Fred Jacobs

A Guide to the Macro Implementation of SNOBOL4

R. E. Griswold

Bell Telephone Laboratories, Incorporated

July 16, 1971

S4D8d

****

This manual corresponds to Version 3 of SNOBOL4.

****

1. Introduction

The SNOBOL4 programming language [1] is implemented in macro-assembly language [2,3]. 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 implementation already written in the macro language, one obtains a version of SNOBOL4 which is largely source-language compatible with other versions implemented in the same way. Nearly all the logic of the SNOBOL4 langauge resides in the program written in the macro language. Thus if one implements the macro language properly, the resulting implementation of SNOBOL4 will be essentially the same as other such implementations.

This paper describes the macro language and contains information necessary for its implementation. Section 2 describes environmental considerations. Section 3 describes the representation of data. Syntax tables and character graphics are described in Section 4. Section 5 is a list of all macro operations with a description of how to implement each. Supplementary information is included in appendices.

2. Environmental Considerations

A. 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 this is handled as an integer which appears in the address fields of descriptors which are arguments to the I/O macros. Unit reference numbers are referred to symbolically in the SNOBOL4 assembly. See the PARMS file in the dicussion 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 include only literals, X, T, and A conversion. Generally speaking, formats occur in "undigested" form. Formats used by OUTPUT are assembled by the FORMAT 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 which may be formed during program execution and hence must be accepted in their undigested form.

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

Where possible, 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 will probably occur in handling undigested formats. When questions arise as to what an operation should do, FORTRAN conventions should be applied. A programmer should expect the same results from SNOBOL4 as from FORTRAN if, for example, he requests a string 200 characters from a file containing 80-character records.

B. 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 will permit execution of small programs. Of the space required, the SNOBOL4 system and its internal data consume about 99K bytes, the FORTRAN I/O routines consume about 14K bytes, and the remainder is devoted to dynamically allocated storage. Allocated storage is handled in machine-independent data units (see the next section) called descriptors which occupy 8 bytes each on the 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.

C. 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 will use dump facilities to provide core dumps requested by the keyword &ABEND if such facilities are available. Time and date are used by SNOBOL4, but they are 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

1. Descriptors.

2. Specifiers.

3. Character Strings.

4. Syntax Table Entries.

A. 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:

1. Address.

2. Flag.

3. Value.

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

i. The Address Field

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

ii. The Flag Field

The flag field is used to represent the state of a number of disjoint conditions and is treated as a set of bits which are individually tested, turned on and turned off. There are five flag bits currently used in SNOBOL4, but space is left for several more.

iii. The Value Field

The value field is used to represent a number of internal quantities which are represented as unsigned integers (magnitudes). These quantities include 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 big as the address field, but must be large enough to represent the size of the largest data aggregate which can be formed.

On the IBM System/360, a descriptor is two words (8 bytes). The first word is the address field. The second word consists of 1 byte for the flag field and 3 bytes for the value field. The 3 bytes (24 bits) for the value field permits representation of data objects as large as $2^{24}-1$ bytes. On the other hand, 2 bytes would limit objects to $2^{16}-1$ bytes. Since on the 360 there are 8 bytes per descriptor, $2^{16}-1$ bytes limits objects to 8191 descriptors which is restrictive. For machines with fewer address units per descriptor, the value field need not be as many bits.

B. 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:

1. Address.
2. Flag.
3. Value.
4. Offset.
5. 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.

C. Character Strings

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

D. 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

A. 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, the graphics used on the IBM System/360 are used as a point of reference.

name	language function	360 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	;)>,] (includes blank and tab)

B. Syntax Tables

The micro-syntax (or "fine structure") of the SNOBOL4 language is analyzed using the operation STREAM (q.v.) which is driven from syntax tables. 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 the next character.

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

The syntax tables for the IBM 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 complete list of syntax table descriptions follows:

```
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
```

Can be modified !

```
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
```

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.

5. The SNOBOL4 Macros

This section contains implementation instructions for each of the macros. The instructions for an operation usually consist of a description of the operation's function, figures indicating data relating to the operation, and programming notes which contain details, and references to other relevant information. The figures consist of stylized represenations of the various data objects and the fields within them.

A. Diagrammatic Representation of Data

Figure 1 is the representation of a descriptor at LOC1. A, F, and V indicate the values of the address, flag, and value fields.

LOC1	A	F	V

Figure 1. Representation of a Descriptor

Figure 2 is the representation of a specifier at LOC2. A, F, V, O, and L indicate the values of the address, flag, value, offset, and length fields.

LOC2	A	F	V	O	L

Figure 2. Representation of a Specifier

Character strings have two representations depending on how many characters are relevant to the description.

Figure 3 is the short representation of a string of L characters at LOC3. C1 and CL are the first and last characters respectively. In this representation, the intermediate characters are not indicated.

LOC3	C1	...	CL

Figure 3. Short Representation of a String

Figure 4 is the long representation of a string of L characters at LOC4. CJ and CJ+1 are relevant characters in the interior of the string. The long representation is used when interior characters must be referred to.

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

Figure 4. Long Representation of a String

Figure 5 is the representation of a syntax table entry. A, T, and P indicate the values of the next table address, type indicator, and put field.

LOC5	A	T	P

Figure 5. Representation of a Syntax Table Entry

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. Fields which are changed have their new value underlined to make such fields easier to locate. Only changed fields are underlined. For example Figure 6 shows a descriptor whose address field is changed. The new value of the address field is A2, and no other fields are changed.

A2		

Figure 6. An Altered Descriptor

Letters are used as abbreviations to differentiate the values which may appear in a field. The six basic fields are indicated by the letters A, F, V, O, L, and C. 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.

The reader should glance through the descriptions which follow to familiarize himself with the ways in which the figures and field notation are used.

B. 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 the macro descriptions which follow, such branch points are underlined in the prototype of the macro call. See ACOMP which follows.

In general, some or all of such 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,GT,EQ,LT
ACOMP   DESCR1,DESCR2,GT,EQ
ACOMP   DESCR1,DESCR2,GT
ACOMP   DESCR1,DESCR2,GT,,LT
ACOMP   DESCR1,DESCR2,,EQ,LT
ACOMP   DESCR1,DESCR2,,EQ
ACOMP   DESCR1,DESCR2,,,LT
```

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.

C. Abbreviations

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

D is used for the addressing width of a descriptor. On the IBM System/360, the machine addressing unit is 1 byte, and D is 8.

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

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

I is used for (signed) integers.

R is used for real numbers.

T is used for indicator entries in syntax tables

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

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

D. Data Type Codes

The SNOBOL4 system has data type codes assigned for integers and real numbers, among others. These codes are indicated in the macro descriptions by IC and RC respectively. The actual global symbols for these codes in the SNOBOL4 system are I and R respectively. The actual symbols are not used in the descriptions to avoid confusion with the abbreviations given above. However in the implementation of the macros, IC should be replaced by I and RC by R.

E. 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.

1. ACOMP (address comparison)

```
ACOMP   DESCR1,DESCR2,GT,EQ,LT
```

ACOMP is used to compare the address fields of two descriptors. See figure 7. The comparison is arithmetic with A1 and A2 being considered as signed integers:

If A1 > A2 transfer is to GT.

If A1 = A2 transfer is to EQ.

If A1 < A2 transfer is to LT.

DESCR1	A1		
DESCR2	A2		

Figure 7. Data Input to ACOMP

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,GT,EQ,LT
```

ACOMPC is used to compare the address field of a descriptor to a constant. See figure 8. The comparison is arithmetic with A being considered as a signed integer.

If A > N transfer is to GT.

If A = N transfer is to EQ.

If A < N transfer is to LT.

DESCR	A		

Figure 8. Data Input to ACOMPC

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. See figures 9 and 10.

SPEC					L
DESCR	I				

Figure 9. Data Input to ADDLG

SPEC					<u>L+I</u>

Figure 10. Data Altered by ADDLG

<u>Programming Notes</u>

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. See figures 11 and 12.

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

Figure 11. Data Input to ADDSIB

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

Figure 12. Data Altered by ADDSIB

Programming Notes

1. ADDSIB is only used by compilation procedures.
2. 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. See figures 13 and 14.

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

Figure 13. Data Input to ADDSON

A2+FATHER	<u>A1</u>	<u>F1</u>	<u>V1</u>
A2+RSIB	<u>A3</u>	<u>F3</u>	<u>V3</u>
A1+LSON	<u>A2</u>	<u>F2</u>	<u>V2</u>
A1+CODE			<u>I+1</u>

Figure 14. Data Altered by ADDSON

Programming Notes

1. ADDSON is only used by compilation procedures.

2. 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. See figures 15 and 16.

DESCR2	A2		
DESCR3	A3		
A2	A4		

Figure 15. Data Input to ADJUST

DESCR1	A3+A4		

Figure 16. Data Altered by ADJUST

Programming Notes

1. A3 is always an address integer.

7. ADREAL (add real numbers)

```
        ADREAL   DESCR1,DESCR2,DESCR3,FAILURE,SUCCESS
```

ADREAL is used to add two real numbers. See figures 17 and 18.

If the result is out of the range available for real numbers, transfer is to FAILURE.

Otherwise transfer is to SUCCESS.

DESCR2	R2	F2	V2
DESCR3	R3		

Figure 17. Data Input to ADREAL

DESCR1	R2+R3	F2	V2

Figure 18. Data Altered by ADREAL

Programming Notes

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

8. AEQL (addresses equal test)

```
AEQL     DESCR1,DESCR2,NE,EQ
```

AEQL is used to compare the address fields of two descriptors. See figure 19. The comparison is arithmetic with A1 and A2 being considered as signed integers:

If A1 = A2 transfer is to EQ.

If A1 ≠ A2 transfer is to NE.

DESCR1	A1		
DESCR2	A2		

Figure 19. Data Input to AEQL

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,NE,EQ
```

AEQLC is used to compare the address field of a descriptor to a constant. See figure 20. The comparison is arithmetic with A being considered as a signed integer.

If A = N transfer is to EQ.

If A ≠ N transfer is to NE.

DESCR	A		

Figure 20. Data Input to AEQLC

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,NE,EQ
```

AEQLIC is used to compare an indirectly specified address field of a descriptor to a constant. See figure 21. The comparison is arithmetic with A1 being considered as a signed integer:

If A2 = N2 transfer is to EQ.

If A2 ≠ N2 transfer is to NE.

DESCR	A1		
A1+N1	A2		

Figure 21. Data Input to AEQLIC

Programming Notes

1. A2 may be a relocatable address.

2. N2 is never negative.

3. See also AEQL, AEQLC, LEQLC, ACOMP, and ACOMPC.

11. APDSP (append specifier)

L C A F V
0 4 8 12

```
APDSP   SPEC1,SPEC2
```

APDSP is used to append one specified string to another specified string. See figures 22 and 23.

SPEC1 1 →	A1			O1	L1
SPEC2 2 →	A2			O2	L2

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

Figure 22. Data Input to APDSP

SPEC1	A1			O1	L1+L2

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

Figure 23. Data Altered by APDSP

Programming Notes

1. If L1 = 0, C21 is placed at O1 + A1.
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. See figure 24.

L	0	0	0
⋮		⋮	
L+(N-1)*D	0	0	0

Figure 24. Data Assembled by ARRAY

Programming Notes

1. All fields of all descriptors assembled by ARRAY <u>must</u> 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. See figures 25 and 26. The flag field of the descriptor at A distinguishes between string structures and blocks.

If F contains the flag STTL, then

7

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

where [X] is the integer part of X and CPD is the numbers of characters stored per descriptor. The constant 4 occurs because of the 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.

Otherwise

F(V) = V + D

DESCR2	A		
A		F	V

Figure 25. Data Input to BKSIZE

DESCR1	<u>F(V)</u>	<u>0</u>	<u>0</u>

Figure 26. Data Altered by BKSIZE

<u>Programming Notes</u>

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. See figure 27.

DESCR			
	I		

Figure 27. Data Input to BKSPCE

Programming Notes

1. See also ENFILE and REWIND.

2. Refer to the section on input and output 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 the operation at LOC. PROC, if given, is the procedure in which LOC occurs.

Programming Notes

1. Refer to the section on program organization and procedure entry points,

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. See figure 28.

DESCR	A		
A+N	LOC		

Figure 28. Data Input to BRANIC

17. BUFFER (assemble buffer of blank characters)

```
LOC     BUFFER   N
```

BUFFER is used to assemble a string of N blank characters. See figure 29.

LOC

Figure 29. Data Assembled by BUFFER

Programming Notes

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

18. CHKVAL (check value)

```
CHKVAL   DESCR1,DESCR2,SPEC,GT,EQ,LT
```

CHKVAL is used to compare an integer to the length of a specifier plus another integer. See figure 30.

If L + I2 > I1 transfer is to GT.

If L + I2 = I1 transfer is to EQ.

If L + I2 < I1 transfer is to LT.

SPEC					L
DESCR1	I1				
DESCR2	I2				

Figure 30. Data Input to CHKVAL

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	Ø
ERROR	8
STOP	16
STOPSH	24

The indicator field of each entry of TABLE is set to T where T is the indicator which corresponds to the value of KEY. See figures 31 and 32.

TABLE		T	
•		•	
•		•	
•		•	
TABLE+Z*E		T	

Figure 31. Data Altered by CLERTB for ERROR, STOP, or STOPSH

TABLE	TABLE	0	
•		•	
•		•	
•		•	
TABLE+Z*E	TABLE	0	

Figure 32. Data Altered by CLERTB for CONTIN

Programming Notes

1. See the section which discusses the structure of syntax tables.

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 SNOBOL4 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 after the program.

MDATA is a file of machine-dependent data. It contains data used in implementation of the macros and for strings which depend on the character set of an individual machine or present other problems which 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 which describes 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 macro and definitions of nine 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. 1 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 which 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 which are titles of string structures.

10. TTL, a flag used to identify descriptors which 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 and 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, 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.

<u>Programming Notes</u>

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. See figures 33, 34, 35, 36, 37, and 38. First set

R1 = A1

R2 = A2

R3 = A6

where R1, R2, and R3 are temporary variables. 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 in 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

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

Figure 33. Initial Data Input to CPYPAT

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

Figure 34. Data Input to CPYPAT for Successive Vaues of R2

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

Figure 35. Data Altered by CPYPAT for Successive Values of R1

R2+4D	A10	F10	V10

Figure 36. Additional Data Input for Successive Values of R2 if V7 = 3

R1+4D	A10	F10	V10

Figure 37. Additional Data Altered for Successive Values of R1 if V7 = 3

DESCR1	R1		

Figure 38. Data Altered when Copying is Complete

22. DATE (get date)

DATE SPEC

DATE is used to obtain the current date. See figure 39. A character representation of the current date is placed in BUFFER.

4 8 12

SPEC	BUFFER	0	0	0	L

BUFFER	C1	...	CL

Figure 39. Data Altered by DATE

Programming Notes

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

    April 1, 1968

    04/01/68

    4:1:68

    68.092

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 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. See figures 40 and 41. A is considered as a signed integer.

DESCR	A		

Figure 40. Data Input to DECRA

DESCR	<u>A-N</u>		

Figure 41. Data Altered by DECRA

Programming Notes

1. A may be 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,NE,EQ
```

DEQL is used to compare two descriptors. See figure 42.

If A1 = A2, F1 = F2, and V1 = V2, transfer is to EQ.

Otherwise transfer is to NE.

DESCR1	A1	F1	V1
DESCR2	A2	F2	V2

Figure 42. Data Input to DEQL

Programming Notes

1. All fields of the two descriptors must be identical for transfer to EQ.

25. DESCR (assemble descriptor)

LOC	DESCR	A,F,V

DESCR assembles a descriptor with specified address, flag, and value fields. See figure 43.

LOC	A	F	V

Figure 43. Data Assembled by DESCR

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,FAILURE,SUCCESS
```

DIVIDE is used to divide one integer by another. Any remainder is ignored. That is, the result is truncated, not rounded. See figures 44 and 45.

If I = 0 transfer is to FAILURE.

Otherwise transfer is to SUCCESS.

DESCR2	A	F	V
DESCR3	I		

Figure 44. Data Input to DIVIDE

DESCR1	A/I	F	V

Figure 45. Data Altered by DIVIDE

Programming Notes

1. A may be a relocatable address.

27. DVREAL (divide real numbers)

```
DVREAL   DESCR1,DESCR2,DESCR3,FAILURE,SUCCESS
```

DVREAL is used to divide one real number by another. See figures 46 and 47.

If R3 = 0 or the result is out of the range available for real numbers, transfer is to FAILURE.

Otherwise transfer is to SUCCESS.

DESCR2	R2	F2	V2
DESCR3	R3		

Figure 46. Data Input to DVREAL

DESCR1	R2/R3	F2	V2

Figure 47. Data Altered by DVREAL

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 which initiated the SNOBOL4 run. See figure 48.

If I is nonzero, a post-mortem dump of user core should be given.

DESCR	I		

Figure 48. Data Input to ENDEX

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 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. See figure 49.

DESCR	I		

Figure 49. Data Input to ENFILE

Programming Notes

1. See also BKSPCE and REWIND.

2. Refer to the section on input and output for a discussion of unit reference numbers.

31. EQU (define 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,FAILURE,SUCCESS
```

EXPINT is used to raise an integer to an integer power. See figures 50 and 51.

If I1 = 0 and I2 is not positive, or if the result is out of the range available for integers, transfer is to FAILURE.

Otherwise transfer is to SUCCESS.

DESCR2	I1	F	V
DESCR3	I2		

Figure 50. Data Input to EXPINT

DESCR1	I1**I2	F	V

Figure 51. Data Altered by EXPINT

33. EXREAL (exponentiate real numbers)

```
        EXREAL   DESCR1,DESCR2,DESCR3,FAILURE,SUCCESS
```

EXREAL is used to raise a real number to a real power. See figures 52 and 53.

If the result is out of the range available for real numbers, transfer is to <u>failure</u>.

Otherwise transfer is to <u>success</u>.

DESCR2	R1	F	V
DESCR3	R2		

Figure 52. Data Input to EXREAL

DESCR1	<u>R1**R2</u>	<u>F</u>	<u>V</u>

Figure 53. Data Altered by EXREAL

34. FORMAT (assemble format string)

```
LOC     FORMAT  'C1...CN'
```

FORMAT is used to assemble the characters of a format. See figure 54.

LOC	C1	...	CN

Figure 54. Data Assembled by FORMAT

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. See figures 55 and 56.

SPEC				O	L

Figure 55. Data Input to FSHRTN

SPEC				O+N	L-N

Figure 56. Data Altered by FSHRTN

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. See figures 57 and 58.

DESCR2	A2		
A2+N	A		

Figure 57. Data Input to GETAC

DESCR1	<u>A</u>		

Figure 58. Data Altered by GETAC

Programming Notes

1. See also PUTAC, GETDC, and PUTDC.

N may be negative!

37. GETBAL (get parenthesis balanced string)

```
        GETBAL   SPEC,DESCR,FAILURE,SUCCESS
```

GETBAL is used to get the specification of a balanced substring. See figures 59 and 60. 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, J = 1.

If CL+1 is a left parenthesis, 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 FAILURE.

Otherwise SPEC is modified as indicated and transfer is to SUCCESS.

SPEC	A			O	L

DESCR	N		

A+O	C1	...	CL	CL+1	...	CL+N

Figure 59. Data Input to GETBAL

SPEC	A			O	L+J

Figure 60. Data Altered by GETBAL

38. GETD (get descriptor)

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

GETD is used to get a descriptor. See figures 61 and 62.

DESCR2	A2		
DESCR3	A3		
A2+A3	A	F	V

Figure 61. Data Input to GETD

DESCR1	<u>A</u>	<u>F</u>	<u>V</u>

Figure 62. Data Altered by GETD

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. See figures 63 and 64.

DESCR2	A2		
A2+N	A	F	V

Figure 63. Data Input to GETDC

DESCR1	<u>A</u>	<u>F</u>	<u>V</u>

Figure 64. Data Altered by GETDC

<u>Programming Notes</u>

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. See figures 65 and 66.

SPEC					L

Figure 65. Data Input to GETLG

DESCR	L	0	0

Figure 66. Data Altered by GETLG

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. See figures 67 and 68. The amount of storage is given by the formula

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

where [X] is the integer part of X 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 sting of L characters.

DESCR2	L		

Figure 67. Data Input to GETLTH

DESCR1	F(L)	0	0

Figure 68. Data Altered by GETLTH

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. See figures 69 and 70.

DESCR2	A		
A			V

Figure 69. Data Input to GETSIZ

DESCR1	V	0	0

Figure 70. Data Altered by GETSIZ

Programming Notes

1. See also SETSIZ.

43. GETSPC (get specifier with constant offset)

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

GETSPC is used to get a specifier. See figures 71 and 72.

DESCR	A1				
A1+N	A	F	V	O	L

Figure 71. Data Input to GETSPC

SPEC	<u>A</u>	<u>F</u>	<u>V</u>	<u>O</u>	<u>L</u>

Figure 72. Data Altered by GETSPC

<u>Programming Notes</u>

1. See also PUTSPC.

44. INCRA (increment address)

```
INCRA    DESCR,N
```

INCRA is used to increment the address field of a descriptor. See figures 73 and 74.

DESCR	A		

Figure 73. Data Input to INCRA

DESCR	A+N		

Figure 74. Data Altered by INCRA

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. See figures 75 and 76. I is considered as an unsigned (nonnegative) integer.

DESCR			I

Figure 75. Data Input to INCRV

DESCR			I+N

Figure 76. Data Altered by INCRV

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 function of this operation is machine and system dependent. Typically, INIT sets program masks and the values of certain registers.

In addition to any initialization required for a particular system and machine, INIT also performs the follwing initialzation 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. TLSGP1 is set to the first descriptor past the end of dynamic storage. Space for dynamic storage may be preallocated or seized 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. See figures 77 and 78.

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

Figure 77. Data Input to 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

Figure 78. Data Altered by INSERT

Programming Notes

1. See also ADDSIB and ADDSON.

2. INSERT is only used by compilation procedures.

48. INTRL (convert integer to real number)

INTRL DESCR1,DESCR2

INTRL is used to convert a (signed) integer to a real number. See figures 79 and 80. R(I) is the real number corresponding to I.

DESCR2	I		

Figure 79. Data Input to INTRL

DESCR1	R(I)	0	RC

Figure 80. Data Altered by INTRL

Programming Notes

1. RC stands for 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. See figures 81 and 82.

DESCR	A		

Figure 81. Data Input to INTSPC

SPEC	BUFFER	0	0	O	L

BUFFER+O	C1	...	CL

Figure 82. Data Altered by INTSPC

Programming Notes

1. C1...CL should be a "normalized" string corresponding to the integer I. That is, it should contain no leading zeroes and 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. See figure 83.

OSTACK	0		
CSTACK	STACK		

Figure 83. Data Altered by ISTACK

Programming Notes

1. STACK is a global 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,GT,EQ,LT
```

LCOMP is used to compare the lengths of two specifiers. See figure 84.

If L1 > L2 transfer is to GT.

If L1 = L2 transfer is to EQ.

If L1 < L2 transfer is to LT.

SPEC1					L1
SPEC2					L2

Figure 84. Data Input to LCOMP

Programming Notes

1. See also ACOMP, RCOMP and LEQLC.

52. LEQLC (length equal to constant test)

```
LEQLC    SPEC,N,NE,EQ
```

LEQLC is used to compare the length of a specifier to a constant. See figure 85. The magnitudes are compared.

If L = N transfer is to <u>EQ</u>.

If L ≠ N transfer is to <u>NE</u>.

SPEC					L

Figure 85. Data Input to LEQLC

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,GT,EQ,LT
```

LEXCMP is used to compare two strings lexicographically (i. e. according to their alphabetical ordering). See figure 86.

If C11...C1N1 > C21...C2M transfer is to GT.

If C11...C1N1 = C21...C2M transfer is to EQ.

If C11...C1N1 < C21...C2M transfer is to LT.

SPEC1	A1			O1	N
SPEC2	A2			O2	M

A1+O1	C11	...	C1N
A2+O2	C21	...	C2M

Figure 86. Data Input to LEXCMP

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 which 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 (except the null string).

4. Two strings are equal only if they are of the same length and 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 GT and LT will specify the same location or both be omitted. Because of the frequency of such use, it is desireable to handle this case specially if a test for equality can be performed more efficiently than the general case.

54. LHERE (define 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

```
LOC     LHERE
        OP
```

is equivalent to

```
LOC     OP
```

55. LINK (link to external function)

```
LINK     DESCR1,DESCR2,DESCR3,DESCR4,FAILURE,SUCCESS
```

LINK is used to link to an external function. See figures 87 and 88. 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 data type expected for the resulting value. The returned value is placed in DESCR1.

If the external function signals failure, transfer is to <u>FAILURE</u>.

Otherwise the transfer is to <u>SUCCESS</u>.

DESCR1			V1
DESCR2	A2		
DESCR3	N		
DESCR4	A4		

Figure 87. Data Input to LINK

DESCR1	<u>A</u>	<u>F</u>	<u>V</u>

Figure 88. Data Altered by LINK

<u>Programming Notes</u>

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" fields of pattern nodes until the end, indicated by a zero field, is reached. This zero field is replaced by I. See figures 89 and 90.

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

Figure 89. Data Input to LINKOR

A+2D+IN	I		

Figure 90. Data Altered by LINKOR

57. LOAD (load external function)

```
LOAD    DESCR,SPEC1,SPEC2,FAILURE,SUCCESS
```

LOAD is used to load an external function. See figures 91 and 92. 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 <u>success</u>.

Otherwise transfer is to <u>failure</u>.

SPEC1	A1			O1	L1
SPEC2	A2			O2	L2

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

Figure 91. Data Input to LOAD

DESCR	<u>A3</u>		

Figure 92. Data Altered by LOAD

<u>Programming Notes</u>

1. LOAD is a system-dependent operation.

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

3. On the IBM/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,<u>FAILURE</u>,<u>SUCCESS</u>

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. See figures 93 and 94. 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 <u>FAILURE</u>.

Otherwise transfer is to <u>SUCCESS</u>.

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			

Figure 93. Data Input to LOCAPT

DESCR1	<u>A+2I*D</u>	<u>F</u>	<u>V</u>

Figure 94. Data Altered by LOCAPT

Programming Notes

1. Note that the address of DESCR1 is set to one descriptor less than the descriptor which is located.

2. See also LOCAPV.

59. LOCAPV (locate attribute pair by value)

LOCAPV DESCR1,DESCR2,DESCR3,<u>FAILURE</u>,<u>SUCCESS</u>

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. See figures 95 and 96. 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 <u>FAILURE</u>.

Otherwise transfer is to <u>SUCCESS</u>.

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			

Figure 95. Data Input to LOCAPV

DESCR1	<u>A+2I*D</u>	<u>F</u>	<u>V</u>

Figure 96. Data Altered by LOCAPV

Programming Notes

1. Note that the address of DESCR1 is set to two descriptors less than the descriptor which 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. See figures 97, 98 and 99.

DESCR	A	F	V
A			I

Figure 97. Data Input to LOCSP

SPEC	A	F	V	4*CPD	I

Figure 98. Data Altered by LOCSP if A ≠ 0

SPEC					0

Figure 99. Data Altered by LOCSP if A = 0

Programming Notes

1. If A = 0, the value of DESCR represents the null (zero length) string and is handled as a special case as indicated. The remainder of SPEC is 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. See figures 100 and 101. The address field of DESCR1 is set to I where

I = minimum(I1,...,IK)

DESCR2	A		
A+2D	N1		
A+3D	I1		
A+N1+2D	N2		
A+N1+3D	I1		
. . .	. . .		
A+NK+2D	0		
A+NK+3D	IK		

Figure 100. Data Input to LVALUE

DESCR1	I	0	0

Figure 101. Data Altered by LVALUE

Programming Notes

1. I1,...,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. See figures 102 and 103.

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

Figure 102. Data Input to MAKNOD

DESCR6	A6	F6	V6	P

Figure 103. Additional Data Input if DESCR6 is Given

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

Figure 104. Data Altered by MAKNOD

A2+4D	A6	F6	V6

Figure 105. Addtional Data Altered if DESCR6 is Given

Programming Notes

1. As indicated, there are two forms of MAKNOD. If DESCR6 is given, an additional descriptor is 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. See figures 106 and 107.

DESCR2	R	F	V

Figure 106. Data Input to MNREAL

DESCR1	-R	F	V

Figure 107. Data Altered by MNREAL

Programming Notes

1. R may be negative.

2. See alsc MNSINT, ADREAL, DVREAL, EXREAL, MPREAL, and SBREAL.

64. MNSINT (minus integer)

```
MNSINT   DESCR1,DESCR2,FAILURE,SUCCESS
```

MNSINT is used to change the sign of an integer.

If -I exceeds the maximum integer, transfer is to FAILURE.

Otherwise transfer is to SUCCESS. See figures 108 and 109.

DESCR2	I	F	V

Figure 108. Data Input to MNSINT

DESCR1	-I	F	V

Figure 109. Data Altered by MNSINT

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. See figures 110 and 111.

DESCR2	A		

Figure 110. Data Input to MOVA

DESCR1	A		

Figure 111. Data Altered by MOVA

Programming Notes

1. See also MOVD and MOVV.

56. MOVBLK (move block of descriptors)

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

MOVBLK is used to move (copy) a block of descriptors. See figures 112 and 113.

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

Figure 112. Data Input to MOVBLK

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

Figure 113. Data Altered by MOVBLK

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. Consequently, descriptors must be moved one at a time starting at the first descriptor in the diagram.

67. MOVD (move descriptor)

```
        MOVD     DESCR1,DESCR2
```

MOVD is used to move a descriptor from one location to another. See figures 114 and 115.

DESCR2	A	F	V

Figure 114. Data Input to MOVD

DESCR1	A	F	V

Figure 115. Data Altered by MOVD

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 which is indirectly specified with an offset constant. See figures 116 and 117.

DESCR1	A1		
DESCR2	A2		
A2+N2	A	F	V

Figure 116. Data Input to MOVDIC

A1+N1	<u>A</u>	<u>F</u>	<u>V</u>

Figure 117. Data Altered by MOVDIC

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. See figures 118 and 119.

DESCR2			V

Figure 118. Data Input to MOVV

DESCR1			<u>V</u>

Figure 119. Data Altered by MOVV

Programming Notes

1. See also MOVA and MOVD.

70. MPREAL (mulitply real numbers)

```
MPREAL   DESCR1,DESCR2,DESCR3,FAILURE,SUCCESS
```

MPREAL is used to multiply two real numbers. See figures 120 and 121.

If the result is out of the range available for real numbers, transfer is to FAILURE.

Otherwise transfer is to SUCCESS.

DESCR2	R2	F2	V2
DESCR3	R3		

Figure 120. Data Input to MPREAL

DESCR1	R2*R3	F2	V2

Figure 121. Data Altered by MPREAL

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. See figure 122.

DESCR	TIME	0	0

Figure 122. Data Altered by MSTIME

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 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,FAILURE,SUCCESS
```

MULT is used to multiply two integers. See figures 123 and 124.

In the event of overflow, transfer is to FAILURE.

Otherwise, transfer is to SUCCESS.

DESCR2	I2	F2	V2
DESCR3	I3		

Figure 123. Data Input to MULT

DESCR1	I2*I3	F2	V2

Figure 124. Data Altered by MULT

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. See figures 125 and 126.

DESCR2	I		

Figure 125. Data Input to MULTC

DESCR1	I*N	0	0

Figure 126. Data Altered by MULTC

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 shift, 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. Figure 127 shows the organizational structure of SNOBOL4 variable storage consisting of OBSIZ linked chains. The links should be rearranged to put the strings in alphabetical order.

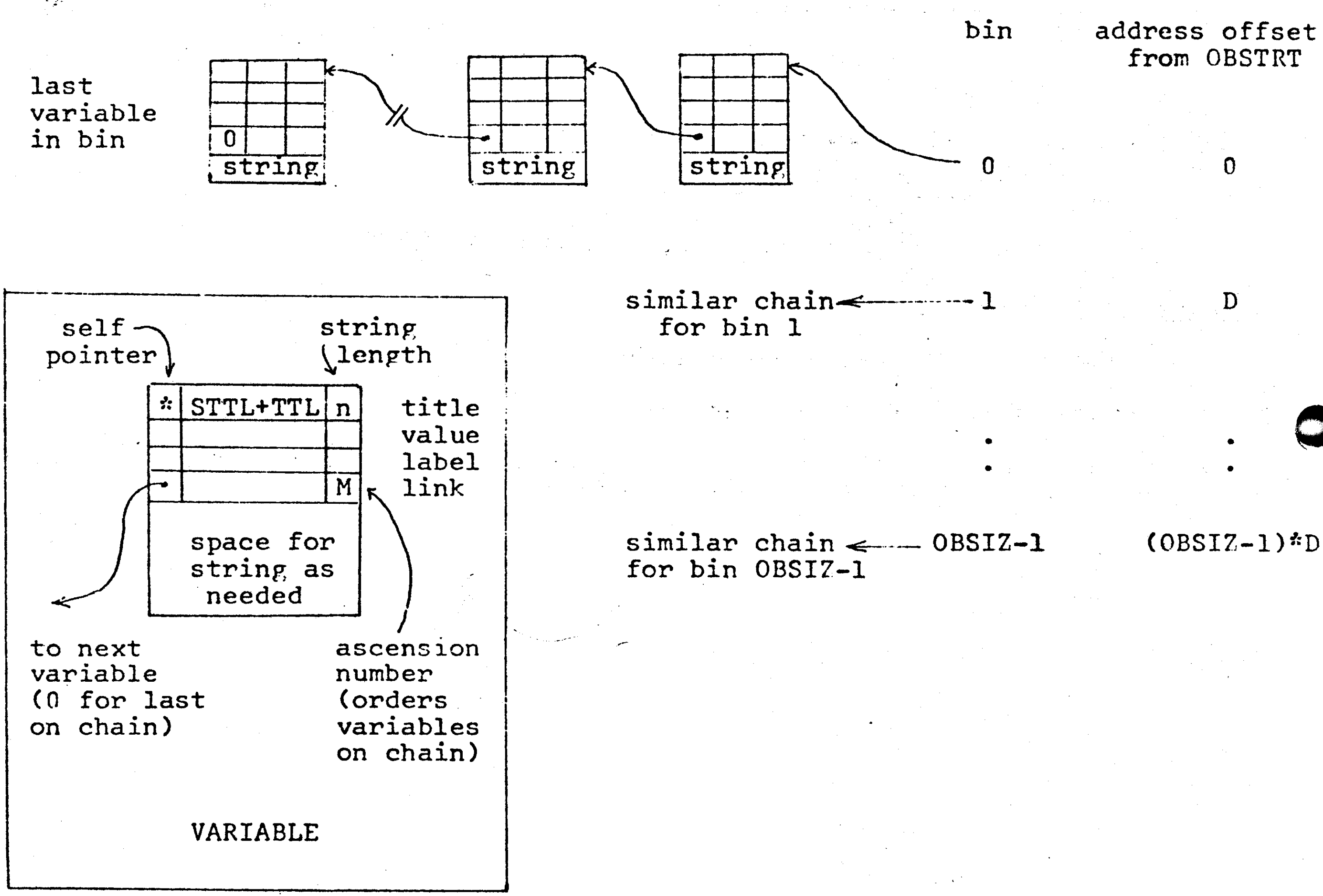

Figure 127. Organization of Variable Storage

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 null strings (zero address fields and value fields containing the global symbol S) may be omitted from the sort. In fact they should be omitted if a sort with factorial properties (such as an exchange sort) is used. A sort with linear properties such as a radix sort is more desirable but more complicated.

4. The ascension number, M, is computed by VARID (q.v.).

75. OUTPUT (output record)

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

OUTPUT is used to output a list of items according to FORMAT. See figure 128. 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.

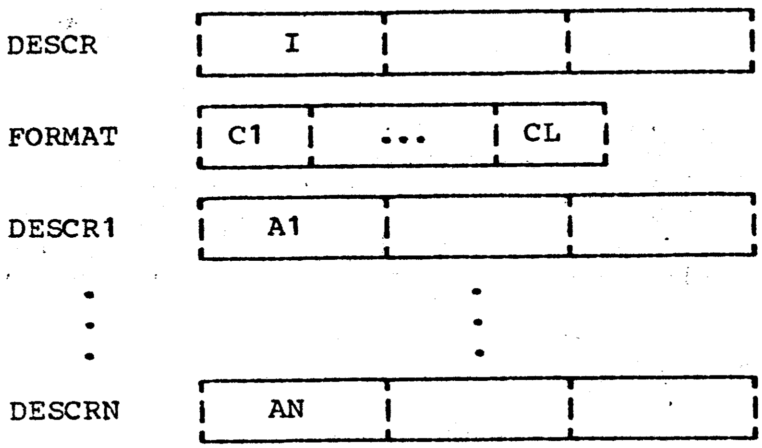

Figure 128. Data Input to OUTPUT

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 which corresponds to the value of KEY. See figures 129, 130 and 131.

SPEC	A			O	L

A+O	C1	...	CL

Figure 129. Data Input to PLUGTB

TABLE+E*C1		T	
⋮		⋮	
TABLE+E*CL		T	

Figure 130. Data Altered by PLUGTB for ERROR, STOP, or STOPSH

TABLE+E*C1	TABLE	0	
⋮		⋮	
TABLE+E*CL	TABLE	0	

Figure 131. Data Altered by PLUGTB for CONTIN

Programming Notes

1. See the section which discusses the structure of syntax tables.

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. See figures 132 and 133.

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

Figure 132. Data Input to POP

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

Figure 133. Data Altered by POP

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 tr ansferring to the global location INTR10 when the condition is detected.

78. PROC (procedure entry)

```
LOC1    PROC    [LOC2]
```

PROC is used to identify a procedure entry point. If LOC2 is omitted, LOC1 is the primary procedure entry point. If LOC2 is present, LOC1 is a secondary entry point in the procedure with primary entry point LOC2.

Programming Notes

1. Procedure entry points may be referred to by RCALL, BRANIC, or BRANCH (in its two argument form).

2. In most implementations, PROC will have no functional use and may be implemented as LHERE. For machines which 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. See figures 134 and 135.

CSTACK	A		

Figure 134. Data Input to PSTACK

DESCR	A-D	0	0

Figure 135. Data Altered by PSTACK

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. See figures 136 and 137.

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

Figure 136. Data Input to PUSH

CSTACK	<u>A+(D*N)</u>		
A+D	<u>A1</u>	<u>F1</u>	<u>V1</u>
.		.	
.		.	
.		.	
A+(D*N)	<u>AN</u>	<u>FN</u>	<u>VN</u>

Figure 137. Data Altered by PUSH

Programming Notes

1. If A + (D * N) > STACK + STSIZE, stack overflow occurs. Transfer should be made to the global 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 with a constant offset. See figures 138 and 139.

DESCR1	A1		
DESCR2	A2		

Figure 138. Data Input to PUTAC

A1+N	<u>A2</u>		

Figure 139. Data Altered by PUTAC

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. See figures 140 and 141.

DESCR1	A1		
DESCR2	A2		
DESCR3	A	F	V

Figure 140. Data Input to PUTD

A1+A2	<u>A</u>	<u>F</u>	<u>V</u>

Figure 141. Data Altered by PUTD

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 with an offset constant. See figures 142 and 143.

DESCR1	A1		
DESCR2	A	F	V

Figure 142. Data Input to PUTDC

A1+N	A	F	V

Figure 143. Data Altered by PUTDC

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. See figures 144 and 145.

DESCR	I		

Figure 144. Data Input to PUTLG

SPEC					I

Figure 145. Data Altered by PUTLG

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. See figures 146 and 147.

DESCR	A1				
SPEC	A	F	V	O	L

Figure 146. Data Input to PUTSPC

A1+N	A	F	V	O	L

Figure 147. Data Altered by PUTSPC

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 with an offset constant. See figures 148 and 149.

DESCR1	A		
DESCR2			V

Figure 148. Data Input to PUTVC

A+N			V

Figure 149. Data Altered by PUTVC

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 which receives value upon return. 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 signalled. See figures 150, 151 and 152. 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 program similar to that shown should be assembled. OP is intended to represent an instruction which stores the value returned by PROC in DESCR.

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

Figure 150. Data Input to 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		

Figure 151. Data Altered by RCALL

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

Figure 152. Return Code at LOC

Programming Notes

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

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

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

4. (DESCR1,...,DESCRN) may be 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 RCALL.

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

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

8. RCALL typically must save program state information. On the IBM 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 on 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 <u>the flag fields of descriptors used to save program state information must be set to zero.</u>

9. See also SELBRA.

88. RCOMP (real comparison)

```
RCOMP    DESCR1,DESCR2,GT,EQ,LT
```

RCOMP is used to compare two real numbers. See figure 153.

If R1 > R2 transfer is to GT.

If R1 = R2 transfer is to EQ.

If R1 < R2 transfer is to LT.

DESCR1	R1		
DESCR2	R2		

Figure 153. Data Input to RCOMP

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. See figures 154 and 155.

DESCR	R		

Figure 154. Data Input to REALST

SPEC	BUFFER	0	0	0	L

BUFFER	C1	...	CL

Figure 155. Data Altered by REALST

Programming Notes

1. C1...CL should represent the real number R as a "normalized" string 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 compatability with real literals and data type conversions, the real number should not be represented with an exponent, although very large or small numbers may require a large number of characters for their representation.

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 given length. See figures 156 and 157.

SPEC2	A2	F2	V2	O2	L2
SPEC3					L3

Figure 156. Data Input to REMSP

SPEC1	A2	F2	V2	O2+L3	L2-L3

Figure 157. Data Altered by REMSP

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. See figures 158 and 159.

```
DESCR     |          |    F     |          |
```

Figure 158. Data Input to RESETF

```
DESCR     |          |  F-FLAG  |          |
```

Figure 159. Data Altered by RESETF

Programming Notes

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

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. See figure 160.

DESCR	I		

Figure 160. Data Input to REWIND

Programming Notes

1. Refer to the section on input and output for a discussion of unit reference numbers.

2. See also BKSPCE and ENFILE.

93. RLINT (convert real number to integer)

RLINT DESCR1,DESCR2,FAILURE,SUCCESS

RLINT is used to convert a real number to an integer. See figures 161 and 162.

If the magnitude of R exceeds the magnitude of the largest integer, trandfer is to FAILURE.

Otherwise transfer is to SUCCESS.

DESCR1	R		

Figure 161. Data Input to RLINT

DESCR2	I(R)	0	IC

Figure 162. Data Altered by RLINT

Programming Notes

1. I(R) is the integer equivalent of the real number R.
2. The fraction part of R is discarded.
3. IC stands for the integer data type code.

94. RPLACE (replace characters)

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

RPLACE is used to replace characters in a string. See figures 163 and 164. 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

$$F(CI) = CI \text{ if } CI \neq C2J \text{ for any } J \ (1 \le J \le L2)$$

$$F(CI) = C3J \text{ if } CI = C2J \text{ for some } J \ (1 \le J \le L2)$$

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

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

Figure 163. Data Input to RPLACE

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

Figure 164. Data Altered by RPLACE

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

$$C2I = C2J = \ldots = C2K \quad (I < J < K)$$

then

$$F(CI) = C3K$$

3. RPLACE is used only in the 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. See figures 165, 166 and 167. The stack is repositioned as shown.

At the location LOC program similar to that shown has been assembled by RCALL. OP represents an instruction which is used by RRTURN to return the value of DESCR.

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

Figure 165. Data Input to RRTURN

CSTACK	A		
OSTACK	A0		
DESCR1	A1	F1	V1

Figure 166. Data Altered by RRTURN

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

Figure 167. Return Code at LOC.

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 which is specified indirectly. See figures 168 and 169.

DESCR	A		
A		F	

Figure 168. Data Input to RSETFI

A		F-FLAG	

Figure 169. Data Altered by RSETFI

Programming Notes

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

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,FAILURE,SUCCESS
```

SBREAL is used to subtract one real number from another. See figures 170 and 171.

If the result is out of the range available for real numbers, transfer is to FAILURE.

Otherwise transfer is to success.

DESCR2	R2	F2	V2
DESCR3	R3		

Figure 170. Data Input to SBREAL

DESCR1	R2-R3	F2	V2

Figure 171. Data Altered by SBREAL

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. See figure 172. Transfer is to LOCI corresponding to I.

DESCR	I		

Figure 172. Data Input to SELBRA

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 $1 \le I \le N + 1$. 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. See figure 173.

DESCR	<u>N</u>		

Figure 173. Data Altered by SETAC

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. See figure 174.

DESCR2			V

Figure 174. Data Input to SETAV

DESCR1	V	0	0

Figure 175. Data Altered by SETAV

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. See figures 176 and 177.

DESCR		F	

Figure 176. Data Input to SETF

DESCR		F+FLAG	

Figure 177. Data Altered by SETF

Programming Notes

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

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. See figures 178 and 179.

DESCR	A		
A		F	

Figure 178. Data Input to SETFI

A		F+FLAG	

Figure 179. Data Altered by SETFI

Programming Notes

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

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. See figure 180.

SPEC					N

Figure 180. Data Altered by SETLC

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. See figures 181 and 182.

DESCR1	A		
DESCR2	I		

Figure 181. Data Input to SETSIZ

A			I

Figure 182. Data Altered by SETSIZ

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. See figures 183 and 184.

SPEC2	A	F	V	O	L

Figure 183. Data Input to SETSP

SPEC1	<u>A</u>	<u>F</u>	<u>V</u>	<u>O</u>	<u>L</u>

Figure 184. Data Altered by SETSP

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. See figures 185 and 186.

DESCR2	I		

Figure 185. Data Input to SETVA

DESCR1			I

Figure 186. Data Altered by SETVA

Programming Notes

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

2. See also SETAV 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. See figure 187.

DESCR			N

Figure 187. Data Altered by SETVC

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. See figures 188 and 189.

SPEC					L

Figure 188. Data Input to SHORTN

SPEC					L-N

Figure 189. Data Altered by SHORTN

Programming Notes

1. L - N is never negative.

109. SPCINT (convert specifier to integer)

```
SPCINT   DESCR,SPEC,FAILURE,SUCCESS
```

SPCINT is used to convert a specified string to an integer. See figures 190 and 191. I 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 FAILURE.

Otherwise transfer is to SUCCESS.

SPEC

A			O	L

A+O

C1	...	CL

Figure 190. Data Input to SPCINT

DESCR

I	0	IC

Figure 191. Data Altered by SPCINT

Programming Notes

1. IC stands for 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. If L = 0, I should be the integer 0.

4. See also INTSPC and SPREAL.

110. SPEC (assemble specifier)

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

SPEC is used to assemble a specifier. See figure 192.

LOC	A	F	V	O	L

Figure 192. Data Assembled by SPEC

111. SPOP (pop specifier from stack)

SPOP (SPEC1,...,SPECN)

SPOP is used to pop a list of specifiers from the system stack. See figures 193 and 194.

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

Figure 193. Data Input to SPOP

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

Figure 194. Data Altered by SPOP

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 global location INTR10 when the condition is detected.

2. See also POP, SPUSH, and PUSH.

112. SPREAL (convert specified string to real number)

SPREAL DESCR,SPEC,<u>FAILURE</u>,<u>SUCCESS</u>

SPREAL is used to convert a specified string into a real number. See figures 195 and 196. R is a signed real number resulting from the conversion of the string C1...CL.

If C1...CL does not represent a real number, or if the real number represents is out of the range available for real numbers, transfer is to FAILURE.

Otherwise transfer is to SUCCESS.

SPEC	A			O	L

A+O	C1	...	CL

Figure 195. Data Input to SPREAL

DESCR	R	0	RC

Figure 196. Data Altered by SPREAL

Programming Notes

1. RC stands for 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 should be the real number 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. See figures 197 and 198.

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

Figure 197. Data Input to SPUSH

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

Figure 198. Data Altered by SPUSH

Programming Notes

1. If A + (S * N) > STACK + STSIZE, stack overflow occurs. Transfer should be made to the global 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. See figures 199 and 200. 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 signalled by the output routine.

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

Figure 199. Data Input to STPRNT

DESCR1	J		

Figure 200. Data Altered by STPRNT

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 at present. It is intended for eventual use in indicating interrupts from a console on which output is being written. DESCR1 can be ignored for the present.

4. See also OUTPUT and STREAD.

115. STREAD (string read)

```
STREAD  SPEC,DESCR,EOF,ERROR,SUCCESS
```

STREAD is used to read a string. See figures 201 and 202. The string C1...CL is read from the file associated with unit reference number I.

If a reading error occurs, transfer is to <u>ERROR</u>.

If an end of file is encountered, transfer is to <u>EOF</u>.

Otherwise transfer is to <u>SUCCESS</u>.

DESCR	I				
SPEC	A			O	L

Figure 201. Data Input to STREAD

A+O	<u>C1</u>	...	<u>CL</u>

Figure 202. Data Altered by STREAD

<u>Programming Notes</u>

1. Note that the length of the string to be read is specified by the data input 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,SUCCESS
```

STREAM is used to locate a syntactic token at the beginning of the string specified by SPEC2. See figures 203, 204, 205, 206, and 207.

> If there is an I ($1 \le I \le L$) such that TI is ERROR, STOP, or STOPSH, and J is the least such I, then
>
> If TJ is ERROR, transfer is to ERROR.
>
> If TJ is STOP or STOPSH, transfer is to SUCCESS.
>
> 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 ($1 \le I \le J$) which is nonzero (i.e. for which a put is specified in the syntax table description for the tables given).

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			

Figure 203. Data Input to STREAM

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

Figure 204. Data Altered by STREAM if Termination is STOP

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

Figure 205. Data Altered by STREAM if Termination is STOPSH

STYPE	O				
SPEC1	A	F	V	O	L

Figure 206. Data Altered by STREAM if Termination is ERROR

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

Figure 207. Data Altered by STREAM if Termination is RUNOUT

Programming Notes

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

2. If L = 0 (i.e. if SPECR2 specifies the null string), RUNOUT occurs. In this case the address field of STYPE should be set to 0.

117. STRING (assemble specified string)

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

STRING is used to assemble a string and a specifier to it. See figure 208.

LOC	A	0	0	0	L

A	C1	...	CL

Figure 208. Data Assembled by STRING

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,FAILURE,SUCCESS
```

SUBSP is used to specify an initial substring of a specified string. See figures 209 and 210.

If L3 ≥ L2 transfer is to SUCCESS.

Otherwise transfer is to FAILURE and SPEC1 is not altered.

SPEC2					L2
SPEC3	A3	F3	V3	O3	L3

Figure 209. Data Input to SUBSP

SPEC1	A3	F3	V3	O3	L2

Figure 210. Data Altered by SUBSP if L3 ≥ L2

119. SUBTRT (subtract addresses)

```
SUBTRT   DESCR1,DESCR2,DESCR3,FAILURE,SUCCESS
```

SUBTRT is used to subtract one address field from another. See figures 211 and 212. A2 and A3 are considered as signed integers.

If A2 - A3 is out of the range available for integers, transfer is to FAILURE.

Otherwise transfer is to SUCCESS.

DESCR2	A2	F2	V2
DESCR3	A3		

Figure 211. Data Input to SUBTRT

DESCR1	A2-A3	F2	V2

Figure 212. Data Altered by SUBTRT

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,FAILURE,SUCCESS
```

SUM is used to add two address fields. See figures 213 and 214. A and I are considered as signed integers.

If A + I is out of the range available for integers, transfer is to FAILURE.

Otherwise transfer is to SUCCESS.

DESCR2	A	F	V
DESCR3	I		

Figure 213. Data Input to SUM

DESCR1	A+I	F	V

Figure 214. Data Altered by SUM

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,FAILURE,SUCCESS

TESTF is used to test a flag field for the presence of a flag. See figure 215.

If F contains FLAG, transfer is to SUCCESS.

Otherwise transfer is to FAILURE.

DESCR

	F	

Figure 215. Data Input to TESTF

Programming Notes

1. See also TESTFI.

122. TESTFI (test flag indirect)

```
        TESTFI   DESCR,FLAG,FAILURE,SUCCESS
```

TESTFI is used to test an indirectly specified flag field for the presence of a flag. See figure 216.

If F contains FLAG, transfer is to <u>SUCCESS</u>.

Otherwise transfer is to <u>FAILURE</u>.

DESCR	A		
A		F	

Figure 216. Data Input to TESTFI

<u>Programming Notes</u>

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 program. 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. See figures 217 and 218. 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.

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

Figure 217. Data Input to TOP

DESCR1	<u>A-(N*D)</u>	<u>F</u>	<u>V</u>
DESCR2	<u>N*D</u>	<u>0</u>	<u>0</u>

Figure 218. Data Altered by TOP

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. See figures 219 and 220.

SPEC2	A	F	V	O	L

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

Figure 219. Data Input to TRIMSP

SPEC1	A	F	V	O	J

Figure 220. Data Altered by TRIMSP

Programming Notes

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

126. UNLOAD (unload external function)

UNLOAD SPEC

UNLOAD is used to unload an external function. See figure 221. C1...CL represents the name of the function that is to be unloaded.

SPEC	A			O	L

A+O	C1	...	CL

Figure 221. Data Input to UNLOAD

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 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 LOADed 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. See figures 222 and 223. K and M are computed by

$$K = F1(C1...CL)$$

$$M = F2(C1...CL)$$

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

$$0 \le K \le (OBSIZ - 1) * D$$

$$0 \le M \le SIZLIM$$

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

SPEC	A			O	L

A+O	C1	...	CL

Figure 222. Data Input to VARID

DESCR	<u>K</u>		<u>M</u>

Figure 223. Data Altered by VARID

Programming Notes

1. K is used to selected one of a number of chains in variable storage. The K are address offsets which 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,GT,EQ,LT
```

VCMPIC is used to compare a value field, indirectly specified with an offset constant, with another value field. See figure 224. V1 and V2 are considered as unsigned integers.

If V1 > V2 transfer is to GT.

If V1 = V2 transfer is to EQ.

If V1 < V2 transfer is to LT.

DESCR1	A1		
DESCR2			V2
A1+N			V1

Figure 224. Data Input to VCMPIC

129. VEQL (value fields equal test)

```
VEQL    DESCR1,DESCR2,NE,EQ
```

VEQL is used to compare the value fields of two descriptors. See figure 225. V1 and V2 are considered as unsigned integers.

If V1 = V2 transfer is to EQ.

If V1 ≠ V2 transfer is to NE.

DESCR1			V1
DESCR2			V2

Figure 225. Data Input to VEQL

Programming Notes

1. See also AEQL and VEQLC.

130. VEQLC (value field equal to constant test)

```
VEQLC    DESCR,N,NE,EQ
```

VEQLC is used to compare the value field of a descriptor to a constant. See figure 226. V is considered as an unsigned integer.

If V = N transfer is to EQ.

If V ≠ N transfer is to NE.

DESCR			V

Figure 226. Data Input to VEQLC

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. See figures 227 and 228.

DESCR1	A		
DESCR2	D*I		

Figure 227. Data Input to ZERBLK

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

Figure 228. Data Altered by ZERBLK

Programming Notes

1. I is always positive.

Appendix 1 - Implementation Notes

A. Optional Macros

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

Operation	Alternative Implementation	Features Disabled
ADREAL[1]	Branch to INTR10.	Real arithmetic
BKSPCE	Branch to UNDF.	The function BACKSPACE
CLERTB[2]	Branch to UNDF.	The functions ANY, NOTANY, SPAN, and BREAK
DATE	Set length of SPEC to 0.	The function DATE
DVREAL[1]	Set address of DESCR2 to 0.	Real arithmetic and post-run statistics
ENFILE	Branch to UNDF.	The function ENDFILE
EXPINT	Branch to UNDF.	Exponentiation of integers
EXREAL[1]	Branch to INTR10.	Real arithmetic
GETBAL	Branch to UNDF.	The primitive pattern BAL
INTRL[1]	Perform no operation.	Real arithmetic
LEXCMP[3]	If <u>GT</u> ≠ <u>LT</u>, branch to UNDF.	The function LGT
LINK[4]	Branch to INTR10.	External functions
LOAD[4]	Branch to UNDF.	External functions

---

[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 for <u>LT</u> = <u>GT</u>.
[4] LINK, LOAD, and UNLOAD should be implemented or not implemented as a group.

MNREAL[1]	Branch to INTR10.	Real arithmetic
MPREAL[1]	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
PLUGTB[2]	Branch to INTR10.	The functions ANY, NOTANY, SPAN, and BREAK
RCOMP[1]	Branch to INTR10.	Real arithmetic
REALST[1]	Branch to UNDF.	Real arithmetic
REWIND	Branch to INTR10.	The function REWIND
RLINT[1]	Branch to INTR10.	Real arithmetic
RPLACE	Branch to INTR10.	The function REPLACE
SBREAL[1]	Branch to INTR10.	Real arithmetic
SPREAL[1]	Take the FAILURE exit.	Real arithmetic
TRIMSP	Branch to INTR10.	The function TRIM
UNLOAD[4]	Perform no operation.	External functions

B. 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.

C. Error Exit 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.

D. Subroutines versus In-Line Code

The choice between implementing macro operations by subroutine call 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 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 360 implementation. Time spent in I/O and system subroutines is not included. A * marks those macros implementated by subroutines in the IBM 360 implementation (including macros that call I/O and system subroutines).

Macro	Count	Time
ACOMP	65	2.952
ACOMPC	57	1.450
ADDLG	7	0.000
ADDSIB	6	0.000
ADDSON	12	0.017
ADJUST	2	0.000
ADREAL	1	0.000
AEQL	17	0.397
AEQLC	173	3.574
AEQLIC	9	0.086
APDSP*	93	0.897
ARRAY	5	-----
BKSIZE	5	1.329
BKSPCE*	1	0.000
BRANCH	348	0.638
BRANIC	5	2.054
BUFFER	5	-----
CHKVAL	3	0.604
CLERTB	4	0.000
COPY	3	-----
CPYPAT*	14	3.021
DATE*	1	0.000
DECRA	60	1.588
DEQL	73	1.346
DESCR	921	-----
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	25	-----
FSHRTN	12	0.000
GETAC	10	0.638
GETBAL*	1	0.172
GETD	47	7.408
GETDC	118	5.025
GETLG	59	0.759

GETLTH	2	0.172
GETSIZ	27	0.397
GETSPC	10	0.017
INCRA	136	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	17	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	33	5.197
LOCSP	79	1.605
LVALUE*	6	0.207
MAKNOD	13	0.172
MNREAL	1	0.000
MNSINT	1	0.034
MOVA	6	0.397
MOVBLK*	14	0.103
MOVD	147	1.985
MOVDIC	7	0.017
MOVV	16	0.811
MPREAL	1	0.000
MSTIME*	8	0.000
MULT	5	0.120
MULTC	18	0.207
ORDVST*	1	0.000
OUTPUT*	27	0.034
PLUGTB	4	0.000
POP	114	4.282
PROC	172	2.365
PSTACK	5	0.034
PUSH	120	3.091
PUTAC	11	0.448
PUTD	29	0.069
PUTDC	132	3.056
PUTLG	9	0.189
PUTSPC	1	0.138
PUTVC	1	0.034
RCALL	343	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	166	0.673
SETAV	32	1.830
SETF	1	0.000
SETFI	5	0.086
SETLC	28	0.034
SETSIZ	7	0.155
SETSP	18	0.155
SETVA	14	0.051
SETVC	30	0.207
SHORTN	4	0.000
SPCINT*	23	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	105	0.759
ZERBLK	3	0.128

Appendix 2 - 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 which 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 which Relate to Recursive Procedures and Stack Management.

ISTACK	POP	PROC	PSTACK	PUSH
RCALL	RRTURN	SPOP	SPUSH	

Macros which Move and Set Descriptors.

GETD	GETDC	MOVBLK	MOVD	MOVDIC
POP	PUSH	PUTD	PUTDC	ZERBLK

Macros which Modify Address Fields of Descriptors.

ADJUST	BKSIZE	GETAC	GETLG	GETLTH
GETSIZ	MOVA	PUTAC	SETAC	SETAV

Macros which Modify Value Fields of Descriptors.

INCRV	MOVV	PUTVC	SETSIZ	SETVA
SETVC				

Macros which Modify Flag Fields of Descriptors.

RESETF	RSETFI	SETF	SETFI

Macros which Perform Integer Arithmetic on Address Fields.

DECRA	DIVIDE	EXPINT	INCRA	MNSINT
MULT	MULTC	SUBTRT	SUM	

Macros which Deal with Real Numbers.

ADREAL	DVREAL	EXREAL	INTRL	MNREAL
MPREAL	RCOMP	REALST	RLINT	SBREAL
SPREAL				

Macros which Move Specifiers.

GETSPC	PUTSPC	SETSP	SPOP	SPUSH

Macros which Operate on Specifiers.

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

Macros which Operate on Syntax Tables.

CLERTB	PLUGTB

Macros which Construct Pattern Nodes.

CPYPAT MAKNOD

Macros which Operate on Tree Nodes.

ADDSIB ADDSON INSERT

Input and Output Macros.

BKSPCE ENFILE FORMAT OUTPUT REWIND
STPRNT STREAD

Macros which Depend on Operating System Facilities.

DATE ENDEX INIT LINK LOAD
MSTIME UNLOAD

Miscellaneous Macros.

LINKOR LOCAPT LOCAPV LVALUE ORDVST
RPLACE SPCINT TOP VARID

Appendix 3 - 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 desireable to perform as many changes as possible by some systematic, mechanical means (preferrably 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 about 6500 80-character card images. All cards are blank in column 72 and contain sequence numbering in columns 73 through 80. There are two kinds of cards: program cards and comment cards. Comment cards have an asterisk (*) in column 1 and descriptive text of various types in columns 2 through 71. All other cards (about 4800 out of the total of 6500) are program cards. Program cards have 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. All program cards have operations which 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. List 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 which terminates the variable field. Such comments begin in column 36.

The following portion of program is typical.

```
*
*       BLOCK MARKING
*
GCM     PROC    ,                  PROCEDURE TO MARK BLOCKS
        POP     BK1CL              RESTORE BLOCK TO MARK FROM
        PUSH    ZEROCL             SAVE END MARKER
GCMA1   GETSIZ  BKDX,BK1CL         GET SIZE OF BLOCK
GCMA2   GETD    DESCL,BK1CL,BKDX   GET DESCRIPTOR
        TESTF   DESCL,PTR,GCMA3    IS IT A POINTER?
        AEQLC   DESCL,0,,GCMA3     IS ADDRESS ZERO?
        TOP     TOPCL,OFSET,DESCL  GET TO TITLE OF BLOCK POINTED TO
        TESTFI  TOPCL,MARK,GCMA4   IS BLOCK MARKED?
GCMA3   DECRA   BKDX,DESCR         DECREMENT OFFSET
        AEQLC   BKDX,0,GCMA2       CHECK FOR END OF BLOCK
        POP     BK1CL              RESTORE BLOCK PUSHED
        AEQLC   BK1CL,0,,RTN1      CHECK FOR END
        SETAV   BKDX,BK1CL         GET SIZE REMAINING
        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
*
```

Appendix 4 - Differences between Version 2 and Version 3

There are three new macro operations included in Version 3 that were not used in Version 2. One macro has been deleted. A number of Version 2 macros have been changed slightly. Corrections and improved descriptions have been supplied for a number a macros. The character classes used to define syntax tables have been extended and revised. The following lists are provided to assist in converting Version 2 implementations to Version 3.

1. Macro Operations New to Version 3

EXREAL, RCOMP, RLINT

2. Changed Macro Operations

COPY, CPYPAT, ENDEX, INIT, LOAD, LOCAPT, LOCAPV, MNSINT, STREAM

3. Changed Macro Formats

AEQLIC, VCMPIC

4. Deleted Macro

DUMP

5. Corrected or Improved Macro Descriptions

ACOMPC, AEQLC, BKSIZE, CLERTB, DECRA, DIVIDE, EXPINT, GETBAL, GETLTH, INCRA, INCRV, INSERT, INTRL, LINK, LVALUE, MULTC, ORDVST, PLUGTB, POP, RCALL, RPLACE, SELBRA, SETAC, SPCINT, SPOP, SPREAL, STREAM, SUBTRT, SUM, VARID

References

1. Griswold, R. E., J. F. Poage, and I. P. Polonsky. The SNOBOL4 Programming Language. Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1969.

2. McIlroy, M. D. "Macro Instruction Extensions of Compiler Languages. Comm. ACM 3 (April, 1960), 214.

3. Strachey, C. "A General Purpose Macro Generator". Comput. J. 8 (Oct. 1965), 255.

Acknowledgement

The SNOBOL4 system was implemented jointly by the author and Messrs. J. F. Poage and I. P. Polonsky. The author is indebted to Messrs. Poage and Polonsky who have made significant contributions to the development of the macro language and designed many of the individual macros described in this report. The author would also like to thank Messrs. R. S. Gaines, Mr. J. F. Gimpel, and W. M. Waite who have provided numerous criticisms and suggestions which were particularly helpful during the evolution of the macro language. Miss P. A. Hamilton, Messrs. L. Osterweil, M. D. Shapiro, L. Wade, and Miss R. A. Weiss have provided many helpful criticisms and corrections to the manuscript.