A FAST IMPLEMENTATION OF SNOBOL4
FOR THE CDC 6000 SERIES COMPUTERS

by

William Richards Sears, III

A Thesis Submitted to the Faculty of the
DEPARTMENT OF COMPUTER SCIENCE
In Partial Fulfillment of the Requirements
For the Degree of
MASTER OF SCIENCE
In the Graduate College
THE UNIVERSITY OF ARIZONA

1974
STATEMENT BY AUTHOR

This thesis has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the library.

Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: [Signature]

APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date shown below:

R. E. Griswold
Professor of Computer Science

[Signature]  November 23, 1974

Date
ACKNOWLEDGMENTS

The author wishes to express his gratitude to Timothy Tye who originally inspired this work. Special recognition should be given to Frederick Druseikis for following and understanding most of the algorithms and for his many suggestions and comments. John Doyle, John Hallyburton, James Gimpel, David Hanson and Murray Sargent III also provided useful implementation suggestions and criticisms.

A special debt of gratitude is owed to my committee, Douglas Brotz, David L. Clark and Ralph E. Griswold. They have contributed heavily to this project both in physical support and in useful advice and encouragement.

I would also like to thank Control Data Corporation, and the staff of the University of Arizona Computer Center for making available resources that made this project feasible.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF ILLUSTRATIONS</td>
<td>vii</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>viii</td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>Motivation</td>
<td>3</td>
</tr>
<tr>
<td>Design Goals</td>
<td>3</td>
</tr>
<tr>
<td>Approach</td>
<td>4</td>
</tr>
<tr>
<td>Prerequisites</td>
<td>5</td>
</tr>
<tr>
<td>2. DATA REPRESENTATION IN SIXBOL</td>
<td>6</td>
</tr>
<tr>
<td>Descriptors</td>
<td>6</td>
</tr>
<tr>
<td>Representation of Source-Language Data</td>
<td>9</td>
</tr>
<tr>
<td>STRINGS</td>
<td>9</td>
</tr>
<tr>
<td>NULL STRINGS</td>
<td>11</td>
</tr>
<tr>
<td>INTEGERS and REALS</td>
<td>12</td>
</tr>
<tr>
<td>PATTERNS</td>
<td>13</td>
</tr>
<tr>
<td>ARRAYS</td>
<td>16</td>
</tr>
<tr>
<td>TABLEs</td>
<td>16</td>
</tr>
<tr>
<td>NAMES</td>
<td>18</td>
</tr>
<tr>
<td>EXPRESSIONs</td>
<td>19</td>
</tr>
<tr>
<td>CODE</td>
<td>20</td>
</tr>
<tr>
<td>Defined Data Types</td>
<td>20</td>
</tr>
<tr>
<td>Symbol Table</td>
<td>21</td>
</tr>
<tr>
<td>3. TRAPPED VARIABLES</td>
<td>26</td>
</tr>
<tr>
<td>Motivation</td>
<td>26</td>
</tr>
<tr>
<td>Implementation</td>
<td>28</td>
</tr>
<tr>
<td>4. STORAGE MANAGEMENT</td>
<td>33</td>
</tr>
<tr>
<td>Memory Layout</td>
<td>33</td>
</tr>
<tr>
<td>Titled Blocks</td>
<td>37</td>
</tr>
<tr>
<td>Storage Allocation</td>
<td>39</td>
</tr>
<tr>
<td>Storage Regeneration</td>
<td>41</td>
</tr>
<tr>
<td>Marking</td>
<td>41</td>
</tr>
<tr>
<td>The Marking Algorithm</td>
<td>42</td>
</tr>
<tr>
<td>Marking Procedures for Various Data Objects</td>
<td>43</td>
</tr>
</tbody>
</table>
### TABLE OF CONTENTS—Continued

<table>
<thead>
<tr>
<th>Class 1: NAME datatype</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class 2: String descriptors</td>
<td>43</td>
</tr>
<tr>
<td>Class 3: Code blocks</td>
<td>44</td>
</tr>
<tr>
<td>Class 4: Data blocks</td>
<td>44</td>
</tr>
<tr>
<td>Class 5: Tables</td>
<td>45</td>
</tr>
<tr>
<td>Cleaning up the SUM</td>
<td>45</td>
</tr>
<tr>
<td>Compression Pass</td>
<td>47</td>
</tr>
<tr>
<td>Optimizations</td>
<td>48</td>
</tr>
</tbody>
</table>

5. REPRESENTATION OF THE OBJECT PROGRAM

- Specification of Operands and Operators | 54 |
- Saving Variables on the Stack | 55 |
- Statement Structure | 57 |
- Instruction Structure of Generated Code | 59 |

6. THE COMPILER

- The Syntax Analyzer | 63 |
- Statement Compilation | 68 |
- Code Generation | 69 |
- Emission of Object Code | 70 |

7. FUNCTION LINKAGE FOR PROGRAMMER-DEFINED FUNCTIONS

- Defined Function Return | 76 |
- OPSYN | 77 |

8. PATTERN MATCHING

- Control Structures and Pattern Blocks | 80 |
- Concatenation | 81 |
- Alternation | 81 |
- Unevaluated Expressions | 84 |
- ABORT | 87 |
- The Subject String | 89 |
- Value Assignment | 89 |
- ARBNO | 93 |
- Implementation of Pattern Matching | 95 |
- Concatenation | 96 |
- Alternation | 99 |
- ARBNO | 102 |
- Naming Operations | 103 |
- Coding Details | 104 |
- The Cursor | 104 |
- Operations Performed at MSCS | 105 |
- Operations Performed at MFAIL | 107 |
# TABLE OF CONTENTS—Continued

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9. TRACING</td>
<td>108</td>
</tr>
<tr>
<td>Trace Function Invocation</td>
<td>109</td>
</tr>
<tr>
<td>Value Traps</td>
<td>111</td>
</tr>
<tr>
<td>Label Tracing</td>
<td>111</td>
</tr>
<tr>
<td>Function Tracing</td>
<td>112</td>
</tr>
<tr>
<td>Keyword Tracing</td>
<td>114</td>
</tr>
<tr>
<td>10. KEYWORDS</td>
<td>118</td>
</tr>
<tr>
<td>Overview</td>
<td>118</td>
</tr>
<tr>
<td>Statement Monitoring Keywords</td>
<td>120</td>
</tr>
<tr>
<td>Trap Setting Keywords</td>
<td>121</td>
</tr>
<tr>
<td>Procedures Associated with Each Keyword</td>
<td>122</td>
</tr>
<tr>
<td>Table Organization for Keywords</td>
<td>125</td>
</tr>
<tr>
<td>11. CODING CONVENTIONS FOR SIXBOL</td>
<td>127</td>
</tr>
<tr>
<td>The Implementation Language</td>
<td>127</td>
</tr>
<tr>
<td>Subroutine Structure</td>
<td>127</td>
</tr>
<tr>
<td>Call and Return Sequences</td>
<td>130</td>
</tr>
<tr>
<td>Logical Control</td>
<td>133</td>
</tr>
<tr>
<td>Coding Source-Language Procedures</td>
<td>135</td>
</tr>
<tr>
<td>Register Usage in SIXBOL</td>
<td>137</td>
</tr>
<tr>
<td>APPENDIX A: REGISTER USAGE IN SIXBOL</td>
<td>139</td>
</tr>
<tr>
<td>A-Registers</td>
<td>140</td>
</tr>
<tr>
<td>X-registers</td>
<td>141</td>
</tr>
<tr>
<td>B-registers</td>
<td>142</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>143</td>
</tr>
</tbody>
</table>
# LIST OF ILLUSTRATIONS

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Symbol Table</td>
<td>23</td>
</tr>
<tr>
<td>2.</td>
<td>Trapped Variable Structure for I/O Associations</td>
<td>29</td>
</tr>
<tr>
<td>3.</td>
<td>Layout of the Allocated Data Area</td>
<td>36</td>
</tr>
<tr>
<td>4.</td>
<td>Layout of Memory in SIXBOL</td>
<td>38</td>
</tr>
<tr>
<td>5.</td>
<td>A Part of Dynamic Storage Before Regeneration</td>
<td>50</td>
</tr>
<tr>
<td>6.</td>
<td>Compression Barriers</td>
<td>50</td>
</tr>
<tr>
<td>7.</td>
<td>A Schematic Representation of a Code Tree</td>
<td>67</td>
</tr>
<tr>
<td>8.</td>
<td>Layout of a DEFINE Block</td>
<td>74</td>
</tr>
<tr>
<td>9.</td>
<td>Stack Configuration During Pattern Matching</td>
<td>88</td>
</tr>
<tr>
<td>10.</td>
<td>Pattern Matching Stack Basing Pointers</td>
<td>91</td>
</tr>
<tr>
<td>11.</td>
<td>Modification of Exit Paths by Concatenation</td>
<td>98</td>
</tr>
</tbody>
</table>
ABSTRACT

SNOBOL4 presents a number of implementation problems due to its powerful string manipulation operations and lack of declarations. Originally, it was written in a machine independent macro language and installed on nearly all large scale scientific computers. Unfortunately, SNOBOL4 systems based on the macro implementation are both slow and large. To alleviate these problems, several alternate implementations have been done, most notably SPITBOL for the IBM 360 and SITBOL for the DECSYSTEM-10. Each of these subsequent implementations achieved a considerable improvement in the overall efficiency of the system, while imposing only minor restrictions on the SNOBOL4 source language.

SIXBOL is designed as a similar system for the Control Data 6000 series computers. Although the architecture of these machines is poorly tailored for a language primarily concerned with string processing, their widespread availability makes a faster and smaller implementation of SNOBOL4 quite desirable. This paper describes implementation techniques needed to implement a full SNOBOL4 system for these machines. Special emphasis is placed on data structures required to make efficient use of the 6000 series hardware. The techniques described are an example of
carefully tailoring the design of a large scale programming project to a particular machine architecture when maximum system efficiency is a primary goal.
CHAPTER 1

INTRODUCTION

SNOBOL4 (Griswold, Poage and Polonsky 1971) is a high-level language that has been implemented on almost all large-scale scientific computers. It has powerful string manipulation facilities, automatic storage allocation and deallocation and extensive program trace facilities. In addition, SNOBOL4 allows runtime access to the symbol table, invocation of the compiler, and operator redefinition.

Because of the dynamic nature of SNOBOL4 and the fact that it contains no declarations, the original implementation of SNOBOL4 (Griswold 1972), referred to in this paper as the macro implementation (Griswold 1971, 1972), was interpretive. This implementation was distributed in a relatively machine-independent macro source, from which an implementation could be completed by defining the macros for a given machine. This produced a large number of highly source-language compatible implementations, but these implementations were not necessarily well tailored to the hardware they were running on in terms of efficient use of machine capabilities. The first attempt at an alternate implementation was SPITBOL (Dewar 1971). This highly successful implementation for the IBM 360/370 used a compiler
that generated machine instructions to implement the operations specified in the source program. Some compromises were made in the SNOBOL4 source language however, and in particular operator redefinition and programmer-defined trace functions were not implemented.

Another successful implementation was SITBOL (Gimpel 1974), which was done for the PDP-10. This interpretive implementation supports almost of the SNOBOL4 language features in a compatible manner. By carefully tailoring the code to the PDP-10 hardware, a much smaller implementation resulted than the macro implementation done for the same machine.

Three other systems that deserve mention are FASBOL (Santos 1971), ELFBOL (Gimpel and Hanson 1973) and CISBOL (Tye 1972). FASBOL is a true compiler, generating inline code for simple operations and producing a loadable relocatable object module that can be linked with separately compiled subroutines. This is achieved by substantial changes to the source language and not implementing certain difficult features such as CODE and OPSYN. ELFBOL has not yet been implemented, although the design is fairly complete and well documented. ELFBOL would be similar to SITBOL with respect to its faithfulness to the SNOBOL4 source language, yet would use a compiler that generates machine instructions. The nature of this code is similar to an interpreter, except that some interpretive overhead is eliminated.
CISBOL was designed for the Control Data 6400, but was never completed. The storage management techniques in SIXBOL are adapted from that paper.

Motivation

The primary motivation for this project is the lack of other efficient implementations of SNOBOL4 for the Control Data 6000 and CYBER 70 and 170 series computers (hereafter referred to as CDC 6000 series). The need for such a system is accentuated by the paucity of software of any kind for performing nonnumeric tasks on these machines. Further, widespread availability of CDC 6000 series computers and the recent announcement of the CYBER 170 series that are software compatible with earlier versions, insures such an implementation of a long and useful life. Finally, a new SNOBOL4 implementation is seen as an ideal context for the development of new implementation techniques for SNOBOL4 language features.

Design Goals

The primary goal of the implementation, hereafter referred to as SIXBOL, is to produce a SNOBOL4 system for the CDC 6000 series that is both smaller and faster than any existing implementation for these machines. A primary emphasis is on source-language compatibility in order to maintain the machine-independent aspects of SNOBOL4 source programs. The implementation is aimed at supporting the
SNOBOL4 source language available in Version 3.7 (Griswold et al. 1971) of the macro implementation. In particular, full tracing facilities, the use of the CODE function, and dynamic redefinition of operators (with two exceptions) are to be permitted.

Another goal will be to develop a system that can be used with either of the standard character sets supported by Control Data depending by setting an assembly switch. Presently, there is very little software available for the CDC 6000 series computers that allows use of the extended character set.

**Approach**

SIXBOL will use a compiler to translate the source program into machine instructions. In spite of this, however, the system will still be conceptually interpretive, since no operations will be compiled in line. This is a natural consequence of a language in which none of the operations are fixed at compile time. Patterns will be implemented in a similar manner, with executable instructions that direct the matching process.

Trapped variables (Gimpel 1974) will be used in a very general manner. Any natural or nonnatural variable can have two classes of traps, and within these classes there can be an arbitrary number of procedures associated with the reference to the variable. This facility will be used in
implementing unrestricted use of tracing and input/output associations, and implementation of keywords.

A notable departure from previous implementations is the elimination of the quickscan mode of pattern matching. It has been noted that almost all programs that run in this mode will also run in fullscan mode, and the implementation of the heuristics is fraught with difficulties. Many patterns can probably be rewritten so that they run faster in fullscan mode anyway, since the heuristic testing is time consuming.

Prerequisites

Understanding this paper requires a knowledge of COMPASS (Control Data Corporation 1973a), the assembler for the CDC 6000 series computers, and the macro implementation (Griswold 1972). The coding examples will be given in COMPASS (Control Data Corporation 1973a), frequently using symbolic register designations. These are defined in Appendix A.
CHAPTER 2

DATA REPRESENTATION IN SIXBOL

Descriptors

SNOBOL4 has a relatively large number of built-in datatypes. In addition to conventional datatypes such as integer, string and real, SNOBOL4 supports patterns, tables, arrays, code and programmer-defined data objects. SNOBOL4 supports each of these, moreover, without any compile-time declarations whatsoever. Any variable in SNOBOL4 can contain any type of data at any time during program execution. Further, objects of different datatypes may be assigned to a given variable at different times.

In order to support this essential language feature, a descriptor is used to represent all source-language data. Two essential pieces of information that a descriptor contains are the type of data and its value. Since most data objects in SNOBOL4 have the property that they may be arbitrarily large, a value is normally stored as a pointer to the actual object. By this technique it is possible to represent an object of arbitrary size in a descriptor of uniform and relatively small size.

Since descriptors are manipulated frequently in SIXBOL, a descriptor that occupies one word is quite
desirable. Fortunately the CDC 6000 series has a 60-bit word, which is quite sufficient for a SNOBOL4 descriptor. In SIXBOL, the descriptor is parcelled out as

```
dt | lp | off | rp
```

The four fields are referred to by the following names:

- `dt`: datatype
- `lp`: left pointer
- `off`: offset
- `rp`: right pointer

The datatype field is then further broken down into subfields as follows:

```
l  r  n  dc
```

The first three subfields are one-bit flags, while the remaining nine bits are the datacode for the descriptor. This nine-bit field uniquely specifies the datacode for the object specified by a descriptor.

The flags are allocated as follows: The first flag, indicated as the `l` flag in the diagram, is called the left pointer flag. The left pointer flag is on whenever the quantity in the left pointer field is a pointer into the allocated data area. This flag is important to the storage management routines. The second flag, indicated as the `r` flag in the diagram, is the right pointer flag. It is
analogous to the left pointer flag and is on whenever the quantity in the right pointer is a pointer to the allocated data area. The third flag, indicated as \( n \) flag, is the **numeric flag**. It is set to zero whenever the descriptor is being used to represent a source-language number so the datacode field may be used to contain part of the value, thus retaining as much precision as possible.

The right pointer is the primary pointer field, and normally contains a pointer to the data object. It is located in the rightmost part of the word because the hardware in the CDC 6000 series makes it easy to perform addressing with these bits.

The left pointer field is a secondary pointer field. It is used for certain types of objects where additional qualification is desirable. This field does not always contain a pointer, however. For example, in a string descriptor this field is used to store the number of characters in the string. When used in this manner, the left pointer flag is not set.

The remaining field in a descriptor is called the **offset field**. This field is 12 bits long and can be used to store a small integer. Usually, it is packed with a bias of 2000 base eight (which will be indicated in this paper as \( 2000B \)) so it can be accessed using the floating point arithmetic hardware. One use of this field is to store the bit offset for a string.
It is of some interest to compare this representation of a descriptor with that used in the macro implementation. There, a descriptor consists of three fields: a T field, V field and an F field where the F field is actually six one-bit fields grouped together. In that implementation, the V field corresponds roughly to the right pointer field, the T field corresponds to the datacode portion of the datatype field.

**Representation of Source-Language Data**

SNOBOL4 supports 10 different datatypes: STRING, INTEGER, REAL, PATTERN, ARRAY, TABLE, NAME, EXPRESSION, defined object and EXTERNAL. For SIXBOL, no attempt will be made to support external objects in the initial implementation.

The class of defined objects appears to be many different types of data at the source-language level, but from an implementation point of view all of these objects are vectors with named fields.

Each datatype in SIXBOL has a descriptor representation and an associated data structure. These are described for each datatype below.

**STRINGS**

A string descriptor is laid out in the following manner:
The datatype field contains 3777B, i.e., the right pointer flag is set, the left pointer flag is not set and a full datatype field is being used. The right pointer field contains a pointer to the first word containing a character in the string, while the offset field contains the bit offset of this character from the leftmost bit in the word. The left pointer field contains the length of the string in characters, and is broken up into two subfields

where \( w \) is the number of whole words required to contain the string, and \( c \) is the number of characters required to complete the count. This is done to avoid frequent use of division since only a floating divide is available and it requires integer operands to be floated, divided, fixed, multiplied and subtracted to get a quotient and a remainder. This is slow, complicated and uses most of the working registers.

A string descriptor for "HELLO WORLD" might take the form
assuming the string were stored in memory starting in the second character of memory location 302B

The offset field is biased by 2000B and always is in the range of 2000B to 2000B + (60 - character width).

The bias is provided so the offset field can be quickly extracted from the descriptor using the CDC 6000 series unpack instruction. This instruction was designed to remove real exponents from floating point numbers. These exponents are biased and the instruction removes this bias. Adding the bias results in the value being ready for use as a shift count after it has been extracted, and makes it easy to insert the offset field using the pack instruction.

NULL STRINGS

The null string, which occurs frequently in SNOBOL4, is given a special representation. A null descriptor has the form
The datacode is the same as for a string but the flags are turned off. Conveniently, there is a hardware test for words of this form since such a word is "positive indefinite" floating point number.

**INTEGERS and REALS**

Integers and reals are discussed together because they share the same datacode. They are distinguished by context when necessary. A numeric descriptor has a datatype field with the first by three bits set to zero. The third bit in the datatype field is a special bit, that, when set to zero, indicates that the descriptor represents a number. This is done so that as many bits as possible can be used to represent the value. The descriptor has the form

```
0 | exp | value
```

All numeric items are stored in floating point format. Integer objects are distinguished from real values by normalizing all real values and leaving integers unnormalized. (A number is considered to be normalized if the high-order bit differs from the sign bit.) For this reason, integer overflow is flagged when this condition is reached during computation. This is convenient because ordinarily integer overflow is ignored on the CDC 6000 series.

To use a numeric descriptor for computation, it is necessary to shift it to the left by three bits. Then the
operands are in normal floating point format and the hardware floating point instructions can be used. Conversion to real is trivial; the operands are simply normalized. Conversion from real to integer requires unpacking, shifting and repacking with a zero exponent. Integer operands are effectively multiplied by eight when they are shifted to the left by three bits. For consistency, real operands are also stored with a constant factor of eight applied to them. Conversion to internal form (i.e., normal CDC 6000 series format) requires that this factor be removed. For integers this is accomplished by shifting, and for reals it is accomplished by subtracting 3 from the exponent.

PATTERNS

Patterns are given the datacode 1, with the flags set to 1,1,1. Each pointer field in the descriptor contains an address in the allocated data area, so both flags are set. The pattern descriptor points to a pattern block that contains executable instructions and some bookkeeping information. The form of the descriptor and the associated block is
The relocation data is used to permit address adjustment when a pattern block is moved or copied to another location in memory. This is because the block contains executable instructions, and some of the address fields reference either other locations in the pattern block or natural variables in floating storage. During pattern building, these bits are used when pattern blocks are copied to adjust addresses within the block so that they continue to point to the same relative locations. During storage
regeneration, these bits are used to maintain both pointers that are internal to the block and pointers to natural variables that may move during compaction.

One bit is allocated for each half word, so it is necessary to allocate one relocation header word for every 30 instruction words in the block. A restriction imposed by this scheme is that instructions containing relocatable addresses must start on halfword boundaries. This is an implementation restriction, since the hardware will allow such instructions to start on the second quarterword boundary as well.

Following the relocation words, there is the block of CDC 6000 series machine instructions that, when executed, direct the matching process. The last descriptor in a pattern block is the title descriptor. The title is used by storage regeneration routines and is required because there are floating addresses with the block that are flagged externally rather than with the normal descriptor flag convention.

The 6000B datatype for title descriptors is a special case. It is used because it represents a negative-indefinite value for which there is a CDC 6000 series hardware test and is thus convenient during storage regeneration. The right pointer field of the pattern block title contains the length of the entire block (including the title descriptor), while the left pointer field contains the
umber of relocation bit words allocated for the block. The
offset field contains the sub-datatype for the title. This
is used so that the type of special processing will be known
when the title is encountered.

ARRAYs

An array descriptor takes on the following form:

\[
\begin{array}{c|c|c}
3002B & \text{len} & n \\
\end{array} \rightarrow \{\text{array block}\}
\]

The right pointer field points to a block of
descriptors whose length is given by the left pointer field.
The first descriptor is used to store the prototype of the
array, which is simply a string. The next \( n \) descriptors
give the length and the lower bounds of the various dimen-
sions in the right pointer and left pointer fields respec-
tively. The number of dimensions is stored in the offset
field of the descriptor (with the usual 2000B bias). The
remainder of the block contains the array elements.

TABLEs

A table descriptor has the form:

\[
\begin{array}{c|c|c}
3003B & \frac{n}{2} + 1 & 0 \\
\end{array} \rightarrow \{\text{header block}\}
\]

The table datacode is 3003, indicating that only the
right pointer field contains a pointer. The left pointer
field contains the length of the bin block that has one word for every two bins plus one descriptor, called the order-chain pointer. It is placed first in the block and contains in the right pointer field a pointer to the last element added to the table. The left pointer field contains a pointer to the first element added. Each table block contains a pointer to the next element added, so it is possible to perform conversion from table to array and maintain chronological order easily.

The bin headers are packed two to a word. The datatype 7770B is an internal datatype used for storage management to flag the action required for table elements. Each table element is placed on a chain computed by a hash function and ordered within that chain with a 12-bit ascension number computed by another hash function. The ascension number is stored in the offset field of the table block header descriptor. The left pointer contains a pointer to the next chronological element in the table. The right pointer field contains the next element on the hash chain.

The table itself takes on the usual form of a hash bin symbol table:
Most names in SNOBOL4 are names of natural variables. In the macro implementation, in which every string is a natural variable, such names are simply string descriptors. In SIXBOL, however, such names are represented by a descriptor type called an *indirect string*. Such descriptors point to the name field of a natural variable. Conversion is made between standard and indirect strings in a uniform manner, depending on the operation being performed. Thus, no difference in the two types of string representation is detectable at the source-language level.
This type of descriptor takes the form:

![Diagram of 4000B descriptor]

The datatype of 4000B is chosen because it represents a negative-infinite floating point quantity and is thus easily tested for. The left pointer field is used so that the flag representation will be consistent. This descriptor is in violation of the nonnumeric usage bit, but this is not a serious problem because these descriptors must be specially tested every time their value is fetched anyway.

Other names in SIXBOL are represented by name descriptors of the form:

![Diagram of 5004B descriptor]

The left pointer is used for consistency with the string name descriptor representation.

**EXPRESSIONs**

Expressions are pointers into a code block giving address is where execution will begin when the expression is evaluated. The expression datacode is 5, and in all other respects, an expression descriptor is similar to a code descriptor described below.
CODE block descriptors have a datacode of 6, with the flags set at 1,1,1. The form of the descriptor is

| 7006B | ttlptr | 0 | transfer |

The code descriptor is similar to a pattern descriptor except for the datacode. The ttlptr points to the code block title, and the transfer address specifies where execution is to begin when the code block is executed. The relationship between the code descriptor and a code block is identical to that of a pattern descriptor and a pattern block. This is natural, since they both represent a block of executable code. The instructions comprising the compiled code are executed by transferring control to the address given in the right pointer field. The actual nature of the code contained therein is described in Chapter 5.

Defined Data Types

Defined object descriptors are given a datacode of 1, with the flags set to 1, 1 and 1. The right pointer gives the location of the first descriptor in the defined object, while the left pointer contains the address of a data definition block. The offset field contains the number of fields for the object.
The data definition block contains a prototype for the defined object followed by the string name of the datatype and the names of the various fields. Each of the field names is stored as an indirect string to insure uniqueness so that the search may be performed to get a field of the correct name without performing string comparison. The prototype descriptor contains all of the fields necessary to create another object of this type, except for the address.

Defined object creation is quite simple. A call is made to get a block of the specified size for the object. The block returned is filled with the arguments passed to the object creation function, and the prototype descriptor is returned after ORing in the block address. Other operations are also simple. FIELD is implemented by indexing appropriately into the definition block and returning the value found. DATATYPE on a defined object simply requires returning the string value contained at the second descriptor of the block.

Note that unlike other implementations of SNOBOL4, no title is required for defined objects. If many objects with few fields are being used, this may save a substantial amount of memory.

Symbol Table

The symbol table is an essential part of the SNOBOL4 system. While most programming languages convert all
source-language symbols into addresses at compile time, SNOBOL4 maintains the symbolic reference facility throughout execution of the source program. Natural variables provide access to all source-language data. In the macro implementation, this link was emphasized by providing that all nonnull strings be included in the symbol table. Though this does offer uniformity, most strings are not actually used as symbols. In SIXBOL, only strings with nondefault attributes are placed in the symbol table.

The SIXBOL natural variable table is organized in a hash-addressed set of chains. The hash code is used to locate on which of 64 chains the variable should be placed. Within each chain, variables are ordered by another 18-bit hash code. This is called an ascension number. The 64 chain headers are stored in 32 words by placing two headers in each word. The header contains the address of the title descriptor of the first variable on the chain. If the chain is empty, a zero value is used. The title descriptor contains both the ascension number and the pointer to the next variable in the chain. The symbol table appears schematically as shown in Figure 1.
Fig. 1. Symbol Table
Each natural variable block is structured as follows:

<table>
<thead>
<tr>
<th>function</th>
<th>(One word of machine code)</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td>block</td>
</tr>
<tr>
<td>name</td>
<td>3777B</td>
</tr>
<tr>
<td>value</td>
<td>dt</td>
</tr>
<tr>
<td>title</td>
<td>6000B</td>
</tr>
</tbody>
</table>

The symbolic name and usage for each of these fields are:

**TITLE** -- NV.TTL, used to bind the natural variable table together

**VALUE** -- NV.VAL, used to store the source-language value of the variable

**NAME** -- NV.NAM, used to store a string descriptor representing the string name of the variable

**LABEL** -- NV.LAB, used to store the label descriptor for the variable

**FUNCTION** -- NV.FNC, Used to store the function linkage for the function associated with this variable. This linkage consists of executable instructions that branch to the appropriate code.

All variables are generated by REFVAR, a service routine used to reference the symbol table. When called, REFVAR hashes the string to locate the appropriate chain on which it searches for the variable. The string name field of each variable block is checked to see if it matches the string being looked up until either a matching name or a
greater ascension number is found. In the former case, the value field of the variable is returned (by location) while in the latter case, a new variable block is inserted at this point on the chain.

When such a variable is newly allocated, the label and function fields are set to undefined label and function procedures. The default value field is the null string.
CHAPTER 3

TRAPPED VARIABLES

Motivation

SNOBOL4 allows any variable, natural or otherwise, to have an input or output association. Any variable may have a value trace placed on it. A reference to a keyword may require that a complex operation be performed. Thus, fetching of the value of some variables in SIXBOL requires that a procedure be executed to obtain the value. This is a common feature of languages such as ALGOL (Naur 1963), but ALGOL requires declarations that permit the compiler to generate special code when a procedure rather than a value is being referenced. In SNOBOL4, this property can be changed dynamically. In fact, any name can have its status changed to that of a procedure.

This procedure must be properly invoked whether or not the reference is direct. Consider

\[
\begin{align*}
X &= \text{ARRAY}(100) \\
\text{INPUT}(.X<5>,5,72) \\
Z &= .X<5> \\
\ldots \\
\ldots \\
Q &= $Z
\end{align*}
\]

which causes a line image to be read from unit 5.
A related process occurs when assignment is made to a variable that is either output associated or value traced. The statement

\[
\text{OUTPUT} = X
\]

causes an output procedure to called, this time with an argument whose value is that of the variable \( X \). This process occurs independently of any input associations that the variable may have. Input associations are procedures that are called when the value of a variable is referenced, while output associations and traces are procedures that are called when an assignment is made to some location. The former will be referred to as value traps; the latter will be referred to as assignment traps. A variable in SNOBOL4 may have several associations at once, as illustrated in the following example:

\[
\begin{align*}
\&\text{TRACE} &= 1000 \\
\text{INPUT}(.\text{OUTPUT},1) \\
\text{OUTPUT}(.\text{OUTPUT},2) \\
\text{TRACE}(.\text{OUTPUT},.\text{VALUE}) \\
\text{OUTPUT} &= \text{OUTPUT} : S(\text{OUTPUT}) \\
\end{align*}
\]

In this program, the variable OUTPUT has three different traps placed upon it: one value trap and two assignment traps. Accessing the value of OUTPUT here results in an input procedure being called, while a assignment to the variable OUTPUT causes an output procedure and a trace procedure to be called.
Implementation

Since there is no restriction on the kind of variable that may be trapped, some uniform representation in a single descriptor (so it will fit in an arbitrary location) is desirable. The object used for this purpose is called a trap descriptor. Whenever the value of a source-language descriptor is needed, the descriptor must be tested for traps before it is used. Similarly, a trap test must be made before any assignment is performed. This testing is very important to the proper operation of the system, since failure to check when a value is accessed causes the wrong value to be fetched, while failure to check before an assignment is made causes the trap to be overwritten.

A trap descriptor, and the associated structure for it, is shown in Figure 2. The descriptor looks much like that used for an indirect string, with a pointer to the value in the left pointer field and a datacode of 4000B. The difference is detected by checking the offset field of the trap descriptor. This field contains an indefinite floating exponent (chosen because there is a hardware test for it) if the variable contains an assignment trap and is negative (sign bit set) if the variable contains a value trap.

When a trapped variable is detected, a call is made to a procedure that interprets the variable. To make coding as simple as possible, these procedures save all working
Fig. 2. Trapped Variable Structure for I/O Associations
registers and return leaving any computed values where they
would have been if the procedure had not been called. Thus,
the input procedure computes a value by performing a read
and returns the value obtained in the register that origi-
nally contained the trap. The calling procedure need not be
concerned about what happens if the trap procedure is
called.

Macros are provided to generate code that performs
the necessary checks. A call to these macros is used every
place in the code where a source-language descriptor is
referenced. Traps must be detected whether or not the type
of trap is applicable to the operation being performed. The
existence of an assignment trap requires special handling
during a fetch operation, as does the existence of a value
trap during an assignment. This code is generated in-line
by the value trap checking macro, since fetches are per-
formed frequently as the the system runs. Since assignment
des performed less frequently, the trap interpretation
routine does the checking.

When there is no applicable trap procedure, the
value in the location specified by the left pointer field of
the trap descriptor is used. Otherwise it is necessary to
save the working registers and invoke the trap procedures
specified by the trap list in sequence. The last procedure
is a special procedure used to end the process. It restores
the registers and insures that everything is left so that
processing can continue in a normal manner when the trap procedure returns.

In the case of an assignment trap, the call is made in the following manner:

```
SA.RG1 VAR        Fetch value
PL    X.RG1,SKIP  Traps are negative
IR    X.RG1,SKIP  And infinite
RJ    ASNTRP      Trap processing
SKIP  SA.RES A.RG1 Perform assignment
```

To preserve transparency, the trap procedure must ultimately reset register A.RG1 to the location where the value is to be saved. Thus, when the operation

```
SA.RES A.RG1
```

is performed, the value is stored in the indirectly referenced location. The convention is made that the value to be assigned is always in X.RES when the assignment trap procedure is called. This is only a slight restriction, since X.RES is the normal value return register for procedures and only X6 and X7 (X.STO and X.RES) may be used for a store operation on the CDC 6000 series.

Value trapping is handled similarly. When a value is needed, the following code is executed (assuming that X.RG2 is the register to be checked):
Note that if the descriptor was really an indirect string, this code fetches the standard string qualifier so that strings are handled in a uniform manner. The routine X2TRP assumes that the value fetch is to be performed from X.RG2. A similar routine, X1TRP, is used to execute value traps specified in register X.RG1.

The trap routines always call procedures in the trap list with two arguments. The first is the location of the trap, the second is some descriptor that is stored in the second descriptor of the two-word trap-list block. In the case of input and output associations, this is an I/O descriptor for the association, while a TRACE trap has a pointer to a trace block. The provision for arguments is to avoid the need for many different but similar procedures. All input associations use the same routine with the different I/O descriptors providing distinction between different files and formats.
CHAPTER 4

STORAGE MANAGEMENT

Memory Layout

The SIXBOL system is divided into several areas. A large fixed block is allocated for the system code, constants and pointers. Another block is allocated for the system stack. The remaining data is divided into two classes depending on whether or not it contains pointers to other objects. This distinction is useful because a practical implementation of SNOBOL4 requires that storage be allocated and de-allocated dynamically. The process for de-allocation, which is referred to as storage regeneration, requires locating every data item that can still be referenced, and hence it is necessary to distinguish between data that can contain pointers and data that cannot.

Strings never contain pointers to other objects, so they are separated into a storage area apart from that used from descriptors. Descriptors may contain pointers, so they are placed in an area that is searched during storage regeneration. The two areas are allocated out of a common region so that the available space may be used in whatever way is required at run time. Free space lies between the two areas and allocation is performed by moving a pointer.
toward the center of the free area. Storage is exhausted when the string area pointer becomes greater than the descriptor area pointer; the storage regeneration routine is then called to try to obtain more free storage. The dynamic storage area is shown schematically in Figure 3.

Strings are allocated in the lower area, (that is, in the area with smaller numeric addresses) because they are frequently formed incrementally and allocation in this manner saves copying and reallocating. Descriptor blocks, on the other hand, are allocated in blocks of known size, and thus can be allocated by decrementing the pointer.

At the high end of the allocated data region is a storage usage map (SUM). The SUM contains a usage bit for every word in core. Thirty-two such bits are used in each SUM word, so one SUM word must be allocated for every 32 every word in the allocated data region. (Thirty-two bits are used because if the number of usage bits in each sum word is a power of two, address computations are much faster.) A SUM word has the following format:

<table>
<thead>
<tr>
<th>mark flags</th>
<th>---</th>
<th>count</th>
</tr>
</thead>
</table>

Following the SUM, and at the highest part of core memory, is the system stack. The stack is placed here to allow for hardware detection of a stack overflow condition.
If the stack does overflow, a memory fault is detected, and a Mode 1 error occurs. This can be trapped using the reprieve facility (Control Data Corporation 1973b). The job is then terminated abnormally with a "Stack Overflow" message if the error processing routine detects that the stack pointer has exceeded the field length.

The remainder of memory is allocated as shown in Figure 4. At the low end is the mask area, which overlays the system communication area after the program begins execution. This area contains 61 words of left-justified masks starting with a zero word, a word with the first bit set, and so forth. The last one is a mask with all sixty bits set. These masks are used frequently both during storage regeneration and string manipulation. On the CDC 5000 series, a table lookup is the simplest way obtain a mask of arbitrary size using as few temporary registers as possible.

Between the masks and the allocated data area is the SIXBOL system, including the program, assembled constants, and fixed data areas. One of these fixed areas is referred to as the basic block because the area of fixed size contains pointers to the allocated data area which may require adjustment or form an access path to data that must be saved.
Fig. 3. Layout of the Allocated Data Area
Titled Blocks

Due to the requirement of the storage allocation routines, a special descriptor, called a title descriptor, is appended to the end (i.e., descriptor with the highest address) of some blocks. Title descriptors are used to indicate that the block requires special handling during storage regeneration. Two types of blocks fall into this category:

- **Type 0**: Natural variable blocks
- **Type 1**: Blocks containing machine instructions

The motivation for a title differs in the two cases. Natural variables have titles because natural variable blocks must be retained in their entirety even if all of the descriptors contain default attributes and only one of the descriptors is referenced. The title serves to bind the block together. Code blocks have titles because the machine instructions do not correspond to the normal descriptor flag conventions, so saving and relocation must be handled in a special way.

Title descriptors have the form:

```
6000B type
```

where type indicates what type of block it belongs to. These are stored internally with a bias of 2000B. The
### Fig. 4. Layout of Memory in SIXBOL

<table>
<thead>
<tr>
<th>Low addresses</th>
<th>High addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Masks</td>
<td></td>
</tr>
<tr>
<td>System and Fixed Data</td>
<td></td>
</tr>
<tr>
<td>Basic Block</td>
<td></td>
</tr>
<tr>
<td>FETs and I/O Buffers</td>
<td></td>
</tr>
<tr>
<td>Allocated String Region</td>
<td></td>
</tr>
<tr>
<td>Allocatable Free Space</td>
<td></td>
</tr>
<tr>
<td>Allocated Block Region</td>
<td></td>
</tr>
<tr>
<td>SUM</td>
<td></td>
</tr>
<tr>
<td>System Stack</td>
<td></td>
</tr>
</tbody>
</table>
pointer fields are available for block-specific uses. For natural variable blocks, the pointer fields contain linkages to other variables in the symbol table. For code blocks the pointer fields contain the length and the number of relocation words associated with the block.

Title descriptors are placed at the ends of blocks because titles are used during the compression phase of storage regeneration. Since descriptor blocks are allocated in the high area of memory, they are compacted by moving them upward. This requires that the descriptor with the highest address in a block be moved first so it is seen before other descriptors in the block. Thus, allocating the descriptor at the end enables the storage manager to take special action at the correct time.

Storage Allocation

Storage allocation in SIXBOL is simple. A pointer is kept to the next available word for each of the string and block storage regions. When memory is needed, an allocation procedure increments the pointer by the number of words requested and returns a pointer to the first word in the block. When the request is larger than the difference between the two storage area pointers, the system must perform a storage regeneration to remove data that is no longer being referenced. If this procedure fails to free enough space, then the program is terminated with a message indicating that insufficient storage is available.
While the process of allocating dynamic storage is simple, its use is full of pitfalls (Griswold, 1972). A misplaced pointer can cause the system to collapse following a storage regeneration. These errors can be difficult to find, since they may manifest themselves long after the original error was made. In particular, it is essential that all blocks be cleared immediately following allocation. Furthermore, when a request for storage is made, it is important that all objects still being used are referenced from a location tended by the regeneration procedures. A problem can easily appear with a temporary pointer left in a register. All active registers containing pointers must be placed on the stack with flags indicating the presence of floating addresses so that they will be properly relocated during the regeneration process.

String allocation requires some care also. Strings are normally allocated at the beginning of a word, since frequently used service procedures run faster for aligned strings. Further, if strings are allocated in whole word chunks, string garbage collection tends to be more complete, since only whole words are retrieved.

One exception to this rule is allowed for string concatenation. If the first argument is the last string allocated, advantage is taken of the fact and the second string is simply appended to the end of it. For this reason, a global string descriptor, LASTSTR, is set up
pointing to the last string allocated. This descriptor must be properly updated by all routines that allocate strings. This optimization is performed because many strings in SNOBOL4 are built as the concatenation of several shorter strings, particularly when performing output.

Storage Regeneration

Storage regeneration in SIXBOL is a two-pass operation that involves marking all descriptors that can be referenced and moving them as far from the center of the allocated data region as possible by copying blocks into space no longer being used.

Marking

The marking process requires that all items that can be referenced have some access path that begins with a descriptor that is either in the basic block area or on the system stack. Included in the basic blocks are such pointers as the set of natural variable hash headers and LASTSTR. The system stack contains intermediate results created during expression evaluation, saved values resulting from programmer-defined function invocation, and the values of registers containing pointers. The natural variables form access paths to anything referenced by them, since all variables can be reached with the indirection operator.

Before any descriptors are marked, each word in the SUM is set to zero. This turns off all mark bits so it can
be determined which descriptors have been reached during the reference search. Every descriptor pointed to from the basic block area is marked, and the marking procedure is called recursively if the marked descriptor itself contains any pointers. The number of descriptors marked depends on the type of descriptor being examined. A descriptor pointing to a block of ten descriptors causes ten descriptors to be marked starting with the one directly pointed to. The marking algorithm is then called ten times, once for each descriptor in the block.

Since the set of natural variables forms the virtual addressing space for the SNOBOL4 source language, these are treated somewhat specially. Since natural variables have a number of attributes that must remain consistent from reference to reference, the requirement is made that if any of the fields are different than the default values, or if any pointers exist to any of the fields in a variable, then that variable must be saved.

The Marking Algorithm

The marking algorithm is basically a recursive graph search. It begins with a specified set of root nodes -- including the natural variable hash headers and the system stack -- and continues with every node referenced from a root node or any node reachable from a root node. As nodes are found, they are marked and the search routine is called
recursively to mark the next set of nodes. Marking allows searching to terminate if there are circular pointers in the structure. Due to the variety of data descriptors used in SIXBOL, the marking algorithm contains many sub-algorithms for marking different types of descriptors. These algorithms are described below.

**Marking Procedures for Various Data Objects**

Class 1: NAME datatype

Only the descriptor pointed to must be marked, along with any descendent descriptors. No problem arises if the name points to a field of a natural variable because a special clean-up pass is made to insure that natural variables are never fragmented. A name that points to a single descriptor in an otherwise unreferenced array causes only that descriptor to be saved. This is acceptable since there is no way to reference the other descriptors from the source language.

Class 2: String descriptors

The word pointed to is marked along with the next n words, where n is the number of descriptors specified in the word count part of the length field of the string descriptor. Additionally, one more descriptor must be marked if the relation

\[ c \cdot \text{cwidth} + \text{offset} > 60 \]
is true, where \( c \) is the character length overflow, \( \text{cwidth} \) is the width of a character in bits, and \( \text{offset} \) is the bits offset of the first character in the first word. The whole formula is

\[
[(c \times \text{cwidth} + \text{offset}) / 60] + w + 1
\]

where \( w \) is the word count in the length field and \([X]\) is interpreted to mean the greatest integer less than or equal to \( X \).

Class 3: Code blocks

The number of words in the block (specified in the right pointer field of the title descriptor) is marked. Then the block itself is searched for pointers by examining the bits in the relocation words contained in the upper part of the code block. These pointers are either to other words in the code block itself (in which case the mark bit will already have been set) or to some descriptor within a natural variable. Since a clean-up pass is made over the natural variables at the end of the mark pass, no other processing need be done at this time.

Class 4: Data blocks

This marking procedure is used for all data blocks whether internal (used only by the SIXBOL system itself) or source-language. The length of these blocks can be determined by examining the descriptor and applying an
algorithm that is a function of the datatype. Each descriptor within the block is marked, and the marking procedure is called recursively for any descriptor that contains pointers itself.

Class 5: Tables

Tables are handled in a manner similar to natural variables. Each hash header is searched for table entry blocks, and those containing nonnull values are marked for saving. The mark procedure is then called recursively on both descriptors within the block. Tables entries with null values need not be saved, since there is no operation that enables one to find out if such an entry exists (conversion to array deletes these). The one exception occurs if the unary name operator has been invoked on such a descriptor, but that reference itself causes the descriptor to be marked and saved.

Cleaning up the SUM

Following the mark pass, all of storage that must be saved has been so marked by a bit in the SUM. The SUM is then processed so that the address adjustment can be computed relatively quickly.

When the mark pass is completed, the count field is zero and bits have been set for every descriptor that can still be referenced. In order to perform pointer adjustment, it is necessary to know how many descriptors are
unused between the current descriptor and the extreme edge of the data area. In Figure 5, a portion of a data region is shown. The descriptor at \((n-4)\) is located such that there is one unused word between its location and the edge of the storage area. When descriptors are moved toward the edge and over the unused one, a reference to that position requires an adjustment of one. Similarly, a reference to the descriptor shown at \((n-7)\) requires an adjustment of two.

To enable this calculation to be performed quickly, the number of unused words represented in all SUM words up to the current one is contained in the count field of each SUM word. This count is performed by starting with the edge SUM word and counting the number of off bits, adding that to a running sum, and placing each partial term in the next SUM word.

This process is done once for each region in the allocated data area, with an ascending sum kept for the string data region and a descending sum for the block region. Included in the partial sum for a given word in the SUM are the bits turned on in the word itself.

Since the string and block data regions grow toward each other, there is a strong possibility that one SUM word will mark some words from each region. This SUM word is called the common SUM word and, since the bit sums for the two regions are not related, the bit sum for this word is
always set to zero. To allow for the computation of the address adjustment in a uniform manner, computations involving the bit sums always reference another SUM word. For a string region reference, bits in the corresponding SUM word are masked from the left to the corresponding bit and counted. This quantity is then added to the bit sum in the preceding SUM word. For a block region reference, bits are masked from the right, and the count of masked bits is added to the sum given in the next SUM word.

The algorithm for computing an adjusted pointer address is as follows:

(1) Determine which segment the pointer references.

(2) Fetch the appropriate SUM word by dividing the offset from the edge of the segment by 2.

(3) Fetch the preceding SUM word to get the accumulated bit sum for all SUM words preceding the one currently being processed.

(4) Count the number of off bits that represent locations between the word referenced in the previous sum word and the location being adjusted.

(5) Add the value obtained in step 3 to the value obtained in step 4.

(6) Adjust the pointer by subtracting the value obtained (if a string address) or adding the value obtained (if a block address).
Compression Pass

To avoid destroying data, the compression pass must be made starting from the edges and moving toward the center. The highest address in the allocated data region is tested first. If it is to be saved, the source and target pointers for the compression operation are both decremented and the descriptor is then tested for pointer flags. Fields flagged as containing pointers are tested to verify that they reference the allocated data region. Such addresses are then adjusted using the algorithm described above.

This process continues until the end of the block segment is reached. As descriptors that are no longer required are encountered, they are passed over by decrementing only the source pointer. When a title is encountered, an appropriate procedure is called to handle the block.

The string area is collected in much the same way, except that no pointer checks are made because strings never contain pointers.

Optimizations

Even though the count can be made in a single instruction, it is still somewhat slow. The process can be speeded up considerably by not relocating pointers known to point to objects that are not going to be moved. During the compression pass, the addresses closest to the center of the data region beyond which no unreferenced data exists are remembered. These points are called the compression
Compression barriers indicate addresses above and below which no pointer adjustment is required. This is shown schematically in Figure 6. An address whose value does not fall between the compression barriers cannot possibly require relocation, since there is no empty space to allow movement.

In practice, the regions between the compression barriers and the extremes of the allocated data area often comprises a substantial amount of floating storage since this data represents the fixed data in SNOBOL4 source programs. Typically included are the compiled program, source-program variables, literals, and patterns built during the initial phase of program execution.

The second technique that can be employed is to regenerate only the string area. Many SNOBOL4 programs generate only string garbage after the initialization phase. A string-only storage regeneration is similar to the complete algorithm except that

1) The recursive search is replaced by a linear pass through memory, searching only for string descriptors.

2) Only the string area is compressed, and only string descriptors require any pointer adjustment. This is done with a second linear pass through the block storage region.

The elimination of the recursive pointer search and half of the storage mapping saves considerable amounts of CPU time.
Fig. 5. A Part of Dynamic Storage Before Regeneration

Fig. 6. Compression Barriers
Selection of a point to try a fast garbage collection is somewhat open to question, but one reasonable algorithm would be to do full collections until little or no core is retrieved in the block region of core. String-only regenerations would then be performed until space becomes cramped, at which point a full collection would again be attempted.

In addition to being useful as an optimization technique, this string-only storage regeneration can be a useful implementation device when attempting to get a part of the system running. Many SNOBOL4 programs certainly will run with no block collection at all, and this algorithm is much simpler than the complete operation.
CHAPTER 5

REPRESENTATION OF THE OBJECT PROGRAM

The compiler for SIXBOL generates machine instructions which, when executed, perform the operations specified in the source program. This approach differs from that taken in the macro implementation and that taken in SITBOL, but is similar to that taken in ELFBOL.

Each SNOBOL4 language operation is specified by a call to a procedure. SIXBOL does not compile any operations in line because of the dynamic definition and redefinition of operations supported by SNOBOL4. Instead, a uniform calling sequence is adopted that is used for the invocation of all SNOBOL4 language operations regardless of the particular syntax used to specify in the source program.

The object code produced is postfix; that is, operands are specified and evaluated before the operation to be performed on them. This type of code results in a simpler runtime support system and somewhat less overhead than the prefix code used in the macro implementation. Unfortunately, however, it is not sufficiently general to allow implementation of all operations in the SNOBOL4 language. The unary * operator, which returns a pointer to the expression that is given to it as an argument without evaluating it, is logically inconsistent with a notation in
which operands are evaluated before the operations are performed. This problem is illustrated in the following example

```
DEFINE("END()")
F = *END()
```

The function END(), if called, terminates the program when transfer is made to the entry label END. If *END() were to be evaluated before the assignment to F, the program would terminate before the operation was completed. This is in clear violation of the semantics of SNOBOL4.

The problem is circumvented by treating unary * as a special case and generating special code for it. When the unary * operator is encountered in the source program, its argument is compiled separately, and a procedure that returns a pointer to this section of code is placed in the main line. As a result of this special action by the compiler, OPSYN of unary * cannot be supported in SIXBOL.

A different problem is encountered with the unary ~ operator. This operator returns the null string if its argument fails, and fails if its argument succeeds. Unlike unary *, there is no requirement that argument evaluation be delayed, since the argument is always evaluated so that its success or failure can be determined. Unfortunately, treating this as an operation requires looking for it anytime a function signals failure. This is a complicated and time-consuming operation, and penalizes many statements
(those that fail) for the sake of the seldom used SNOBOL4 language feature of OPSYN of unary ~. For these reasons, special code will be generated by the compiler for this operator and OPSYN of unary ~ will not be permitted in SIXBOL.

**Specification of Operands and Operators**

SNOBOL4 is a language with a preponderance of binary and unary operations. Only four primitive operations — REPLACE, OPSYN, APPLY and ITEM — have more than two arguments. (As in SITBOL (Gimpel 1974), pattern matching with replacement is treated as two operations). For this reason, the function calling sequence is designed to make calls to procedures of one or two arguments particularly efficient by allocating two register-pairs for argument passing. Additional arguments, if provided, are passed on the system stack.

A call has four parts:

1) FETCH2 Load last argument into a register
2) FETCH1 Load the next-to-last argument
3) SETCNT Setting of the argument count
4) JUMP Branch to the procedure

The argument fetches may involve either a reference to a natural variable or to a previously computed result. When a natural variable is referenced, the address of the value field of the variable is placed in the A-register associated with the argument being fetched. This simultaneously causes the value of the variable to be loaded
into the corresponding X-register. If the function being called requires its arguments by name, the address in the A-register is used, while if the value is needed, the descriptor in the X-register is used.

Previously computed results are fetched by popping the top of the stack -- an operation that requires three instructions. All computed results are left on the stack with two entries, a location and a value. Such results are called name-value pairs. The value computed is always placed first. The second part of the pair, the name descriptor, contains the location where the value returned can be found. This will be the location of a variable if return is by name and the address of the location on the stack containing the value if return is by value.

The two types of return can be distinguished by comparing the value of the name location to the bottom address of the system stack. Addresses falling below it are valid names, while those above it are not.

Saving Variables on the Stack

As mentioned previously, some SNOBOL4 language operations have more than two arguments. For an operation of n arguments, the first n-2 of them are to be passed in the top 2*(n-2) locations of the system stack as name-value pairs. To do this, code is generated to place variables on the stack. The code generated for
REPLACE(A,B,C)

consists of the following macro operations

PUSH A
FETCH2 C
FETCH1 B
SETCNT 2
JUMP REPLACE

If the first argument to REPLACE had been the result of some operation, the result would have already been on the stack, so no PUSH code would have been needed. The statement

REPLACE(TRIM(A),B,C)

generates

FETCH2 A
SETCNT 1
JUMP TRIM
FETCH2 C
FETCH1 B
SETCNT 3
JUMP REPLACE

Note that the last argument is fetched before the one preceding it. The reason for this is obvious if one considers a statement such as

A* B + C* D

which generates
 bytes stacked. When the arguments to PLUS are being fetched from the stack, the last argument (that is the result of C * D) is at the top, so it must be obtained first.

**Statement Structure**

SNOBOL4 is a statement-structured language, with each statement having an associated identifying number, as well as success and failure goto locations. The former goto is taken if all operations in a statement succeed, while the latter is taken at the point when an operation fails. An actual goto need not be specified for either of these: the next statement is used for any missing goto fields.

Unlike most other implementations of SNOBOL4, no statistic tallying is performed at the beginning of each statement. This is performed during goto processing. The number of statements executed since the last goto is computed by subtracting the statement number of the statement to which a transfer was last made from the current statement number. &STCOUNT is then incremented by this quantity plus
one. A last transfer trace word, LSTNOTR, which contains the source and destination statement numbers, is used so the &STCOUNT can be updated when gotos are performed, and &LASTNO can be computed if needed. &STLIMIT is also checked when a goto is processed to verify that the statement execution limit has not been exceeded. (This limit may be exceeded by a small amount (the upper bound is the length of the source program) but its primary purpose of breaking hopelessly looping programs is still maintained.)

Statement initialization requires two instructions. The first sets up statement basing in register B.FRT. This address can be used to fetch the current statement number if needed, and is the address where transfer is made in the event of statement failure.

The second instruction sets up the function linkage register, B.LNK. When a source-language procedure returns, the value of B.LNK is incremented by 2 and return is made to that location. Since source-language procedure calls are normally two words long, linkage is provide from operation to operation without requiring setting of this register in the object code.

Following statement initialization comes the instructions compiled to represent the statement itself. This is followed by a statement trailer, which consists of three parts: success goto, statement number and block basing, and the failure goto. The success goto consists
either of a call to the goto function if an S-goto is provided, a branch around the failure goto if only an F-goto is provided and is null if an unconditional goto (U-goto) field appears. The statement number and block title pointer is sandwiched between the goto fields and is provided so the current statement number and the location of the title for the block containing the statement can be easily located by fetching the memory cell at B.FRT-1. This word contains the two instructions

\[
\begin{align*}
SB0 & \quad \text{block title address} \\
SB0 & \quad \text{statement number}
\end{align*}
\]

and is always in the location immediately preceding the failure goto part of the statement. The SB0 instructions are both no-operations, and are provided so execution can fall through the word if necessary.

The failure transfer part generates code only if an F-goto or an unconditional goto is specified. If neither appears, execution continues with the next statement. An unconditional goto is placed in the failure goto part of the statement. If the statement succeeds, execution simply proceeds through the statement number word.

**Instruction Structure of Generated Code**

Until now, the operations for which the compiler generates code have been discussed in relatively high-level
terms. This section describes exactly what machine instructions are used to implement the various operations.

There are two types of argument fetches. The first is used to fetch the value of a natural variable, while the second is used to fetch the value of the last computed result. The code for the first, which is denoted symbolically as a FETCHn operation, is

$$SA.RGn \ variable\_addr+NV.VAL$$

Registers are referred to symbolically in SIXBOL, and the register designators for argument registers are RG1 for the next-to-last argument and RG2 for the last argument. The term in the address field "variable_addr" refers to the address of the variable in dynamic storage while the term "NV.VAL" refers to the offset of the value field within the natural variable.

The code for a fetch of a computed result, which is designated as a POPn operation, takes the form

$$SA.RGn \ B.STK-B.ONE$$
$$SB.STK \ A.RGn-B.ONE$$
$$SA.RGn \ X.RGn$$

In this code there is a reference to register B.ONE, a register that always contains the constant 1, and B.STK, the stack pointer. Since the stack pointer always points to the location of the next available free word, popping the last value requires that it be decremented by one. The resetting
of B.STK also involves a second decrement because the returned result consisted of two words. Finally, the value is fetched by referencing the location pointed to by the name part of the returned result.

In actual code, one more instruction is frequently required since the popping of an argument requires 45 bits of instructions and the standard calling sequence assumes that two words are required to set up the call. In the event that more than two words are required, it is necessary to increment the linkage register by 1. This is done most efficiently by the instruction

\[ \text{SB.LNK } \text{B.LNK+B.ONE} \]

The operation of SETCNT is implemented in one instruction. A B-register, B.CNT, is used to transmit the argument count. The code for setting this is

\[ \text{SB.CNT } \text{number of args} \]

The actual transfer to the function itself is done with an unconditional branch instruction. No saving of the return address is done here, since the linkage register is set at statement initialization and is incremented by 2 every source-language procedure. If a function call that requires more than two words of code is compiled, instructions to adjust the linkage register are output so that it remains synchronized with the object code.
From the standpoint of a compiler writer, SNOBOL4 is a relatively simple language. Each statement consists of five parts, any of which may be omitted. The basic parts are: the label, subject, pattern, replacement and goto. Although all of these parts are optional, the subject must appear in order for a pattern or replacement to appear. Determination of which part is being examined can be made by examining one character if the last part processed is known.

Within these basic parts, there is a substructure that is easily described as an operator grammar. Several relatively simple schemes exist for parsing this type of syntax. In SIXBOL, a variation of recursive descent is used because the semantic routines associated with the parse are conveniently conceived of within this framework.

During the syntax scan, the compiler builds an internal representation of the statement in the form of code trees. The motivation for this comes from the fact that the code produced in SIXBOL is primarily concerned with the last arguments to a function, and these are of course encountered last in the syntax scan. The use of code trees allows easy determination of the relative positions of operands.
After code trees have been built for each part of the statement, a routine is called to list them and generate the object code.

The Syntax Analyzer

The syntax analyzer scans the source program, determines the role of each token, and builds code trees. There are two scanning routines, called ELMNT and EXPR. ELMNT is used to scan elements, which are variable references, function calls, literal references, array and table references, unary operator references and parenthesized expressions. EXPR is called to scan either elements or sequences of elements separated by binary operators.

The distinction between the two types of syntactic scanning is useful for two reasons: First, the distinction is made in the syntax of SNOBOL4 because the subject of a statement is defined to be an element rather than an expression. Second, this logical division makes the operation of the compiler somewhat clearer. ELMNT and EXPR are mutually recursive, since evaluation of a parenthesized expression (an element) is done with a call to EXPR, and analysis of the elements of an expression is done by ELMNT.

EXPR must make precedence distinctions between the various operators to cause correct operator associations to be made when operators of varying precedences are encountered in an expression. This requires scanning ahead before generating any code for a binary operator expression. In
EXPR, this is done be performing a one-operator lookahead. Rescanning of operators is avoided by providing that EXPR may be called in either of two modes -- referred to as left mode and right mode.

Left mode is used whenever EXPR is being called to evaluate an expression not bounded on the left by a binary operator, while right mode is used to evaluate an expression when the left operand and the operator have already been scanned. Internally, the two modes are distinguished by the arguments passed to EXPR. Zero arguments signal a left mode call, while nonzero arguments signal a right mode call. In left mode evaluation, the first argument and operator must be scanned by EXPR and evaluation continues until a terminator is found. In right mode evaluation, the first argument and operator are not scanned, since they are already known, and evaluation terminates if either a terminator or an operator of lower precedence than the one passed in the call is encountered. In the latter case, this operator is returned to the caller so it can be used in subsequent evaluation.

The operation of EXPR can be seen in the following example. Consider the following SIXBOL expression

\[(A + B * C + D)\]

After the left parenthesis has been scanned, EXPR is called in left mode. The first operand is processed (by
ELMNT) and the + operator is scanned. Then the next element is processed and a lookahead is performed to determine the precedence of the next operator. In this example, it is *, which has a higher precedence than +. Precedence rules require that the expression bound by the * operator be processed first, so EXPR is called recursively in right mode using the second operand scanned as the left operand and the operator * as arguments. This second incarnation of EXPR scans the next operand (C) and looks ahead at the next operator, which is +. Since this is of lower precedence than *, the expression B * C is processed and this second incarnation of EXPR returns the processed expression and the lookahead operator to its caller. Now the first incarnation of EXPR binds its original left operand with the + that it scanned and the result returned by the second call to EXPR. These results are now joined together and EXPR looks for another operand to join with the + operator that the second incarnation of EXPR found. The variable D is located, and the operator lookahead is again performed. This time a terminator is found (the right parenthesis) so EXPR joins the left part of the expression with the + operator and the last operand D and returns the tree representing the complete expression.

EXPR and ELMNT both return code trees. These trees are linked together as more elements and expressions are parsed. The trees are built of nodes that are formed of two words:
The first word is a descriptor that contains either a pointer to a natural variable (for literal and variable references) or a functional descriptor that contains an argument count in the left pointer field and a pointer to the function linkage descriptor in the right pointer field. The second word of a code tree node consists of a pointer used to link nodes together to form trees. The LSON field is used only for functional nodes, and points to the first element in a linked argument list that is chained together with the RSIB fields. The end of this list is signaled by a zero RSIB field. An example of a code tree for the expression given in the last example is shown in Figure 1. The recursive descent parsing technique is motivated by the need to retain pointers to the various subtrees until they are completely linked to the other nodes. In the example, the first operator parsed, a +, was not linked to its right sibling until the end of the parse, when the variable D was encountered.
Fig. 7. A Schematic Representation of a Code Tree
Statement Compilation

Statement compilation begins by searching for a label. This construction is detected by a letter or digit as the first character in a line. If a label is found, the address where the next code will be placed is inserted into the label field of the variable specified. The subject is scanned next, using the parsing routine ELMNT. Following the subject, a search is made for a binary operator. If a character that starts an element is located instead, the compiler generates a pattern-matching operator and calls EXPR to get the pattern. When EXPR returns, the subject and pattern trees are joined, and a scan is made for a replacement. If an equal sign is found, then the assignment operator tree is built using the subject pattern and replacement parts. If an equal sign is found before an element start character, EXPR is called to get the replacement, and the subject and replacement trees are linked with an assignment functional node.

After a code tree for the expression part of the statement has been generated, the goto field is processed. The compiler can generate three goto trees, although only two are allowed in a legal statement. If an unconditional goto is found, a parse tree for a U-goto is saved and similarly for the S- and F-gotos. A check is then made to be sure that not both a U- and F- or S-gotos exist, and the parse stage of the compilation is complete.
Code Generation

Code generation begins by generating the statement heading, code for the statement expression and finally code for the S- and F-goto trees. The routine that lists code trees is called PUBTRE, mnemonic for PUBLish code TREes. PUBTRE begins by testing the root node of the tree to see if it is the rightmost node (indicated by a null RSIB field). If it is, a test is made to see if it is a variable or a function call. Function nodes require that the argument trees be listed first, which is done by recursively calling PUBTRE on the LSON node. Variable nodes generate code to either fetch the name-value pair in the appropriate argument passage register or to save the name-value pair on the stack, depending on whether or not the argument is in one of the last two positions.

When a node that represents the next-to-last argument is located, evaluation of the node is made but generation of argument fetch code is delayed until after the last argument is evaluated. This is done with another recursive call to PUBTRE. Then the next-to-last argument fetch is actually generated. (The motivation behind this is described in the last chapter.) At this point there are a variety of special cases that may exist due to an attempt to produce the smallest code possible and still retain the correct relationship between the code size and the linkage
register settings. These are all handled at the point where the next-to-last argument fetch code is being output.

**Emission of Object Code**

The code emitter in SIXBOL is an interpreter that uses a language specifically tailored to the generation of CDC 6000 series machine instructions. Operations include output of 15- and 30-bit instructions, saving of forward reference pointers, as well as forcing word alignment and ensuring that sufficient room remains in the current code block for code that must be generated contiguously. Each operation has an argument, plus various modifiers that are used to specify whether the instruction must be aligned and how the address part of 30-bit instruction is to be interpreted.

The basic operations that this interpreter allows are:

- `.15 (code)` output 15-bit instruction
- `.30 (code)` output 30-bit instruction
- `.FUP (n)` Insure that n words follow
- `.W (loc)` Output word in location loc
- `.PLUG (addr)` Plug forward reference
- `.XCT (loc)` Call subroutine
- `.DONE` End output of this block

Several modes are available for the address fields of 30-bit instructions. They are:
R  Set relocation flag for this address
I  Fetch address portion indirectly
S  Save the location where this instruction is generated for later reference in location loc.

The S-flag is used in conjunction with the .PLUG operation to handle forward references. An F-flag that is used to force the current instruction to a new word may also be used with any command except .DONE.

Code modules are saved as blocks of code emitter commands. When a specific section of code is to be output, the compilation output interpreter is called with the address of the module block and the location of an argument block, if any are required. These arguments are fetched through the use of the I-flag in output directives.

An example of a code module for statement heading is

```
.FUP 2           Insure 2 contiguous cells
.30 (SB.FRT 0),SR Relocatable forward reference
.30 (SB.LNK 1),SR Similar, to start linkage
.PLUG 1          Satisfy forward reference 1
.DONE            End of block
```

The motivation for the interpretive code emission is the advantage of flexibility. This form makes it easy to understand the code that is being output for a given statement, and thus presumably easier to modify should errors or improvements be found later. The nature of the compiled code has been re-thought several times, and there
is little reason to believe that no further changes will be desired.
CHAPTER 7

FUNCTION LINKAGE FOR PROGRAMMER-DEFINED FUNCTIONS

Programmer-defined functions present special problems, since they require the evaluation of one statement to be suspended, and another statement to be initiated. Before branching to the specified transfer address for the defined function, the current environment must be saved along with the values of arguments and local variables. Then the argument and local variables must be given their initial values.

Invocation of all programmer-defined functions involves this same process so they are all handled by a common function invocation routine, DEFFNC. Such functions differ in name, local and argument variable lists, and entry point. The link descriptor for a programmer-defined function contains a reference to the define block and a jump to DEFFNC. This is done with two instructions

\[
\begin{align*}
\text{SA3} & \quad \text{DEFBLOK} & \quad \text{get definition block} \\
\text{JP} & \quad \text{DEFFNC} & \quad \text{invoke the function}
\end{align*}
\]

The define block for a function contains a list of variables to be used as the function name, arguments and local variables as well as the entry point label identifier. Such a block is laid out as shown in Figure 8.

73
<table>
<thead>
<tr>
<th>3770</th>
<th>len</th>
<th>a</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3776</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3776</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3776</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 8. Layout of a DEFINE Block
The right pointer field of the first descriptor in the block contains a pointer to the remainder of the block. This is used by the storage regeneration procedures in saving the block properly. It is needed because no length information appears in the linkage descriptor and this may be the only pointer referencing the define block. The second descriptor contains a pointer to the entry point label, and the third descriptor contains a pointer to the function name. Pointers to the argument variables follow in the order they are referenced in the define prototype. Local variables occur at the end of the block.

When a function is called, DEFFNC starts by swapping the current value of the argument variables for the values to be passed. A check must be made for traps both in the actual arguments and the variables to be used by the function. In the former case, value traps must be honored, while in the later case, it is sufficient to note that the trap is there and place the actual argument in the value location of the trap. As the arguments are being evaluated and inserted, the current values of the variables must be saved on the system stack so they may be restored when the function returns.

Local variables are handled in the same way as arguments except that the initial value for all local variables is null. Omitted trailing arguments are treated as local variables (i.e., are initialized to null), while excess arguments are just evaluated.
The address of the define block is saved on the stack following the last of the saved local variables. Proper return also requires that B.LNK and B.FRT be saved (with flags indicating floating addresses) and the old stack base is saved. The stack base is then reset to the current location so that statement failure will not clear the function linkage from the stack. DEFFNC then initiates the function by picking up the label descriptor address in the define block and executing a GOTO to that label after indicating no label tracing is to be performed with a negative second argument.

**Defined Function Return**

Defined function return involves capturing the returned value, restoring the old values of variables contained in the define block (again testing for traps so that they will not be overwritten), and setting up the correct return to the system. SNOBOL4 allows three different types of defined-function returns: RETURN, FRETURN and NRETURN. The processing required to restore variables and release stack space is common to all of them, only the return to the system is different. When a transfer is mode to the return procedure, an indicator is set to signal the type of return it will make.

Restoring of values begins by rebasing the stack and restoring the old values of B.LNK and B.FRT. The define block pointer is at the top of the stack and it is used to
find the last variable specified. Values are restored starting with the last local variable and continuing until the function name is encountered. Its current value is placed in X.RES (anticipating return) and the value of the variable before the call was made is restored. The symbol table and stack have now been returned to their original condition (except for programmer-caused side effects) and the indicated return can be made.

Return is handled in the normal manner. For standard RETURN, the value is already in X.RES so the current value of B.STK is placed in A.RES and transfer is made through B.LNK. FRETURN is handled by transfer to FAIL, which ultimately returns control through B.FRT. NRETURN requires that a representation of a location as value be converted to an internal return by name. This is equivalent to the unary $ operator, so transfer of control is passed to the internal procedure for unary $ and it handles the final return.

**OPSYN**

As in the macro implementation, OPSYN is quite simple to implement in SIXBOL because all operations are accessed through linkage descriptors. In most cases, all that must be done is to copy this descriptor to some specified location.

OPSYN can be used to change definitions of three types of operations: Functions, unary operators and binary
operators. The type of operation being changed affects which linkage descriptor is used. Function linkages are contained in the natural variables, while unary and binary operator linkages are stored in operator table's.

Operator tables are linear arrays of linkage descriptors. A call to UNOPORD or BNOPORD is made to get the index into the link descriptor vector. These functions take a string argument and either return the index into the table if the operator exists or signal failure if it does not.

OPSYN uses the third argument to determine which table should be searched. If this argument is null, the function linkage is obtained from the natural variable table and copied into the appropriate target variable. If the third argument to OPSYN is 1 or 2, UNOPORD or BNOPORD is called on each argument before using the symbol table. The locations obtained from these searches are used for the definition change.

The only time a simple descriptor copy does not work for OPSYN is when the definition about to be copied is a traced defined function. In this case, the trace is removed before the descriptor is moved. Determining whether the linkage contains a trace is simple, since all traced functions link to DFNCTR. A simple address comparison indicates the presence of the trace. If a trace exists, the original descriptor is reconstructed as
The address of the definition block is given by the first descriptor in the function trace block.
CHAPTER 8

PATTERN MATCHING

Control Structures and Pattern Blocks

Patterns in SIXBOL consist of machine instructions that specify what operations are to be performed during a pattern match. Performing a pattern match involves calling the pattern as a procedure with the subject string as an argument. Success or failure is indicated by the procedure and this determines the outcome of the match. As the pattern executes, certain side effects may occur; immediate and conditional assignment may be made and any section of SNOBOL4 program may be executed as a result of evaluation of an unevaluated expression.

The instructions in a pattern block do not perform any matching, but instead merely direct the matching process. All matching is done with matching procedures that are called from the pattern block. The pattern block itself specifies the control structure of a pattern, i.e., the order in which various parts of the pattern are to be used depending on success or failure of previous parts. This approach is somewhat different from the technique described by Gimpel (Gimpel 1973).
Concatenation

From an implementation standpoint, a pattern consists of a number of components — calls to matching procedures — that are connected together using certain control structures. The simplest control structure in patterns is sequentiation. This control structure is implied when the pattern building operation of concatenation is performed. For example, the pattern

"*" SPAN(" ") SPAN("BC") RPOS(0)

specifies four matching procedures that are to be called in sequence. The entire pattern fails if any component fails. Success is signaled only if each component is successfully executed in sequence.

Alternation

Another control structure that may be used to connect pattern components is alternation. A pattern such as

A | B

specifies that a match with this pattern is to be considered successful if either A or B signals success. In matching, A is called first. If it signals success, B is never called, but if A signals failure then B is called. Thus, B is only called when A fails. The process associated with
Alternation is more complex, however, when more than one alternative is found in a pattern. In the example

\[(A \lor B) (C \lor D \lor E)\]

Alternation of \(A \lor B\) implies that either \(A\) or \(B\) will satisfy the pattern, while the pattern \((C \lor D \lor E)\) can be satisfied by any of \(C\), \(D\) or \(E\). Thus there are six possible combinations of calls that can result in a successful path through the pattern. This can be diagrammed as

\[
\begin{align*}
\text{The diagram can be interpreted as follows:} \\
\text{Sequanciation is indicated by components lying on the same path. For example} \\
A & \rightarrow B & \rightarrow C
\end{align*}
\]

Indicates that \(A\) must be followed by \(B\) which must be followed by \(C\). Alternation is indicated by a node with
multiple exit paths, followed by a common return. Thus,

\[ A \cup B \]

represents \( A \cup B \).

In SIXBOL a pattern is a program that implements a search for a successful path through the pattern graph. The algorithm is as follows. Sequentiation requires calling each component or pattern in the specified order. Alternation involves saving all alternative paths on a stack when an alternation node is reached, then proceeding with the first path from the node. If a component ever fails, the last alternative is popped from the top of the stack and execution begins with the indicated alternative. The process terminates successfully if the end of the pattern is reached, and fails if a component fails and no more alternatives remain on the stack.

Assuming that the component FAIL was added to the last example, the structure would then be
When this pattern is executed, the following components are called

A, C, FAIL
D, FAIL
E, FAIL
B, C, FAIL
D, FAIL
E, FAIL

Note that alternatives are tried starting with those most recently encountered. When the search algorithm backs up, it tries to back up as little as possible in its search for a successful path.

Unevaluated Expressions

A third structure available to a pattern is the unevaluated expression. An unevaluated expression is a pattern whose value is determined at the point when the match is to be performed. When an such a component is evaluated, the value returned is then used as part of a pattern. This facility is implemented in a way analogous to a subroutine call (which it is, since a pattern is really a program). Return information is saved on the stack and the pattern resulting from the unevaluated expression is called (executed). Like any other procedure called during the matching process, it may either fail or succeed, and like other components, it may leave alternates that must be tried in the event of failure later on. This is shown in an example such as
P = X *F FAIL
F = A | B
STR P

When the match is performed, the matching procedure associated with X is called. The unevaluated F is evaluated. The result, A | B, is then used in the pattern by calling the pattern recursively. If A succeeds, the primitive component FAIL signals failure and the stack is checked for an alternative. B is the only alternative in this pattern, which is located in F. If this alternative were simply picked up and matching were to continue at that point, B could finish the match successfully merely by signaling success itself. This would result from the fact that the pattern F is a subpattern of P, and thus its signal of success is nested within the calling pattern P.

This presents no problem on the first call, since it is known that the result of evaluating a deferred component is being referenced and the appropriate linkage can be set up. For alternatives, however, some special provision must be made. This is done by capping the stack with a flag that signals that alternatives contained below the cap belong to a pattern that was called from the current structure and using these alternatives will require restoring of the linkage information so that a proper return may be made. In the previous example, the stack would have the following structure after the match of component A.
Note that the alternates included in the unevaluated expression $F$ are surrounded by the cap on one side and the linkage information on the other. The linkage information is used to return to the calling pattern ($P$) when evaluation of $F$ is complete.

The use of a capped stack allows the system to handle unevaluated expressions in the same storage block that other stack functions are handled. This is possible since the nature of unevaluated expressions is such that information about alternates is always added and removed in a last-in, first-out manner. This is not true however, of the return linkage. This may be needed before all of the alternates generated by the unevaluated expression are consumed. A pattern such as

$$F = A \mid B \mid C \mid D \mid E$$

when used as part of $P$ (see the example above) may be called five times at different places. The ability to find linkage information quickly is important to the running speed of the system, so they are laced together with pointers. Because
such linkage information is woven through the stack, this organization is referred to as a *laced* stack. A laced stack is analogous to a multi-dimensional array in that it consists of two or more nested structures contained in the same space.

The representation of the stack after the matching process has completed evaluation of the external pattern $F$ is shown in Figure 9. This figure also shows that the first call is handled in the same way as subsequent calls, with a cap that prevents system information from being overwritten. The cap for the level-zero call links to a procedure to be called to terminate the pattern match unsuccessfully. The current link pointer is used when successful matching is complete to perform the return.

**ABORT**

ABORT is a special pattern component that terminates a match unsuccessfully. It may be encountered at any time in the matching process. Whenever it is encountered, the matching process terminates.

ABORT may be encountered when nested pattern invocations have occurred to some arbitrary level, so proper clearing of the stack requires that a pointer be maintained to the stack location where the matching process began. ABORT clears the stack to that point and terminates the matching process unsuccessfully.
Fig. 9. Stack Configuration During Pattern Matching
The Subject String

Pattern matching is performed on a string. A cursor is used to indicate how far the match has advanced along the string during the operation, and to indicate where any indicated replacement should be performed if it terminates successfully.

When an alternate is tried, the cursor position must be restored to the value it had when the alternative was first placed on the stack. This is simple to implement; whenever an alternative is added to the stack, the current cursor position is stored with it. When the alternative is to be executed, the global cursor position is restored to the value so saved. With this in mind, a more complete stack diagram can be described. This is shown in Figure 10.

Value Assignment

SNOBOL4 provides a facility for assigning substrings matched by components of a pattern to source-language variables. This feature requires that cursor pairs be remembered. Two kinds of value assignment are supported, immediate and conditional. The immediate assignments are performed as they are encountered, while the conditional assignments are saved until the entire matching process terminates and are only performed if the match is successful.
Because implementation of immediate value assignment is simpler, it will be discussed first. An immediate value assignment structure is built with the binary operator $. A pattern such as

\[ X \ A \ $ \ B \ Y \]
generates several components internally. Three special ones are used in the immediate naming process: save_cursor, which saves the current cursor on the stack; pop_cursor, which removes the most recently added cursor position pair from the stack; and assign_cursor(x), which assigns the substring represented by the cursor pair at the top of the stack to the variable x.

\[ X \ (\text{save} \_ \text{cursor} \mid \text{pop} \_ \text{cursor}) \ A \]
\[ + \]
\[ \text{assign} \_ \text{cursor}(B) \ Y \]

When evaluation of X is complete, the cursor is saved so that it may be used if A is matched successfully. The alternative pop_cursor is provided in case A fails; in this event, the cursor saved is removed so that it will not interfere with other assignments remaining in the pattern. If A succeeds, the procedure assign_cursor(B) is called which pops the last cursor position saved, generates a string descriptor using the cursor positions and the subject string, and assigns the string to the natural variable B.
ABORT Pointer

Alternates to P

Current Pointer

linkage for P

linkage for \*F

cursor B
cursor C
cursor D
cursor E
cap

subsequent to \*F

Fig. 10. Pattern Matching Stack Basing Pointers
It also removes the pop_cursor alternative since the cursor pair it was to remove is no longer there.

The old cursor values are maintained by lacing, and an external pointer is used to locate the most recent entry. Also associated with the entry is a back pointer to the previous precursor position. This value is used to update the top of the cursor list whenever an entry is removed (popped) from the list.

Use of this structure presents no problem even when complicated nesting is required. A pattern such as

\[ \text{NULL} \; \text{\$} \; X \; \text{\$} \; Y \; \text{\$} \; Z \]

maps into

\[
\begin{align*}
(\text{save\_cursor} & \mid \text{pop\_cursor}) \\
+ \quad (\text{save\_cursor} & \mid \text{pop\_cursor}) \\
+ \quad (\text{save\_cursor} & \mid \text{pop\_cursor}) & \text{NULL} \\
+ \quad \text{assign\_cursor}(X) \\
+ \quad \text{assign\_cursor}(Y) \\
+ \quad \text{assign\_cursor}(Z)
\end{align*}
\]

Conditional assignments are more complicated because the actual value assignment is not made until the pattern match is successfully completed. Until then, any provisionally matched component may lose that status as a result of failures later on. Consider the situation encountered during the matching with the following pattern.

\[ P = (A \; . \; B \; | \; C \; . \; D) \; X \]
In the event that X fails following matching of component A, the conditional assignment to B must not be made.

Implementation of this operation requires internal components for maintaining a list of provisional value assignments during matching. If the match is successful, these provisional assignments are performed starting with the first. The extra list managing procedures required are add_assignment, which adds a provisional assignment to the provisional list, and remove_assignment, which removes the latest entry. This list is stored as by linking the various entries together on the system stack in the same way the the cursor pair list is maintained for value assignment.

Conditional value assignment is similar to immediate assignment in that a save_cursor component is placed before the tagged component and an assignment procedure is added following it. An extra component is required to remove the name association if the matched component is subsequently backed into. This is shown in the generated components for X . N

which are

\[
(\text{save}_\text{cursor} \upharpoonright \text{pop}_\text{cursor}) \ X \\
+ \ (\text{add}_\text{assignment}(N) \upharpoonright \text{remove}_\text{assignment})
\]

The remove_assignment procedure is always backed into before any alternates to X are reached.
ARBNO

ARBNO allows for repeated instances of a pattern, and is somewhat analogous to an iteration facility. Functionally, it provides an inexhaustible supply of alternates. The pattern resulting from ARBNO(P) causes a new pointer to P to be placed on the stack as an alternative each time P matches. A special case must be made, however, on the first time through, since the operation requires that P not be invoked unless it is backed into. ARBNO(P) can be diagrammed as follows:

![Diagram](image)

The order of alternates is important, of course, and the convention that alternates are tried in a clockwise direction will be used here.

This implementation of ARBNO is quite efficient, and requires no special handling since it is expressed in terms of operations that have already been defined. Since it is efficient, it may be freely used in defining the primitive patterns ARB, BAL and SUCCEED. The definitions for these
primitive functions are:

\[
\begin{align*}
\text{ARB} &= \text{ARBNO}(\text{LEN}(1)) \\
\text{BAL} &= \text{GBAL} \text{ARBNO}(\text{GBAL}) \\
\text{SUCCEED} &= \text{ARBNO}(	ext{NULL})
\end{align*}
\]

The internal primitive pattern GBAL is simulated by the SNOBOL4 statements

\[
\text{GBAL} = \text{NOTANY}("()") \mid "(" \text{ARBNO}(\#\text{GBAL}) ")"
\]

but is of course implemented in a more efficient manner by scanning for parentheses, counting each left parenthesis as 1 and each right parenthesis as -1 then terminating the scan when the count reaches zero or less than zero. If the count is zero GBAL matches successfully, otherwise it fails.

**Implementation of Pattern Matching**

Like the code that represents the source program, each primitive pattern is represented as a call to some procedure. Although the operations are fixed at pattern-building time, space considerations make coding even simple matching operation in-line undesirable.

A typical pattern component is the one created by the pattern-valued function LEN. Such a block takes on the form
Each component starts by setting a linkage register to the return point and picking up an argument. The procedure is entered with a jump, while return is handled through the linkage register if the match procedure is successful. Failure is handled by transfer to a routine called MFAIL.

Concatenation builds a structure that implements sequentiation. As a simple example, consider

LEN(1) "A"

The code to implement this pattern takes the form

```
SB.LNK TAG TAG1 set first return
SBO 0 filler
SA.RG1 ARG1 arg to LEN
JP MLEN call match routine
ARG1 VFD 2/1,10/12,42/0 character is 12 bits
TAG1 SB.LNK TAG2
SBO 0 filler
SA.RG1 ARG2 arg to match string
JP MLIT
ARG2 (string descriptor for "A")
TAG2 JP MSCS announce success
```
pattern-building function for concatenation copies all but the last word of the first pattern as well as all of the second pattern. Relocation is performed using the relocatable address flags contained in each pattern block. Each address so flagged is checked to see that it points to a location inside the block and such addresses are adjusted by the displacement of the move.

In complicated patterns, there may be several exits to the success routine MSCS. The pattern concatenation procedure must locate all of these and redirect them to link to the pattern being added. Fortunately, these exits are easy to find, since they all exist in words with the same bit pattern. A simple linear search is performed on the first pattern argument for exit branches, and these are appropriately modified. This is shown schematically in Figure 11. Note that the last exit is never modified, it is simply eliminated so that the matching process flows into the next component.
Fig. 11. Modification of Exit Paths by Concatenation
Alternation

Alternation joins two patterns together so that the result implements the alternation operation. Code must be generated to save the address of the alternate and the current cursor position on the stack and execute the first component. Successful match of either component causes transfer to MSCS.

Schematically, the operation is

![Diagram of alternation operation]

Note that no modification of the components themselves takes place except for address relocation required during the copy operation. Each pattern still terminates from the same point that it originally would
have. The block resulting from alternation consists of code to push the second argument’s address on the stack as an alternate as well as the two blocks joined together.

Multiple alternates are handled with several calls to the alternation-construction procedure, exactly as specified by the source language. If the diagram just given was created as the intermediate result of the expression

\[
\text{PAT} = A \mid B \mid C
\]

the next pattern built would be
The first alternate pushed on the stack is \( \gamma \), the address for component C; the second is \( \beta \), the address of component B. Since the history list is a LIFO structure, the first alternate tried will be B, followed by C. If A were to have any alternates of its own, these would be pushed later still, so that they would be tried before A returned failure and B was called.

Since patterns consist of machine instructions, the alternation construction operation must generate machine code to perform the PUSH described. Remembering that the system stack pointer is located in register B.STK, and that this register always points to the first free location on the stack, the operation of saving the cursor and alternate address is as follows:

\[
\begin{align*}
\text{SBO} & \quad 1 & \text{number of alternates} \\
\text{SX.STO} & \quad \text{ALT} & \text{address of alternate} \\
\text{BX.STO} & \quad \text{X.CUR+X.STO} & \text{cursor and address} \\
\text{SA.STO} & \quad \text{B.STK} & \text{push} \\
\text{SB.STK} & \quad \text{A.ST0+1} & \text{adjust stack pointer}
\end{align*}
\]

Here ALT is the location of code for the alternate. If a second alternate needs to be pushed, the following code can be used:

\[
\begin{align*}
\text{SBO} & \quad 2 & \text{count of alternates} \\
\text{SX.STO} & \quad \text{ALT2} & \text{second alternative} \\
\text{BX.STO} & \quad \text{X.CUR+X.STO} & \text{merge with cursor} \\
\text{SA.STO} & \quad \text{B.STK} & \text{save on stack} \\
\text{SX.STO} & \quad \text{ALT1} & \text{1st alternate} \\
\text{BX.STO} & \quad \text{X.CUR+X.STO} & \text{merge with cursor} \\
\text{SA.STO} & \quad \text{A.ST0+B1} & \text{save on stack} \\
\text{SB.STK} & \quad \text{S.STK+1} & \text{increment pointer}
\end{align*}
\]
The address of the next available stack position is computed from the address of the last store when several addresses must be pushed in a series. The first operation is a no-op, since $B_0$ is always zero. It is used so that the number of alternates can be determined quickly during pattern building. This does not cause any inefficiency because the header must be allocated in whole words and the thirty bits used for this count are left over.

**ARBNO**

The pattern built by ARBNO takes the form

```
JP P.END  first match null
P.ST BSS 0  put pattern arg here
  .
  .  (argument to ARBNO)
P.END BSS 0
SX.STO P.STO  where P begins
BX.STO X.CUR+X.STO  merge cursor
SA.STO B.STK  B.STK+1
SB.STK B.STK+1  adjust stack pointer
SB0 0  filler
JP MSCS  announce success
```

The operation invoked on the first pass is one of merely saving the address of $P$ and the current cursor. If backed into, $P$ matches once and new alternates are placed on the stack along with the cursor position just computed. Note that ARBNO's implicit alternatives are tried before any alternatives laid down as alternates to $P$ on previous invocations are tried, exactly as specified in the example.

The fact that ARBNO is pattern iteration is clearly brought out in the implementation. Only one stack location is ever used by the iteration itself, regardless of how many times it repeats. (Of course, the pattern P may itself leave an arbitrary number of untried alternatives on each pass through it).

Naming Operations

The constructor procedures for binary $ and . are quite simple. The concatenation operation is invoked twice, once to append a header to the tagged pattern, then again to attach the trailer. The header is used to save the cursor prior to matching, and provides an alternative that removes the unneeded cursor position from the stack should the component fail. The trailer code either places the assignment on a list of assignments to be performed if conditional assignment was specified, or performs the assignment. The name argument is always placed in the trailer code, which has the form

```
SB.LNK  P.END
SA.RG2  DESCR
  JP    IVAP.
  DESCR  DATA (descr)
P.END  JP    MSCS
```

For conditional assignment, the routine is CVAP. instead of IVAP.
While IVAP. must actually perform the assignment, CVAP. merely links the assignment with other assignments on a chain. The chain consists of blocks of the form

<table>
<thead>
<tr>
<th>Name or Expression Descriptor</th>
<th>last name association</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cursor position 1</td>
<td>strip procedure</td>
</tr>
<tr>
<td>Cursor position 2</td>
<td></td>
</tr>
</tbody>
</table>

The name or expression descriptor is whatever the second argument to binary $ was when the pattern was built. (Names are converted to value form.) The first cursor position descriptor holds the cursor position saved by the header pattern. The second descriptor links back to the next assignment and contains the post-cursor value. The last descriptor is the special alternate that is placed on the stack to remove this block if it is ever backed into as an alternate.

**Coding Details**

The Cursor

The cursor is carried throughout pattern matching in an X-register denoted symbolically as X.CUR. It has the form

```
2000B | words | bits | 0
```
This form is convenient because it makes character addressing within the subject string simple and fast. The right pointer flag is set because alternates are frequently placed in the low-order 18 bits and pushed on the stack. Since patterns are in floating storage, such addresses must be flagged to insure proper pointer adjustment if a storage regeneration should occur. This datatype only allows for pointer adjustment, which is acceptable since the argument pattern is still in existence and the link is sufficient to save the block.

Operations Performed at MSCS

MSCS is reached whenever a pattern completes successfully. This can occur at the base level or when an unevaluated expression component completes. In the former case, termination processing continues and conditional assignments are performed next. In the latter case, however, control must now return to the pattern that called the unevaluated expression in the first place. Before control is returned, the cap is placed on the stack. The cap has the form
RESTAR is a routine that bumps the function call level for matching by picking up the backpointer for the unevaluated expression pattern. If an alternate from the deferred pattern again succeeds, MSCS places the back pointer and RESTAR cap on the stack again. If all of the alternatives from the unevaluated expression are exhausted, the system picks up the other cap, which has the form

XMFAIL is a routine that removes this block and updates the pattern recursion pointer to the location indicated by the pointer labeled next. The back pointer described for the previous diagram points to the descriptor pair shown here.
The return address is a pointer into the pattern that calls the unevaluated expression.

Operations Performed at MFAIL

Primitive pattern components indicate match failure by transfer to MFAIL. This routine simply pops the stack, restores the cursor from the word picked up, and transfers to the address given in the right-most 18 bits of the word.
CHAPTER 9

TRACING

SNOBOL4 allows tracing of a variety of events at the source-language level; assignment to specified locations, transfer to a source-language label, invoking of programmer-defined functions and certain system actions with associated protected keywords may all be monitored through appropriate calls to the primitive function TRACE.

A call to the TRACE function has four arguments: a location to be monitored, the type of tracing to be performed, an optional tag and the procedure that is to be invoked when the traced event occurs. From an implementation standpoint, there are eight different events that can be traced:

1. Assignment to a source-language variable
2. Branch to source-language label
3. Call of a programmer-defined function
4. Execution resulting in a conditionally fatal error (setting &ERRTYPE)
5. Call or return of a function (setting &FNCLEVEL)
6. Execution of a statement (setting &STCOUNAT)
7. Failure of a statement (setting &STFCOUNT)
The keyword tracing facility is treated specially for each one of these because in SIXBOL there is no location associated with a protected keyword. In return for the complexity resulting from this design, the source-language facilities specified for the language can be implemented with virtually no overhead to users who choose not to use these features. In SIXBOL, tracing a protected keyword is usually performed by changing one word of the system at runtime. The code that is changed cause the trace to be executed. No time is spent testing locations to find out whether a trace has been set during normal system operation.

Tracing of functions is handled by changing the branch addresses in the linkage fields of traced functions to the location of a procedure that can handle the trace. Return tracing is handled by changing the return addresses before calling DEFFNC. Label tracing is handled with a flag in the label descriptor.

**Trace Function Invocation**

When a trap is triggered, all active registers are saved using certain conventions. A.RG1 and A.RG2 are assumed to contain addresses and are saved with appropriate flags (it should be remembered that no particular difficulty is encountered if either A1 or A2 contains a value that is outside of the allocated data area -- the storage regeneration procedures always range-check pointers to make sure they point to locations between the compression barriers;
other locations are simply left unchanged). X.RG1 and X.RG2 are assumed to contain arguments and are saved. B.LNK and B.FRT are assumed to contain return addresses and are saved accordingly.

Recall that assignment traps are always invoked with two arguments; the first specifies the location of the trap and the second is associated with the particular assignment trap procedure invocation. For tracing, this argument points to block of two descriptors.

```
{proc}
```

The first descriptor points to the procedure associated with the trace, while the second contains the TAG, if it was specified when the trace was made.

When the trace invocation routine first gets control, the value of &TRACE is tested and if it is nonzero an attempt is made to convert the address of the trap into an indirect string. If this can be done (by looking where the natural variable title should be and checking whether or not it is really there), it is used as the first argument of the trace procedure. Otherwise, a source-language NAME descriptor is generated. The second argument is simply the value of the second word of the trace block.
Before the trace procedure is called, the values of B.FRT, B.LNK and &TRACE are saved. Then the trace routine is called as a procedure. When the trace procedure returns B.FRT, B.LNK and &TRACE are restored, &TRACE is decremented by one and the trace invocation procedure itself returns.

**Value Traps**

Value tracing traps are handled using a trapped variable. Since a variable may have been associated with an indefinite number of trap procedures, no problem is encountered in tracing an unprotected keyword or a variable with an output association. The order in which trap procedures are executed must be maintained. Keyword traps are executed first, then trace traps, and finally I/O associations. When a call to the TRACE function is made, the variable is checked to see if a trap already exists. If it does, the TRACE trap linkage is added to the trap procedure list, otherwise a new trap list is formed and a trap is set for the variable.

Traces that have null tags and use the standard tracing procedure all share the same trace block. If either of these arguments is defined in the TRACE call however, a special trace block is built to specify the nondefault parameter or parameters.

**Label Tracing**

A label descriptor normally has the following layout:
When a trace is placed on the label, the address of the trace block is placed in the left pointer field of the label descriptor like this:

```
dt  0  stno  {stmt loc}
```

This of course requires setting of the left pointer flag, which is the left-most bit in the descriptor. This has the convenient property that there is a hardware test for negative words that can be used to test for label traps. If one is found, the trace invocation procedure is called using the block whose address is specified in the label descriptor.

**Function Tracing**

ALL, RETURN and FUNCTION tracing are handled by changing the function linkage descriptor. A function linkage descriptor for a programmer-defined function normally has the form:

```
SA3 DEFBLOK  JP DEFFNC
```

For tracing this is changed to
here location the TRBLOK contains a seven-descriptor block of the form:

```
+-----------------+-----------------+
| 7770  | *+1   | 2006B      |
+-----------------+-----------------+
| {DEFBLOK}       |
+-----------------+-----------------+
| {CALL trace proc} |
+-----------------+-----------------+
| {CALL trace tag}  |
+-----------------+-----------------+
| {RETURN trace proc} |
+-----------------+-----------------+
| {RETURN trace tag} |
+-----------------+-----------------+
| {FUNCTION trace proc} |
+-----------------+-----------------+
| {FUNCTION trace tag} |
+-----------------+-----------------+
```

When the traced function procedure DFNCTR is called, checks the various fields of the TRBLOK to determine which, if any, actions must be performed. Function tracing is trapped by changing the transfer address in the DEFBLOK point to a function trace invocation procedure. In order to do this without destroying system integrity, the internal function level is bumped by saving the return address pair, LNK and B.FRT, on AUXSTK (a special system stack used for synchronous processes like function tracing). These registers are then set so DFNCTR will regain control following argument evaluation.

After the arguments have been evaluated by DEFFNC in the normal manner, control passes to the location specified
in the define block. This now contains a location inside DFNCTR. DFNCTR restores the address in the define block and invokes trace procedures in the normal manner. When CALL or FUNCTION tracing is finished, traps can be set in B.LNK and B.FRT to insure that DFNCTR regains control again when the programmer defined function returns. If FUNCTION or RETURN tracing is specified, the appropriate trace procedure is invoked. When it returns, B.LNK and B.FTRN are restored and control returns to the object program.

Keyword Tracing

Each of the traceable protected keywords monitors some important system function. The implementation philosophy used in SIXBOL allows keywords to be implemented as functions that compute the source-language value when it is requested, hence there is no uniform manner in which unprotected keywords are assigned values. Fortunately, of the four traceable keywords, only &STCOUNT does not have associated with it the execution of some distinct piece of code in the SIXBOL system. When a trace is placed on &ERRTYPE, &FNCLEVEL or &STFCOUNT, a trap is placed in code that is normally associated with the event to be monitored. The trap is set as follows: at the beginning of the code that implements the process, a jump is inserted that transfers control to a special routine to handle a trace on the keyword. The first thing such a routine must do is execute
the instructions overwritten by the trap. The value of &TRACE is tested and decremented, and if the value is nonzero, tracing is invoked using a trace block set aside for the keyword. Upon return from the trace procedure, execution continues at the location following the trap.

The &ERRTYPE trace trap is placed in the ERROR processing routine. The trap follows testing for a positive value of &ERRLIMIT. The &STFCOUNT trap is placed in statement failure processing. Trap code for &STFCOUNT is complicated by the necessity to test the value of B.FRT to insure that a statement failure offset is contained therein. Some routines, such as unary ~ and DFNCTR, insert special addresses in B.FRT so that they can regain control if failure is signalled later on. Such address can be detected because they are always negative. If one of these is found, tracing is not performed.

&FNCLEVEL is modified in two places; it is incremented in DEFFNC and decremented in the defined function return code. Traps for this keyword must be placed in both location.

Tracing of &STCOUNT is difficult because there is no single point in the system that is entered at statement initialization. The approach in SPITBOL (Dewar 1971) was to modify the compiled program, but this is impractical in SIXBOL because the only process associated with statement initialization is the setting of B.FRT. This process only
requires 30 bits of machine instructions and no suitable replacement code exists that will fit in the space (A subroutine call with an argument requires at least 60 bits).

In SIXBOL, the approach is to modify the program one statement at a time. The trace procedure starts the process by locating the next instruction, which is done by starting with the word pointed to by B.FRT and searching for an instruction of the form:

SB.FRT BO+addr

The search never requires testing more than one word because the code at the failure transfer address is either a goto (one word in length) or the next statement itself. When the statement is found, the first word is saved and a trap is written so the &STCOUNT trace procedure will be invoked if execution continues with that statement.

In addition to the next statement trap, a trap is also placed in the GOTO procedure in case execution does not continue with the next statement. One of these traps will be reached, and the &STCOUNT trace routine will gain control before the next statement is executed. When this procedure is called, the next statement trap is removed and a new trap is placed in the statement following the one about to be executed. This statement is found by looking at the statement initialization instruction and using the failure offset address to locate the next instruction. This process
continues until the trace is removed by STOPTR or &TRACE becomes equal to zero. The trace procedure itself is executed in the normal manner using a trace block set aside for &STCOUNT tracing.
CHAPTER 10

KEYWORDS

Overview

The keyword facility is a symbol table for certain values that allows specification or examination of global system parameters. In the macro implementation, these variables are handled as simple storage cells that are checked and set by the system in a straightforward manner. The value of keywords such as &STNO and &FNCLEVEL are incremented and decremented each time the associated actions occurs. Parameter keywords such as &ANCHOR are tested each time the operations are performed. The differentiation between protected and unprotected keywords is handled by return signals, and required conversion to integer by assignment is handled by special-case code.

In SIXBOL, many of the keywords are handled in special ways, so that it is difficult to implement them with a simple mechanism. Since the & operator definition may be changed, it is impossible to compile special code, and it would defeat the goals of the project if restrictions were placed on the context in which keywords could be used. Special code to check for keywords would be inefficient and fraught with implementation problems.
A general facility that can handle these problems already exists in SIXBOL -- the trapped variable. Every keyword can be represented by some process that will result in the setting or fetching of its current value, and that procedure can be specified by a trapped variable. Since a trapped variable has associated with it a specific location in memory, such operations as

`.TRIM`

and

`PAT $ &ANCHOR`

present no special problems.

As described in Chapter 3, a trapped variable has associated with it both an assignment procedure and an access procedure. For protected keywords, the assignment procedure is an error handler that issues an appropriate error message. Unprotected keywords may have any one of several procedures associated with the assignment trap, depending on the particular keyword. Keywords that have a simple integer representation call a procedure that converts the value to be assigned to an integer. Other keywords are implemented by planting traps in the SIXBOL system, so the assignment trap for these keywords sets and clears such traps, depending on the value assigned.

The access procedures for the various keywords are different, depending on the particular keyword that is being
implemented. The default procedure is used for keywords that have an integer representation such as &ANCHOR and &TRIM, while more sophisticated routines are required for keyword values that must be computed such as &STCOUNT and &STNO.

**Statement Monitoring Keywords**

Several keywords are implemented using special implementation techniques because it was felt that a simple integer representation in a memory cell would have an adverse affect on the overall running speed of the system. In particular, the protected keywords that are related to statement processing fall into this class. There are three keywords whose value must be available for every statement: &STNO, &LASTNO and &STCOUNT.

Rather than updating these keywords every time a statement is executed, the system maintains sufficient information to allow computation of these values should they ever be referenced. The value of &STNO, for example, is found by looking in the memory location just preceding the failure offset.

Computation of &LASTNO and &STCOUNT must be performed using a cell that contains information about the last nonsequential transfer of control. This cell, called LSTNOTR, is updated every time a goto is performed, and contains the address from where and to where the last goto
was made. Whenever LSTNOTR is changed, the value of STCOUNT is updated by adding the number of statements executed since the previous goto to it. Any time the current value of &STCOUNT is needed, the number of statements since the last goto is computed by taking the difference between the current statement number and the last transfer statement and adding that to the statement count sum saved by the system.

The value of &LASTNO is also found using LSTNOTR. The last transfer statement is compared with the current statement number, and if equal, it is known that the last statement number is the old statement number part of LSTNOTR. Otherwise, the last statement number can be computed by subtracting one from the current statement number.

**Trap Setting Keywords**

Several keywords change the way the system runs. Examples of these are &TRIM, &ANCHOR, and &INPUT. While the overhead involved to implement a commonly used feature such a &TRIM in a straightforward way seems reasonable, it is less clear that such overhead is tolerable for a feature that is seldom used. One approach is to eliminate the feature from the implementation, but this conflicts directly with the goals of the project. Another is to offer a compiler switch, but this is esthetically unappealing and burdens the programmer with unnecessary trivial decisions.

The approach taken in SIXBOL is to implement such features in a way that their existence does not affect
programs not using them. This is done by dynamically altering the system slightly when the feature is referenced, and is referred to as trap setting.

Traps always have the following form: At the point in the code where the trap is to be performed, a jump to the trap code is inserted. This code must execute the missing instruction word before it returns to the location following the trap. The trap is cleared by restoring the former code.

Each trap has associated with it a two-word trap block

```
JP   TRAP
<argument>
```

When the keyword value is made positive, the first entry in the block is copied into the appropriate location to set the trap. Whenever the keyword is set to zero, the second entry is used. It is not necessary to know what the current trap status is when this is done. For the keywords &TRACE and &FTRACE, the action occurs both when the system sets the value and when it is set from the source program.

**Procedures Associated with Each Keyword**

The procedures called for each keyword are listed below. Default access is to return the value in the location referenced by the left pointer of the trap, while default assignment is to convert to integer and save the value in that location.
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Assign</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;ABEND</td>
<td>default</td>
<td>default</td>
</tr>
<tr>
<td>&amp;ABORT</td>
<td>error</td>
<td>default</td>
</tr>
<tr>
<td>&amp;ALPHABET</td>
<td>error</td>
<td>default</td>
</tr>
<tr>
<td>&amp;ANCHOR</td>
<td>convert to integer, set appropriate trap</td>
<td>default</td>
</tr>
<tr>
<td>&amp;ARB</td>
<td>error</td>
<td>default</td>
</tr>
<tr>
<td>&amp;BAL</td>
<td>error</td>
<td>default</td>
</tr>
<tr>
<td>&amp;CODE</td>
<td>default</td>
<td>default</td>
</tr>
<tr>
<td>&amp;DUMP</td>
<td>default</td>
<td>default</td>
</tr>
<tr>
<td>&amp;ERRLIMIT</td>
<td>default</td>
<td>default</td>
</tr>
<tr>
<td>&amp;ERRTYPE</td>
<td>error</td>
<td>default</td>
</tr>
<tr>
<td>&amp;FAIL</td>
<td>error</td>
<td>default</td>
</tr>
<tr>
<td>&amp;FENCE</td>
<td>error</td>
<td>default</td>
</tr>
</tbody>
</table>
&FNLEVEL
assign:  error
access:  convert from CDC 6000 series internal form to integer

&FTRACE
assign:  set trap and convert to internal form
access:  convert from CDC 6000 series internal form to integer

&FULLSCAN
assign:  default
access:  default

&INPUT
assign:  convert to integer and set trap
access:  default

&LASTNO
assign:  error
access:  compute using LSTNOTR

&MAXLENGTH
assign:  convert to string length field
access:  convert to SIXBOL integer

&OUTPUT
assign:  convert to integer and set trap
access:  default

&REM
assign:  error
access:  default

&RTNTYPE
assign:  error
access:  This is left as an integer by the system, so a conversion from this index to string is performed:
0 -> "FRETURN"
1 -> "RETURN"
2 -> "NRETURN"

&STCOUNT
assign:  error
access:  computed using STCOUNT sum, current statement number and LSTNOTR.

&STFCOUNT
assign:  error
access:  convert to SIXBOL integer form
&STLIMIT  
assign: default  
access: default

&STNO
assign: error  
access: obtained by referencing B.FRT-1, then converting to SIXBOL integer

&SUCCEED
assign: error  
access: default

&TRACE
assign: convert to 6000 integer and set trap  
access: convert to SIXBOL integer form

&TRIM
assign: convert to integer and set trap  
access: default

Table Organization for Keywords
Since keywords are referenced infrequently, a storage scheme that uses little space and is simple to implement is desired. A simple structure for doing this is a pair list, accessed by a simple linear search. Each entry consists of an indirect string descriptor, followed by the keyword trapped variable.

To find a keyword, the name field of the natural variable is located and its address is set up as an indirect string. The linear search is then fairly fast because only a simple compare is required, since the indirect representation of a string is unique. Finding the name field of the natural variable is easy, since the argument to unary & must be a name. Testing for a title verifies that the name is a natural variable, since the title descriptor is always in
the same position relative to a name descriptor of a natural variable.
CHAPTER 11

CODING CONVENTIONS FOR SIXBOL

The Implementation Language

SIXBOL is coded in COMPASS, the assembly language for the CDC 6000 series. The macro facility is used frequently, and a style of programming is used that yields control structures similar to those found in BCPL (Richards 1969). The assembler is used both to generate object code and to perform consistency checks on the SIXBOL program. Heavy use is made of symbolic constants, particularly when referencing parameters relating to character sizes, since a design goal of the project is support for both six- and twelve-bit character sets.

Subroutine Structure

Three types of procedures are used in SIXBOL, each of which uses the same calling sequence. The procedure types are:

(1) SPROC -- directly callable from the source language

(2) RPROC -- not directly callable from the source program, but capable of calling other procedures

(3) PROC -- not capable of calling other procedures
PROCs are distinguished from RPROCs for efficiency. PROCs normally implement relatively simple operations and calls to them may occur in frequently executed sections of code. RPROCs save linkage information on the system stack and are therefore capable of being recursive.

All procedures may allocate temporary storage at runtime, subject to the restriction that the amount of such storage be known at assembly time. The procedure heading macros may be used to define symbols that refer to this dynamically allocated space, and consistency checks are made to aid in locating incorrect references to such symbols.

Procedure heading macros exist for each type of procedure. A call takes one of the forms

\[
\text{name \{
\begin{array}{l}
\text{SPROC} \\
\text{RPROC} \\
\text{PROC}
\end{array}
\} \text{temps,returns}
\]

Here "name" is the name of the procedure being defined, "temps" is the list of temporary locations needed, and "returns" is the number of nonstandard returns the procedure can make. Neither argument is required. If the "temps" argument is omitted, the procedure allocates no temporary space. If the "returns" argument is omitted only the standard return is permitted.

A temporary variable list is specified by enclosing the list in parentheses. Vectors may be specified by providing an address expression following an equal (\(=\)) sign in
the list. For example

\[ \text{ZPROC RPROC (X,Y,Z=A+B)} \]

specifies that three temporary symbols are to be defined. X and Y are each allocated one cell apiece, while Z is allocated the number of cells given by the quantity \((A+B)\). A and B must evaluate to a previously defined absolute assembler expression for the call to be legal.

References are made to such variables by basing the reference from the stack pointer. Allocation is performed by decrementing a counter. In the above example, X is referenced with an offset of \(-1\), Y is referenced with an offset of \(-2\) and Z would be referenced with an offset of \((-2-A-B)\). When coding such a reference, it it is written as \#T\#symbol. In COMPASS, the \#T\# is a micro reference that expands to a pre-defined value. For the assembly of SIXBOL, this value is \"B.STK+]\". The B.STK part of the expression bases the reference from the stack pointer, and the right bracket is a "corruption prefix" applied to the symbols given in the procedure defining macro call and to references to such symbols. This provides a check on references, since the assembler differentiates between the symbols \]X and X.

A temporary variable reference thus might take the form

\[ \text{SA1 } \#T\#X \]
**Call and Return Sequences**

Procedures are called by saving the return address in a B-register, B.LNK, and jumping to the entry point. Saving of the return address must be explicitly coded since no branch-and-save program counter instruction exists on the CDC 6000 series. Since more than one instruction is required, the CALL macro is provided. The form of this call is

```
CALL proc,returns
```

where "proc" is the name of the procedure being called and "returns" is a list of nonstandard return addresses to be used. The second argument may be omitted if the procedure has no nonstandard returns; otherwise, the correct number of returns must be provided. The latter restriction is enforced by using a corruption suffix on procedures that use nonstandard returns. This suffix is generated by counting the number of returns provided in the returns list.

For a simple call, with no nonstandard returns, the code generated is

```
SB.LNK RTNPT
JP =XPROC
RTNPT BSS 0
```

The procedure is entered using the following instructions
for an RPROC, the code is modified somewhat

Entry to a procedure that allocates temporary storage is similar, except that instead of incrementing the stack pointer by only 1, it is incremented by the number of temporary cells required. Therefore, no additional instructions are required to allocate and deallocate space.

Certain restrictions must be observed when writing procedures of this type. First, the stack pointer must be changed except when calling another procedure; otherwise temporary storage basing will be disrupted. Further, so that procedures calling a given procedure will not be disrupted, all procedures must restore the stack pointer to the value it had when the procedure was entered. In general, this means that procedures must be written to use a fixed amount of space that is declared at assembly time.
The second consideration that must be given to stack allocated storage is that the stack is one of the regions tended by the storage regeneration procedures. This means that all data stored in the stack area must conform to the standard descriptor layouts that these procedures can handle. Unused descriptors left on the stack may cause data blocks in the allocated data area to be kept around after they are no longer needed so this should also be considered before a procedures returns control to its caller. Stack storage is not systematically cleared before or after its use. These pointers do not present a catastrophic problem, however, as long as they originally conformed to SIXBOL data conventions. Until they are overwritten by subsequent procedure calls, they are tended by the storage routines in the normal manner and remain valid but unneeded descriptors. The only problem posed is they may prevent space occupied by unneeded blocks from being released for other purposes.

When calling a procedure that can give several returns, the optional returns are saved in a transfer vector following the call. Each entry in the vector consists of a branch to the address given in the call. The vector is stored in reverse order, so signaling the n-th return is done by transferring n locations before the address specified in the linkage register. This is best seen by an example. A call of
CALL IOCHK,(EOF,ERR)

would expand to

SB.LNK RTNPT
JP =XIOCHK#2
JP ERR
JP EOF
RTNPT BSS 0

This example shows that a corruption suffix of 2 was added to the procedure indicating that two nonstandard returns are expected. The first, which is designated as RETURN 1, is taken if an end-of-file condition is encountered, while the second, designated as RETURN 2, is taken if an error is found. The routine therefore has three possible return points. The standard return is taken by transferring to B.LNK, the first is taken by transferring to B.LNK-1 and the third is taken by transferring to B.LNK-2. Since the jump instruction that bases from a B-register allows for an immediate operand, this return structure fits well within the order code of the CDC 6000 series.

Logical Control

Included in the macro text for SIXBOL are several macros for implementing control structures. This allows coding without the use of the branch instruction and consequent inventing of symbol names in some instances.

An IF macro, spelled $IF to distinguish it from a COMPASS pseudo-op, is used to provide a control structure similar to the IF-THEN-ELSE found in many languages. The
The condition must be a test that can be performed in a single machine instruction — the code following the $IF is executed if the specified condition is met. The instructions under the $IF are terminated with a $ operation (a valid COMPASS macro call). If an else-clause is required, it can be specified by using the $ELSE macro. Code between the $IF macro and the $ELSE macro is executed if the specified condition is met, while code between the $ELSE and the terminating $ macro is executed if the condition is not met. The $IF, $ELSE and $ macros may be nested up to about 100 levels if desired.

An example of $IF is shown for a section of code testing whether a descriptor has a datatype of STRING. If the object is not a string, the routine STREP is called to get a string representation for it.

```
$IF  IR,X.RG1

CALL STREP
BX.RG1 X.RG7
$

String descriptors are out of range
Get string representation
Move result to X.RG1
End of conditional
```

The $IF macro generates a label identifier and a branch to it, leaving the identifier on a stack. When the $ macro is encountered, the most recent label is popped from the stack and it is defined at the current location. The $IF macro must reverse the condition on which the branch is made before generating the branch around the conditional —
this is done with a table of all conditional operations and their complements.

In addition, a form of iteration is provided with the $DO and $OD macros. The call to $DO allows a hardware testable condition to be specified which, if met, causes the code between it and a matching $OD call to be executed. $OD causes transfer to be made back to the preceding $DO. The code produced creates a do-while control structure that is relatively efficiently implemented. A test is made before the loop is ever entered to determine whether it should be executed the first time. The branch instruction at the bottom then makes the complementary test to determine whether execution should continue. This done in the same manner as the $IF instruction complement is performed.

Coding Source-Language Procedures

Source-language procedures are those that are directly called from the compiled object code. The structure of this code is quite rigid and a certain discipline must be followed when writing procedures of this type.

When a call is made from the object code, the last argument is passed in X.RG1, the next-to-last is passed in X.RG2 and the argument count is passed in B.CNT. The linkage register is set at statement initialization and must be incremented by the called procedure. Further, this register points to floating storage, so it must be saved on the stack with flags set to indicate this. Otherwise it could be not
be properly adjusted should a storage regeneration move the block whence the call was made.

Use of the SPROC procedure heading macro causes B.LNK to be saved with flags and be incremented by 2. It is the procedure's responsibility to test that the correct number of arguments were passed, and this is normally done with the ARG= macro. This macro generates code to call a subroutine that adjusts the argument count to be that specified if the number given in the actual call does not correspond to the number expected. The action of this subroutine is to add trailing nulls if the count is short or throw away arguments if the count is too large.

The next important action that a source-language procedure must perform is to test for value traps in any arguments that the value is used. These must be evaluated in the correct order, since traps frequently have side effects. Any procedures that assign values to source-language variables must also check for traps before the assignment is made. Both types of traps require special handling, since careless assignment to a trapped variable will overwrite the trap.

Procedures that create source-language strings must be careful to update the LASTSTR pointer, since this is used by concatenation to optimize the generation of strings built in stages. Failure to update this pointer properly might confuse the concatenation operation by making it appear as
if a certain string was at the edge of the string area when
in fact it was not. Procedures that create strings are
CONVERT, DATE, DUPL, REPLACE and TIME as well as the oper-
ations of concatenation and input.

Finally, all procedures must place the result they
return on the system stack. Procedures not returning a
useful value generate a null string to act as filler and
enable correct operation of the argument passing mechanism.

Register Usage in SIXBOL

Register allocation on the CDC 6000 series is
complicated by the fact that functionally there are eight
different types of registers. They are normally divided
into three groups; A registers, which are used for
referencing memory; B-registers, which are index registers;
and X-registers, which are 60-bit registers designed for
computation. Register allocation is further complicated by
the fact that there are two types of B-registers and 3 types
each of A- and X-registers. B-registers are used by a
variety of instructions for indexing, shifting and branch-
ing. They are all functionally similar except for B0,
which is always set to zero. A-registers are used for
specifying the addresses of memory references. When any of
A1, A2, A3, A4 or A5 are set to a value, a memory fetch
occurs, with the result being place in the corresponding
X-register. A6 and A7 are used in a converse fashion to
cause the contents of X6 and X7 to be stored in memory.
Register A0 is not related to X0, and can be used as an index register, albeit with a much smaller set of possible instructions than are available for a B-register. All of the X-registers are functionally similar, but their connection with the various types of A-registers restricts somewhat how they are used.

The wide variety of idiosyncratic properties makes the use of a multi-tier register allocation system such as that used in SITBOL (Gimpel, 1974) somewhat less effective on the CDC 6000 series. To a large extent, register selection is forced by the need to use certain registers with certain instructions. However, the relationship between the A- and X-registers makes possible an elegant way of transmitting both a location and a value in fast and uniform way.

To encourage the uniform use of registers throughout the system, each is given a symbolic name. References to the registers are then made symbolically, as is done in this paper. This makes possible the reallocation of registers to a limited extent, and perhaps reduces the number of keyboarding errors that are not flagged by the assembler. Registers names are given in Appendix A.
APPENDIX A

REGISTER USAGE IN SIXBOL
### A-Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0 A.SBP</td>
<td>Contains the stack base for the current source-program function level.</td>
</tr>
<tr>
<td>A1 A.RG1</td>
<td>First argument passage. Used in conjunction with X.RG1.</td>
</tr>
<tr>
<td>A2 A.RG2</td>
<td>Second argument passage. Used in conjunction with X.RG2.</td>
</tr>
<tr>
<td>A3 A.SCR</td>
<td>Used for general scratch when memory fetches are required.</td>
</tr>
<tr>
<td>A4 A.SCT</td>
<td>Similar to above, but considered of higher privilege.</td>
</tr>
<tr>
<td>A5 A.CUR</td>
<td>Cursor pointer. Used in pattern matching and compiling.</td>
</tr>
<tr>
<td>A6 A.STO</td>
<td>Used as a scratch register when memory stores are required.</td>
</tr>
<tr>
<td>A7 A.RES</td>
<td>Used to return an address if one is computed by a procedure.</td>
</tr>
</tbody>
</table>
### X-registers

<table>
<thead>
<tr>
<th>X0</th>
<th>X.MSK</th>
<th>Used for generating field isolation masks; also general scratch register.</th>
</tr>
</thead>
<tbody>
<tr>
<td>X1</td>
<td>X.RG1</td>
<td>First argument passage</td>
</tr>
<tr>
<td>X2</td>
<td>X.RG2</td>
<td>Second argument passage</td>
</tr>
<tr>
<td>X3</td>
<td>X.SCR</td>
<td>Used for general scratch</td>
</tr>
<tr>
<td>X4</td>
<td>X.SCT</td>
<td>Similar to X.SCR but considered of higher privilege.</td>
</tr>
<tr>
<td>X5</td>
<td>X.CUR</td>
<td>Used in pattern matching to carry the cursor, and in compiling to hold the input stream.</td>
</tr>
<tr>
<td>X6</td>
<td>X.STO</td>
<td>Used for scratch work, especially in conjunction with A.STO</td>
</tr>
<tr>
<td>X7</td>
<td>X.RES</td>
<td>Used to return values from procedures. It can also be used for scratch while in a procedure. Procedures that do not return results should normally avoid using it.</td>
</tr>
</tbody>
</table>
B-registers

B0  Constrained by hardware to contain zero.
B1  B.ONE  Constrained by convention to contain the value 1.
B2  B.BCT  A scratch register of fairly high privilege. It should not be used to hold floating addresses.
B3  B.SC2  A scratch register of medium privilege.
B4  B.SC1 or B.CNT A scratch register of low privilege.
B5  B.STK  Stack pointer. Always points to the next free word on the stack.
B6  B.LNK  Used for subroutine linkage.
B7  B.FRT  Used to carry source-program basing and failure offset address.
REFERENCES


SANTOS, PAUL JOSEPH JR. (1971) "FASBOL, A SNOBOL4 Compiler" Electronics Research Laboratory Memorandum, University of California, Berkeley, California, December.
