-- --

S4D58

Implementing SNOBOL4 in SIL; Version 3.11

Ralph E. Griswold

Department of Computer Science
The University of Arizona
Tucson, Arizona 85721

February 1981

Implementing SNOBOL4 in SIL; Version 3.11

Ralph E. Griswold

Department of Computer Science
The University of Arizona
Tucson, Arizona 85721

1. Introduction

The SNOBOL4 programming language is implemented in macro-assembly language called SIL (SNOBOL4 Implementation Language). This macro language is largely machine-independent and is designed so that it can be implemented on a variety of computers. Thus, an implementation of the SNOBOL4 programming language can be obtained by implementing the much simpler macro language. By implementing the macro language, and using the SNOBOL4 system already written in the macro language, one obtains a version of SNOBOL4 that is largely source-language compatible with other versions implemented in the same way. Nearly all the logic of the SNOBOL4 language resides in the program written in the macro language. Thus if the macro language is implemented properly, the resulting implementation of SNOBOL4 is essentially the same as other such implementations.

This paper describes the macro language and contains information necessary for its implementation. Information given here related to Version 3.11 of the SIL source, although it applies equally well to any modification of the basic Version 3. Section 2 describes environmental considerations. Section 3 describes the representation of data. Syntax tables and character graphics are described in Section 4. Section 5 explains the method used to describe the macro operations. Section 6 is a list of all macro operations with a description of how to implement each one. Section 7 contains miscellaneous implementation notes. Supplementary information, including a list of other documentation, is given in appendices.

2. Environmental Considerations

2.1. Input and Output

SNOBOL4 is designed to perform all input and output through FORTRAN IV routines. A SNOBOL4 object program has much the same I/O facilities as a FORTRAN IV object program. Specification of I/O is thus largely machine-independent both at the source-language level and at the implementation level.

Files are referred to by their FORTRAN unit reference numbers. In SNOBOL4 unit reference numbers are integers that appear in data that is given in arguments to macros that perform input and output. Unit reference numbers are referred to symbolically in the SNOBOL4 assembly. See the `PARMS` file in the discussion of the `COPY` macro.

Input, performed by `STREAD`, uses only A conversion, with lengths being specified. Output is controlled by formats. Output is performed by `OUTPUT` and `STPRNT`. The output done by the SNOBOL4 system specifies H-type literals, A, I, and, in one case, F conversion. Programmer formats should only literals, X, T, and A conversion. Generally speaking, formats occur in ‘‘‘undigested’’’ form. Formats used by `OUTPUT` are assembled by the `FORMAT` macro and are intended to be simply character strings representing undigested formats. `FORMAT` may, however, assemble any convenient representation of the format. Formats used by `STPRNT` are strings that may be formed during program execution and hence must be accepted in their undigested form.

There are three other I/O related operations that correspond to their FORTRAN counterparts. These are `BKSPCE`, `ENFILE`, and `REWIND`.

The easiest way to implement SNOBOL4 I/O is to use FORTRAN calling sequences for corresponding operations and link the FORTRAN I/O library with the SNOBOL4 system. The main difficulties usually occur in handling undigested formats. When questions arise as to what an operation should do, FORTRAN conventions should be applied. A programmer should get the same results from SNOBOL4 as from FORTRAN if, for example, a string of 200 characters is requested from a file containing 80-character records.

2.2. Storage Requirements

The SNOBOL4 system itself is very large and SNOBOL4 programs typically require large amounts of dynamically allocated storage. The magnitude of these requirements may be determined from the implementation for the IBM System/360. This system requires a user partition of about 200K bytes (characters) to run large programs. A partition of about 170K bytes permits execution of small programs. Of the space required, the SNOBOL4 system and its internal data consume about 100K bytes, the FORTRAN I/O routines consume about 14K bytes, and the remainder is devoted to dynamically allocated storage. Allocated storage is referred to in machine-independent data units (see the next section) called descriptors that occupy 8 bytes each on the IBM System/360. A production system should be able to provide about 10,000 descriptors of dynamically allocated storage. Because of the large amount of space required for dynamic storage, overlay techniques for the program itself can only partially reduce the requirements for physical storage. Virtual memory systems may display poor performance if SNOBOL4 is run with inadequate amounts of physical storage.

2.3. Other Considerations

SNOBOL4 makes few other demands on its operating system environment. Facilities should be provided so that the SNOBOL4 system can be called and can return to the operating system under which it operates. SNOBOL4 uses dump facilities to provide core dumps requested by the keyword `&ABEND` if such facilities are available. Time and date information is used by SNOBOL4, but it is not essential.

3. Representation of Data

There are a few basic types of data used in the SNOBOL4 system, and a number of aggregates of the basic types. The basic types of data are:

descriptors
specifiers
character strings
syntax table entries

3.1. Descriptors

Descriptors are used to represent all pointers, integers, and real numbers. A descriptor may be thought of as the basic ‘‘‘word’’’ of SNOBOL4. Descriptors consist of three fixed-length fields:

address
flag
value

The size and position of these fields is determined from the data they must represent and the way that they are used in the various operations. The following paragraphs describe some specific requirements.

3.1.1. Address Field

The address field of a descriptor must be large enough to address any descriptor, specifier, or program instruction within the SNOBOL4 system. (Descriptors do not have to address individual characters of strings. See Section 3.2.) The address field must also be large enough to contain any integer or real number (including sign) that is to be used in a SNOBOL4 program. The address field is the most frequently used field of a descriptor and is used frequently for addressing and integer arithmetic and it should be positioned so that these operations can be performed efficiently.

3.1.2. Flag Field

The flag field is used to represent the states of a number of disjoint conditions and is treated as a set of bits that are individually tested, turned on, and turned off. Five flag bits used in SNOBOL4.

3.1.3. Value Field

The value field is used to represent a number of internal quantities that are represented as unsigned integers (magnitudes). These quantities the encoded representation of source-language data types, the length of strings, and the size (in address units) of various data aggregates. The value field need not be as large as the address field, but it must be large enough to represent the size of the largest data aggregate that can be formed.

On the IBM System/360, a descriptor is two words (eight bytes). The first word is the address field. The second word consists of one byte for the flag field and three bytes for the value field. The three bytes (24 bits) for the value field permits representation of data objects as large as 2^{24}-1 bytes. On the other hand, two bytes would limit objects to 2^{16}-1 bytes. Since on the IBM System/360 there are eight bytes per descriptor, 2^{16}-1 bytes would limit objects to 8191 descriptors, which would be too restrictive. For machines with fewer address units per descriptor, the value field need not be as large.

3.2. Specifiers

Specifiers are used to refer to character strings. Almost all operations performed on character strings are handled through operations on specifiers. All specifiers are the same size and have five fields:

address
flag
value
offset
length

Specifiers and descriptors may be stored in the same area indiscriminately, and are indistinguishable to many processes in the SNOBOL4 system. As a result, specifiers are composed of two descriptors. One descriptor is used in the standard way to provide the address, flag, and value fields. The other descriptor is used in a nonstandard way. Its address field is used to represent the offset of an individual character from the address given in the specifier’s address field. The value field of this other descriptor is used for the length.

3.3. Character Strings

Character strings are represented in packed format, as many characters per descriptor as possible. Storage of character strings in SNOBOL4 dynamic storage is always in storage units that are multiples of descriptors.

3.4. Syntax Table Entries

Syntax tables are necessarily somewhat machine dependent. Consequently, implementation of these tables is done individually for each machine. A description of the table requirements is given in the next section.

4. Syntax Tables and Character Graphics

4.1. Characters

The SNOBOL4 language permits the use of any character that can be represented on a particular machine. There are certain characters that have syntactic significance in the source language. The card codes, graphics, and internal representations vary from machine to machine. For each machine, representations are chosen for each of the syntactically significant characters. Such characters and sets of characters are given descriptive names to avoid dependence on a particular machine. In the list that follows, ASCII graphics are used as a point of reference.

function	*name*	*graphics*
ALPHANUMERIC	digit and letter	ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz 0123456789
AT	operator	@
BLANK	separator and operator	blank and tab
BREAK	dot and underscore	. _
CMT	comment card	*
CNT	continue card	+ .
COLON	goto designator and dimension separator	:
COMMA	argument separator	,
CTL	control card	-
DOLLAR	operator	$
DOT	operator	.
DQUOTE	literal delimiter	"
EOS	statement terminator	;
EQUAL	assignment	=
FGOSYM	failure goto designator	F
KEYSYM	operator	&
LEFTBR	reference and goto delimiter	< [
LEFTPAREN	expression delimiter	(
LETTER	letter	ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz
MINUS	operator	-
NOTSYM	operator	~
NUMBER	digit	0123456789
ORSYM	operator	|
PERCENT	operator	%
PLUS	operator	+
POUND	operator	#
QUESYM	operator	?
RAISE	operator	^
RIGHTBR	reference and goto delimiter	>]
RIGHTPAREN	expression delimiter	)
SGOSYM	success goto designator	S
SLASH	operator	/
SQUOTE	literal delimiter	'
STAR	operator	*
TERMINATOR	expression terminator	;) > ,] blank and tab

4.2. Syntax Tables

The lexical syntax of the SNOBOL4 language is analyzed using the operation STREAM (q.v.) which is driven from syntax tables. The syntax tables provide a representation of a finite state machine used during lexical analysis. See Reference 3 in Appendix B for a more detailed discussion.

In a syntax table there is an entry for each character at a position corresponding to the numerical value of the internal encoding of that character. The syntax table entry specifies the action to be taken if that character is encountered. The actions are:

1. CONTIN, indicating that the current syntax table is to be used for processing the next character.

2. GOTO(TABLE), indicating that TABLE is to be used for processing the next character.

3. `STOP`, indicating that `STREAM` should terminate with the last character examined to be included in the accepted string.

4. `STOPSH`, indicating the `STREAM` should terminate with the last character examined *not* to be included in the string accepted.

5. `ERROR`, indicating that `STREAM` should terminate with an error indication.

6. `PUT(ADDRESS)`, indicating that `ADDRESS` is to be placed in the address field of the descriptor `STYPE`.

The classes of characters for which actions are to be taken are given in `FOR` designations. `CONTIN` and `GOTO(TABLE)` provide information about the next table to use and are typically represented by addresses in syntax table entries. `STOP`, `STOPSH`, and `ERROR` are type indicators used to stop the streaming process.

`SNABTB` is used in pattern matching for `ANY(CS)`, `BREAK(CS)`, `NOTANY(CS)`, and `SPAN(CS)`. `SNABTB` is modified during execution by the macros `CLERTB` and `PLUGTB` (q.v.). The other syntax tables are not modified.

Two representative syntax table descriptions follow. A complete list is given in Appendix A.

```
BEGIN IBLKTB
FOR(BLANK) GOTO(FRWDTB)
FOR(EOS) PUT(EOSTYP) STOP
ELSE ERROR
END IBLKTB

BEGIN VARBTB
FOR(ALPHANUMERIC,BREAK) CONTIN
FOR(LEFTPAREN) PUT(LPTYP) STOPSH
FOR(COMMA) PUT(CMATYP) STOPSH
FOR(RIGHTPAREN) PUT(RPTYP) STOPSH
ELSE ERROR
END VARBTB
```

The syntax tables for the IBM System/360 implementation are generated from such descriptions using a (SNOBOL4) program in which the character classes and the order of the internal character codes are parameters. The use of some kind of automatic technique to generate the syntax tables is advisable, both to ensure accuracy and because of the large amount of data involved.

5. Describing the Macros

This section explains the method of describing the macros. The instructions for implementing an operation usually consist of a description of the operation's function, figures indicating data relating to the operation, and programming notes that contain details and references to other relevant information. The figures consist of stylized representations of the various data objects and the fields within them.

5.1. Diagrammatic Representation of Data

The representation of a descriptor at `LOC1` is shown below. `A`, `F`, and `V` indicate the values of the address, flag, and value fields.

LOC1	A	F	V

The representation of a specifier at `LOC2` is shown below. `A`, `F`, `V`, `O`, and `L` indicate the values of the address, flag, value, offset, and length fields.

LOC2	A	F	V	O	L

Character strings have two representations depending on how many characters are relevant to the description. The short representation of a string of `L` characters is shown below. `C1` and `CL` are the first and last characters, respectively. In this representation, the intermediate characters are indicated by dots.

LOC3	C1	...	CL

The long representation of a string of `L` characters at `LOC4` is shown below. `CJ` and `CJ+1` are relevant characters in the interior of the string. The long representation is used when such interior characters must be specified.

LOC4	C1	...	CJ	CJ+1	...	CL

The representation of a syntax table entry is shown below. `A`, `T`, and `P` indicate values of the next table address, type indicator, and put field as specified by the `PUT` action.

LOC5	A	T	P

Various values and expressions may occur in the fields of data objects. Fields are left blank when their value is not used in an operation. In data objects that are changed by an operation, unchanged fields are left blank. For example, if the figure below referred to a descriptor to be changed, the new value of the address field would be `A2`, and no other fields would be changed.

A2		

Letters are used as abbreviations to differentiate the values that may appear in a field. The seven basic fields are indicated by the letters `A`, `F`, `V`, `O`, `L`, `T`, and `P`. Numerical suffixes (which may be thought of as subscripts) are used as necessary to distinguish between values of the same type. Thus, for example, `A1`, `A32`, and `AN` might be used to refer to addresses, `F1` and `F2` to flags, and so on. To make further distinctions where appropriate, `I` and `R` are used to indicate integers and real numbers, respectively.

5.2. Branch Points

Program labels are included in the argument lists of many macros. These addresses are points to which control may be transferred, depending on data supplied to the macros. In general, some or all of the branch points may be omitted in a macro call. An omitted branch point signifies that control is to pass to the next macro in line if the condition corresponding to the omitted branch point is satisfied. For example `ACOMP` is called in the following forms:

```
ACOMP DESCR1,DESCR2,GTLOC,EQLOC,LTLOC
ACOMP DESCR1,DESCR2,GTLOC,EQLOC
ACOMP DESCR1,DESCR2,GTLOC
ACOMP DESCR1,DESCR2,GTLOC,,LTLOC
ACOMP DESCR1,DESCR2,,EQLOC,LTLOC
ACOMP DESCR1,DESCR2,,EQLOC
ACOMP DESCR1,DESCR2,,,LTLOC
```

where `GTLOC`, `EQLOC`, and `LTLOC` are addresses to which `ACOMP` may branch. `ACOMP` is not called with all three branch points omitted, since that is not a meaningful operation. Other macros such as `SUM` (q.v.) are often called with all branch points omitted.

Implementation of the macros must take omission of branch points into consideration. Alternate expansions, conditioned by the omission of branch points, may be used to generate more efficient code.

5.3. Abbreviations

Several abbreviations are used in the descriptions that follow. These are:

1. `D` is used for the addressing width of a descriptor. On the IBM System/360, the machine addressing unit is one byte, and `D` is eight.

2. `S` is used for the addressing width of a specifier; `S = 2D`.

3. `CPD` is used for the number of characters stored per descriptor.

4. `I` is used for (signed) integers.

5. `R` is used for real numbers.

6. `E` is used for the address width of a syntax table entry.

7. `Z` is used to indicate the number of the last character in collating sequence. Characters are ‘‘‘numbered’’’ from 0 to `Z`.

The data type codes `I` and `R` are defined in the SIL source program. The other codes are machine dependent. See the `COPY` macro. by `R` and `I` respectively. These symbols are defined in

5.4. Programming Notes

Programming notes are provided for some macro operations. The notes are intended to point out special cases, indicate implementation pitfalls, and to provide information about conditions that can be used to improve the efficiency of the implementation.

6. The Macros

---

1. `ACOMP` (address comparison)

```
              ACOMP       DESCR1,DESCR2,GTLOC,EQLOC,LTLOC
```

`ACOMP` is used to compare the address fields of two descriptors. The comparison is arithmetic with `A1` and `A2` being considered as signed integers. If `A1 > A2`, transfer is to `GTLOC`. If `A1 = A2`, transfer is to `EQLOC`. If `A1 < A2`, transfer is to `LTLOC`.

Data Input to `ACOMP`:

DESCR1	A1		
DESCR2	A2		

Programming Notes:

1. A1 and A2 may be relocatable addresses.

2. See also LCOMP, ACOMPC, AEQL, AEQLC, and AEQLIC.

---

2. ACOMPC **(address comparison with constant)**

```
                ACOMPC       DESCR,N,GTLOC,EQLOC,LTLOC
```

ACOMPC is used to compare the address field of a descriptor to a constant. The comparison is arithmetic with A being considered as a signed integer. If A > N, transfer is to GTLOC. If A = N, transfer is to EQLOC. If A < N, transfer is to LTLOC.

Data Input to ACOMPC:

DESCR	A		

Programming Notes:

1. A may be a relocatable address.

2. N is never negative.

3. N is often 0.

4. See also ACOMP, AEQL, AEQLC, and AEQLIC.

---

3. ADDLG **(add to specifier length)**

```
                ADDLG        SPEC,DESCR
```

ADDLG is used to add an integer to the length of a specifier.

Data Input to ADDLG:

SPEC					L
DESCR	I				

Data Altered by ADDLG:

SPEC					L+I

Programming Notes:

1. `I` is always positive.

---

4. `ADDSIB` (add sibling to tree node)

```
            ADDSIB      DESCR1,DESCR2
```

`ADDSIB` is used to add a tree node as a sibling to another node.

Data Input to `ADDSIB`:

DESCR1	A1		
DESCR2	A2	F2	V2
A1+FATHER	A3	F3	V3
A1+RSIB	A4	F4	V4
A3+CODE			I

Data Altered by `ADDSIB`:

A2+RSIB	A4	F4	V4
A2+FATHER	A3	F3	V3
A1+RSIB	A2	F2	V2
A3+CODE			I+1

Programming Notes:

1. `ADDSIB` is only used by compilation procedures.
2. `FATHER`, `RSIB`, and `CODE` are symbols defined in the source program.
3. See also `ADDSON` and `INSERT`.

5. ADDSON (add son to tree node)

```
              ADDSON      DESCR1,DESCR2
```

ADDSON is used to add a tree node as a son to another node.

Data Input to ADDSON:

DESCR1	A1	F1	V1
DESCR2	A2	F2	V2
A1+LSON	A3	F3	V3
A1+CODE			I

Data Altered by ADDSON:

A2+FATHER	A1	F1	V1
A2+RSIB	A3	F3	V3
		⋮	
A1+LSON	A2	F2	V2
A1+CODE			I+1

Programming Notes:

1. ADDSON is only used by compilation procedures.
2. FATHER, LSON,RSIB, and CODE are symbols defined in the source program.
3. See also ADDSIB and INSERT.

6. ADJUST (compute adjusted address)

```
              ADJUST      DESCR1,DESCR2,DESCR3
```

ADJUST is used to adjust the address field of a descriptor.

Data Input to ADJUST:

DESCR2	A2		
DESCR3	A3		
A2	A4		

Data Altered by ADJUST:

DESCR1	A3+A4		

Programming Notes:

1. A3 is always an address integer.

---

7. ADREAL (add real numbers)

```
              ADREAL       DESCR1,DESCR2,DESCR3,FLOC,SLOC
```

ADREAL is used to add two real numbers. If the result is out of the range available for real numbers, transfer is to FLOC. Otherwise transfer is to SLOC.

Data Input to ADREAL:

DESCR2	R2	F2	V2
DESCR3	R3		

Data Altered by ADREAL:

DESCR1	R2+R3	F2	V2

Programming Notes:

1. See also DVREAL, EXREAL, MNREAL, MPREAL, and SBREAL.

---

8. AEQL (addresses equal test)

```
              AEQL         DESCR1,DESCR2,NELOC,EQLOC
```

AEQL is used to compare the address fields of two descriptors. The comparison is arithmetic with A1 and A2 being considered as signed integers: If A1 = A2, transfer is to EQLOC. Otherwise transfer is to NELOC.

Data Input to AEQL:

DESCR1	A1		
DESCR2	A2		

Programming Notes:

1. A1 and A2 may be relocatable addresses.

2. See also VEQL, AEQLC, LEQLC, AEQLIC, ACOMP, and ACOMPC.

---

9. AEQLC **(address equal to constant test)**

```
            AEQLC      DESCR,N,NELOC,EQLOC
```

AEQLC is used to compare the address field of a descriptor to a constant. The comparison is arithmetic with A being considered as a signed integer. If A = N, transfer is to EQLOC. Otherwise transfer is to NELOC.

Data Input to AEQLC:

DESCR	A		

Programming Notes:

1. A may be a relocatable address.

2. N is never negative.

3. N is often 0.

4. See also LEQLC, AEQL, AEQLIC, ACOMP, and ACOMPC.

---

10. AEQLIC **(address equal to constant indirect test)**

```
            AEQLIC     DESCR,N1,N2,NELOC,EQLOC
```

AEQLIC is used to compare an indirectly specified address field of a descriptor to a constant. The comparison is arithmetic with A1 being considered as a signed integer. If A2 = N2, transfer is to EQLOC. Otherwise transfer is to NELOC.

Data Input to `AEQLIC`:

DESCR	A1		
A1+N1	A2		

Programming Notes:

1. `A2` may be a relocatable address.
2. `N2` is never negative.
3. `N1` is always zero.
4. See also `AEQL`, `AEQLC`, `LEQLC`, `ACOMP`, and `ACOMPC`.

---

11. `APDSP` (append specifier)

```
                APDSP         SPEC1,SPEC2
```

`APDSP` is used to append one specified string to another specified string.

Data Input to `APDSP`:

SPEC1	A1			O1	L1
SPEC2	A2			O2	L2
A1+O1	C11	...	C1L1		
A2+O2	C21	...	C2L2		

Data Altered by `APDSP`:

SPEC1	A1			O1	L1+L2	
A1+O1	C11	...	C1L1	C21	...	C2L2

Programming Notes:

1. If `L1 = 0`, `C21` is placed at `A1+O1`.
2. The storage following `C1L1` is always adequate for `C21...C2L2`.

12. ARRAY (assemble array of descriptors)

```
L              ARRAY         N
```

ARRAY is used to assemble an array of descriptors.

Data Assembled by ARRAY:

L	0	0	0
	.	.	.
L+(N-1)*D	0	0	0

Programming Notes:

1. All fields of all descriptors assembled by ARRAY *must* be zero when program execution begins.

13. BKSIZE (get block size)

```
               BKSIZE       DESCR1,DESCR2
```

BKSIZE is used to determine the amount of storage occupied by a block or string structure. The flag field of the descriptor at A distinguishes between string structures and blocks. If F contains the flag STTL, then

```
F(V)=D*(4+[(V-1)/CPD+1])
```

where [V] is the integer part of V and CPD is the number of characters stored per descriptor. The constant 4 occurs because there are 4 descriptors (including the title) in a string structure in addition to the string itself. The expression in brackets represents the number of descriptors required for a string of V characters. If F does not contain the flag STTL, then F(V) = V+D.

Data Input to BKSIZE:

DESCR2	A		
A		F	V

Data Altered by BKSIZE:

DESCR1	F(V)	0	0

Programming Notes:

1. See also GETLTH.

14. `BKSPCE` (backspace record)

```
               BKSPCE        DESCR
```

`BKSPCE` is used to back space one record on the file associated with unit reference number `I`.

Data Input to `BKSPCE`:

DESCR	I		

Programming Notes:

1. See also `ENFILE` and `REWIND`.
2. Refer to Section 2.1 for a discussion of unit reference numbers.

15. `BRANCH` (branch to program location)

```
               BRANCH        LOC,PROC
```

`BRANCH` is used to alter the flow of program control by branching to `LOC`. If `PROC` is given, it is the procedure in which `LOC` occurs. If `PROC` is omitted, `LOC` is in the current procedure.

Programming Notes:

1. See also `PROC`.

16. `BRANIC` (branch indirect with offset constant)

```
               BRANIC        DESCR,N
```

`BRANIC` is used to alter the flow of program control by branching indirectly to the operation at `LOC`.

Data Input to `BRANIC`:

DESCR	A		
A+N	LOC		

Programming Notes:

1. `N` is always zero

17. BUFFER (assemble buffer of blank characters)

```
LOC            BUFFER       N
```

BUFFER is used to assemble a string of N blank characters.

Data Assembled by BUFFER:

LOC		...	

Programming Notes:

1. All characters of the string assembled by BUFFER *must* be blank (not zero) when program execution begins.

18. CHKVAL (check value)

```
               CHKVAL       DESCR1,DESCR2,SPEC,GTLOC,EQLOC,LTLOC
```

CHKVAL is used to compare an integer to the length of a specifier plus another integer. If L+I2 > I1, transfer is to GTLOC. If L+I2 = I1, transfer is to EQLOC. If L+I2 < I1, transfer is to LTLOC.

Data Input to CHKVAL:

SPEC					L
DESCR1	I1				
DESCR2	I2				

Programming Notes:

1. I1, I2, and L are always positive integers.

2. CHKVAL is used only in pattern matching.

19. CLERTB (clear syntax table)

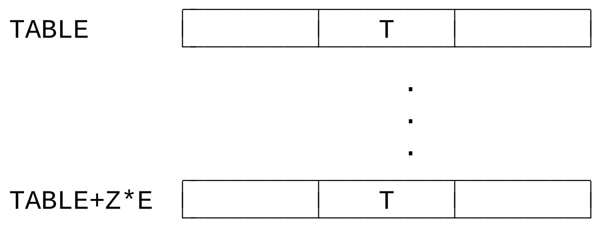
```
               CLERTB       TABLE,KEY
```

CLERTB is used to set the indicator fields of all entries of a syntax table to a constant. KEY may be one of four values:

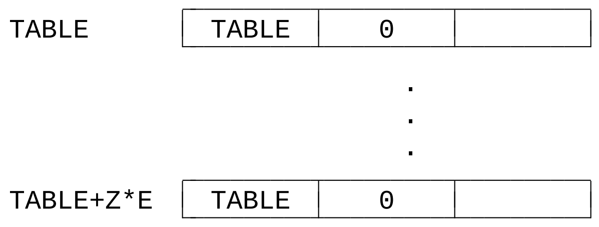
```
CONTIN
ERROR
STOP
STOPSH
```

The indicator field of each entry of `TABLE` is set to `T` where `T` is the indicator that corresponds to the value of `KEY`.

Data Altered by `CLERTB` for `ERROR`, `STOP`, or `STOPSH`:



Data Altered by `CLERTB` for `CONTIN`:



Programming Notes:

1. See Section 4.2.

2. See also `PLUGTB`.

---

20. `COPY` (copy file into assembly)

```
              COPY         FILE
```

`COPY` is used to copy a file of machine-dependent data into the source program. `COPY` occurs three times in the assembly:

```
COPY    MDATA
COPY    MLINK
COPY    PARMS
```

`MLINK` and `PARMS` are copied at the beginning of the SNOBOL4 assembly. `MDATA` is copied in the data region.

`MDATA` is a file of machine-dependent data. It contains data used in the implementation of the macros and for strings that depend on the character set of an individual machine or that represent other problems that prevent a machine-independent representation. These are:

1. `ALPHA`, a string that consists of all characters arranged in the order of their internal numerical

representation (collating sequence).

2. AMPST, a string consisting of a single ampersand, or whatever character is used to represent the keyword operator in the source language.

3. COLSTR, a string of two characters consisting of a colon followed by a blank.

4. QTSTR, a string consisting of a single quotation mark, or whatever character is used to represent a quotation mark in the source language.

These strings of characters are pointed to by the specifiers ALPHSP, AMPSP, COLSP, and QTSP respectively.

MLINK is a file of entry points and external symbol names that describe linkages used to access machine-language subroutines and I/O packages.

PARMS is a file of machine-dependent constants (equivalences). It contains constants used in the implementation of the macros and definitions of symbols. These are:

1. ALPHSZ, the number of characters in the character set for the machine. (ALPHSZ is 256 for the IBM System/360.)

2. CPA, the number of characters per machine addressing unit. (CPA is 1 for the IBM System/360, i.e., one character per byte.)

3. DESCR, the address width of a descriptor.

4. FNC, a flag used to identify function descriptors.

5. MARK, a flag used to identify descriptors that are marked titles.

6. PTR, a flag used to identify descriptors pointing into SNOBOL4 dynamic storage.

7. SIZLIM, the value of the largest integer that can be stored in the value field of a descriptor.

8. SPEC, the address width of a specifier.

9. STTL, a flag used to identify descriptors that are titles of string structures.

10. TTL, a flag used to identify descriptors that are titles of blocks.

11. UNITI, the number of the standard input unit. UNITI is 5 for the IBM System/360 implementation.

12. UNITO, the number of the standard print output unit. UNITO is 6 for the IBM System/360 implementation.

13. UNITP, the number of the standard punch output unit. UNITP is 7 for the IBM System/360 implementation.

CSTACK and OSTACK, the current end old stack pointers, respectively, should be defined in one of the COPY files. These pointers may either be in registers or in the address fields of descriptors, depending on how the stack management macros are implemented (see PUSH and RCALL, e.g.). If these pointers are implemented as registers, they should be defined in PARMS. If they are implemented in storage locations, they should be defined in MDATA.

Programming Notes:

1. COPY may be implemented in a variety of ways. COPY may, for example, simply expand into the data required, depending on the value of its argument as given above.

2. Any of the COPY segments can be used to incorporate other machine-dependent data.

---

21. CPYPAT (copy pattern)

```
            CPYPAT      DESCR1,DESCR2,DESCR3,DESCR4,DESCR5,DESCR6
```

CPYPAT is used to copy a pattern. First set

```
R1 = A1
R2 = A2
R3 = A6
```

where R1, R2, and R3 are temporary locations. Sections of the pattern are copied for successive values of R1 and R2. After copying each section, set

```
R3 = R3-(1+V7)*D
```

Then set

```
R1 = R1+(1+V7)*D
R2 = R2+(1+V7)*D
```

If R3 > 0, continue, copying the next section. Otherwise the operation is complete. The final value of R1 is inserted in the address field of DESCR1.

The functions F1 and F2 are defined as follows:

```
F1(X) = 0 if X = 0
F1(X) = X+A4 otherwise

F2(X) = A5 if X = 0
F2(X) = X+A4 otherwise
```

Initial Data Input to CPYPAT:

DESCR1	A1		
DESCR2	A2		
DESCR3	A3		
		.	
		.	
		.	
DESCR4	A4		
DESCR5	A5		
DESCR6	A6		

Data Input to CPYPAT for Successive Values of R2:

R2+D	A7	F7	V7
R2+2D	A8	0	V8
R2+3D	A9	0	V9

Data Altered by CPYPAT for Successive Values of R1:

R1+D	A7	F7	V7
R1+2D	F1(A8)	0	F2(V8)
R1+3D	A9+A3	0	V9+A3

Additional Data Input for Successive Values of R2 if V7 = 3:

R2+4D	A10	F10	V10

Additional Data Altered for Successive Values of R1 if V3 = 7:

R1+4D	A10	F10	V10

Data Altered when Copying is Complete:

DESCR1	R1		

22. DATE (get date)

	DATE	SPEC

DATE is used to obtain the current date. A character representation of the current date is placed in BUFFER.

Data Altered by DATE:

SPEC	BUFFER	0	0	0	L

BUFFER	C1	...	CL

Programming Notes:

1. The choice of representation for the date is not important so far as the source language is concerned. Thus

```
April 1, 1981
04/01/81
4:1:81
81.092
```

are all acceptable.

2. BUFFER is local to DATE and its old contents may be overwritten by a subsequent use of DATE.

3. DATE is used only in the SNOBOL4 DATE function.

4. Implementation of DATE, as such, is not essential. In this case, DATE should set the length of SPEC to zero and do nothing else.

23. DECRA (decrement address)

	DECRA	DESCR,N

DECRA is used to decrement the address field of a descriptor. A is considered as a signed integer.

Data Input to DECRA:

DESCR	A		

Data Altered by DECRA:

DESCR	A-N		

Programming Notes:

1. `A` maybe a relocatable address.

2. `N` is always positive.

3. `N` is often 1 or `D`.

4. `A-N` may be negative.

5. See also `INCRA`.

---

24. `DEQL` **(descriptor equal test)**

```
              DEQL          DESCR1,DESCR2,NELOC,EQLOC
```

`DEQL` is used to compare two descriptors. If `A1 = A2`, `F1 = F2`, and `V1 = V2`, transfer is to `EQLOC`. Otherwise transfer is to `NELOC`.

Data Input to `DEQL`:

DESCR1	A1	F1	V1
DESCR2	A2	F2	V2

Programming Notes:

1. All fields of the two descriptors must be *identical* for transfer to `EQLOC`.

---

25. `DESCR` **(assemble descriptor)**

```
LOC           DESCR         A,F,V
```

`DESCR` assembles a descriptor with specified address, flag, and value fields.

Data Assembled by `DESCR`:

LOC	A	F	V

Programming Notes:

1. Any or all of `A`, `F`, and `V` may be omitted. A zero field must be assembled when the corresponding argument is omitted.

26. DIVIDE (divide integers)

```
            DIVIDE      DESCR1,DESCR2,DESCR3,FLOC,SLOC
```

DIVIDE is used to divide one integer by another. Any remainder is discarded. That is, the result is truncated, not rounded. If I = 0, transfer is to FLOC. Otherwise transfer is to SLOC.

Data Input to DIVIDE:

DESCR2	A	F	V
DESCR3	I		

Data Altered by DIVIDE:

DESCR1	A/I	F	V

Programming Notes:

1. A may be a relocatable address.

27. DVREAL (divide real numbers)

```
            DVREAL      DESCR1,DESCR2,DESCR3,FLOC,SLOC
```

DVREAL is used to divide one real number by another. If R3 = 0 or the result is out of the range available for real numbers, transfer is to FLOC. Otherwise transfer is to SLOC.

Data Input to DVREAL:

DESCR2	R2	F2	V2
DESCR3	R3		

Data Altered by DVREAL:

DESCR1	R2/R3	F2	V2

Programming Notes:

1. In addition to use in source-language arithmetic, DVREAL is used in the computation of statistics published at the end of a SNOBOL4 run.

2. See also ADREAL, EXREAL, MNREAL, MPREAL, and SBREAL.

28. END (end assembly)

```
                END
```

END is used to terminate assembly of the SNOBOL4 system. It occurs only once and is the last card of the assembly.

29. ENDEX (end execution of SNOBOL4 run)

```
                ENDEX        DESCR
```

ENDEX is used to terminate execution of a SNOBOL4 run. ENDEX is the last instruction executed and is responsible for returning properly to the environment that initiated the SNOBOL4 run. If I is nonzero, a post-mortem dump of user core should be given.

Data Input to ENDEX:

DESCR	I		

Programming Notes:

1. If a dump is not given, the keyword &ABEND will not have its specified effect. Nothing else will be affected.

2. On the IBM System/360, if I is nonzero, an abend dump is given with a user code of I.

3. See also INIT.

30. ENFILE (write end of file)

```
                ENFILE       DESCR
```

ENFILE is used to write an end-of-file on (close) the file associated with unit reference number I.

Data Input to ENFILE:

DESCR	I		

Programming Notes:

1. See also BKSPCE and REWIND.

2. Refer to Section 2.1 for a discussion of unit reference numbers.

31. EQU (symbol equivalence)

```
SYMBOL       EQU          N
```

EQU is used to assign, at assembly time, the value of N to SYMBOL.

32. EXPINT (exponentiate integers)

```
             EXPINT      DESCR1,DESCR2,DESCR3,FLOC,SLOC
```

EXPINT is used to raise an integer to an integer power. If I1 = 0 and I2 is not positive, or if the result is out of the range available for integers, transfer is to FLOC. Otherwise transfer is to SLOC.

Data Input to EXPINT:

DESCR2	I1	F	V
DESCR3	I2		

Data Altered by EXPINT:

DESCR1	I1**I2	F	V

33. EXREAL (exponentiate real numbers)

```
             EXREAL      DESCR1,DESCR2,DESCR3,FLOC,SLOC
```

EXREAL is used to raise a real number to a real power. If the result is not a real number or is out of the range available for real numbers, transfer is to FLOC. Otherwise transfer is to SLOC.

Data Input to EXREAL:

DESCR2	R1	F	V
DESCR3	R2		

Data Altered by EXREAL:

DESCR1	R1**R2	F	V

34. FORMAT (assemble format string)

```
LOC            FORMAT       'C1...CL'
```

FORMAT is used to assemble the characters of a format.

Data Assembled by FORMAT:

LOC			
LOC	C1	...	CL

Programming Notes:

1. The characters assembled by FORMAT are treated as an '''undigested''' format by FORTRAN IV routines.

35. FSHRTN (foreshorten specifier)

```
               FSHRTN       SPEC,N
```

FSHRTN is used to exclude initial characters from a string specification.

Data Input to FSHRTN:

SPEC				O	L

Data Altered by FSHRTN:

SPEC				O+N	L-N

Programming Notes:

1. L-N is never negative.

2. See also REMSP.

36. GETAC (get address with offset constant)

```
               GETAC        DESCR1,DESCR2,N
```

GETAC is used to get an address field with an offset constant.

Data Input to `GETAC`:

DESCR2	A2		
A2+N	A		

Data Altered by `GETAC`:

DESCR1	A		

Programming Notes:

1. `N` may be negative.

2. See also `PUTAC`, `GETDC`, and `PUTDC`.

---

37. `GETBAL` **(get parenthesis balanced string)**

```
            GETBAL      SPEC,DESCR,FLOC,SLOC
```

`GETBAL` is used to get the specification of a balanced substring. The string starting at `CL+1` and ending at `CL+N` is examined to determine the shortest balanced substring `CL+1,...,CL+J`. `J` is determined according to the following rules:

If `CL+1` is not a parenthesis, then `J` = 1.

If `CL+1` is a left parenthesis, then `J` is the least integer such that `CL+1...CL+J` is balanced with respect to parentheses in the usual algebraic sense.

If `CL+1` is a right parenthesis, or if no such balanced string exists, transfer is to `FLOC`. Otherwise `SPEC` is modified as indicated and transfer is to `SLOC`.

Data Input to `GETBAL`:

SPEC	A			O	L	
DESCR	N					
A+O	C1	...	CL	CL+1	...	CL+N

Data Altered by `GETBAL`:

SPEC	A			O	L+J

38. `GETD` (get descriptor)

```
            GETD          DESCR1,DESCR2,DESCR3
```

`GETD` is used to get a descriptor.

Data Input to `GETD`:

DESCR2	A2		
DESCR3	A3		
A2+A3	A	F	V

Data Altered by `GETD`:

DESCR1	A	F	V

Programming Notes:

1. See also `GETDC`, `PUTD`, and `PUTDC`.

39. `GETDC` (get descriptor with offset constant)

```
            GETDC         DESCR1,DESCR2,N
```

`GETDC` is used to get a descriptor with an offset constant.

Data Input to `GETDC`:

DESCR2	A2		
A2+N	A	F	V

Data Altered by `GETDC`:

DESCR1	A	F	V

Programming Notes:

1. See also `GETD`, `PUTDC`, and `PUTD`.

40. GETLG (get length of specifier)

```
                GETLG       DESCR,SPEC
```

GETLG is used to get the length of a specifier.

Data Input to GETLG:

SPEC					L

Data Altered by GETLG:

DESCR	L	0	0

Programming Notes:

1. See also PUTLG.

41. GETLTH (get length for string structure)

```
                GETLTH      DESCR1,DESCR2
```

GETLTH is used to determine the amount of storage required for a string structure. The amount of storage is given by the formula

```
F(L)=D*(3+[(L-1)/CPD+1])
```

where [L] is the integer part of L and CPD is the numbers of characters stored per descriptor. The constant 3 accounts for the three descriptors in a string structure in addition to the string itself. The expression in brackets represents the number of descriptors required for a string of L characters.

Data Input to GETLTH:

DESCR2	L		

Data Altered by GETLTH:

DESCR1	F(L)	0	0

Programming Notes:

1. See also BKSIZE.

42. GETSIZ (get size)

```
            GETSIZ     DESCR1,DESCR2
```

GETSIZ is used to get the size from the value field of a title descriptor.

Data Input to GETSIZ:

DESCR2	A		
A			V

Data Altered by GETSIZ:

DESCR1	V	0	0

Programming Notes:

1. See also SETSIZ.

43. GETSPC (get specifier with constant offset)

```
            GETSPC     SPEC,DESCR,N
```

GETSPC is used to get a specifier.

Data Input to GETSPC:

DESCR	A1				
A1+N	A	F	V	O	L

Data Altered by GETSPC:

SPEC	A	F	V	O	L

Programming Notes:

1. See also PUTSPC.

44. `INCRA` (increment address)

```
                    INCRA         DESCR,N
```

`INCRA` is used to increment the address field of a descriptor.

Data Input to `INCRA`:

DESCR	A		

Data Altered by `INCRA`:

DESCR	A+N		

Programming Notes:

1. `A` may be a relocatable address.
2. `A` is never negative.
3. `N` is always positive.
4. `N` is often 1 or `D`.
5. See also `DECRA` and `INCRV`.

45. `INCRV` (increment value field)

```
                    INCRV         DESCR,N
```

`INCRV` is used to increment the value field of a descriptor. `I` is considered as an unsigned (nonnegative) integer.

Data Input to `INCRV`:

DESCR			I

Data Altered by `INCRV`:

DESCR			I+N

Programming Notes:

1. N is always positive.
2. N is often 1.
3. See also INCRA.

---

46. INIT (initialize SNOBOL4 run)

```
                INIT
```

INIT is used to initialize a SNOBOL4 run. INIT is the first instruction executed and is responsible for performing any initialization necessary. The operation is machine and system dependent. Typically, INIT sets program masks and the values of vertain registers.

In addition to any initialization required for a particular system and machine, INIT also performs the following initialization for the SNOBOL4 system. Dynamic storage is initialized. The address fields of FRSGPT and HDSGPT are set to point to the first descriptor in dynamic storage. The address field of TLSGP1 is set to the first descriptor past the end of dynamic storage. Space for dynamic storage may be preallocated or obtained from the operating system by INIT. The timer is initialized for subsequent use by the MSTIME macro (q.v.).

Programming Notes:

1. See also ENDEX.

---

47. INSERT (insert node in tree)

```
                INSERT      DESCR1,DESCR2
```

INSERT is used to insert a tree node above another node.

Data Input to INSERT:

DESCR1	A1	F1	V1
DESCR2	A2	F2	V2
A1+FATHER	A3	F3	V3
A3+LSON	A4	F4	V4
A2+CODE			I

Data Altered by `INSERT`:

`A1+FATHER`	`A2`	`F2`	`V2`
`A4+RSIB`	`A2`	`F2`	`V2`
`A2+FATHER`	`A3`	`F3`	`V3`
`A2+LSON`	`A1`	`F1`	`V1`
`A2+CODE`			`I+1`

Programming Notes:

1. Since the fields of the descriptor at `A1+FATHER` are used in the data to be altered, care should be taken not to modify this descriptor until its former values have been used.

2. `INSERT` is only used by compilation procedures.

3. `FATHER`, `LSON`, `RSIB`, and `CODE` are symbols defined in the source program.

4. See also `ADDSIB` and `ADDSON`.

---

48. `INTRL` (convert integer to real number)

```
                  INTRL         DESCR1,DESCR2
```

`INTRL` is used to convert a (signed) integer to a real number. `R(I)` is the real number corresponding to `I`.

Data Input to `INTRL`:

`DESCR2`	`I`		

Data Altered by `INTRL`:

`DESCR1`	`R(I)`	`0`	`R`

Programming Notes:

1. `R` is a symbol defined in the source program and is the code for the real data type.

49. INTSPC (convert integer to specifier)

```
            INTSPC      SPEC,DESCR
```

INTSPC is used to convert a (signed) integer to a specified string.

Data Input to INTSPC:

DESCR	I		

Data Altered by INTSPC:

SPEC	BUFFER	0	0	O	L

BUFFER+O	C1	...	CL

Programming Notes:

1. C1...CL should be a ‘‘‘normalized’’’ string corresponding to the integer I. That is, it should contain no leading zeroes and should begin with a minus sign if I is negative.

2. BUFFER is local to INTSPC and its contents may be overwritten by a subsequent use of INTSPC.

3. See also SPCINT.

50. ISTACK (initialize stack)

```
            ISTACK
```

ISTACK is used to initialize the system stack.

Data Altered by ISTACK:

OSTACK	0		
CSTACK	STACK		

Programming Notes:

1. STACK is a program symbol whose value is the address of the first descriptor of the system stack.

2. See also PSTACK, RCALL, and RRTURN.

51. LCOMP (length comparison)

```
            LCOMP       SPEC1,SPEC2,GTLOC,EQLOC,LTLOC
```

LCOMP is used to compare the lengths of two specifiers. If L1 > L2, transfer is to GTLOC. If L1 = L2, transfer is to EQLOC. If L1 < L2, transfer is to LTLOC.

Data Input to LCOMP:

SPEC1					L1
SPEC2					L2

Programming Notes:

1. See also ACOMP, RCOMP, and LEQLC.

52. LEQLC (length equal to constant test)

```
            LEQLC       SPEC,N,NELOC,EQLOC
```

LEQLC is used to compare the length of a specifier to a constant. If L = N, transfer is to EQLOC. Otherwise transfer is to NELOC.

Data Input to LEQLC:

SPEC					L

Programming Notes:

1. L and N are never negative.

2. See also LCOMP, AEQLC, and AEQLIC.

53. LEXCMP (lexical comparison of strings)

```
            LEXCMP      SPEC1,SPEC2,GTLOC,EQLOC,LTLOC
```

LEXCMP is used to compare two strings lexicographically (i.e. according to their alphabetical ordering). If C11...C1N1 < C21...C2M, transfer is to GTLOC. If C11...C1N1 = C21...C2M, transfer is to EQLOC. If C11...C1N1 > C21...C2M, transfer is to LTLOC.

Data Input to LEXCMP:

SPEC1	A1			O1	N
SPEC2	A2			O2	M
A1+O1	C11	...	C1N		
A2+O2	C21	...	C2M		

Programming Notes:

1. The lexicographical ordering is machine dependent and is determined by the numerical order of the internal representation of the characters for a particular machine.

2. A string that is an initial substring of another string is lexicographically less than that string. That is ABC is less than ABCA.

3. The null (zero-length) string is lexicographically less than any other string.

4. Two strings are equal if and only if they are of the same length and are identical character by character.

5. By far the most frequent use of LEXCMP is to determine whether two strings are the same or different. In these cases GTLOC and LTLOC will specify the same location or both be omitted. Because of the frequency of such use, it is desirable to handle this case specially, since a test for equality usually can be performed more efficiently than the general test.

---

54. LHERE (location here)

```
LOC           LHERE
```

LHERE is used to establish the equivalence of LOC as the location of the next program instruction.

Programming Notes:

1. LHERE is equivalent to the familiar EQU *. Similarly

```
LOCLHERE
OP
```

is equivalent to

```
LOCOP
```

55. `LINK` (link to external function)

```
                LINK        DESCR1,DESCR2,DESCR3,DESCR4,FLOC,SLOC
```

`LINK` is used to link to an external function. `A2` is a pointer to an argument list of `N` descriptors. `A4` is the address of the external function to be called. `V1` is the date type expected for the resulting value. The returned value is placed in `DESCR1`. If the external function signals failure, transfer is to `FLOC`. Otherwise transfer is to `SLOC`.

Data Input to `LINK`:

DESCR1			V1
DESCR2	A2		
DESCR3	N		
DESCR4	A4		

Data Altered by `LINK`:

DESCR1	A	F	V

Programming Notes:

1. `LINK` is a system-dependent operation.
2. `LINK` need not be implemented if `LOAD` is not. In this case, `LINK` should branch to `INTR10`.
3. See also `LOAD` and `UNLOAD`.

56. `LINKOR` (link ‘‘‘or’’’ fields of pattern nodes)

```
                LINKOR      DESCR1,DESCR2
```

`LINKOR` links through ‘‘‘or’’’ (alternative) fields of pattern nodes until the end, indicated by a zero field, is reached. This zero field is replaced by `I`.

Data Input to `LINKOR`:

DESCR1	A		
DESCR2	I		
A+2D	I1		
A+2D+I1	I2		
	.		
	.		
	.		
A+2D+IN	0		

Data Altered by `LINKOR`:

A+2D+IN	I		

---

57. `LOAD` **(load external function)**

```
                LOAD          DESCR,SPEC1,SPEC2,FLOC,SLOC
```

`LOAD` is used to load an external function. `C11...C1L1` is the name of the external function to be loaded from a library. `C21...C2L2` is the name of the library. `A3` is the address of the entry point. If the external function is loaded, transfer is to `SLOC`. Otherwise transfer is to `FLOC`.

Data Input to `LOAD`:

SPEC1	A1			O1	L1
SPEC2	A2			O2	L2

A1+O1	C11	...	C1L1
A2+O2	C21	...	C2L2

Data Altered by `LOAD`:

DESCR	A3		

Programming Notes:

1. LOAD is a system-dependent operation.

2. LOAD need not be implemented as such. If it is not, the built-in function LOAD will not be available, and an error comment should be generated by branching to UNDF.

3. On the IBM System/360, LOAD uses the OS macro LOAD to bring an external function from the library whose DDNAME is specified by C21...C2L2.

4. See also LINK and UNLOAD.

---

58. LOCAPT (locate attribute pair by type)

```
                LOCAPT      DESCR1,DESCR2,DESCR3,FLOC,SLOC
```

LOCAPT is used to locate the ‘‘‘type’’’ descriptor of a descriptor pair on an attribute list. Descriptors on an attribute list are in ‘‘‘type-value’’’ pairs. Odd-numbered descriptors are ‘‘‘type’’’ descriptors. The list starting at A+D is searched, comparing descriptors at A+D, A+3D, ... for the first descriptor whose value is equal to the value of DESCR3. If a descriptor equal to DESCR3 is not found, transfer is to FLOC. Otherwise transfer is to SLOC.

Data Input to LOCAPT:

DESCR2	A	F	V
DESCR3	A3	F3	V3
A			2K*D
A+D	A11	F11	V11
		.	
		.	
		.	
A+D+2I*D	A3	F3	V3
		.	
		.	
		.	
A+2K*D			

Data Altered by LOCAPT:

DESCR1	A+2I*D	F	V

Programming Notes:

1. Note that the address of `DESCR1` is set to one descriptor less then the descriptor that is located.

2. See also `LOCAPV`.

---

59. `LOCAPV` **(locate attribute pair by value)**

```
              LOCAPV      DESCR1,DESCR2,DESCR3,FLOC,SLOC
```

`LOCAPV` is used to locate the ‘‘‘value’’’ descriptor of a descriptor pair on an attribute list. Descriptors on an attribute list are in ‘‘‘type-value’’’ pairs. Even-numbered descriptors are ‘‘‘value’’’ descriptors. The list starting at `A+D` is searched, comparing descriptors at `A+2D`, `A+4D`, ... for the first descriptor whose value is equal to the value of `DESCR3`. If a descriptor equal to `DESCR3` is not found, transfer is to `FLOC`. Otherwise transfer is to `SLOC`.

Data Input to `LOCAPV`:

DESCR2	A	F	V
DESCR3	A3	F3	V3
A			2K*D
A+2D	A12	F12	V12
		.	
		.	
		.	
A+2D+2I*D	A3	F3	V3
		.	
		.	
		.	
A+2K*D			

Data Altered by `LOCAPV`:

DESCR1	A+2I*D	F	V

Programming Notes:

1. Note that the address of `DESCR1` is set to two descriptors less than the descriptor that is located.

2. See also `LOCAPT`.

60. LOCSP (locate specifier to string)

```
            LOCSP       SPEC,DESCR
```

LOCSP is used to obtain a specifier to a string given in a string structure. CPD is the number of characters per descriptor.

Data Input to LOCSP:

DESCR	A	F	V
A			I

Data Altered by LOCSP if A $\neq$ O:

SPEC	A	F	V	4*CPD	I

Data Altered by LOCSP if A = O:

SPEC					0

Programming Notes:

1. If A = O, the value of DESCR represents the null (zero-length) string and is handled as a special case as indicated. The other fields of SPEC are unchanged in this case.

61. LVALUE (get least length value)

```
            LVALUE      DESCR1,DESCR2
```

LVALUE is used to get the least value of address fields in a chain of pattern nodes. The address field of DESCR1 is set to I where

```
I = min(I0,...,IK)
```

Data Input to LVALUE:

DESCR2	A		
A+2D	N1		
A+3D	I0		
A+N1+2D	N2		
A+N1+3D	I1		

.
.
.

A+NK+2D	0		
A+NK+3D	IK		

Data Altered by LVALUE:

DESCR1	I	0	0

Programming Notes:

1. I0,...,IK are all nonnegative.

2. A is never zero, but N1 may be.

---

62. MAKNOD (make pattern node)

```
                MAKNOD      DESCR1,DESCR2,DESCR3,DESCR4,DESCR5,DESCR6
```

MAKNOD is used to make a node for a pattern. DESCR6 may be omitted. If it is, one less descriptor is modified, but the two forms are otherwise the same.

Data Input to MAKNOD:

DESCR2	A2	F2	V2
DESCR3	A3		
DESCR4	A4		
DESCR5	A5	F5	V5

Additional Data Input if `DESCR6` is Given:

DESCR6	A6	F6	V6

Data Altered by `MAKNOD`:

DESCR1	A2	F2	V2
A2+D	A5	F5	V5
A2+2D	A4		
A2+3D	A3		

Additional Data Altered if `DESCR6` is Given:

A2+4D	A6	F6	V6

Programming Notes:

1. As indicated, there are two forms of `MAKNOD`. If `DESCR6` is given, an additional descriptor if modified, but otherwise the two forms are the same.

2. `DESCR1` must be changed *last*, since `DESCR6` may be the same descriptor as `DESCR1`.

3. `MAKNOD` is used only for constructing patterns.

---

63. `MNREAL` (minus real number)

```
            MNREAL      DESCR1,DESCR2
```

`MNREAL` is used to change the sign of a real number.

Data Input to `MNREAL`:

DESCR2	R	F	V

Data Altered by `MNREAL`:

DESCR1	-R	F	V

Programming Notes:

1. `R` may be negative.

2. See also `MNSINT`, `ADREAL`, `DVREAL`, `EXREAL`, `MPREAL`, and `SBREAL`.

64. MNSINT (minus integer)

```
                MNSINT       DESCR1,DESCR2,FLOC,SLOC
```

MNSINT is used to change the sign of an integer. If -I exceeds the maximum integer, transfer is to FLOC. Otherwise transfer is to SLOC.

Data Input to MNSINT:

DESCR2	I	F	V

Data Altered by MNSINT:

DESCR1	-I	F	V

Programming Notes:

1. I may be negative.

2. See also MNREAL.

65. MOVA (move address)

```
                MOVA         DESCR1,DESCR2
```

MOVA is used to move an address field from one descriptor to another.

Data Input to MOVA:

DESCR2	A		

Data Altered by MOVA:

DESCR1	A		

Programming Notes:

1. See also MOVD and MOVV.

66. `MOVBLK` (move block of descriptors)

```
               MOVBLK      DESCR1,DESCR2,DESCR3
```

`MOVBLK` is used to move (copy) a block of descriptors.

Data Input to `MOVBLK`:

DESCR1	A1		
DESCR2	A2		
DESCR3	D*N		
A2+D	A21	F21	V21
	.	.	.
A2+(D*N)	A2N	F2N	V2N

Data Altered by `MOVBLK`:

A1+D	A21	F21	V21
	.	.	.
A1+(D*N)	A2N	F2N	V2N

Programming Notes:

1. Note that the descriptor at `A1` is not altered.

2. The area into which the move is made may overlap the area from which the move is made. This only occurs when `A1` is less than `A2`. Care must be taken to handle this case correctly.

67. `MOVD` (move descriptor)

```
               MOVD        DESCR1,DESCR2
```

`MOVD` is used to move (copy) a descriptor from one location to another.

Data Input to `MOVD`:

`DESCR2`	A	F	V

Data Altered by `MOVD`:

`DESCR1`	A	F	V

Programming Notes:

1. See also `MOVA` and `MOVV`.

---

68. `MOVDIC` (move descriptor indirect with constant offset)

```
              MOVDIC      DESCR1,N1,DESCR2,N2
```

`MOVDIC` is used to move a descriptor that is indirectly specified with an offset constant.

Data Input to `MOVDIC`:

`DESCR1`	A1		
`DESCR2`	A2		
`A2+N2`	A	F	V

Data Altered by `MOVDIC`:

`A1+N1`	A	F	V

Programming Notes:

1. See also `MOVD`, `GETDC`, and `PUTDC`.

---

69. `MOVV` (move value field)

```
              MOVV        DESCR1,DESCR2
```

`MOVV` is used to move a value field from one descriptor to another.

Data Input to `MOVV`:

`DESCR2`			V

Data Altered by `MOVV`:

`DESCR1`			`V`

Programming Notes:

1. See also `MOVA` and `MOVD`.

---

70. `MPREAL` **(multiply real numbers)**

```
                 MPREAL       DESCR1,DESCR2,DESCR3,FLOC,SLOC
```

`MPREAL` is used to multiply two real numbers. If the result is out of the range available for real numbers, transfer is to `FLOC`. Otherwise transfer is to `SLOC`.

Data Input to `MPREAL`:

`DESCR2`	`R2`	`F2`	`V2`
`DESCR3`	`R3`		

Data Altered by `MPREAL`:

`DESCR1`	`R2*R3`	`F2`	`V2`

Programming Notes:

1. See also `ADREAL`, `DVREAL`, `EXREAL`, `MNREAL`, and `SBREAL`.

---

71. `MSTIME` **(get millisecond time)**

```
                 MSTIME       DESCR
```

`MSTIME` is used to get the millisecond time.

Data Altered by `MSTIME`:

`DESCR`	`TIME`	`0`	`0`

Programming Notes:

1. The origin with respect to which the time is obtained is not important. The SNOBOL4 system deals only with differences in times.

2. The time units should be milliseconds, but accuracy is not critical.

3. `MSTIME` is used in program tracing, the SNOBOL4 `TIME` function, and in statistics printed upon termination of a SNOBOL4 run.

4. It is not critically important that `MSTIME` be implemented as such. If it is not, the address field of `DESCR` should be set to zero also.

5. See also `INIT`.

---

72. `MULT` (multiply integers)

```
               MULT         DESCR1,DESCR2,DESCR3,FLOC,SLOC
```

`MULT` is used to multiply two integers. In the event of overflow, transfer is to `FLOC`. Otherwise, transfer is to `SLOC`.

Data Input to `MULT`:

DESCR2	I2	F2	V2
DESCR3	I3		

Data Altered by `MULT`:

DESCR1	I2*I3	F2	V2

Programming Notes:

1. The test for success and failure is used in only two calls of this macro. Hence the code to make the check is not needed in most cases.

2. `DESCR1` and `DESCR2` are often the same.

3. See also `MULTC` and `DIVIDE`.

73. MULTC (multiply address by constant)

```
            MULTC       DESCR1,DESCR2,N
```

MULTC is used to multiply an integer by a constant.

Data Input to MULTC:

DESCR2	I		

Data Altered by MULTC:

DESCR1	I*N	0	0

Programming Notes:

1. I*N never exceeds the range available for integers.

2. DESCR1 and DESCR2 are often the same.

3. N is often D, which typically may be implemented by a , or simply by no operation if D is 1 for a particular machine.

4. See also MULT.

74. ORDVST (order variable storage)

```
            ORDVST
```

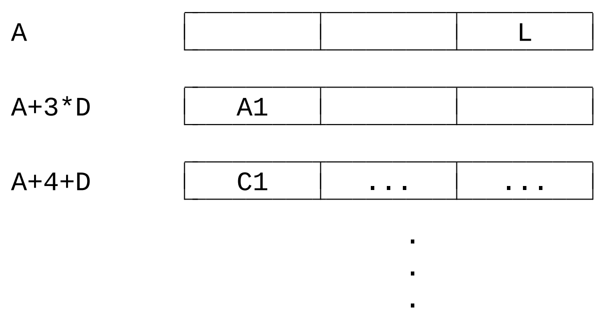
ORDVST is used to alphabetically order variables in SNOBOL4 dynamic storage. Variables are organized in a number of bins, each bin containing a linked list of variables as shown below. OBEND = OBSTRT+(OBSIZ-1)*D, where OBSIZ is the number of bins and is defined in the source program.

Bins of Variables:

OBSTRT	A1		
OBSTRT+D	A2		
		.	
		.	
		.	
OBEND	AN		

The addresses A1, A2, ..., AN point to the first variable in each bin. A zero value for any of these addresses indicates there are no variables in that bin. Within each bin, variables are linked together.

Relevant Parts of a Variable:



`L` is the length of the string. The string itself begins at `A+4*D` and occupies as many descriptor locations as are necessary. `A1` is a link to the next variable in the bin. A zero value of `A1` indicates the end of the chain for that bin.

Programming Notes:

1. `ORDVST` is used only in ordering variables for a programmer-requested post-mortem dump of variable storage. `ORDVST` need not be implemented as such, but may simply perform no operation. In this case, the post-mortem dump will not be alphabetized, but will be otherwise correct.

2. If `ORDVST` *is* implemented, it is easiest to put all variables in one long chain starting at `OBSTRT`. The address fields of the descriptors `OBSTRT+D`,...,`OBSTRT+(OBSIZ-1)*D` should then be set to zero.

3. Since dynamic storage may contain many variables, some care must be taken to assure that the sorting procedure is not excessively slow. Variables whose values are the null string (zero address field and value field containing the program symbol `S`) should be omitted from the sort.

4. Since any character may appear in a string, the value of `I` must be used to determine the length of the string in a variable — characters following the string in the last descriptor are undefined.

---

75. `OUTPUT` (output record)

```
              OUTPUT      DESCR,FORMAT,(DESCR1,...,DESCRN)
```

`OUTPUT` is used to output a list of items according to `FORMAT`. The output is put on the file associated with unit reference number `I`. The format `C1...CL` may specify literals and the conversion of integers and real numbers given in the address fields `A1`,...,`AN`.

Data Input to OUTPUT:

DESCR	I		
FORMAT	C1	...	CL
DESCR1	A1		
	.		
	.		
	.		
DESCRN	AN		

Programming Notes:

1. See also STPRNT.

---

76. PLUGTB (plug syntax table)

```
            PLUGTB      TABLE,KEY,SPEC
```

PLUGTB is used to set selected indicator fields in the entries of a syntax table to a constant. KEY may be one of four values:

```
CONTIN
ERROR
STOP
STOPSH
```

The indicator fields of entries corresponding to C1,...,CL are set to T where T is the indicator that corresponds to the value of KEY.

Data Input to PLUGTB:

SPEC	A			O	L
A+O	C1	...	CL		

Data Altered by PLUGTB for ERROR, STOP, or STOPSH:

TABLE+E*C1		T	
		.	
		.	
		.	
TABLE+E*CL		T	

Data Altered by `PLUGTB` for `CONTIN`:

`TABLE+E*C1`	`TABLE`	`0`	
	.		
	.		
	.		
`TABLE+E*CL`	`TABLE`	`0`	

Programming Notes:

1. See Section 4.2.

2. See also `CLERTB`.

---

77. `POP` (pop descriptors from stack)

```
             POP          (DESCR1,...,DESCRN)
```

`POP` is used to pop a list of descriptors off the system stack.

Data Input to `POP`:

`CSTACK`	`A`		
`A`	`A1`	`F1`	`V1`
	.		
	.		
	.		
`A-D*(N-1)`	`AN`	`FN`	`VN`

Data Altered by `POP`:

`CSTACK`	`A-(N*D)`		
`DESCR1`	`A1`	`F1`	`V1`
	.		
	.		
	.		
`DESCRN`	`AN`	`FN`	`VN`

Programming Notes:

1. If `A-(N*D) < STACK`, stack underflow occurs. This condition indicates a programming error in the implementation of the macro language. An appropriate diagnostic message indicating an error may be obtained by transferring to the program location `INTR10` if the condition is detected.

---

78. `PROC` (procedure entry)

```
LOC1          PROC          LOC2
```

`PROC` is used to identify a procedure entry point. `LOC2` may be omitted, in which case `LOC1` is the primary procedure entry point. If `LOC2` is given, `LOC1` is a secondary entry point in the procedure with primary entry point `LOC2`.

Programming Notes:

1. Procedure entry points are referred to by `RCALL`, `BRANIC`, and `BRANCH` (in its two-argument form).

2. In most implementations, `PROC` has no functional use and may be implemented as `LHERE`. For machines that have a severely limited program basing range (such as the IBM System/360), `PROC` may be used to perform required basing operations.

---

79. `PSTACK` (post stack position)

```
              PSTACK        DESCR
```

`PSTACK` is used to post the current stack position.

Data Input to `PSTACK`:

`CSTACK`	A		

Data Altered by `PSTACK`:

`DESCR`	A-D	0	0

Programming Notes:

1. See also `ISTACK`.

80. PUSH (push descriptors onto stack)

```
                PUSH          (DESCR1,...,DESCRN)
```

PUSH is used to push a list of descriptors onto the system stack.

Data Input to PUSH:

CSTACK	A		
DESCR1	A1	F1	V1
		.	
		.	
		.	
DESCRN	AN	FN	VN

Data Altered by PUSH:

CSTACK	A+(D*N)		
A+D	A1	F1	V1
		.	
		.	
		.	
A+(D*N)	AN	FN	VN

Programming Notes:

1. If A+(D*N) > STACK+STSIZE, stack overflow occurs. Transfer should be made to the program location OVER, which will result in an appropriate error termination.

2. See also SPUSH, POP, and SPOP.

81. PUTAC (put address with offset constant)

```
                PUTAC         DESCR1,N,DESCR2
```

PUTAC is used to put an address field into a descriptor located at a constant offset.

Data Input to PUTAC:

DESCR1	A1		
DESCR2	A2		

Data Altered by PUTAC:

A1+N	A2		

Programming Notes:

1. See also GETAC, PUTVC, PUTD, and PUTDC.

---

82. PUTD (put descriptor)

```
                PUTD          DESCR1,DESCR2,DESCR3
```

PUTD is used to put a descriptor.

Data Input to PUTD:

DESCR1	A1		
DESCR2	A2		
DESCR3	A	F	V

Data Altered by PUTD:

A1+A2	A	F	V

Programming Notes:

1. See also PUTDC, PUTAC, PUTVC, and GETD.

---

83. PUTDC (put descriptor with constant offset)

```
                PUTDC         DESCR1,N,DESCR2
```

PUTDC is used to put a descriptor at a location with a constant offset.

Data Input to PUTDC:

DESCR1	A1		
DESCR2	A	F	V

Data Altered by PUTDC:

A1+N	A	F	V

Programming Notes:

1. See also PUTD, PUTAC, PUTVC, and GETD.

---

84. PUTLG **(put specifier length)**

```
                PUTLG       SPEC,DESCR
```

PUTLG is used to put a length into a specifier.

Data Input to PUTLG:

DESCR	I		

Data Altered by PUTLG:

SPEC					I

Programming Notes:

1. I is always nonnegative.

2. See also GETLG.

---

85. PUTSPC **(put specifier with offset constant)**

```
                PUTSPC      DESCR,N,SPEC
```

PUTSPC is used to put a specifier.

Data Input to PUTSPC:

DESCR	A1				
SPEC	A	F	V	O	L

Data Altered by PUTSPC:

A1+N	A	F	V	O	L

Programming Notes:

1. See also GETSPC.

---

86. PUTVC **(put value field with offset constant)**

```
            PUTVC       DESCR1,N,DESCR2
```

PUTVC is used to put a value field into a descriptor at a location with a constant offset.

Data Input to PUTVC:

DESCR1	A		
DESCR2			V

Data Altered by PUTVC:

A+N			V

Programming Notes:

1. See also PUTAC, PUTDC, and PUTD.

---

87. RCALL **(recursive call)**

```
            RCALL       DESCR,PROC,(DESCR1,...,DESCRN),(LOC1,...,LOCM)
```

RCALL is used to perform a recursive call. DESCR is the descriptor that receives the value upon return from the call. PROC is the procedure being called. DESCR1,...,DESCRN are descriptors whose values are passed to PROC. LOC1,...,LOCM are locations to transfer to upon return according to the return exit signaled. The old stack pointer (A0) is saved on the stack, the current stack pointer becomes the old stack pointer, and a new current stack pointer is generated as indicated. The return location LOC is saved on the stack so that the return can be properly made. The values of the arguments DESCR1,...,DESCRN are placed on the stack. Note that their order is the *opposite* of the order that would be obtained by using PUSH.

At the return location LOC, code similar to that shown should be assembled. OP represents an instruction that stores the value returned by PROC in DESCR.

Data Input to RCALL:

CSTACK	A		
OSTACK	A0		
DESCR1	A1	F1	V1
		⋮	
DESCRN	AN	FN	VN

Data Altered by RCALL:

A+D	A0	0	0
A+2D	LOC	0	0
A+3D	AN	FN	VN
		⋮	
A+D*(2+N)	A1	F1	V1
CSTACK	A+(2+N)*D		
OSTACK	A		

Return Code at LOC:

```
LOC       OP           DESCR1
          BRANCH       LOC1
              .
              .
              .
          BRANCH       LOCM
```

Programming Notes:

1. RCALL and RRTURN are used in combination, and their relation to each other must be thoroughly understood in order to implement them correctly.

2. Ordinarily OP is an instruction to store the value returned by RRTURN.

3. DESCR sometimes is omitted. In this case, any value returned by RRTURN is ignored and OP should perform no operation.

4. (DESCR1,...,DESCRN) sometimes is entirely omitted. In this case N should be taken to be zero in interpreting the figures.

5. Any of the locations LOC1,...,LOCM may be omitted. As in the case of operations with omitted conditional branches, control then passes to the operation following the RCALL.

6. The return indicated by RRTURN may be M+1, in which case control is passed to the operation following the RCALL.

7. The return indicated by RRTURN is never greater than M+1.

8. RCALL typically must save program state information. On the IBM System/360, this consists of the location LOC and a base register for the procedure containing the RCALL. This information is pushed onto the stack. In pushing information onto the stack, care must be taken to observe the rules concerning the use of descriptors. The rest of the SNOBOL4 system treats the stack as descriptors, and the flag fields of descriptors used to save program state information *must be set to zero*.

9. See also SELBRA.

---

88. RCOMP **(real comparison)**

```
              RCOMP        DESCR1,DESCR2,GTLOC,EQLOC,LTLOC
```

RCOMP is used to compare two real numbers. If R1 > R2, transfer is to GTLOC. If R1 = R2, transfer is to GTLOC. If R1 < R2, transfer is to LTLOC.

Data Input to RCOMP:

DESCR1	R1		
DESCR2	R2		

Programming Notes:

1. See also ACOMP and LCOMP.

89. REALST **(convert real number to string)**

```
                REALST      SPEC,DESCR
```

REALST is used to convert a real number into a specified string.

Data Input to REALST:

DESCR	R		

Data Altered by REALST:

SPEC	BUFFER	0	0	0	L

BUFFER	C1	...	CL

Programming Notes:

1. C1...CL should represent the real number R in the SNOBOL4 fashion, containing a decimal point and having at least one digit before the decimal point, zeroes being added as necessary. If R is negative, the string should begin with a minus sign. For compatibility with real literals and data type conversions, the real number should not be represented in exponent form, although very large or small real numbers may require a large number of characters for their representation otherwise.

2. The number of digits (and hence the size of BUFFER) required is machine dependent and depends on the range available for real numbers.

3. BUFFER is local to REALST and its contents may be overwritten by a subsequent use of REALST.

4. See also INTSPC and SPREAL.

90. REMSP **(specify remaining string)**

```
                REMSP       SPEC1,SPEC2,SPEC3
```

REMSP is used to obtain a remainder specifier resulting from the deletion of a specified length at the end.

Data Input to REMSP:

SPEC2	A2	F2	V2	O2	L2
SPEC3					L3

Data Altered by REMSP:

SPEC1	A2	F2	V2	O2+L3	L2-L3

Programming Notes:

1. SPEC1 and SPEC3 may be the same.

2. L2-L3 is never negative.

3. See also FSHRTN.

---

91. RESETF **(reset flag)**

```
               RESETF       DESCR,FLAG
```

RESETF is used to reset (delete) a flag from a descriptor.

Data Input to RESETF:

DESCR		F	

Data Altered by RESETF:

DESCR		F-FLAG	

Programming Notes:

1. Only FLAG is removed from the flags in F. Any other flags are left unchanged.

2. If F does not contain FLAG, no data is altered.

3. See also RSETFI and SETFI.

---

92. REWIND **(rewind file)**

```
               REWIND       DESCR
```

REWIND is used to rewind the file associated with the unit reference number I.

Data Input to REWIND:

DESCR	I		

Programming Notes:

1. Refer to Section 2.1 for a discussion of unit reference numbers.

2. See also `BKSPCE` and `ENFILE`.

---

93. `RLINT` (convert real number to integer)

```
              RLINT        DESCR1,DESCR2,FLOC,SLOC
```

`RLINT` is used to convert a real number to an integer. If the magnitude of `R` exceeds the magnitude of the largest integer, transfer is to `FLOC`. Otherwise transfer is to `SLOC`.

Data Input to `RLINT`:

`DESCR2`	`R`		

Data Altered by `RLINT`:

`DESCR1`	`I(R)`	`0`	`I`

Programming Notes:

1. `I(R)` is the integer equivalent of the real number `R`.

2. The fractional part of `R` is discarded.

3. `I` is a symbol defined in the source program and is the code for the integer data type.

---

94. `RPLACE` (replace characters)

```
              RPLACE       SPEC1,SPEC2,SPEC3
```

`RPLACE` is used to replace characters in a string. `SPEC2` specifies a set of characters to be replaced. `SPEC3` specifies the replacement to be made for the characters specified by `SPEC2`. The replacement is described by the following rules. For `I = 1`,...,`L`

`F(CI) = CI` if `CI ≠ C2J` for any `J (1 ≤ J ≤ L2)`
`F(CI) = C3J` if `CI = C2J` for some `J (1 ≤ J ≤ L2)`

Data Input to RPLACE:

SPEC1	A1			O1	L
SPEC2	A2			O2	L2
SPEC3	A3			O3	L2

.
.
.

A1+O1	C1	...	CL
A2+O2	C21	...	C2L2
A3+O3	C31	...	C3L2

Data Altered by RPLACE:

A1+O1	F(C1)	...	F(CL)

Programming Notes:

1. L may be zero.

2. If there are duplicate characters in C21...C2L2, replacement should be made corresponding to the last instance of the character. That is, if

```
C2I = C2J = ... = C2K  (I < J < K)
```

then

```
F(CI) = C3K
```

3. RPLACE is used only in the SNOBOL4 REPLACE function. It is not essential that RPLACE be implemented as such. If it is not, RPLACE should transfer to UNDF to provide an appropriate error comment.

---

95. RRTURN (recursive return)

```
                 RRTURN       DESCR,N
```

RRTURN is used to return from a recursive call. DESCR is the descriptor whose value is returned. The stack pointers are repositioned as shown. At the location LOC, code similar to that shown is assembled by the RRCALL to which return is to be made. OP represents an instruction that is used by RRTURN to return the value of DESCR. Control is transferred to LOCN corresponding to N given in the RRTURN.

Data Input to `RRTURN`:

OSTACK	A		
A+D	A0		
A+2D	LOC		
DESCR	A1	F1	V1

Data Altered by `RRTURN`:

CSTACK	A		
OSTACK	A0		
DESCR1	A1	F1	V1

Return Code at `LOC`:

```
LOC      OP          DESCR1
         BRANCH      LOC1
            .
            .
            .
         BRANCH      LOCM
```

Programming Notes:

1. `RCALL` and `RRTURN` are used in combination, and their relation to each other must be thoroughly understood.

2. `DESCR` may be omitted. In this case, `OP` should not be executed.

---

96. `RSETFI` (reset flag indirect)

```
            RSETFI       DESCR,FLAG
```

`RSETFI` is used to reset (delete) a flag from a descriptor that is specified indirectly.

Data Input to `RSETFI`:

DESCR	A		
A		F	

Data Altered by `RSETFI`:

`A`		`F-FLAG`	

Programming Notes:

1. Only `FLAG` is removed from the flags in `F`. Any other flags are left unchanged.

2. If `F` does not contain `FLAG`, no data is altered.

3. See also `RESETF` and `SETFI`.

---

97. `SBREAL` **(subtract real numbers)**

```
                SBREAL      DESCR1,DESCR2,DESCR3,FLOC,SLOC
```

`SBREAL` is used to subtract one real number from another. If the result is out of the range available for real numbers, transfer is to `FLOC`. Otherwise transfer is to `SLOC`.

Data Input to `SBREAL`:

`DESCR2`	`R2`	`F2`	`V2`
`DESCR3`	`R3`		

Data Altered by `SBREAL`:

`DESCR1`	`R2-R3`	`F2`	`V2`

Programming Notes:

1. See also `ADREAL`, `DVREAL`, `EXREAL`, `MNREAL`, and `MPREAL`.

---

98. `SELBRA` **(select branch point)**

```
                SELBRA      DESCR,(LOC1,...,LOCN)
```

`SELBRA` is used to alter the flow of program control by selecting a location from a list and branching to it. Transfer is to `LOCI` corresponding to `I`.

Data Input to `SELBRA`:

`DESCR`	`I`		

Programming Notes:

1. Any of the locations may be omitted. As in the case of operations with omitted conditional branches, control then passes to the operation following `SELBRA`.

2. If `I = N+1`, control is passed to the operation following `SELBRA`.

3. `I` is always in the range `1 ≤ I ≤ N+1`. For debugging purposes, it may be useful to verify that `I` is within this range.

---

99. `SETAC` **(set address to constant)**

```
                SETAC       DESCR,N
```

`SETAC` is used to set the address field of a descriptor to a constant.

Data Altered by `SETAC`:

DESCR	N		

Programming Notes:

1. `N` may be a relocatable address.

2. `N` is often 0, 1, or `D`.

3. `N` is never negative.

4. See also `SETVC`, `SETLC`, and `SETAV`.

---

100. `SETAV` **(set address from value field)**

```
                SETAV       DESCR1,DESCR2
```

`SETAV` sets the address field of one descriptor from the value field of another.

Data Input to `SETAV`:

DESCR2			V

Data Altered by `SETAV`:

DESCR1	V	0	0

Programming Notes:

1. See also `SETAC`

101. `SETF` (set flag)

```
                SETF          DESCR,FLAG
```

`SETF` is used to set (add) a flag in the flag field of `DESCR`.

Data Input to `SETF`:

`DESCR`		`F`	

Data Altered by `SETF`:

`DESCR`		`F+FLAG`	

Programming Notes:

1. `FLAG` is added to the flags already present in `F`. The other flags are left unchanged.

2. If `F` already contains `FLAG`, no data is altered.

3. See also `SETFI`.

102. `SETFI` (set flag indirect)

```
                SETFI         DESCR,FLAG
```

`SETFI` is used to set (add) a flag in the flag field of a descriptor specified indirectly.

Data Input to `SETFI`:

`DESCR`	`A`		
`A`		`F`	

Data Altered by `SETFI`:

`A`		`F+FLAG`	

Programming Notes:

1. `FLAG` is added to the flags already present in `F`. The other flags are left unchanged.

2. If `F` already contains `FLAG`, no data is altered.

3. See also `SETF` and `RSETFI`.

---

103. `SETLC` **(set length of specifier to constant)**

```
             SETLC        SPEC,N
```

`SETLC` is used to set the length of a specifier to a constant.

Data Altered by `SETLC`:

SPEC					N

Programming Notes:

1. `N` is never negative.

2. `N` is often 0.

3. See also `SETAC`.

---

104. `SETSIZ` **(set size)**

```
             SETSIZ       DESCR1,DESCR2
```

`SETSIZ` is used to set the size into the value field of a title descriptor.

Data Input to `SETSIZ`:

DESCR1	A		
DESCR2	I		

Data Altered by `SETSIZ`:

A			I

Programming Notes:

1. `I` is always positive and small enough to fit into the value field.
2. See also `GETSIZ`

---

105. `SETSP` (set specifier)

```
                SETSP       SPEC1,SPEC2
```

`SETSP` is used to set one specifier equal to another.

Data Input to `SETSP`:

SPEC2	A	F	V	O	L

Data Altered by `SETSP`:

SPEC1	A	F	V	O	L

---

106. `SETVA` (set value field from address)

```
                SETVA       DESCR1,DESCR2
```

`SETVA` is used to set the value field of one descriptor from the address field of another.

Data Input to `SETVA`:

DESCR2	I		

Data Altered by `SETVA`:

DESCR1			I

Programming Notes:

1. `I` is always positive and small enough to fit into the value field.
2. See also `SETVA` and `SETVC`.

107. SETVC (set value to constant)

```
                SETVC       DESCR,N
```

SETVC is used to set the value field of a descriptor to a constant.

Data Altered by SETVC:

DESCR			N

Programming Notes:

1. N is always positive and small enough to fit into the value field.

2. See also SETVA and SETAC.

108. SHORTN (shorten specifier)

```
                SHORTN      SPEC,N
```

SHORTN is used to shorten the specification of a string.

Data Input to SHORTN:

SPEC					L

Data Altered by SHORTN:

SPEC					L-N

Programming Notes:

1. L-N is never negative.

109. SPCINT (convert specifier to integer)

```
                SPCINT      DESCR,SPEC,FLOC,SLOC
```

SPCINT is used to convert a specified string to a integer. I(S) is a signed integer resulting from the conversion of the string C1...CL. If C1...CL does not represent an integer or if the integer it represents is too large to fit the address field, transfer is to FLOC. Otherwise transfer is to SLOC.

Data Input to SPCINT:

SPEC	A			O	L

A+O	C1	...	CL

Data Altered by SPCINT:

DESCR	I(S)	0	I

Programming Notes:

1. I is a symbol defined in the source program and is the code for the integer data type.

2. C1...CL may begin with a sign (plus or minus) and may contain indefinite number of leading zeros. Consequently the value of L itself does not determine whether the integer represented is too large to fit into an address field.

3. A sign alone is not a valid integer.

4. If L = 0, I(S) should be the integer 0.

5. See also INTSPC and SPREAL.

---

110. SPEC (assemble specifier)

```
LOC           SPEC          A,F,V,O,L
```

SPEC is used to assemble a specifier.

Data Assembled by SPEC:

LOC	A	F	V	O	L

---

111. SPOP (pop specifier from stack)

```
              SPOP          (SPEC1,...,SPECN)
```

SPOP is used to pop a list of specifiers from the system stack.

Data Input to SPOP:

CSTACK	A				
A+D-S	A1	F1	V1	O1	L1
			.		
			.		
			.		
A+D-(N*S)	AN	FN	VN	ON	LN

Data Altered by SPOP:

CSTACK	A-(N*S)				
SPEC1	A1	F1	V1	O1	L1
			.		
			.		
			.		
SPECN	AN	FN	VN	ON	LN

Programming Notes:

1. If `A-(N*S) < STACK`, stack underflow occurs. This condition indicates a programming error in the implementation of the macro language. An appropriate error termination for this error may be obtained by transferring to the program location `INTR10` if the condition is detected.

2. See also `POP`, `SPUSH`, and `PUSH`.

---

112. SPREAL (convert specified string to real number)

```
            SPREAL      DESCR,SPEC,FLOC,SLOC
```

SPREAL is used to convert a specified string into a real number. `R(S)` is a signed real number resulting from the conversion of the string `S = C1`. If `C1...CL` does not represent a real number, or if the real number it represents is out of the range available for real numbers, transfer is to `FLOC`. Otherwise transfer is to `SLOC`.

Data Input to SPREAL:

SPEC	A			O	L
A+O	C1	...	CL		

Data Altered by `SPREAL`:

DESCR	R(S)	0	R

Programming Notes:

1. `R` is a symbol defined in the source program and is the code for the real data type.

2. `C1,...,CL` may begin with a sign (plus or minus) and may contain an indefinite number of leading zeros. `C1,...,CL` will contain a decimal point if it represents a real number, and have at least one digit before the decimal point.

3. If `L = 0`, `R(S)` should be the real number 0.0.

4. See also `SPCINT` and `INTRL`.

---

113. `SPUSH` (push specifiers onto stack)

```
            SPUSH        (SPEC1,...,SPECN)
```

`SPUSH` is used to push a list of specifiers onto the system stack.

Data Input to `SPUSH`:

CSTACK	A				
SPEC1	A1	F1	V1	O1	L1
			.		
			.		
			.		
SPECN	AN	FN	VN	ON	LN

Data Altered by `SPUSH`:

CSTACK	A+(S*N)				
A+D	A1	F1	V1	O1	L1
			.		
			.		
			.		
A+D+S*N-S	AN	FN	VN	ON	LN

Programming Notes:

1. If `A+(S*N) > STACK+STSIZE`, stack overflow occurs. Transfer should be made to the program location `OVER`, which will result in an appropriate error termination.

2. See also `PUSH`, `POP`, and `SPOP`.

---

114. `STPRNT` **(string print)**

```
              STPRNT       DESCR1,DESCR2,SPEC
```

`STPRNT` is used to print a string. The string `C11...C1L` is printed on the file associated with unit reference number `I`. `C21...C2M` is the output format. `J` is an integer specifying a condition signaled by the output routine.

Data Input to `STPRNT`:

DESCR2	A				
A+D	I				
A+2D	A2				
A2			M		
A2+4D	C21	...	C2M		
SPEC	A1			O1	L
A1+O1	C11	...	C1L		

Data Altered by `STPRNT`:

DESCR1	J		

Programming Notes:

1. The format `C21...C2M` is a FORTRAN IV format in ‘‘‘undigested’’’ form. See `FORMAT`.

2. Both `C11...C1L` and `C21...C2M` begin at descriptor boundaries.

3. The condition `J` set in the address field of `DESCR1` is not used.

4. See also `OUTPUT` and `STREAD`.

115. STREAD (string read)

```
            STREAD     SPEC,DESCR,EOF,ERROR,SLOC
```

STREAD is used to read a string. The string C1...CL is read from the file associated with unit reference number I. If an end-of-file is encountered, transfer is to EOF. If a reading error occurs, transfer is to ERROR. Otherwise transfer is to SLOC.

Data Input to STREAD:

DESCR	I				
SPEC	A			O	L

Data Altered by STREAD:

A+O	C1	...	CL

Programming Notes:

1. Note that the length of the string to be read is specified by the data provided to STREAD. If the record read is not of length L, FORTRAN IV conventions regarding truncation or reading of additional records should be followed.

2. See also STPRNT.

116. STREAM (stream for token)

```
            STREAM     SPEC1,SPEC2,TABLE,ERROR,RUNOUT,SLOC
```

STREAM is used to locate a syntactic token at the beginning of the string specified by SPEC2. If there is an I (1 ≤ I ≤ L) such that TI is ERROR, STOP, or STOPSH, and J is the least such I, then if TJ is ERROR, transfer is to ERRROR, while if if TJ is STOPSH, transfer is to SLOC. Otherwise transfer is to RUNOUT.

In the figures that follow, J is the least value of I for which TI is STOP or STOPSH. P is the last value of P (1 ≤ I ≤ J) that is nonzero (i.e. for which a PUT is specified in the syntax table description for the tables given). If no PUT is specified, P is zero.

Data Input to STREAM:

SPEC2	A	F	V	O	L	
A+O	C1	...	CJ	CJ+1	...	CL
TABLE+E*C1	A2	T1	P1			
A2+E*C2	A3	T2	P2			
		.				
		.				
		.				
AL+E*CL		TL	PL			

Data Altered by STREAM if Termination is STOP:

STYPE	P				
SPEC1	A	F	V	O	J
SPEC2	A	F	V	O+J	L-J

Data Altered by STREAM if Termination is STOPSH:

STYPE	P				
SPEC1	A	F	V	O	J-1
SPEC2	A	F	V	O+J-1	L-J+1

Data Altered by STREAM if Termination is ERROR:

STYPE	0				
SPEC1	A	F	V	O	L

Data Altered by STREAM if Termination is RUNOUT:

STYPE	P				
SPEC1	A	F	V	O	L
SPEC2	A	F	V	O	0

Programming Notes:

1. Termination with `STOP` or `STOPSH` may occur on the last character, `CL`.

2. If `L = 0` (i.e. if `SPEC2` specifies the null string), `RUNOUT` occurs. In this case the address field of `STYPE` should be set to 0.

3. See Section 4.2.

---

117. `STRING` **(assemble specified string)**

```
LOC          STRING      'C1...CL'
```

`STRING` is used to assemble a string and a specifier to it.

Data Assembled by `STRING`:

LOC	A	0	0	0	L
A	C1	...	CL		

Programming Notes:

1. Note that `LOC` is the location of the specifier, not the string. The string may immediately follow the specifier, or it may be assembled at a remote location.

---

118. `SUBSP` **(substring specification)**

```
             SUBSP       SPEC1,SPEC2,SPEC3,FLOC,SLOC
```

`SUBSP` is used to specify an initial substring of a specified string. If `L3 ≥ L2`, transfer is to `SLOC`. Otherwise transfer is to `FLOC` and `SPEC1` is not altered.

Data Input to `SUBSP`:

SPEC2					L2
SPEC3	A3	F3	V3	O3	L3

Data Altered by `SUBSP` if `L3 ≥ L2`:

SPEC1	A3	F3	V3	O3	L2

119. SUBTRT (subtract addresses)

```
                SUBTRT       DESCR1,DESCR2,DESCR3,FLOC,SLOC
```

SUBTRT is used to subtract one address field from another. A2 and A3 are considered as signed integers. If A2-A3 is out of the range available for integers, transfer is to FLOC. Otherwise transfer is to SLOC.

Data Input to SUBTRT:

DESCR2	A2	F2	V2
DESCR3	A3		

Data Altered by SUBTRT:

DESCR1	A2-A3	F2	V2

Programming Notes:

1. A2 and A3 may be relocatable addresses.

2. The test for success and failure is used in only one call of this macro. Hence the code to make the check is not needed in most cases.

3. DESCR1 and DESCR2 are often the same.

4. See also SUM.

120. SUM (sum addresses)

```
                SUM          DESCR1,DESCR2,DESCR3,FLOC,SLOC
```

SUM is used to add two address fields. A and I are considered as signed integers. If A+I is out of the range available for integers, transfer is to FLOC. Otherwise transfer is to SLOC.

Data Input to SUM:

DESCR2	A	F	V
DESCR3	I		

Data Altered by SUM:

DESCR1	A+I	F	V

Programming Notes:

1. A may be a relocatable address.

2. The test for success and failure is used in only one call of this macro. Hence the code to make the check is not needed in most cases.

3. DESCR1 and DESCR2 are often the same.

4. See also SUBTRT.

---

121. TESTF **(test flag)**

```
            TESTF       DESCR,FLAG,FLOC,SLOC
```

TESTF is used to test a flag field for the presence of a flag. If F contains FLAG, transfer is to SLOC. Otherwise transfer is to FLOC.

Data Input to TESTF:

DESCR		F	

Programming Notes:

1. See also TESTFI.

---

122. TESTFI **(test flag indirect)**

```
            TESTFI      DESCR,FLAG,FLOC,SLOC
```

TESTFI is used to test an indirectly specified flag field for the presence of a flag. If F contains FLAG, transfer is to SLOC. Otherwise transfer is to FLOC.

Data Input to TESTFI:

DESCR	A		
A		F	

Programming Notes:

1. See also TESTF.

123. `TITLE` (title assembly listing)

```
              TITLE       'C1...CN'
```

`TITLE` is used at assembly time to title the assembly listing of the SNOBOL4 system. `TITLE` should cause a page eject and title subsequent pages with `C1...CN`.

Programming Notes:

1. `TITLE` need not be implemented as such. It may simply perform no operation.

124. `TOP` (get to top of block)

```
              TOP         DESCR1,DESCR2,DESCR3
```

`TOP` is used to get to the top of a block of descriptors. Descriptors at `A`, `A-D`,...,`A-(N*D)` are examined successively for the first descriptor whose flag field contains the flag `TTL`. Data is altered as indicated, where `F3N` is the first field to contain `TTL`.

Data Input to `TOP`:

`DESCR3`	`A`	`F`	`V`
`A-(N*D)`		`F3N`	
		. . .	
`A-D`		`F31`	
`A`		`F30`	

Data Altered by `TOP`:

`DESCR1`	`A-(N*D)`	`F`	`V`
`DESCR2`	`N*D`	`0`	`0`

Programming Notes:

1. `N` may be 0. That is, `F30` may contain `TTL`.

125. `TRIMSP` (trim blanks from specifier)

```
            TRIMSP     SPEC1,SPEC2
```

`TRIMSP` is used to obtain a specifier to the part of a specified string up to a trailing string of blanks.

Data Input to `TRIMSP`:

SPEC2	A	F	V	O	L

A+O	C1	...	CJ	CJ+1	...	CL

Data Altered by `TRIMSP`:

SPEC1	A	F	V	O	J

Programming Notes:

1. If `CL` is not blank, `J = L`.

2. If `L = 0`, `TRIMSP` is equivalent to `SETSP`.

126. `UNLOAD` (unload external function)

```
            UNLOAD     SPEC
```

`UNLOAD` is used to unload an external function. `C1...CL` represents the name of the function that is to be unloaded.

Data Input to `UNLOAD`:

SPEC	A			O	L

A+O	C1	...	CL

Programming Notes:

1. `UNLOAD` is a system-dependent operation.

2. `UNLOAD` need not be implemented as such. If it is not, it should perform no operation, since the SNOBOL function `UNLOAD`, which uses the macro `UNLOAD`, has a valid use in undefining existing, but non-external, functions.

3. `UNLOAD` should do nothing if the function `C1...CL` is not a `LOAD`ed function.

4. See also `LOAD` and `LINK`.

127. VARID (compute variable identification numbers)

```
            VARID       DESCR,SPEC
```

VARID is used to compute two variable identification numbers from a specified string. K and M are computed by

```
K = F1(C1...CL)
M = F2(C1...CL)
```

where F1 and F2 are two (different) functions that compute pseudo-random numbers from the characters C1...CL. The numbers computed should be in the ranges

```
0 ≤ K ≤ (OBSIZ-1)*D
0 ≤ M ≤ SIZLIM
```

where OBSIZ is a program symbol defining the number of chains in variable storage and SIZLIM is a program symbol defining the largest integer that can be stored in the value field of a descriptor.

Data Input to VARID:

SPEC	A			O	L

A+O	C1	...	CL

Data Altered by VARID:

DESCR	K		M

Programming Notes:

1. K is used to select one of a number of chains in variable storage. The K are address offsets that must fall on descriptor boundaries.

2. M is used to order variables (string structures) within a chain. See ORDVST.

3. The values of K and M should have as little correlation as possible with the characters C1...CL, since the ‘‘‘randomness’’’ of the results determines the efficiency of variable access.

4. One simple algorithm consists of multiplying the first part of C1...CL by the last part, and separating the central portion of the result into K and M.

5. L is always greater than zero.

128. VCMPIC (value field compare indirect with offset constant)

```
            VCMPIC      DESCR1,N,DESCR2,GTLOC,EQLOC,LTLOC
```

`VCMPIC` is used to compare a value field, indirectly specified with an offset constant, with another value field. `V1` and `V2` are considered as unsigned integers. If `V1 > V2`, transfer is to `GTLOC`. If `V1 = V2`, transfer is to `EQLOC`. If `V1 < V2`, transfer is to `LTLOC`.

Data Input to `VCMPIC`:

DESCR1	A1		
DESCR2			V2
A1+N			V1

---

129. `VEQL` **(value fields equal test)**

```
            VEQL        DESCR1,DESCR2,NELOC,EQLOC
```

`VEQL` is used to compare the value fields of two descriptors. `V1` and `V2` are considered as unsigned integers. If `V1 = V2`, transfer is to `EQLOC`. Otherwise transfer is to `NELOC`.

Data Input to `VEQL`:

DESCR1			V1
DESCR2			V2

Programming Notes:

1. See also `AEQL` and `VEQLC`.

---

130. `VEQLC` **(value field equal to constant test)**

```
            VEQLC       DESCR,N,NELOC,EQLOC
```

`VEQLC` is used to compare the value field of a descriptor to a constant. `V` is considered as an unsigned integer. If `V = N`, transfer is to `EQLOC`. Otherwise transfer is to `NELOC`.

Data Input to `VEQLC`:

DESCR			V

Programming Notes:

1. `N` is never negative.

2. See also `AEQLC` and `VEQL`.

---

131. `ZERBLK` **(zero block)**

```
                 ZERBLK        DESCR1,DESCR2
```

`ZERBLK` is used to zero a block of `I+1` descriptors.

Data Input to `ZERBLK`:

DESCR1	A		
DESCR2	D*I		

Data Altered by `ZERBLK`:

A	0	0	0
		.	
		.	
		.	
A+(D*I)	0	0	0

Programming Notes:

1. `I` is always positive.

7. Implementation Notes

7.1. Optional Macros

There are several macros that are used in noncritical parts of the SNOBOL4 language. Some macros are used only to implement certain built-in functions. Others are required only for minor executive operations. The following list includes macros for which implementation is optional. For these macros, simple alternative implementations are suggested and the language features disabled are indicated. In selecting macros for inclusion in this list, a judgement was made concerning what features could be disabled and still leave SNOBOL4 a useful language.

Macro	*Alternative Implementation*	*Features Disabled*
ADREAL1—	Branch to INTR10	Real arithmetic
BKSPCE	Branch to UNDF	The function BACKSPACE
CLERTB2—	Branch to UNDF	The functions ANY, NOTANY, SPAN, and BREAK
DATE	Set length of SPEC to 0	The function DATE
DVREAL1—	Set address of DESCR2 to 0	Real arithmetic and post-run statictics
ENFILE	Branch to UNDF	The function ENDFILE
EXPINT	Branch to UNDF	Exponentiation of integers
EXREAL1—	Branch to INTR10	Real arithmetic
GETBAL	Branch to UNDF	The built-in pattern BAL
INTRL1—	Perform no operation	Real arithmetic
LEXCMP3—	If GTLOC ≠ LTLOC, branch to UNDF	The function LGT
LINK4—	Branch to INTR10	External functions
LOAD4—	Branch to UNDF	External functions
MNREAL1—	Branch to INTR10	Real arithmetic
MPREAL1—	Branch to INTR10	Real arithmetic
MSTIME	Set address of DESCR to 0	The function TIME, trace timing, post-run statistics
ORDVST	Perform no operation	Alphabetization of post-run dump
PLUGTB2—	Branch to INTR10	The functions ANY, NOTANY, SPAN,and BREAK
RCOMP1—	Branch to INTR10	Real arithmetic
REALST1—	Branch to UNDF	Real arithmetic
REWIND	Branch to INTR10	The function REWIND
RLINT1—	Branch to INTR10	Real arithmetic
RPLACE	Branch to INTR10	The function REPLACE
SBREAL1—	Branch to INTR10	Real arithmetic
SPREAL1—	Take the FAILURE exit	Real arithmetic
TRIMSP	Branch to INTR10	The function TRIM

1—All operations relating to real arithmetic should be implemented or not implemented as a group.

2—CLERTB and PLUGTB should be implemented or not implemented as a pair.

3—LEXCMP must be properly implemented if LTLOC is the same as GTLOC.

4—LINK, LOAD, and UNLOAD should be implemented or not implemented as a group.

UNLOAD4—	Perform no operation	External functions

7.2. Machine-Dependent Data

In addition to the data given in the COPY files (q.v.) there are several format strings that generally have to be changed to suit a particular machine. The strings defined by FORMAT (which occur at the end of the source file) are in this category. The two strings CRDFSP and OUTPSP defined by STRING are also machine dependent.

7.3. Error Exits for Debugging

During the debugging phases, it is good programming practice to test for certain conditions that should not occur, but typically do if there is an error in the implementation. Stack underflow is typical. Transfer to the label INTR10 upon recognition of such an error causes the SNOBOL4 run to terminate with the message ERROR IN SNOBOL4 SYSTEM. Following this message, the statement number in which the error occurred is printed, as well as requested dumps and termination statistics that may be helpful in debugging.

7.4. Subroutines Versus In-Line Code

The choice between implementing macro operations by subroutine calls or in-line code depends on a number of factors, including the machine and its environment. The size of the SNOBOL4 system usually encourages subroutine implementations of the more complicated operations. The following information, obtained by program analysis and dynamic performance measurements, may be helpful in making these decisions. Column 1 lists the macro operations in alphabetical order, including non-executable macros. Column 2 gives the number of times each each macro operation occurs in the SNOBOL4 program. Column 3 gives the percentage of time spent in each (executable) macro during execution of a typical set of programs on the IBM System/360 implementation. Time spent in I/O and operating system subroutines is not included. A * marks those macros that are implementated by subroutines in the IBM System/360 implementation (including macros that call I/O and system subroutines).

ACOMP	65	2.952
ACOMPC	61	1.450
ADDLG	8	0.000
ADDSIB	6	0.000
ADDSON	12	0.017
ADJUST	2	0.000
ADREAL	1	0.000
AEQL	18	0.397
AEQLC	177	3.574
AEQLIC	10	0.086
APDSP*	93	0.897
ARRAY	5	————
BKSIZE	5	1.329
BKSPCE*	1	0.000
BRANCH	354	0.638
BRANIC	5	2.054
BUFFER	5	————
CHKVAL	4	0.604
CLERTB	4	0.000
COPY	3	————
CPYPAT*	14	3.021
DATE*	1	0.000
DECRA	66	1.588
DEQL	73	1.346
DESCR	920	————
DIVIDE	4	0.000

DVREAL	2	0.000
END	1	———–
ENDEX*	1	0.000
ENFILE*	1	0.000
EQU	69	———–
EXPINT	1	0.000
EXREAL*	1	0.000
FORMAT	26	———–
FSHRTN	12	0.000
GETAC	10	0.638
GETBAL*	1	0.172
GETD	53	7.408
GETDC	113	5.025
GETLG	59	0.759
GETLTH	2	0.172
GETSIZ	28	0.397
GETSPC	10	0.017
INCRA	140	5.577
INCRV	1	0.000
INIT*	1	0.138
INSERT	1	0.000
INTRL	7	0.000
INTSPC*	25	0.552
ISTACK	2	0.000
LCOMP	5	0.000
LEQLC	18	0.103
LEXCMP*	12	2.624
LHERE	14	———–
LINK*	1	0.000
LINKOR	1	0.000
LOAD*	1	0.000
LOCAPT	21	1.467
LOCAPV	32	5.197
LOCSP	80	1.605
LVALUE*	6	0.207
MAKNOD	13	0.172
MNREAL	1	0.000
MNSINT	1	0.034
MOVA	7	0.397
MOVBLK*	13	0.103
MOVD	155	1.985
MOVDIC	7	0.017
MOVV	16	0.811
MPREAL	1	0.000
MSTIME*	8	0.000
MULT	6	0.120
MULTC	18	0.207
ORDVST*	1	0.000
OUTPUT*	28	0.034
PLUGTB	4	0.000
POP	118	4.282
PROC	173	2.365
PSTACK	5	0.034
PUSH	124	3.091

PUTAC	11	0.448
PUTD	33	0.069
PUTDC	126	3.056
PUTLG	9	0.189
PUTSPC	1	0.138
PUTVC	1	0.034
RCALL	342	8.927
RCOMP	6	0.000
REALST*	10	0.000
REMSP	7	0.448
RESETF	3	0.000
REWIND*	1	0.000
RLINT	2	0.000
RPLACE*	1	0.000
RRTURN	21	6.182
RSETFI	2	0.000
SBREAL	1	0.000
SELBRA	18	0.017
SETAC	169	0.673
SETAV	33	1.830
SETF	1	0.000
SETFI	5	0.086
SETLC	28	0.034
SETSIZ	7	0.155
SETSP	23	0.155
SETVA	14	0.051
SETVC	28	0.207
SHORTN	4	0.000
SPCINT*	24	0.069
SPEC	30	————
SPOP	4	0.000
SPREAL*	13	0.000
SPUSH	4	0.000
STPRNT*	15	0.051
STREAD*	4	0.051
STREAM*	35	0.656
STRING	152	————
SUBSP	3	0.362
SUBTRT	22	0.189
SUM	67	1.709
TESTF	24	1.899
TESTFI	9	0.707
TITLE	24	————
TOP	4	0.241
TRIMSP	2	0.069
UNLOAD*	1	0.000
VARID	1	0.897
VCMPIC	1	0.535
VEQL	3	2.158
VEQLC	106	0.759
ZERBLK	3	0.128

7.5. Classification of Macro Operations

In the following sections, the macro operations are classified according to the way they are used.

Assembly Control Macros:

COPY	END	EQU	LHERE	TITLE

Macros that Assemble Data:

ARRAY	BUFFER	DESCR	FORMAT	SPEC
STRING				

Branch Macros:

BRANCH	BRANIC	SELBRA

Comparison Macros:

ACOMP	ACOMPC	AEQL	AEQLC	AEQLIC
CHKVAL	DEQL	LCOMP	LEQLC	LEXCMP
RCOMP	TESTF	TESTFI	VCMPIC	VEQL
VEQLC				

Macros that Relate to Recursive Procedures and Stack Management:

ISTACK	POP	PROC	PSTACK	PUSH
RCALL	RRTURN	SPOP	SPUSH	

Macros that Move and Set Descriptors:

GETD	GETDC	MOVBLK	MOVD	MOVDIC
POP	PUSH	PUTD	PUTDC	ZERBLK

Macros that Modify Address Fields of Descriptors:

ADJUST	BKSIZE	DECRA	GETAC	GETLG
GETLTH	GETSIZ	INCRA	MOVA	PUTAC
SETAC	SETAV			

Macros that Modify Value Fields of Descriptors:

INCRV	MOVV	PUTVC	SETSIZ	SETVA
SETVC				

Macros that Modify Flag Fields of Descriptors:

RESETF	RSETFI	SETF	SETFI

Macros that Perform Integer Arithmetic on Address Fields:

DECRA	DIVIDE	EXPINT	INCRA	MNSINT
MULT	MULTC	SUBTRT	SUM	

Macros that Deal with Real Numbers:

```
ADREAL      DVREAL      EXREAL      INTRL       MNREAL
MPREAL      RCOMP       REALST      RLINT       SBREAL
SPREAL
```

Macros that Move Specifiers:

```
GETSPC      PUTSPC      SETSP       SPOP        SPUSH
```

Macros that Operate on Specifiers:

```
ADDLG       APDSP       FSHRTN      GETBAL      INTSPC
LOCSP       PUTLG       REMSP       SETLC       SHORTN
STREAM      SUBSP       TRIMSP
```

Macros that Operate on Syntax Tables:

```
CLERTB      PLUGTB
```

Macros that Construct Pattern Nodes:

```
CPYPAT      MAKNOD
```

Macros that Operate on Tree Nodes:

```
ADDSIB      ADDSON      INSERT
```

Input and Output Macros:

```
BKSPCE      ENFILE      FORMAT      OUTPUT      REWIND
STPRNT      STREAD
```

Macros that Depend on Operating System Facilities:

```
DATE        ENDEX       INIT        LINK        LOAD
MSTIME      UNLOAD
```

Miscellaneous Macros:

```
LINKOR      LOCAPT      LOCAPV      LVALUE      ORDVST
RPLACE      SPCINT      TOP         VARID
```

7.6. Format of the SNOBOL4 Source File

One problem in implementing SNOBOL4 for a particular machine involves putting the macro language program into a form suitable for the assembler for that machine. This typically involves making a number of format changes and correcting a few special cases by hand. It is desirable to perform as many changes as possible by some systematic, mechanical means (preferably with a program) so that new versions of the macro language program can be converted into the required form easily, thus facilitating the incorporation of updates in the SNOBOL4 language. A systematic, mechanical technique also minimizes random errors inevitably introduced by human interference. Such random errors are particularly dangerous in such an implementation, since most of the logic of the system is at a level divorced from the implementation of the macro language. This section describes the format of the macro language program in order to make the necessary format changes easier to determine.

The SNOBOL4 assembly source file consists of 6611 80 character card images. All card images are blank in column 72 and contain sequence numbering in columns 73 through 80. Updates to the source file are given in terms of these sequence numbers, so care should be taken not to destroy this information. There are two kinds of card images: program text and comments. Comments have an asterisk (*) in column 1 and descriptive text of various types in columns 2 through 71. All other card images (about 4850 out of the total of 6611) are program text. Program text has a field format as follows:

1. Columns 1 through 6: label field. A program label, if present, begins in column 1. All labels begin with a letter, followed by letters or digits. Labels are from two through six characters in length. If a program card has no label, the label field is blank.

2. Column 7: blank.

3. Columns 8 through 13: operation field. Program text has operations that begin in column 8. Operations consist of from three to six letters.

4. Columns 14 and 15: blank.

5. Columns 16 through 71: variable field. A list of operands appears in the variable field starting in column 16. The list consists of items separated by commas. The last item in the list is followed by a blank. If there are no operands, there is a comma in column 16 and a blank in column 17. Items in the operand list may take several forms:

a. Identifiers, which satisfy the requirements of program labels.

b. Integer constants.

c. Arithmetic expressions containing identifiers and constants.

d. Lists of items enclosed in parentheses. Lists are not nested, i.e. lists do not occur as items within lists.

e. Character literals, consisting of characters enclosed in single quotation marks. Quotation marks do not occur within literals, but commas, parentheses, and blanks may. This fact must be taken into account in analyzing the variable field.

f. Nulls, or items of zero length. Nulls represent explicitly omitted arguments to macro operations.

Comments may occur following the blank that terminates the variable field. Such comments begin in column 36 or subsequently.

The following portion of program is typical.

```
*________________________________________________________________*                 00000821
*                                                                                  00000822
*         Block Marking                                                            00000823
*                                                                                  00000824
GCM       PROC      ,                        Procedure to mark blocks              00000825
          POP       BK1CL                    Restore block to mark from            00000826
          PUSH      ZEROCL                   Save end marker                       00000827
GCMA1     GETSIZ    BKDX,BK1CL               Get size of block                     00000828
GCMA2     GETD      DESCL,BK1CL,BKDX         Get descriptor                        00000829
          TESTF     DESCL,PTR,GCMA3          Is it a pointer?                      00000830
          AEQLC     DESCL,0,,GCMA3           Is address zero?                      00000831
          TOP       TOPCL,OFSET,DESCL        Get to title of block pointed to      00000832
          TESTFI    TOPCL,MARK,GCMA4         Is block marked?                      00000833
GCMA3     DECRA     BKDX,DESCR               Decrement offset                      00000834
          AEQLC     BKDX,0,GCMA2             Check for end of block                00000835
          POP       BK1CL                    Restore block pushed                  00000836
          AEQLC     BK1CL,0,,RTN1            Check for end                         00000837
          SETAV     BKDX,BK1CL               Get size remaining                    00000838
```

```
         BRANCH    GCMA2                    Continue processing
*_
GCMA4    DECRA     BKDX,DESCR               Decrement offset
         AEQLC     BKDX,0,,GCMA9            Check for end
         SETVA     BK1CL,BKDX               Insert offset
         PUSH      BK1CL                    Save current block
GCMA9    MOVD      BK1CL,TOPCL              Set poiner to new block
         SETFI     BK1CL,MARK               Mark block
         TESTFI    BK1CL,STTL,GCMA1         Is it a string?
         MOVD      BKDX,TWOCL               Set size of string to 2
         BRANCH    GCMA2                    Join processing
*_
```

Acknowledgement

The SIL version of SNOBOL4 was implemented jointly by the author, Jim Poage, and Ivan Polonsky. Other individuals, too numerous to mention here, have provided many helpful criticisms and corrections of this document.

```
BEGIN BIOPTB
FOR(PLUS) PUT(ADDFN) GOTO(TBLKTB)
FOR(MINUS) PUT(SUBFN) GOTO(TBLKTB)
FOR(DOT) PUT(NAMFN) GOTO(TBLKTB)
FOR(DOLLAR) PUT(DOLFN) GOTO(TBLKTB)
FOR(STAR) PUT(MPYFN) GOTO(STARTB)
FOR(SLASH) PUT(DIVFN) GOTO(TBLKTB)
FOR(AT) PUT(BIATFN) GOTO(TBLKTB)
FOR(POUND) PUT(BIPDFN) GOTO(TBLKTB)
FOR(PERCENT) PUT(BIPRFN) GOTO(TBLKTB)
FOR(RAISE) PUT(EXPFN) GOTO(TBLKTB)
FOR(ORSYM) PUT(ORFN) GOTO(TBLKTB)
FOR(KEYSYM) PUT(BIAMFN) GOTO(TBLKTB)
FOR(NOTSYM) PUT(BINGFN) GOTO(TBLKTB)
FOR(QUESYM) PUT(BIQSFN) GOTO(TBLKTB)
ELSE ERROR
END BIOPTB

BEGIN CARDTB
FOR(CMT) PUT(CMTTYP) STOPSH
FOR(CTL) PUT(CTLTYP) STOPSH
FOR(CNT) PUT(CNTTYP) STOPSH
ELSE PUT(NEWTYP) STOPSH
END CARDTB

BEGIN DQLITB
FOR(DQUOTE) STOP
ELSE CONTIN
END DQLITB

BEGIN ELEMTB
FOR(NUMBER) PUT(ILITYP) GOTO(INTGTB)
FOR(LETTER) PUT(VARTYP) GOTO(VARTB)
FOR(SQUOTE) PUT(QLITYP) GOTO(SQLITB)
FOR(DQUOTE) PUT(QLITYP) GOTO(DQLITB)
FOR(LEFTPAREN) PUT(NSTTYP) STOP
ELSE ERROR
END ELEMTB

BEGIN EOSTB
FOR(EOS) STOP
ELSE CONTIN
END EOSTB

BEGIN FLITB
FOR(NUMBER) CONTIN
FOR(TERMINATOR) STOPSH
ELSE ERROR
END FLITB
```

```
BEGIN FRWDTB
FOR(BLANK) CONTIN
FOR(EQUAL) PUT(EQTYP) STOP
FOR(RIGHTPAREN) PUT(RPTYP) STOP
FOR(RIGHTBR) PUT(RBTYP) STOP
FOR(COMMA) PUT(CMATYP) STOP
FOR(COLON) PUT(CLNTYP) STOP
FOR(EOS) PUT(EOSTYP) STOP
ELSE PUT(NBTYP) STOPSH
END FRWDTB

BEGIN GOTFTB
FOR(LEFTPAREN) PUT(FGOTYP) STOP
FOR(LEFTBR) PUT(FTOTYP) STOP
ELSE ERROR
END GOTFTB

BEGIN GOTOTB
FOR(SGOSYM) GOTO(GOTSTB)
FOR(FGOSYM) GOTO(GOTFTB)
FOR(LEFTPAREN) PUT(UGOTYP) STOP
FOR(LEFTBR) PUT(UTOTYP) STOP
ELSE ERROR
END GOTOTB

BEGIN GOTSTB
FOR(LEFTPAREN) PUT(SGOTYP) STOP
FOR(LEFTBR) PUT(STOTYP) STOP
ELSE ERROR
END GOTSTB

BEGIN IBLKTB
FOR(BLANK) GOTO(FRWDTB)
FOR(EOS) PUT(EOSTYP) STOP
ELSE ERROR
END IBLKTB

BEGIN INTGTB
FOR(NUMBER) CONTIN
FOR(TERMINATOR) PUT(ILITYP) STOPSH
FOR(DOT) PUT(FLITYP) GOTO(FLITB)
ELSE ERROR
END INTGTB

BEGIN LBLTB
FOR(ALPHANUMERIC) GOTO(LBLXTB)
FOR(BLANK,EOS) STOPSH
ELSE ERROR
END LBLTB
```

```
BEGIN LBLXTB
FOR(BLANK,EOS) STOPSH
ELSE CONTIN
END LBLXTB

BEGIN NBLKTB
FOR(TERMINATOR) ERROR
ELSE STOPSH
END NBLKTB

BEGIN NUMBTB
FOR(NUMBER) GOTO(NUMCTB)
FOR(PLUS,MINUS) GOTO(NUMCTB)
FOR(COMMA) PUT(CMATYP) STOPSH
FOR(COLON) PUT(DIMTYP) STOPSH
ELSE ERROR
END NUMBTB

BEGIN NUMCTB
FOR(NUMBER) CONTIN
FOR(COMMA) PUT(CMATYP) STOPSH
FOR(COLON) PUT(DIMTYP) STOPSH
ELSE ERROR
END NUMCTB

BEGIN SNABTB
FOR(FGOSYM) STOP
FOR(SGOSYM) STOPSH
ELSE ERROR
END SNABTB

BEGIN SQLITB
FOR(SQUOTE) STOP
ELSE CONTIN
END SQLITB

BEGIN STARTB
FOR(BLANK) STOP
FOR(STAR) PUT(EXPFN) GOTO(TBLKTB)
ELSE ERROR
END STARTB

BEGIN TBLKTB
FOR(BLANK) STOP
ELSE ERROR
END TBLKTB
```

```
BEGIN UNOPTB
FOR(PLUS) PUT(PLSFN) GOTO(NBLKTB)
FOR(MINUS) PUT(MNSFN) GOTO(NBLKTB)
FOR(DOT) PUT(DOTFN) GOTO(NBLKTB)
FOR(DOLLAR) PUT(INDFN) GOTO(NBLKTB)
FOR(STAR) PUT(STRFN) GOTO(NBLKTB)
FOR(SLASH) PUT(SLHFN) GOTO(NBLKTB)
FOR(PERCENT) PUT(PRFN) GOTO(NBLKTB)
FOR(AT) PUT(ATFN) GOTO(NBLKTB)
FOR(POUND) PUT(PDFN) GOTO(NBLKTB)
FOR(KEYSYM) PUT(KEYFN) GOTO(NBLKTB)
FOR(NOTSYM) PUT(NEGFN) GOTO(NBLKTB)
FOR(ORSYM) PUT(BARFN) GOTO(NBLKTB)
FOR(QUESYM) PUT(QUESFN) GOTO(NBLKTB)
FOR(RAISE) PUT(AROWFN) GOTO(NBLKTB)
ELSE ERROR
END UNOPTB

BEGIN VARATB
FOR(LETTER) GOTO(VARBTB)
FOR(COMMA) PUT(CMATYP) STOPSH
FOR(RIGHTPAREN) PUT(RPTYP) STOPSH
ELSE ERROR
END VARATB

BEGIN VARBTB
FOR(ALPHANUMERIC,BREAK) CONTIN
FOR(LEFTPAREN) PUT(LPTYP) STOPSH
FOR(COMMA) PUT(CMATYP) STOPSH
FOR(RIGHTPAREN) PUT(RPTYP) STOPSH
ELSE ERROR
END VARBTB

BEGIN VARTB
FOR(ALPHANUMERIC,BREAK) CONTIN
FOR(TERMINATOR) PUT(VARTYP) STOPSH
FOR(LEFTPAREN) PUT(FNCTYP) STOP
FOR(LEFTBR) PUT(ARYTYP) STOP
ELSE ERROR
END VARTB
```

Appendix B — Available Implementation Material

There is a substantial amount of material available to the would–be installer of the SIL implementation of SNOBOL4. Much of the basic documentation is given in a book that is available through book suppliers. The rest of the material is available from the University of Arizona:

Ralph E. Griswold
Department of Computer Science
University Computer Center
The University of Arizona
Tucson, Arizona 85721
U.S.A.

telephone: (602) 626–1829

There is no charge for this material but magnetic tapes must be supplied with requests for machine–readable material.

Documents with identifying numbers should be requested by number.

1. Version 3.11 SIL source code and syntax table descriptions in machine–readable form. This material is available in a variety of tape formats. The standard distribution is 9–track, 1600 bpi, unlabeled fixed–blocked, EBCDIC.
2. S4D54c: *Transporting the SIL Version of SNOBOL4; An Overview*. Gives a brief description of the processing of implementing the SIL version of SNOBOL4; suggested reading prior to serious work on the implementation.
3. *The Macro Implementation of SNOBOL4; A Case Study of Machine–Independent Software Development*. (author: Ralph E. Griswold, publisher: W. H. Freeman & Co.) A description of the SIL version of SNOBOL4 that describes data structures, algorithms, the SIL macros, and gives examples from the IBM 360 and CDC 6000 implementations. This book is available from book sellers. The price is approximately $25.00. The terminology used in this book is different from that used in the actual SIL source. See S4D59 below.
4. *Corrigenda for The Macro Implementation of SNOBOL4*. Corrections to the Freeman book listed above.
5. S4D59: *Comparison of Terminologies for the SIL Implementation of SNOBOL4*. Explains the differences between terminology of the Freeman book and that actually used in the machine–readable SIL program.
6. S4D26c: *Source and Cross–Reference Listings for the SIL Implementation of SNOBOL4; Version 3.11*. Listing of SNOBOL4 written in SIL. This document is primarily useful for its cross reference to program symbols.
7. S4D20a: *IBM 360 Macro Definitions for Version 3 of SNOBOL4*. Listing of the IBM 360 macro definitions for SIL operations; primarily useful as an example of an existing implementation. The macro definitions are also available in machine–readable form.
8. S4D19a: *IBM 360 Subroutines for Version 3 of SNOBOL4*. Listing of the IBM 360 subroutines that support SIL operations; primarily useful as an example of an existing implementation. The subroutines are also available in machine–readable form.
9. S4D57: *Implementations of SNOBOL4*. Compilation of SNOBOL4 implementations, including those done in SIL; primarily useful as a source of contacts with other SIL implementors.