions, converts them into trees, and prints the result in canonical form.

```
TEST EXP = TRIM(INPUT) : F(END) OUTPUT = 'EXPRESSION: ' EXP T = TREE(\text{EXP}) OUTPUT = 'CANONICAL FORM: ' LCTR(T) : (TEST)
```

**END**

SUCCESSFUL COMPILATION
The SNOBOL4 Programming Language

S4D4a

November 22, 1967

EXPRESSION: \[\frac{(A+B)\cdot C}{(D-E)\cdot F}\]
CANONICAL FORM: \[\frac{\cdot (\cdot (A, B), C), \cdot (-(D, E), F)}{\cdot (\cdot (A, B), C), \cdot (-(D, E), F)}\]

EXPRESSION: \[A^{\cdot 2} + (C\cdot (D^{\cdot 3}))\]
CANONICAL FORM: \[\cdot (\cdot (A, 2), \cdot (C, \cdot (D, 3)))\]

EXPRESSION: \[(\text{COUNT} + (\text{TOTAL} - \text{DELTA})) \cdot \text{SCALE}\]
CANONICAL FORM: \[\cdot (\cdot (\text{COUNT}, -(\text{TOTAL}, \text{DELTA})), \cdot \text{SCALE})\]