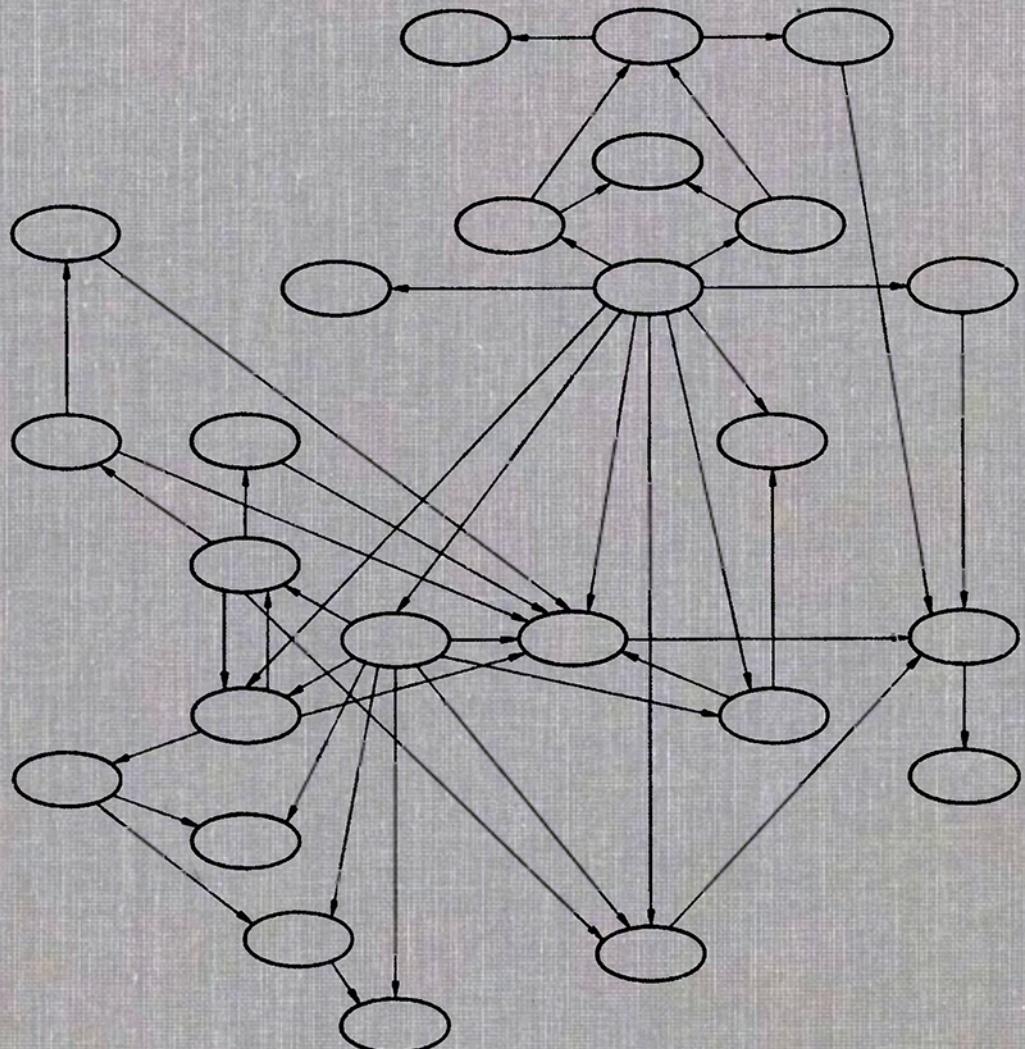


The Macro Implementation of **SNOBOL4**

Ralph E. Griswold



The Macro Implementation of **SNOBOL4**

*A Case Study of Machine-Independent
Software Development*

Ralph E. Griswold

The University of Arizona



W. H. FREEMAN AND COMPANY
San Francisco

Copyright © 1972 by W. H. Freeman and Company

No part of this book may be reproduced
by any mechanical, photographic, or electronic process,
or in the form of a phonographic recording,
nor may it be stored in a retrieval system, transmitted,
or otherwise copied for public or private use
without written permission of the publisher.

Printed in the United States of America

International Standard Book Number: 0-7167-0447-1

Library of Congress Catalog Card Number: 72-184097

Contents

preface ix

part I THE SNOBOL4 LANGUAGE 1

chapter 1 Background 3

- 1.1 The Original SNOBOL Language 3
- 1.2 SNOBOL3 5
- 1.3 The Impetus for SNOBOL4 6

chapter 2 Language Features 8

- 2.1 Basics 9
- 2.2 Built-In Operations 11
- 2.3 Function Definition 14
- 2.4 Data Types 17
- 2.5 Structured Aggregates of Variables 20
- 2.6 Keywords 22
- 2.7 Operations Relating to Variables 23
- 2.8 Patterns and Pattern Matching 25
- 2.9 Translation During Execution 36

- 2.10 Input and Output 37
- 2.11 Tracing 39
- 2.12 Errors and Error Handling 40
- 2.13 A Simple SNOBOL4 Program 41

chapter 3 Factors Affecting the Implementation 44

- 3.1 Language Features 44
- 3.2 Other Considerations 46

part II THE SNOBOL4 SYSTEM 49

chapter 4 Structure of the SNOBOL4 System 51

- 4.1 Approach to the Implementation 51
- 4.2 System Organization 52

chapter 5 Data Representation 56

- 5.1 Basic Data Units 56
- 5.2 Data Structures 59

chapter 6 Interpretation 64

- 6.1 Prefix Code 64
- 6.2 Procedure Linkage 66
- 6.3 The Basic Interpretive Process 68
- 6.4 Function Procedures 69
- 6.5 Control of Program Flow 75

chapter 7 Translation 79

- 7.1 Syntax 79
- 7.2 The Input Text Stream 84
- 7.3 The Translation Process 85

chapter 8 Implementation of Representative Features 96

- 8.1 Strings and Variables 96
- 8.2 Defined Functions 100
- 8.3 Defined Data Types 105
- 8.4 Arrays 110
- 8.5 Tables 112
- 8.6 Keywords 115
- 8.7 Patterns and Pattern Matching 116
- 8.8 Translation During Execution 131

8.9	Input and Output	133
8.10	Tracing	135
8.11	Errors and Error Handling	138
8.12	Other Operations	139

chapter 9 Storage Management 142

9.1	Allocation	142
9.2	Storage Regeneration	145

part III THE MACRO IMPLEMENTATION 157**chapter 10 The Macro Language 159**

10.1	The SNOBOL4 Implementation Language	160
10.2	A Description of SIL	160
10.3	SNOBOL4 Written in SIL	176

chapter 11 The Implementation of SIL on Two Machines 182

11.1	The IBM System/360 Implementation	182
11.2	The CDC 6000 Series Implementation	201
11.3	Comparison of the IBM and CDC Implementations	228

part IV RETROSPECT AND PROSPECT 231**chapter 12 History of the SNOBOL4 Project 233**

12.1	Evolution of the Language	233
12.2	Evolution of the Implementation	237
12.3	Evolution of SIL	239
12.4	Implementation of SNOBOL4 on Other Machines	240
12.5	Interaction of Language Design and Implementation	242

chapter 13 Evaluation of the Results 244

13.1	The User's Viewpoint	244
13.2	The Implementor's Viewpoint	248
13.3	The Author's Viewpoint	249
13.4	Areas for Improvement	250

chapter 14 Alternatives 252

14.1	A SNOBOL4 Compiler	252
14.2	SNOBOL4 Machines	255

appendices 259

- appendix A The Syntax of SNOBOL4 261
- appendix B Syntax Tables 265
- appendix C SIL Macros 273
- appendix D Research Problems 276
- appendix E Solutions to Selected Exercises 278

glossary: Implementation Terms and Symbols 291**references 298****index 301**

Preface

SNOBOL4 is a computer programming language which contains a number of features not commonly found in other programming languages. This book is a description of the implementation of SNOBOL4, including its history, internal structures and algorithms, and a critical evaluation. The name SNOBOL is derived from StriNg Oriented and symBOlic Language.

This book was undertaken in an attempt to help fill a gap in the literature on computer science. Books which describe actual implementations in depth are scarce, although the interest in this kind of material is extensive. One suspects that a contributing factor to this scarcity is the fact that most implementations have flaws that become painfully obvious when extensive documentation is provided. Such flaws are, however, almost inevitable at the present state of the art, and rarely is there time and continuity of effort available to correct them completely. SNOBOL4 is no exception. Yet such flaws provide some of the most valuable material for the student. This book is written in that spirit.

The author now knows another reason for the scarcity of implementation literature: It is very difficult to describe. A program of appreciable size is usually more complex than most people realize. In a developing technology, new terminology must be invented and explained. The mass of irrelevant detail must be stripped away to expose underlying structure and principle. This book took much

longer to write than originally expected because the organization, the approach, the material to be included, and the form of presentation all had to be revised many times.

The material should be useful to individuals with diverse interests. Implementors of SNOBOL4 surely will find interesting material. Students of software design and implementation may find the book helpful, both for the techniques used and as a case history. Language designers will find plentiful material about the effects of language features on implementation techniques. The zealous SNOBOL4 programmer may even find some of the information useful in understanding the language and in improving his programming techniques.

The author wishes to make clear what this book is *not*. It is not a program logic manual (PLM to IBMers) for SNOBOL4. It is not a complete description of the SNOBOL4 system or the macro language used to implement it. It is not an implementor's guide. It does not even include a complete listing of the SNOBOL4 source program. Any of the features listed above would have produced a work of unmanageable proportions. (The complete source listing of SNOBOL4 alone runs some 120 pages.) Instead, essential features and principles are discussed. Enough source material and macro-language information are included to give the reader a feeling for the nature of the system. The reader who wishes to explore the SNOBOL4 system in detail may refer to the working documentation of the SNOBOL4 project in references 1-7.

The terminology used in this book is somewhat different from that used in other descriptions of SNOBOL4. More standard and somewhat more formal language is used here, assuming that the reader has experience with other systems and will more readily understand the presentation if the terminology is familiar. An example is the use of "built-in" instead of "primitive," as used in reference 8.

To a small group of "insiders," a much more serious departure will be evident. The terminology used to describe the macro implementation is far different from that formerly used in developing and documenting the system. Inevitably, errors of notation and poor mnemonics arise in an evolving and complicated system. A constant change of programming notation is an overwhelming burden to implementors. As a result, many poorly chosen terms have become imbedded in the SNOBOL4 system. A new notation has been used in this book in an attempt to make the material easier to understand. The author extends his apologies to those individuals who are familiar with earlier notation and who may find the present material annoying and perplexing. Reference 6 explains the change in terminology and provides a cross reference for old and new terms. The material in this book corresponds to Version 3.7 of SNOBOL4.

The serious student may want to browse through this book before plunging in. A large amount of material is presented, which may be approached in various ways, depending on specific interest.

In Part I, the context for the implementation is presented. The first chapter provides historical background. A description of the language follows, discussing

essential aspects and features that are significant to the implementation. This material is put in perspective with a summary of the factors that most influenced the structure of the implementation.

Part II describes the implementation of the SNOBOL4 system from a machine-independent point of view. A description of the overall structure of the system is followed by a discussion of data objects and structures. The following chapters describe the interpretation process and the translator that converts the source-language program into a format that is suitable for interpretation. The implementation of several important language features is then described in detail. A chapter on storage management, including allocation and storage regeneration, completes Part II.

Part III describes the way in which the machine-independent implementation, described in Part II, is actually accomplished. The macro language used to implement SNOBOL4 is covered in some detail. A second chapter describes the implementation of the macro language on two machines, reducing earlier material to concrete examples.

The implementation of SNOBOL4 is considered in retrospect in Part IV. A history of the machine-independent implementation provides insight into the actual processes involved. An evaluation of the results follows. Finally, alternative approaches to the implementation of SNOBOL4 are considered.

Exercises are provided in appropriate sections of the book, mostly in Part II. Some exercises are relatively simple, designed to focus attention on important points in the text. Other exercises require working out algorithms in detail. Solutions to most of the exercises are given in Appendix E. Also included in the appendices are a list of research problems and various reference materials which supplement the text.

The implementation of SNOBOL4 is the product of the ideas and efforts of many individuals. My colleagues, James Poage and Ivan Polonsky, made major contributions to all parts of SNOBOL4, from language design through the final implementation. It is with the deepest appreciation that I acknowledge the importance of their work and my indebtedness to them.

Significant parts of the SNOBOL4 system were written by Al Breithaupt, Bernard Dickman, Paul Jensen, and Robert Yates. Contributions to the pilot implementation for the IBM 360 were made by Irving Benyacar, James Gimpel, Martha Ann Hawkins, Al Jones, J. C. Noll, Dorothy Teitelbaum, and Lee Varian.

Many individuals provided ideas and inspiration during the development of the machine-independent macro language. Particularly valuable contributions were made by David Farber, Stockton Gaines, James Gimpel, Kenneth Moody, Leon Osterweil, Michael Shapiro, Larry Wade, and William Waite.

The machine-independent macro language has now been implemented for a variety of computers. The list of individuals who participated in these implementations is too long to give here, and there would inevitably be unintended omissions. Their contribution in reducing an idea to practice is, nevertheless,

most valuable. My thanks go to them with the hope that the rewards of their labors compensate for the agonies they suffered.

The work described in this book was done, for the most part, at Bell Telephone Laboratories, Incorporated. The support of Bell Laboratories, including massive amounts of computer time, made SNOBOL4 possible.

Generous contributions to the preparation of this book have been made by several persons. The section on the implementation of pattern matching draws heavily on material developed by James Gimpel. The sections on the CDC 6000 series implementation were written by Michael Shapiro, who also provided the material on the design of his SNOBOL machine.

My special thanks go to Ben Wegbreit, who provided unusually perceptive and constructive criticisms of preliminary drafts of the manuscript. Critical readings by William Dietrich, James Gimpel, Drew Morris, and Michael Shapiro have helped in correcting many errors and improving the presentation. I, of course, assume responsibility for all remaining errors.

The most important acknowledgment is last. My deepest and most heartfelt thanks go to my wife, Madge. She read and corrected the manuscript through many iterations, giving generously of her time to serve as critic, copyeditor, proofreader, and keyboarder. Most of all, she provided constant encouragement and support during all the trials and frustrations of producing a book. Without her help, this book would never have been completed.

Ralph E. Griswold

May, 1972

The Macro Implementation of SNOBOL4

THE SNOBOL4 LANGUAGE

Background

1.1 THE ORIGINAL SNOBOL LANGUAGE

The idea for the first SNOBOL [9] language was born out of frustration. In 1962, the author and his colleagues were laboring to develop programs for manipulating symbolic formulas. (A particularly vivid recollection concerns the intricacies of factoring symbolic polynomials in many variables.) The languages then available were woefully inadequate for the problem. In desperation, the tasks at hand were set aside in order to build better tools.

The designers of SNOBOL had no idea where their initial tool-building efforts would lead. In retrospect, the tools have turned out to be more interesting than the uses that motivated their design. What started as a small effort to get programming leverage on a restricted set of problems turned into the development of a series of novel programming languages, eventually leading to SNOBOL4, which is now widely available in the computing community.

Details of the developments leading from SNOBOL to SNOBOL4 are not relevant here. A few remarks, however, will provide a perspective for the situation that existed at the beginning of the SNOBOL4 project and so strongly influenced its final outcome.

The original SNOBOL was designed to be an extremely simple, high-level language for manipulating strings of symbols. There was only one type of data in SNOBOL, the string. Even arithmetic was performed on strings of numerals. SNOBOL contained three simple types of statements: assignment, pattern matching, and replacement. An indirect referencing operator permitted assignment of value to any string, including strings created during execution. With pattern matching came the concept of success and failure of operations, and corresponding control of program flow based on sensing this success or failure. There was very little else to the language. There were no declarations, for example. SNOBOL could be learned quickly and was easy to use, except for the limitations imposed by its restricted set of operations.

Implementation was another matter. Strings were rampant, and any of them could be used as a variable. Pattern matching was relatively complicated compared with other facilities available at the time. The implementation was quite ad hoc. When the implementation of SNOBOL was initiated, the original motivation for a symbol manipulation tool for specific purposes still remained. Other thoughts had crept in, of course, but the project was still small and oriented toward use by a few individuals. The first system consisted of a translator, an interpreter, and an allocator. The translator converted the source-language program into an internal table of flags and pointers. The interpreter was driven from this table, executing the corresponding language operations. The allocator allocated space for strings as they were created and “garbage collected” when the need arose.

The coding of the SNOBOL system was done in assembly language (BEFAP) for the IBM 7090. The implementation of the system was facilitated by use of a package of string-manipulating macros [10]. All in all, the system was a hodge-podge of good and bad code written by several individuals (sometimes working independently) and was barely in shape for use by anyone but the implementors. The translator would accept any input whatsoever; every construction was valid *syntactically*, although very few constructions were valid *semantically*. The SNOBOL translator would happily accept a COMIT program, but the interpreter would have instant indigestion.

Despite the deficiencies of SNOBOL in language design and implementation, it attracted quite a bit of attention. The authors suddenly found their home-grown tool in public use, which sometimes proved as much of an embarrassment as a reward. The weaknesses of SNOBOL were apparent. The implementation, although moderately free of bugs, was not suitable for general use: In addition to the catholic appetite of the translator, diagnostic messages were often meaningless or, worse, “cute.”

1.2 SNOBOL3

The main language defect in SNOBOL was the lack of built-in functions. Determining the length of a string was grotesque, for example. A new implementation with the addition of built-in functions produced SNOBOL2, which existed as such for only a few weeks in the developmental stage. Programmer-defined recursive functions were added to make SNOBOL3 [11].

The implementation of SNOBOL3 was similar to that of SNOBOL in many respects. A significant attempt was made to improve the quality of the implementation, both internally and externally, as the system appeared to the user. The translator was more general and the tables generated for the interpreter were more uniform. The interpreter was simplified logically and extended to handle recursive operations. The allocator routines were generalized, and the use of storage was more sophisticated. SNOBOL3, like SNOBOL, was coded in macro-assembly language for the IBM 7090/94. More of the code was written in macro calls, with a number of macro operations defined specifically for the functions underlying the implementation.

SNOBOL3 was much more widely used than SNOBOL and was eventually distributed to some thirty or forty installations with 7090/94s. The interest in SNOBOL3 was sufficient to motivate programmers with other kinds of equipment to attempt their own implementations. SNOBOL3 for the IBM 7040/44 was easily achieved using the software support that extended the 7040 operation set to the 7090 operation set. Independent implementations were undertaken for other machines. Most notably successful were those for the Burroughs 5500, the CDC 3600, the RCA 601, the SDS 930, and the IBM 1620. These systems were developed largely from the SNOBOL3 language specifications, since the assembly-language source code for the IBM 7090/94 was not well documented and provided little help to implementors for other machines. The 1620 implementation was particularly interesting because of the small capacity of the 1620 compared with the 7090. The 1620 implementation was purely interpretive, and the source program was repeatedly analyzed during execution. The result was extremely slow execution; nevertheless, this system was useful to many programmers. As they put it, "it works," which is after all the most basically important matter.

One unpleasant result of the independently developed implementations of SNOBOL3 was the introduction of language dialects. The various implementors, either intentionally or by accident, introduced differences: additions, changes, and omissions. Consequently, the various versions were distressingly incompatible. Language incompatibility across machine boundaries is a familiar problem, and more will be said about this subject later. Another matter was the inevitability of errors in the different implementations. Here a word about language support is in order.

Unlike established languages such as FORTRAN, the SNOBOL languages were not supported by a computer manufacturer. SNOBOL was distributed by

the authors from Bell Laboratories and through SHARE, the IBM user group. The IBM 7090/94 version of SNOBOL3 was distributed and maintained solely by the authors from Bell Laboratories. Implementations for other machines were supported in various ways, and sometimes not at all. An implementor sometimes developed SNOBOL3 to a stage of completion and then moved on, leaving his system, or remnants of it, behind. The designers of the original Bell Laboratories SNOBOL3 system, therefore, were recipients of questions and complaints about “their language,” although they often had nothing to do with the implementation in question.

After the implementation of SNOBOL3, there was a period of reflection and gestation, which eventually led to SNOBOL4. While SNOBOL3 achieved much wider popularity and use than SNOBOL, several essential defects were still evident. One was the limited range of pattern-matching primitives: simple alternatives could not be incorporated in a pattern. Another deficiency was the lack of arithmetic operations on real numbers; while arithmetic is hardly the central issue in string manipulation problems, programmers often need to compute averages and such. A third deficiency was the paucity of data types. Arithmetic was still performed on strings of numerals. There were no arrays or other structures that are so useful in many programming problems.

1.3 THE IMPETUS FOR SNOBOL4

For a while, language development took the form of extensions to SNOBOL3. SNOBOL3 permitted the addition of separately assembled machine-language functions [12]. Using this facility, new data types were added for manipulating trees [13] and linked lists [14]. During this period, the authors were besieged with requests and suggestions for language extensions. While some of the suggestions were idiosyncratic and even ludicrous, many were clearly worthwhile. The majority of these suggestions fell into two categories. The first consisted of extensions to the pattern-matching facilities. The second consisted of suggestions for various new data types: linked lists, directed graphs, and so forth. Evidence accumulated for the desirability of a new language that would provide the additional facilities so much in demand. Incorporating all the suggestions or even a reasonable fraction of them as additions to SNOBOL3 would have produced a monstrosity. Consequently, ways were sought to provide general mechanisms, simpler in conception, that would permit the programmer to construct what he needed.

Crystallization of the SNOBOL4 project came, however, from another source: the advent of the third-generation machines. These machines held great promise: faster CPUs and larger memories. Virtual memories offered essentially unlimited space. The third-generation machines would, of course, displace the machines then running SNOBOL3. With the experience of developing and maintaining

SNOBOL and SNOBOL3 (with the problems of its independent, incompatible implementations), there was an evident need for extensions beyond SNOBOL3. In this situation, there seemed little reason to undertake, or encourage others to undertake, implementations of SNOBOL3 on the new machines. Thus, the SNOBOL4 project was born.

This discussion provides some historical background as it is relevant to the SNOBOL4 implementation project. The nature of the language itself strongly influenced the implementation. Chapter 2 discusses language features, with emphasis on facilities that most significantly affect the implementation.

Language Features

An understanding of the implementation of any programming language requires an understanding of the language itself. This is particularly true of SNOBOL4, which differs in so many respects from more familiar languages such as COBOL, FORTRAN, and PL/I. The understanding required to study the implementation is different, however, from the understanding required to use the language. Proficiency in the use of the language is not important, nor is a grasp of good coding techniques or typical “tricks.” On the other hand, some of the more esoteric and infrequently used facilities must be appreciated because they significantly affect the implementation. This chapter is an introduction to SNOBOL4 for the person interested in its implementation, but not necessarily in its use. See [8] for a complete description of the language. The reader who is already familiar with SNOBOL4 may skip this chapter.

At the end of each section there are notes about language features that are significant implementation considerations.

2.1 BASICS

A SNOBOL4 program consists of a sequence of statements which are executed in order unless control is transferred elsewhere under program control. A statement consists of three parts, separated by blanks:

label rule goto

The *label* identifies the statement so that control can be transferred to that statement if desired. The *rule* performs various program actions and is the part of the statement where computation takes place. The *goto* specifies transfer of control, conditionally or unconditionally. All parts of a statement are optional and may be omitted. The parts are separated by blanks. SNOBOL4 has no block structure in the static sense. A program is just a sequence of statements (terminated by an *end* statement). All labels are global and known throughout the program. There are no declarations of any kind.

There are three basic kinds of statements: (1) assignment, (2) pattern matching, and (3) replacement. The form of an assignment statement is

subject = object

in which the *object* is evaluated and made the value of the *subject*. The *object* is typically an expression, and may be quite complicated. The *subject* is a variable which is usually specified by an identifier, but may also be an expression (i.e., computed). Typical assignment statements are

```
SET    D    =    7
      C    =    2 * D * 3.14159      : (START)
```

in which C and D are identifiers, representing variables. SET is the label of the first statement. The second statement has a goto specifying transfer to a statement labeled START. A colon separates the goto from the rest of the statement. Labels in the goto are enclosed in parentheses.

Various types of data are represented syntactically in the source language in different ways. Integers and real numbers are represented in ways similar to other programming languages. Strings are enclosed in quotation marks (which serve as convenient delimiters, permitting the inclusion of characters that would otherwise have syntactic significance). For example,

```
CAPTION    =    'PER CAPITA INCOME'
```

assigns the string PER CAPITA INCOME to the variable CAPTION. Double quotation marks can also be used to delimit string literals. Thus,

```
CAPTION    =    "PER CAPITA INCOME"
```

is equivalent to the statement above. The two types of quotation marks permit

the inclusion of quotation marks within a string. Thus,

```
SQUOTE = ""
```

assigns a single quotation mark to the variable SQUOTE.

Pattern matching permits the examination of the structure of strings. The form of a pattern-matching statement is

```
subject pattern
```

in which the value of the subject is examined for the pattern. The value of the subject must be a string. A simple pattern-matching statement is

```
CAPTION 'INCOME'
```

which examines the value of CAPTION to see if it contains the string INCOME. The subject is usually a variable, as in the example above, but may be any string-valued expression.

The replacement statement is essentially a combination of pattern matching and assignment, in which the matched string is replaced, thus changing the value of the subject variable. The form of the replacement statement is

```
subject pattern = object
```

in which the pattern is matched against the value of the subject variable and the matched string is replaced by the object (if the match is successful). Continuing the example above, the statement

```
CAPTION 'INCOME' = 'GROSS'
```

changes the value of CAPTION to PER CAPITA GROSS.

If an object is omitted, the zero-length null string is assigned. For example, the statement

```
CAPTION 'INCOME' =
```

simple deletes the string INCOME if it occurs in CAPTION.

Pattern matching may succeed or fail. Several language operations possess this property, and success or failure during evaluation is an important part of the language. The most immediate application of success or failure is in controlling program flow through the use of conditional gotos. An S or F may be prefixed to indicate transfer conditional on success or failure, respectively. For example,

```
CAPTION 'INCOME' = 'GROSS' :S(YES)F(NO)
```

transfers to the statement labeled YES if CAPTION contains the string INCOME and transfers to the statement labeled NO otherwise.

Statements ordinarily terminate at the end of a line. However, several state-

ments may be placed on one line, separated by semicolons. An example is

```
M = 1; N = 0; P = M
```

A statement may be continued beyond the end of a line by placing a + at the beginning of the next line. An example is

```
OUTPUT = 'THE RESULT OF THE CALCULATION IS '
+      M * N
```

There is no limit to the number of continuation lines in a statement. A line beginning with a * is a comment. There are a few control statements, indicated by a - at the beginning of a line. An example is

```
-UNLIST
```

which suppresses listing of the source program when it is processed.

Notes: There are no limitations on the lengths of strings, variables, identifiers, or on the complexity of statements. The subject, pattern, object, and goto may be arbitrarily complicated and contain expressions of any kind. Computations may be performed in the goto, although failure of such a computation is an error, and the result of evaluating the goto must be a label.

2.2 BUILT-IN OPERATIONS

2.2.1 Functions

Many language operations are performed by built-in functions. For example,

```
SIZE(S)
```

calls the built-in function SIZE which computes the length of the string S and returns this length as value. Thus,

```
WIDTH = SIZE(CAPTION)
```

assigns the integer 16 to WIDTH if INCOME has been replaced by GROSS, as indicated above. Another built-in function is TRIM, which deletes trailing blanks from a string. Thus,

```
T = TRIM(S)
```

assigns to T the string obtained by deleting trailing blanks from S. DUPL is a function that duplicates a string a specified number of times. For example, the value of DUPL('=', 6) is =====.

2.2.2 Predicates

Predicates are built-in functions that test various conditions. If the condition tested for is satisfied, the predicate succeeds and returns a null (zero-length) string as value. If the condition is not satisfied, the predicate fails. Such failure is similar to failure in pattern matching and can be used to perform operations conditionally or control program flow. A typical predicate is LT(N,M), which fails if N is not less than M. Thus,

```
N = LT(N,10) N + 1 :F(OUT)
```

increments N, provided N is less than 10, and control passes to the next statement in line. Concatenation of the null string returned by LT does not affect the result. If N is not less than 10, LT fails, and execution of the statement stops. The assignment is not performed, and control is transferred to the statement labeled OUT.

Other typical predicates are IDENT, which succeeds only if its two arguments are identical, and LGT, which fails if its first argument is not lexically greater than its second.

2.2.3 Operators

Both unary and binary operators are used for commonly performed operations. Arithmetic operations are the most familiar, as illustrated in previous examples. The most common binary operation is the concatenation of strings. Concatenation is so common that no explicit operator is required: the two strings are written in succession, separated by blanks to avoid syntactic ambiguity. An example is

```
HEAD = CAPTION ' FOR NEW YORK'
```

which concatenates a string literal onto the value of CAPTION. As a result, the value of HEAD becomes PER CAPITA GROSS FOR NEW YORK. Since the blank is used as a binary operator, and is also used to separate the parts of statements, blanks have important syntactic significance in SNOBOL4.

An interesting unary operator is negation, indicated by \neg . This operator is a predicate that succeeds if its argument fails, and fails if its argument succeeds. A typical use of \neg is to reverse the meaning of another predicate. For example, \neg IDENT(X,Y) succeeds if X and Y are different.

Some operators perform different operations depending on the data types of their operands. The most familiar examples of such polymorphous operators are the arithmetic operators which perform integer arithmetic on integer operands and real arithmetic on real operands.

Operator precedence determines which operators bind most closely to their operands. Arithmetic operators are most familiar. The expression

$A + B * C$

is equivalent to

$A + (B * C)$

since multiplication has higher precedence than addition. Most operators associate to the left, but some associate to the right, depending on conventional mathematical usage. Thus,

$A - B - C$

$A ! B ! C$

are equivalent to

$(A - B) - C$

$A ! (B ! C)$

where $!$ is the symbol for exponentiation.

Parentheses can be used to override the usual precedence and associativity of binary operators and to delimit the operands of unary operators.

Section 7.3.3 contains a table of all binary operators. The set of unary operator symbols is the same as the set of binary operator symbols, except that the blank is not a unary operator. Binary operators must be surrounded by blanks. For example,

$A - B$

indicates subtraction of two numbers, but

$A -B$

is a concatenation in which the second operand is the negative of a number.

A number of operators have no predefined meaning. Section 2.3.3 describes how operators can be given new definitions.

Notes: There are many other built-in functions, predicates, and operators. The important ones are described in other sections. It should be noted, however, that the only difference between a function and an operator is syntax. The unary operator $-$, for example, is simply a convenient, abbreviated notation for a function of one argument. Thus, functions and operators are equivalent, so far as internal operation is concerned. In the analysis of the source language, the two are treated differently, of course. Any argument of any function or operator can be any expression, however complicated. If trailing arguments are omitted in the call of a built-in function, the omitted arguments are assumed to be null strings.

2.3 FUNCTION DEFINITION

2.3.1 Programmer-Defined Functions

SNOBOL4 provides facilities for the programmer to define his own functions and then to use them in a manner quite similar to built-in functions. Defined functions are specified during program execution by calling the built-in function **DEFINE**.

```
DEFINE(P,E)
```

defines a function described by the prototype P with entry point E. The prototype specifies the function name and its formal arguments (used to pass values when the function is called). The format of the prototype is illustrated by

```
DEFINE('F(X1,X2)', 'FSTART')
```

in which the prototype **F(X1,X2)** defines a function F with formal arguments X1 and X2. After executing this defining statement, the function is called like a built-in function. An example would be

```
F('A',5)
```

Corresponding to the prototype above, the string A would be assigned to X1, and the integer 5 would be assigned to X2.

The procedure to be executed when a defined function is called is written in SNOBOL4 and must be part of the program in which the function is called. By convention, when a defined function is called, control is transferred to the statement whose label is the second argument of **DEFINE** (which is **FSTART** in this case). If the second argument is omitted, the function name (F in the example above) is used instead. Execution of the procedure continues until transfer is made to a label reserved for returning from the function. For example, the label **RETURN** indicates ordinary return from the function. Upon transfer to **RETURN**, the current value of the function name (F in this case) is returned as value by convention. Transfer to **FRETURN** signals failure of the function call and produces the same effect as the failure of a built-in predicate.

The values of all variables specified in the prototype are saved when the function is called (F, X1, and X2 in the example) and restored on return. Defined functions may therefore be used recursively.

Local variables, whose values are protected during the function call, may be listed at the end of the prototype. Thus,

```
DEFINE('F(X1,X2)L1,L2,L3')
```

defines a function with three local variables, L1, L2, and L3, in this case.

Consider the following programmer-defined function that deletes all occurrences of a character C from a string S. The defining statement is

```
DEFINE('DELETE(S,C)')
```

Since the second argument is omitted, the entry point is DELETE. The procedure executed when DELETE is called is

```
DELETE S C = :S(DELETE)
DELETE = S : (RETURN)
```

The first statement looks for the character C in the string S. If C is found, it is deleted by replacing it with the null string, and the first statement is executed again. Eventually, the pattern match fails, in which case the value of S is assigned to the name of the function and returned as value.

Notes: There is no limit on the number of different functions that can be defined. A function may be redefined as often as desired. Any primitive function can be redefined. Any argument of a programmer-defined function can be any expression, however complicated. All arguments are passed by value. Programmer-defined functions usually return a value but may return a name by transferring to NRETURN.

2.3.2 External Functions

SNOBOL4 can add functions from external libraries. Such external functions, which are not part of the SNOBOL4 system, are typically written in assembly language. External functions can be used to add to the built-in operations available in SNOBOL4, to provide machine-dependent facilities, and to provide more efficient versions of programmer-defined functions.

An external function is added by executing the built-in function LOAD, which has a prototype describing the function and the name of a library where the function is located. For example,

```
LOAD( 'LOG(REAL,INTEGER)REAL' , 'SNOLIB' )
```

might be used to add an external function LOG from the library SNOLIB. The first argument of LOAD specifies in parentheses the data types of the arguments (a REAL value and an INTEGER base in this case). Following the parentheses, the data type of the value returned is given (REAL, in this case). After executing LOAD as described above, LOG may be used like a built-in function. For example,

```
X = LOG(46.76,10)
```

computes the logarithm of 46.76 to the base 10, assigning the value 1.66987 to X.

The function UNLOAD may be used to dispose of an external function when it is no longer needed. The use of UNLOAD is illustrated by

```
UNLOAD( 'LOG' )
```

UNLOAD frees the space used by the external function and undefines the function to prevent its future use.

Notes: External functions are machine-dependent in their implementation. FORTRAN and assembly language can be used for programming external functions on most systems. Support of other languages varies. The form of external function libraries is also machine-dependent. There is no inherent limit on the number of external functions that can be loaded or on the number or size of the libraries. UNLOAD may be used to undefine built-in and defined functions as well as external functions. The space occupied is only released in the case of external functions, however. For a complete description of external functions, see reference 15.

2.3.3 Synonyms

Synonyms for functions or operators can be created using the built-in function OPSYN. For example,

```
OPSYN( 'LESS' , 'LT' )
```

makes LESS a synonym for the built-in function LT. After executing this statement, LESS behaves just like LT. Thus,

$$N = \text{LESS}(N, 50) \quad N + 1$$

increments N if N is less than 50.

Synonyms can be extended to operators by adding a third argument, which may have the value 1 or 2 corresponding to unary or binary operators, respectively. For example,

```
OPSYN( '|' , '-' , 1 )
```

makes the unary | a synonym for the unary -. Subsequently |X behaves like -X. A function name can be made a synonym for an operator, and vice versa. Thus,

```
OPSYN( '|' , 'SIZE' , 1 )
```

makes the unary | a synonym for the built-in function SIZE. Similarly,

```
OPSYN( 'PLUS' , '+' , 2 )
```

makes PLUS a synonym for the binary operator +. Subsequently, PLUS(N, M) behaves like N + M.

Notes: Synonyms can be used for all types of functions and operators. There is no limit to the number of synonyms. Operators that have no built-in meaning can be used as synonyms for frequently used functions, thus extending the usefulness of the convenient operator notation.

2.4 DATA TYPES

2.4.1 Built-In Data Types

SNOBOL4 has many data types and facilities for defining others. The built-in data types are:

```
STRING
INTEGER
REAL
PATTERN
ARRAY
TABLE
NAME
EXPRESSION
CODE
```

The first three are illustrated in examples presented earlier. The remaining ones are described in following sections. Certain functions and operations create objects of these data types, and there are operations for performing appropriate manipulations on each type.

Automatic conversion of data types occurs in many contexts. In arithmetic operations, for example, strings of numerals are converted to numbers (integer or real) if possible. In mixed-mode expressions, integers are converted to reals. In most string contexts (such as concatenation and pattern matching), integers and reals are automatically converted to strings. Explicit conversion between objects of different types can be performed in many cases by using the built-in function CONVERT. CONVERT has two arguments: The first is the object to be converted, and the second is the name of the data type to which conversion is to be made. For example, the value of

```
CONVERT(2.3, 'INTEGER')
```

is the integer 2. Figure 2.4.1 illustrates the conversions supported by CONVERT.

Notes: CONVERT fails if a specified conversion is not defined or is not possible. Any type of data may be passed as an argument of a programmer-defined function. Data type checking is performed on the operands of built-in operations and functions. If an illegal data type is encountered, the erroneous condition is noted (see Section 2.12). Real numbers are represented as strings with decimal points (in F format) rather than with an exponent (E format). The data type name of any object can be obtained by using the built-in function DATATYPE. The value of DATATYPE(2.0 + 3) is, for example, REAL.

data type of argument	data type returned								
	S	I	R	P	A	T	N	E	C
STRING	X	X	X				X	X	
INTEGER	X	X	X						
REAL	X	X	X						
PATTERN	X			X					
ARRAY	X				X	X			
TABLE	X				X	X			
NAME	X						X		
EXPRESSION	X							X	
CODE	X								X

Figure 2.4.1
Conversions performed by CONVERT.

2.4.2 Defined Data Types

New data types can be defined using the built-in function DATA. The argument of DATA is a prototype specifying the new data type name and fields defined on this new data type. For example,

```
DATA( 'COMPLEX(R, I)' )
```

defines a data type COMPLEX with a field R (for *real part*) and a field I (for *imaginary part*). As a result of executing DATA as illustrated above, a new data type, COMPLEX, is created. In addition, three new functions are automatically defined: COMPLEX, R, and I. COMPLEX is an *object creation function*, and R and I are *field reference functions* defined on the data type COMPLEX. The complex number $2.0 + 3.5i$ is created and assigned to C by the statement

```
C = COMPLEX(2.0, 3.5)
```

The real part of C is referred to as R(C) and the imaginary part as I(C). Thus, the sum of the real and imaginary parts of C is obtained by executing

```
SUM = R(C) + I(C)
```

The value of SUM is 5.5. New values can be assigned to the fields in a similar manner. Thus, if the statement

```
I(C) = I(C) + 7.1
```

is executed, the value of C becomes $2.0 + 10.6i$.

The way in which fields are interpreted and used is left to the programmer. In the example above, the fields are treated as parts of an arithmetic entity. A different use is the building of structures. For example,

```
DATA( 'NODE(LAST, NEXT, VALUE)' )
```

might be used to define a data type NODE used to build linked lists. The LAST

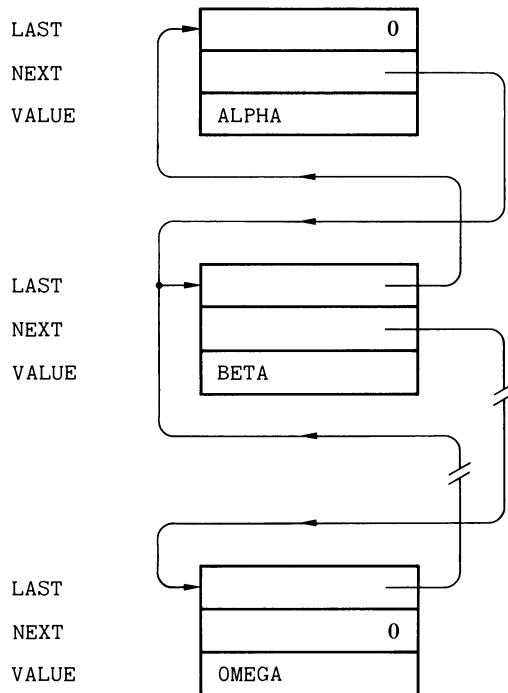


Figure 2.4.2
A linked list of NODES.

and NEXT fields are used for linking nodes together, while the VALUE field contains the value of the node. Figure 2.4.2 illustrates a structure that might be built in such a way.

Notes: There is a limit (899) on the number of different programmer-defined data types, but no inherent limit on the number of different objects of a given data type. The limit on the number of different programmer-defined data types is designed to avoid conflicts of internal representation with external data types (see the next section). There is no limit on the number of fields that can be defined. The same field name may be defined on several different data types, and in different positions if desired (VALUE is a typical example since it is a useful mnemonic). The same data-type name can be redefined as often as desired (within the limit of 899 executions of DATA mentioned above), and the built-in data types can be redefined. Such redefinition does not affect previously existing objects. The built-in function DATATYPE can be used for defined as well as built-in data types. No operations exist specifically for defined data types, but programmer-defined functions can be created for desired operations (for example, arithmetic on complex numbers).

2.4.3 External Data Types

External functions can construct objects whose data types are not known to the SNOBOL4 system. In the LOAD prototype previously described, any omitted or unknown data type name is assumed to be external. Thus, external functions can create and return objects of types unknown to the SNOBOL4 system. Boolean vectors are typical. Data types of values returned by external functions are not checked by the SNOBOL4 system, but operations that require specific data types perform the necessary data-type checking. External data types cannot be used as operands for built-in operators that require specific built-in data types. However, external functions can be added to operate on objects with external data types.

Notes: The mechanism for constructing external data types is necessarily machine-dependent, and care must be taken to avoid conflicts with types known to SNOBOL4. The external function mechanism for each machine specifies the necessary information. There is no way to communicate to the SNOBOL4 system the *name* of an external data type. All externally created data types are considered to belong to the class EXTERNAL.

2.5 STRUCTURED AGGREGATES OF VARIABLES

The simplest form of a variable is an identifier (i.e., a string), illustrated by SUM in the statement

$$\text{SUM} = \text{SIZE(T1)} + \text{SIZE(T2)}$$

Strings are called *natural variables* in SNOBOL4. There are also structures composed of variables. Variables in these structures can be assigned values just like identifiers. Arrays, tables, and programmer-defined aggregates are such structures.

2.5.1 Arrays

Arrays are created during program execution (not declared) by calling the built-in function ARRAY. Like DEFINE, ARRAY has a prototype argument that describes the array to be created. For example,

$$\text{X} = \text{ARRAY('10')}$$

creates an array of ten variables and assigns this array of variables to X. The effect is to create ten subscripted variables, which can be referred to as X(1), X(2), ..., X(10).

ARRAY has an optional second argument which is used for the initial value of the array variables. Thus,

```
Y = ARRAY('20',1)
```

creates an array of 20 variables, each of which have the initial value 1. If the second argument is omitted, the initial value is the null string. After creation of an array, values are assigned to the array variables in the same ways that they are assigned to other variables. For example,

```
X(3) = 'TOP'
```

assigns the string TOP to the third subscripted variable (also called the third element of the array X).

The value of X in this example is an array, and the subscripted variables are referenced by a special syntactic notation applied to X. If the statement

```
W = X
```

is executed, both W and X then have the *same* array as value. Thus, W(1) and X(1) are the *same* variable. Assigning a value to W(1) assigns that value to X(1) also.

Reference to a subscripted array variable fails if the index is out of range of the array. Thus, the statements

```
I = 1
LOOP X(I) = I * 3.14159 :F(OUT)
      I = I + 1           :(LOOP)
```

assign successive products of pi to the subscripted variables of the array X until the statement labeled LOOP fails when I reaches 11. Control is then transferred to OUT.

An array may have more than one dimension. This is indicated in the prototype by listing the sizes of the dimensions separated by commas. For example,

```
Z = ARRAY('3,4')
```

creates a two-dimensional array of 12 elements which can be referred to as Z(1,1), . . . , Z(3,4).

Notes: There is no limit on the number of dimensions that an array can have. There is no inherent limit on the size of an array. There are facilities for specifying lower bounds other than 1. There is no requirement that all the subscripted variables of an array have the same type of data as value. Thus, one subscripted variable may have an integer value, another a string, and so on. An array is a data object and can be the value of any variable. Thus, a subscripted variable can have an array as value.

2.5.2 Tables

Tables are superficially similar to arrays in that they are aggregates of structured variables, and the variables are referenced similarly. Tables can have only one dimension, however, and the subscripts need not be integers. A table is created by executing the built-in function TABLE. For example,

```
Q = TABLE( )
```

creates a table and assigns it to Q. A table has no fixed size, and any data object may be used as a “subscript.” For example,

```
Q('STATE') = 3
```

assigns the value 3 to the STATEth element of Q. The number of variables in a table grows as new subscripts are used. In a sense, tables may be thought of as associative arrays.

Notes: The “subscripts” and the values of the subscripted variables can be of any data type. There is no inherent limit on the size of a table; space is provided as needed. There are provisions for specifying the initial size of a table and the growth quantum to be used if more space is necessary.

2.5.3 Defined Data Types

A defined data object is an aggregate of variables, as mentioned earlier. The variables are referenced by the field functions. That is, a field function applied to a defined data object is a variable. Thus,

```
R(C) = 1.0
```

assigns the value 1.0 to the R field of C.

2.6 KEYWORDS

Certain strings called keywords provide means for communicating between the running SNOBOL4 program and the SNOBOL4 system. A keyword is identified by the prefixed unary operator &. Keywords fall into two categories, depending on whether or not their value can be changed by the program.

2.6.1 Protected Keywords

Protected keywords have values that cannot be changed by the program. Examples are &STNO and &STFCOUNT, which contain the number of the statement

currently being executed and the number of statements that have failed, respectively. Such keywords are useful for diagnostic purposes. For example, the statement

```
OUTPUT = 'FAILURE COUNT IS ' &STFCOUNT
```

would print out a message such as

```
FAILURE COUNT IS 27
```

&STNO and &STFCOUNT are changed automatically by the SNOBOL4 system as program execution proceeds. &ALPHABET, whose value is a string of all characters in collating sequence, is a constant keyword whose value never changes.

2.6.2 Unprotected Keywords

The values of unprotected keywords may be changed by the running program. Such keywords are usually used to set program modes or change the value of certain limits. &STLIMIT, whose value is the limit on the number of statement executions, is an example. The statement

```
&STLIMIT = 1000
```

sets a limit so that program execution stops when 1000 statements have been executed, which might be useful for debugging.

Notes: There are many keywords available. Several others are described in the following sections. A keyword reference results from applying the unary operator & to a string. The string need not be given explicitly but may be computed.

2.7 OPERATIONS RELATING TO VARIABLES

2.7.1 Indirect Referencing

Indirect referencing, represented by the unary operator \$, is an interesting and important unary operation. To understand indirect referencing, and especially its implications in the implementation, the concept of a variable in SNOBOL4 must be understood. In the examples above, the variables are represented by identifiers (C, D, CAPTION, and so forth). Identifiers are themselves strings. Conversely, in SNOBOL4 all strings are variables. Consider the statements

```
COLOR = 'BLUE'  
BLUE = 'WALL'
```

In these statements, there are three strings: COLOR, BLUE, and WALL. BLUE is

used both as a string value (the value of COLOR) and as a variable (that has the value WALL). Indirect referencing permits the use of a string value as a variable. \$COLOR is a level of indirectness on COLOR and is BLUE. The value of \$COLOR (i.e., the value of BLUE) is WALL.

Indirect referencing is a useful programming tool, permitting associative references and the formation of structured relationships among data. For example, if

```
BLUE    =    'WALL'
RED    =    'FLOOR'
WHITE   =    'CEILING'
```

then the value of \$COLOR is WALL, FLOOR, or CEILING, depending on whether the value of COLOR is BLUE, RED, or WHITE, respectively.

Identifiers which appear explicitly in the program cannot contain certain characters (such as colon) because of ambiguity with other constructions. However, indirect referencing permits any (nonnull) string to be used as a natural variable. Thus, in the statements

```
N = 1
M = 3
$(N :: M) = TEMP
```

the indirect reference uses the string 1:3 as a variable.

A string is the name of a variable, and variables with the same name are unique. This is illustrated by the statements

```
X = 'AB' 'C'
Y = 'A' 'BC'
```

As the result of executing these statements, X and Y have the *same* value, ABC, even though the values are constructed differently.

Notes: Indirect referencing can be applied to any type of variable, whether it is a natural variable, a member of a structured aggregate, or an unprotected keyword.

2.7.2 The Name Operator

The location or *name* of a variable is useful in several contexts. The unary . returns the name of a variable, which may be used to pass the variable to a function, for example. An illustration is

```
Z = .X(3)
```

which assigns to Z the location of the subscripted variable X(3). Assignment to X(3) subsequently may be made by using the indirect reference operator on Z.

For example,

```
$Z      =    17
```

assigns 17 to X(3). Thus \$ is the inverse of the . operator. It is also instructive to note that quotation marks perform the same function as the name operator, when applied to a string. Quotation marks permit the inclusion of characters that otherwise would have syntactic significance in the source program. The two statements

```
UNIT    =    'PAGE'  
UNIT    =    .PAGE
```

are equivalent. However, the two statements

```
UNIT    =    'PAGE HEADER'  
UNIT    =    .PAGE HEADER
```

are not equivalent since the blank in the second statement signifies concatenation (of the literal string PAGE with the value of the variable HEADER).

Notes: The names (locations) of all variables, natural, keyword, or structural, explicit or computed, are accessible to the program.

2.8 PATTERNS AND PATTERN MATCHING

Pattern matching is the most powerful and most complicated feature of SNOBOL4. The simplest pattern is a string, as illustrated in previous examples. Pattern structures, however, permit the description of much more complicated relationships among the parts of a string. There are built-in pattern structures and a variety of operators and built-in functions whose values are pattern structures.

2.8.1 Fundamentals of Pattern Matching

Alternation is a commonly used operation, creating a pattern structure that matches one or another alternative. The binary operator | is used to indicate alternatives. For example,

```
TYPE    =    'NET' | 'GROSS'
```

creates a pattern TYPE that matches either NET or GROSS. This pattern can be used in pattern matching in the same way a single string is used. Thus,

```
CAPTION  TYPE    =    'RESULTANT'
```

can be used to replace either NET or GROSS by RESULTANT (if either occurs in CAPTION).

Patterns can be concatenated like strings. The result is a pattern that matches the parts of the concatenation in order. Continuing the example above,

```
HEAD = TYPE '--REVENUE'
```

constructs a pattern that matches either NET--REVENUE or GROSS--REVENUE. Figure 2.8.1 illustrates the structure of such a pattern and indicates the order in which matching is attempted. NET and GROSS are *alternates*, as indicated by the dashed arrow. The string --REVENUE is a *subsequent* of the alternates.

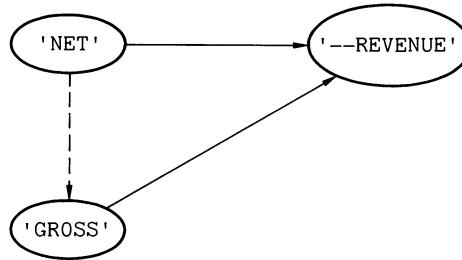


Figure 2.8.1
A simple pattern.

Concatenation has higher precedence than alternation so that the expression

$$A\ B \mid C\ D$$

is equivalent to

$$(A\ B) \mid (C\ D)$$

A more complicated example illustrates the way in which alternates and subsequents can be built into a pattern and the order of events in pattern matching. Consider the following statement.

$$P = ('B' \mid 'BR' \mid 'BRE') ('E' \mid 'A') 'D'$$

This pattern contains three subsequents. The first two contain alternates. Figure 2.8.2 illustrates the structure of P. This pattern will match any of the strings BED, BAD, BRED, BRAD, BREED, or BREAD. The order in which the matching is attempted is determined by the order of the alternates and subsequents in the pattern. In general, matching starts at the top left corner. When a component matches, the next subsequent to the right is attempted. If a match can be obtained for a series of components leading from the left through the right, the entire pattern match is successful. If a component does not match, the alternate below it is tried. If no alternate exists, the matching process backs up, rejecting the previously matched component and seeking an alternate for it. Thus, a match for the string BRAD corresponds to matching components 2, 5, and 6, while BREED corresponds to 3, 4, and 6. Matching can be thought of as an attempt to thread a

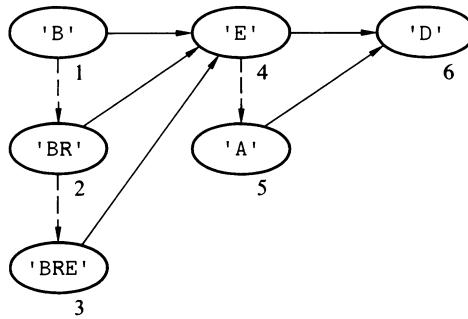


Figure 2.8.2
The structure of a pattern.

path through the pattern components from left to right, according to the alternate-subsequent structure, while moving from left to right in the subject string being matched. The current position in the pattern is identified by a *pattern pointer*, and the position in the string being matched is indicated by a *cursor*.

Figures 2.8.3 through 2.8.9 indicate the movement of the pattern pointer and the cursor as the pattern P is applied to the string BRAD. The subsequent and alternate pointers, which are implicit in the arrangement of the components, are omitted for clarity.

Ordinarily, if the first component of a pattern does not match at the beginning of the subject string, the cursor is advanced and a match for the first component is attempted at the second character of the subject string, and so on. Suppose the subject string is HELLO BRADLEY. The pattern above matches BRAD after the initial cursor position is advanced six characters. In this mode, pattern matching is *unanchored*. By assigning an integer value greater than zero to the keyword &ANCHOR, the initial position of the cursor is fixed at the beginning of the subject string, *anchoring* the match. In the anchored mode, the pattern can match only an initial substring of the subject string. The initial, default value of &ANCHOR is 0.

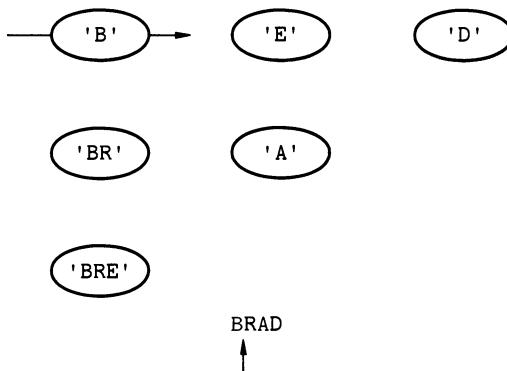


Figure 2.8.3
A match for B.

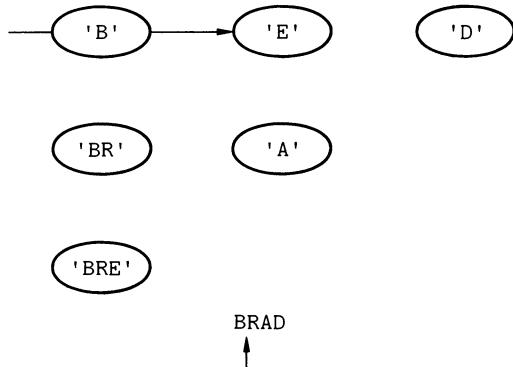


Figure 2.8.4
An attempt to match BE.

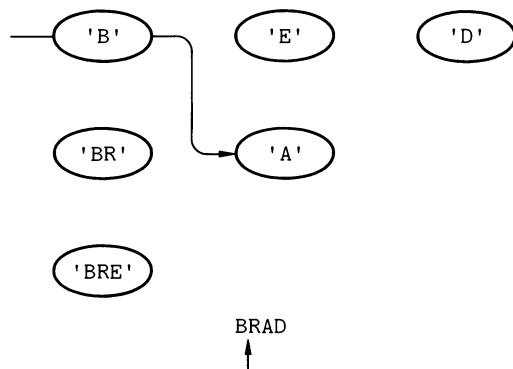


Figure 2.8.5
An attempt to match BA.

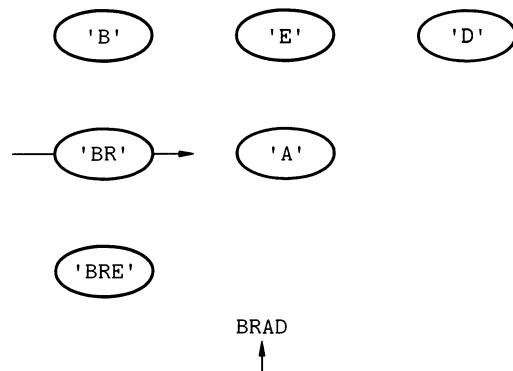


Figure 2.8.6
A match for BR.

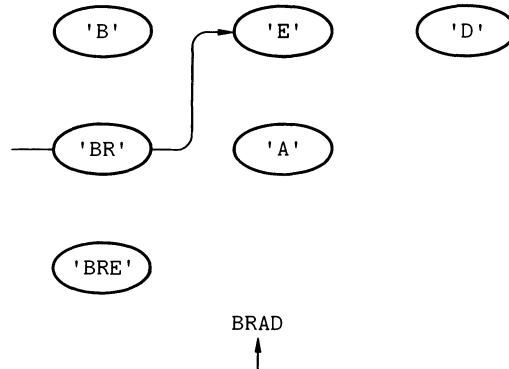


Figure 2.8.7
An attempt to match for BRE.

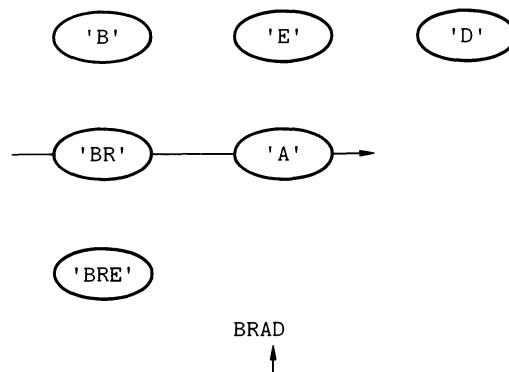


Figure 2.8.8
A match for BRA.

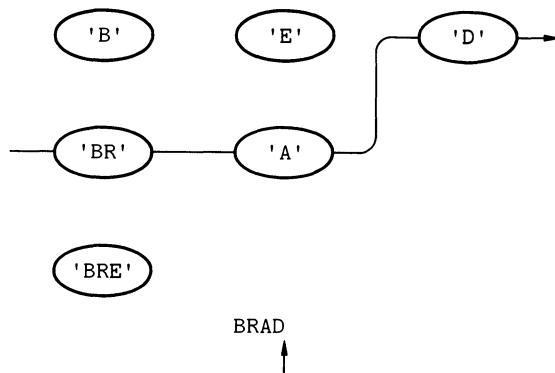


Figure 2.8.9
A complete match for BRAD.

Although this example contains only simple strings as pattern components, the general mechanism for pattern matching and the way the pattern pointer and cursor move are the same for all patterns. A description of more complicated pattern components follows.

2.8.2 Pattern-Matching Assignment

There is a facility for assigning substrings matched by components of a pattern to variables. This provides a way to determine what parts of the subject string are matched and in what way. It also provides a means of breaking a string into substrings.

Consider the pattern P used in the preceding section. This pattern matches any of the strings BED, BAD, BRED, BRAD, BREED, or BREAD. Suppose the statement

```
TEXT      P
```

succeeds. The particular string matched is still not known. If, for example, the value of TEXT is BREAD AND BUTTER, the string matched is BREAD, but if TEXT is BRADLEY LEAVES, the string matched is BRAD.

A variable can be associated with a pattern by use of the binary . operator. The first operand of this operator is the pattern, and the second is a variable to which the matched substring is assigned. For example, as a result of executing

```
TEXT      P . WORD
```

the substring matched by P is assigned to WORD. For the examples of TEXT mentioned above, the resulting value of WORD would be BREAD and BRAD, respectively.

The assignment operator has higher precedence than alternation or concatenation. Thus,

```
A B . N | C
```

is equivalent to

```
(A (B . N)) | C
```

As this example illustrates, variables can be associated with individual components of a pattern. For example,

```
W = 'D' ('A' | 'I' | 'U') . V 'D'
```

matches DAD, DID, or DUD. If it matches one of these strings, the vowel in the middle is assigned to V.

Assignments are made only to components that successfully match, and only if the entire pattern matches. Thus, the binary . operator is called the conditional assignment operator.

There is a similar binary operator, \$, for immediate assignment which assigns the substring matched by a component regardless of whether or not the entire pattern eventually matches. Thus,

```
W1 = 'D' ('A' | 'I' | 'U') $ V 'D'
```

assigns A to V if used to match DAR, even though W1 itself fails to match DAR.

The cursor position can be determined by another form of value assignment that occurs during pattern matching. The unary operator @ creates a pattern that matches the null string but assigns the current cursor position to its operand. For example,

```
TEXT P @L
```

assigns to L the position of the cursor after matching P. If the value of TEXT is BRADLEY LEAVES, the value of L is 4. A use of both kinds of value assignment is illustrated by the pattern

```
Q = P . WORD @L
```

which matches any string that P matches and assigns the matched string to WORD and its length to L.

Notes: There is no limit to the number of associations or cursor position operators that can be incorporated into a pattern. Such associations do not affect the matching done by other components of the pattern.

2.8.3 Built-In Patterns

Many structural properties of strings cannot be described readily by specific character strings. Built-in patterns are provided to extend the properties that can be described.

ARB is a pattern that matches any (“arbitrary”) string of characters. In pattern matching, ARB first matches the null string and then longer strings if necessary so that its subsequent can match. For example, in the statement

```
'TORTURE' 'T' ARB . MIDDLE 'E'
```

the value ORTUR is assigned to MIDDLE. ARB is generally useful in filling in between known components.

BAL is a pattern that matches any string of characters that is balanced with respect to parentheses. An example is the pattern

```
FUNCTION = ARB . NAME '(' BAL . ARGS ')'
```

which might be used to match function calls. If, for example,

```
EXP = 'SIN(2*(X+3))'
```

then as a result of executing

```
EXP      FUNCTION
```

the values of NAME and ARGS become SIN and $2 * (X + 3)$, respectively.

FENCE is a quite different kind of built-in pattern. FENCE matches the null string when encountered as a pattern component but causes the pattern match to fail if its subsequent fails and backs up. Consider the following statement which constructs a pattern to match the concatenation of two functions.

```
FC      =      FUNCTION FENCE FUNCTION
```

If the first occurrence of FUNCTION matches but the second does not, FENCE forces failure of the entire pattern match without attempting an alternate match for the first FUNCTION. FENCE is useful for short circuiting the pattern-matching algorithm to avoid useless and time-consuming attempts to find alternate matches.

In some cases, it may be desirable to force a pattern match to fail if a certain situation is encountered. This may be accomplished by using the built-in pattern ABORT, which causes the entire pattern match to fail if it is encountered. The built-in pattern FAIL fails to match but does not cause the entire pattern match to fail. FAIL is useful in forcing matching for alternates that otherwise would not be attempted.

Notes: There are other built-in patterns for various purposes. The built-in patterns are the initial values of the variables ARB, BAL, FENCE, and so forth. These names are not reserved and other values may be assigned to them. Each built-in pattern has a corresponding protected keyword (&ARB, &BAL, &FENCE, and so forth). Since the values of these keywords cannot be changed, the built-in patterns are always available.

2.8.4 Pattern-Valued Functions

There are a number of built-in functions that construct pattern structures. An example is LEN(N), which constructs a pattern that matches any string of N characters, regardless of what those characters are. LEN(N) simply moves the cursor right N characters. Thus,

```
TRIGRAM    =    LEN(3) . GRAM
```

constructs a pattern that matches three characters and assigns the result to GRAM.

Two functions construct pattern structures that move the cursor to specified positions. TAB(N) moves the cursor past the Nth character of the string. RTAB(N) moves the cursor up to the Nth character from the right end of the string. Thus,

```
PARTS    =    TAB(10) . HEAD RTAB(0) . TAIL
```

constructs a pattern that assigns the first ten characters of a string matched to HEAD and the remainder to TAIL.

Two functions construct patterns that test the cursor position. POS(N) succeeds and matches the null string if the cursor is positioned at the Nth character when POS(N) is encountered. Otherwise, POS(N) fails. This causes pattern matching to back up, seeking an alternate for the last component matched. RPOS(N) is similar, except that the position is measured from the right end of the string. The patterns constructed by POS(N) and RPOS(N) are like predicates, except that if the specified condition is not met, they simply fail to match, but do not necessarily cause the entire pattern match to fail.

Four functions, ANY, NOTANY, BREAK, and SPAN, construct pattern structures in which matching depends on the occurrence of particular characters.

ANY(S) matches any single character that appears in S. For example,

```
VOWEL = ANY('AEIOU')
```

might be used to match vowels. NOTANY(S) constructs the converse pattern.

```
NONVOWEL = NOTANY('AEIOU')
```

constructs a pattern that matches any single character that is not a vowel (such as a blank).

SPAN(S) constructs a pattern that matches any span of consecutive characters that appear in S. The pattern

```
WORD = SPAN('ABCDEFGHIJKLMNOPQRSTUVWXYZ')
```

might be used to match words.

BREAK(S) is the converse of SPAN(S). BREAK(S) matches any string up to, but not including, any character in S.

```
LOCATE = BREAK('ABCDEFGHIJKLMNOPQRSTUVWXYZ')
```

might be used to locate the position of the first letter in a word.

A more complicated pattern-valued function is ARBNO(P), which matches an arbitrary number of consecutive occurrences of anything that P matches. P may be any pattern. Thus,

```
MOD7 = ARBNO(LEN(7))
```

constructs a pattern that matches any string whose length is a multiple of seven (including the zero-length null string).

2.8.5 Unevaluated Expressions

Previous sections have described a number of operations that construct patterns. These include alternation, concatenation, value assignment, built-in patterns, and pattern-valued functions. These operations can be used in combination to construct extremely complicated patterns. Such patterns can be assigned to vari-

ables which are then used in pattern-matching statements or given explicitly in pattern-matching statements. For example,

```
TRIGRAM = LEN(3) . GRAM
```

```
.
```

```
.
```

```
.
```

```
LOOP TEXT TRIGRAM =
```

```
.
```

```
.
```

```
.
```

and

```
.
```

```
.
```

```
.
```

```
LOOP TEXT LEN(3) . GRAM =
```

```
.
```

```
.
```

```
.
```

are two ways of accomplishing the same thing. The second way requires constructing the pattern every time the statement labeled LOOP is executed. The first way is, therefore, preferable in most cases. On the other hand, the values of pattern components may vary during program execution. For example,

```
LOOP TEXT LEN(N) . GRAM =
```

might be used in a situation where N has various values at different times. Using the first method, for example,

```
NGRAM = LEN(N) . GRAM
```

the value of N at the time NGRAM is created is used in the construction of the pattern, and the pattern does not change even if N is subsequently changed.

This difficulty can be avoided by using *unevaluated expressions* obtained by using the unary operator *, which defers evaluation of its argument. An example is

```
NGRAM = LEN(*N) . GRAM
```

which creates a pattern in which N is left unevaluated. When NGRAM is used in pattern matching, the current value of N is used. Thus,

```
LOOP TEXT NGRAM =
```

matches strings of various lengths, depending on the value of N at the time.

Unevaluated expressions permit the imbedding of pattern components whose values are determined during pattern matching, not when the pattern is formed.

Thus, the pattern does not need to be reconstructed simply because of the change of a value. Any argument of a pattern-valued function can be an unevaluated expression. Consider the following statements:

```
LETTER = 'ABCDEFGHIJKLMNPQRSTUVWXYZ'
DUPL   = BREAK(LETTER) LEN(1) $ C BREAK(*C) . GAP
```

DUPL combines immediate value assignment with an unevaluated expression to assign a letter to C, which is subsequently used to search for a second occurrence of that letter. The string between the two occurrences is assigned to GAP.

Any pattern component can itself be an unevaluated expression. Thus,

```
ITH = *A(I)
```

matches the Ith element of A. I is evaluated at the time pattern matching occurs. Another use is illustrated by

```
LOCI = P . *A(I)
```

which matches P and assigns the result to the Ith element of A, where I is evaluated when LOCI is used in pattern matching.

A rather surprising by-product of unevaluated expressions is the ability to construct recursive patterns. Thus,

```
P = *P 'A' | 'B'
```

matches strings of the form

```
B
BA
BAA
BAAA
.
.
.
```

This facility is particularly useful in building patterns that match language constructions whose definitions are inherently recursive. The definition of a SNO-BOL4 expression is an example (see Appendix A).

Any expression can be the operand of the unevaluated expression operator. Thus, a programmer-defined function can be executed during pattern matching. Such a function may, of course, perform pattern matching itself.

2.8.6 Heuristics

One of the problems inherent in pattern matching is the large number of alternatives that may be tried in attempting to match a complicated pattern. The worst case occurs when the pattern eventually fails.

In an attempt to minimize unnecessary processing, certain heuristics are incorporated in the pattern-matching process. Briefly stated, these heuristics are:

- (1) The number of characters remaining in the subject string is continually compared with the minimum number of characters required to match the remainder of the pattern. Pattern matching is terminated in failure when it is recognized that too few characters remain.
- (2) The cursor position is advanced in the unanchored mode only if sufficient characters remain in the subject string to make a match possible.
- (3) The substring matched by ARB is not extended if too few characters remain to match the rest of the pattern.
- (4) The substring matched by ARBNO(P) is not extended if P last matched the null string.

These heuristics prevent attempts to match sections of a pattern if those sections cannot lead to success of the entire pattern match. This applies even if individual components in those sections might match.

To prevent looping of recursive patterns containing unevaluated expressions, an additional heuristic assumes that any unevaluated expression matches at least one character.

Ordinarily, the heuristics described above are used during pattern matching. In some cases, however, these heuristics may produce unexpected results or interfere with intended matches. The heuristics are by-passed if the keyword &FULLSCAN is greater than zero. Thus,

`&FULLSCAN = 1`

turns off the pattern-matching heuristics. The default setting of &FULLSCAN is 0.

2.9 TRANSLATION DURING EXECUTION

Two functions, EVAL and CODE, provide the unusual ability to translate strings into executable program statements during execution.

The argument of EVAL is a string representing a SNOBOL4 expression. EVAL translates that string, executes it, and returns the resulting value. For example,

`X = EVAL('I + SIZE(J)')`

is equivalent to

`X = I + SIZE(J)`

A typical use of EVAL is to evaluate SNOBOL4 expressions read in as data.

The argument of CODE is a string of SNOBOL4 statements separated by semicolons. CODE translates the statements, constructs the corresponding program,

and returns the result as value. This value has the data type CODE. For example,

```
C = CODE('LOOP OUTPUT = A(I) :F(OUT); I = I + 1 :(LOOP)')
```

translates two statements and assigns the resulting code to C. This new section of program can now be executed by transferring to LOOP. When the array reference fails, control is transferred to OUT, presumably somewhere else in the program.

An alternate method of executing the program created by CODE is the use of the *direct goto*, indicated by enclosing angular brackets rather than parentheses. A direct goto specifies a CODE-valued variable rather than a label. In the example above, a transfer to the newly created program could be accomplished by a goto of the form

```
:<C>
```

As in the previous example, transfer to OUT occurs when the statement labeled LOOP fails.

CODE provides a very general way to add to a program during execution. In theory, strings to be converted to CODE could be computed in such a way as to "grow" the program. In practice, CODE is usually used to execute SNOBOL4 statements that are included in data.

2.10 INPUT AND OUTPUT

2.10.1 I/O-Associated Variables

Input and output are accomplished by associating variables with files. The variable INPUT is associated with the standard input file. Whenever the value of INPUT is needed in a computation, one record is read from the input file. For example,

```
X = INPUT :F(EOF)
```

reads a record which automatically becomes the new value of INPUT and, consequently, is assigned to X also. When the input file is exhausted, an attempt to reference the value of INPUT fails, indicating the end-of-file condition. In the statement above, control is transferred to EOF.

OUTPUT is similarly associated with the standard print file. Whenever a value is assigned to OUTPUT, a copy of that value is printed. For example,

```
OUTPUT = INPUT :F(EOF)
```

reads a data card and prints it. OUTPUT retains its value and can be used like any other variable. Printed output is simply a by-product of assigning a value to the variable OUTPUT. Single-spaced carriage control is provided for printing.

PUNCH is associated with the standard punch file, but no carriage control is supplied for punched records.

2.10.2 Input and Output Associations

INPUT, OUTPUT, and PUNCH have built-in associations that connect them with the standard files. SNOBOL4 uses FORTRAN conventions, and the standard input, output, and punch files are designated by the numbers 5, 6, and 7, respectively. The programmer may specify associations for other variables and other files.

The built-in function INPUT is used for making input associations. The form of the function is INPUT(N,F,L), where N is the name of the variable to be associated with the file numbered F. L specifies the length of records to be read. An example of an input association is

```
INPUT('IN',10,72)
```

which associates the variable IN with file 10 and specifies a record length of 72. After executing this statement, references to IN cause 72-character records to be read from file 10. For reference, the built-in association for INPUT corresponds to

```
INPUT('INPUT',5,80)
```

Output associations are similar, except that the third argument is a FORTRAN format specification rather than a length. For example,

```
OUTPUT('OUT',20,'(100A1)')
```

associates OUT with file 20 and specifies 100-character record output with A-conversion. The built-in associations for OUTPUT and PUNCH correspond to

```
OUTPUT('OUTPUT',6,'(1X,132A1)')
OUTPUT('PUNCH',7,'(80A1)')
```

respectively.

Any type of variable can be associated. For example, if A is an array,

```
OUTPUT(.A<5>,6,'(1X,132A1)')
```

associates the fifth variable in the array A so that whenever a value is assigned to A<5>, that value is printed out also.

Data-type conversion takes place automatically on output. If the type of data is not a string or convertible to a string, its data-type name is output instead. For example,

```
OUTPUT      =      LEN(3)
```

prints

PATTERN

Notes: There is no limit to the number of associations or the number of variables that can be associated with one file. Any association, including

the built-in ones, can be changed. There is a facility for detaching associations. While a variable can be input-associated or output-associated with only one file at any given time, a variable may be both input- and output-associated at the same time. The results may be difficult to predict, however. Two keywords, &INPUT and &OUTPUT, can be used to turn all input and output off and on under program control. Other functions permit writing ends-of-file, rewinding files, and backspacing files. I/O is done by FORTRAN routines.

2.11 TRACING

2.11.1 Trace Associations

Tracing facilities are provided to aid program development and debugging. Trace association can be made to provide diagnostic printout for various situations. The built-in function TRACE is executed to form an association. In its simplest form, the call has the form TRACE(N, T), where N is the name of a variable to be traced and T is the type of tracing desired. An example is

```
TRACE( 'X' , 'VALUE' )
```

which establishes an association so that a message is printed whenever the value of X is changed. The message identifies the variable, its new value, and the location in the program where the change in value occurred. Other types of tracing are FUNCTION and KEYWORD. Function tracing prints a message when the named programmer-defined function is called, giving its arguments and the level of the call. On return from the function, another message is printed, giving the type of return and the value. Keyword tracing permits tracing of certain protected keywords whenever their value is changed by the SNOBOL4 system. Tracing &STN0, for example, provides a printed trace of each statement as it is executed.

Any type of variable can be traced. Since only natural variables have string names, a third argument can be provided to be used as an identifying tag when printing the trace message. For example,

```
TRACE( .A<3> , 'VALUE' , 'THIRD ELEMENT' )
```

associates the third element of A for value tracing. Printed trace messages include the tag THIRD ELEMENT.

2.11.2 Programmer-Defined Trace Procedures

The name of a programmer-defined function can be given as the fourth argument of the TRACE function. If this is done, the defined function is called when the traced condition arises instead of using the built-in procedure that ordinarily

prints trace messages. For example, if

```
TRACE( 'SUM' , 'VALUE' , , 'MYTRACE' )
```

is executed, an assignment of value to SUM causes the function MYTRACE to be called. The name of the variable being traced is passed to the function.

This facility for programmer-defined trace procedures not only permits the inclusion of functions to print more elaborate trace messages, but also permits execution of arbitrarily complex procedures when a traced condition occurs.

2.11.3 Trace Modes

Tracing is controlled by the keyword &TRACE. If &TRACE is zero, tracing is turned off and no trace association has any effect. If &TRACE is greater than zero, all trace associations that have been made are operational. Whenever a traced condition is encountered, the trace message is printed (or programmer-defined trace procedure called, if specified), and the value of &TRACE is decremented by one. For example,

```
&TRACE      =      100
```

turns on the tracing mode. Each trace decrements this value, and tracing automatically turns off after 100 traces. Of course, the value of &TRACE can be reset at any time.

Another keyword, &FTRACE, turns on tracing of *all* programmer-defined functions. Like &TRACE, &FTRACE is automatically decremented and turns off automatically.

Notes: There is a facility for removing individual trace associations in addition to turning off all tracing. There is no limit to the number of trace associations that can be made.

2.12 ERRORS AND ERROR HANDLING

Several types of errors may occur during the execution of a SNOBOL4 program.

Errors may be detected during the translation of the source program. Such errors are noted and the offending statements are flagged in the listing of the source program. Most such errors are syntactic, due to erroneous constructions (failure to surround binary operators by blanks is common for programmers new to SNOBOL4). The duplicate appearance of the same label is a semantic error detected during translation. If the number of errors detected during translation exceeds 50, translation ceases and execution is not attempted. Otherwise, execution is attempted regardless of errors detected during translation. If an erroneous statement is subsequently executed, execution terminates with an error message.

Various errors may occur during execution. Such errors range from an inappro-

priate data type in an operation to exceeding storage capacity. Execution errors normally result in termination of execution with an appropriate error message. Two levels of severity are recognized, however: nonsevere errors that are semantically erroneous but permit continued execution, and severe errors that prevent continued execution. If the keyword &ERRLIMIT is greater than zero, a nonsevere error is treated as a failure, causing the statement in which it occurs to fail instead of terminating execution. For each such error, &ERRLIMIT is decremented. If a nonsevere error occurs while &ERRLIMIT is zero, execution terminates with the appropriate error message. Thus,

```
&ERRLIMIT      =      10
```

permits ten nonsevere errors. The eleventh error causes termination of execution.

Each error has a number which is automatically assigned to the protected keyword &ERRTYPE when the error occurs. &ERRTYPE can be traced as a keyword. Therefore, it is possible to use a programmer-defined trace procedure to intercept nonsevere errors, taking desired actions for individual types of errors.

2.13 A SIMPLE SNOBOL4 PROGRAM

The following program, although using only a few of the features described in the preceding sections, illustrates a typical application and common programming techniques.

The problem is to analyze text, counting the number of occurrences of words of various lengths. The following program accomplishes this task.

```

*
*      INITIALIZATION
*
LETTERS      =      'ABCDEFGHIJKLMNPQRSTUVWXYZ'          1
WPAT        =      BREAK(LETTERS) SPAN(LETTERS) . WORD        2
DEFINE('WORD()')
LENGTH      =      ARRAY(10,0)                                3
*
*      PROCESS TEXT
*
RUN      UNIT      =      WORD()           :F(PRINT)          5
LENGTH(SIZE(UNIT)) = LENGTH(SIZE(UNIT)) + 1 : (RUN)       6
*
*      PRINT RESULTS
*
PRINT    I      =      1                                     7
OUTPUT    =
OUTPUT    =      'WORD-LENGTH COUNT:'                      8
OUTPUT    =                                     9
                                         10

```

PLOOP	OUTPUT	=	I	' :	LENGTH(I)	:F(END)	11
	I	=	I + 1	:	(PLOOP)		12
*							
*	WORD ISOLATION FUNCTION						
*							
WORD	LINE	WPAT	=	:S(RETURN)			13
	OUTPUT	=	INPUT	:	F(FRETURN)		14
	LINE	=	OUTPUT	:	(WORD)		15
*							
END							16

WPAT is a pattern used by the function WORD. In statement 13 WPAT locates a word and assigns it to the variable WORD. The word is deleted from LINE by replacing it with the null string. The function WORD returns a word each time it is called. If WPAT fails to match, a new line is read and printed in statement 15. WORD fails when the input is exhausted. LENGTH is an array that keeps count of the words of length 1 to 20. Note that the prototype is given as an integer, which is automatically converted to a string. Each word length is tallied in LENGTH in succession. When WORD fails, the results are printed by looping through the array LENGTH.

A sample output from the program follows.

```
'TWAS BRILLIG, AND THE SLITHY TOVES
DID GYRE AND GIMBLE IN THE WABE;
ALL MIMSY WERE THE BOROGOVES,
AND THE MOME RATHS OUTGRABE .
```

WORD-LENGTH COUNT:

```
1 : 0
2 : 1
3 : 9
4 : 5
5 : 3
6 : 2
7 : 1
8 : 1
9 : 1
10 : 0
```

Exercises

- 2.13.1** What happens if a word of more than ten characters is encountered?
- 2.13.2** What defects are there in the pattern WPAT?
- 2.13.3** Modify the program so that lengths with zero count are not printed.
- 2.13.4** How would the program be affected, if, in creating the array LENGTH, the second argument to ARRAY were omitted?
- 2.13.5** Add statements to the program to provide trace printout whenever a word of length four is encountered.

Factors Affecting the Implementation

3.1 LANGUAGE FEATURES

SNOBOL4 differs markedly from most other programming languages in its lack of declarations. Declarations are largely a concession to implementation and provide information to the compiler (translator) so that proper code can be generated and storage allocated in advance of execution. Programmers become used to such requirements and may even think of them as useful programming tools.

The freedom from declarations in SNOBOL4 is a result of the design goal that, where possible, concessions should not be made to the implementation, but rather that every effort should be made to make life easier for the SNOBOL4 programmer.

Note in particular that the following language features are supported without declarations: (1) recursive procedures, (2) arrays, (3) tables, and (4) programmer-defined data types.

Most languages fix the data type of each variable. Usually this is accomplished by a declaration, with defaults sometimes based on the nature of the identifier associated with the variable. SNOBOL4 not only has no type declarations, but also has no restrictions on the data type of the value of any variable. In particular, a variable may have different types of values at different times. This freedom from type restrictions is extended to structure variables. There is no requirement for type homogeneity that is so typical of other languages. Thus, one element of an array can be an integer, another element a string, and so on. Most striking is the programmer's ability to create variables during execution. Every nonnull string, however it is created, is a variable and can be assigned any value. Structured aggregates of variables are also created during execution. Thus, an executing program typically uses variables that do not appear in the source program.

The lack of declarations for variables has several important effects on the implementation. Since any variable may have any type of value, space for variable values must be adequate to accommodate any type of variable. Furthermore, functions must check the data types of their operands since there can be no guarantee from the form of the program that correct types will be supplied during execution. Closely related to data-type checking are the automatic data-type conversions provided in many instances. Polymorphous operators not only have to check, and possibly convert, data types, but also select appropriate subroutines according to data types supplied.

The ability to define functions during execution, redefine functions, and in general to make any function or operator a synonym for any other has several implications. The name of every built-in function must be available during execution in case that function is redefined. Furthermore, access to every type of function and operator procedure must be sufficiently uniform to permit synonyms.

The ability to associate I/O and trace properties with variables has significant implications. Most frequently, the values of &INPUT and &OUTPUT are greater than zero, permitting automatic input and output. In this case, every fetch of the value of a variable requires a check for an input association. Every assignment of value to a variable requires a check for an output association. Similar checks must be made for trace associations if the trace mode is in effect. Furthermore, such associations can be made for every type of variable, including array and table references.

The fact that some operators require their operand by name while others require it by value creates interesting problems and biases the implementation toward a prefix form of processing, where operators evaluate their operands, rather than the conventional postfix form where operands are evaluated before the operator is executed. The concept of failure during evaluation is important in this respect also, since failure prevents evaluation of subsequent expressions.

Pattern matching is a facility so complicated that it is like a language unto itself. The major problems are providing a unified context for pattern matching, handling the construction of patterns during execution, and implementing unevaluated expressions.

The features mentioned above are the result of the general philosophy that the programmer should have maximum flexibility during program execution: As little as possible should be fixed by the source program. In SNOBOL4 this takes the form of definition and redefinition of procedures during execution, of the creation of structures dynamically with sizes computable during execution, and so forth. The ultimate flexibility is the ability to create and execute program statements during execution.

Implicit in the descriptions of language features given in the preceding sections is the automatic management of storage. Since there are no declarations, there is little information in a program about the storage that will be required during execution. The creation of arrays and tables obviously demands storage, which can only be allocated when the creation occurs. Pattern building is another operation demanding dynamic allocation of storage. Strings also require allocation. All storage allocation is handled automatically. There are no specific allocate or free statements. In line with the philosophy of making things easier for the programmer, storage management is transparent to the executing program. This requires a technique for reclaiming space occupied by data objects that are no longer needed.

There are many situations in which a programmer may make errors. Design goals require the detection of such errors and the provision of meaningful diagnostic messages. In particular, the SNOBOL4 system must not malfunction because of a programmer error, however unlikely or obscure that error may be. This requirement is particularly stringent in a language in which the meanings of so many program operations are changeable during execution.

3.2 OTHER CONSIDERATIONS

The design of SNOBOL4 cannot be separated from the method by which it was implemented. The designers undertook the SNOBOL4 project with only the nucleus of a language design and with the opinion that the best final design could only come from experimentation and testing. A viable, if incomplete, implementation was essential to this approach. The designers intended to, and did, develop language features, test them in use, modify them, even discard some. SNOBOL4 was not designed by one committee to be implemented by another. Rather it evolved in a (more or less) controlled environment. Some features, such as the CODE data type and the negation operator, were not designed until long after the structure of the implementation had been fixed. Nevertheless, the anticipation that such features would be added significantly affected the approach to the implementation.

Experience with SNOBOL3 made it clear that the usefulness of SNOBOL4 would be greatly enhanced by the availability of compatible implementations on many different machines. To make implementations on different machines

a reality, ease of implementation is essential. Hence, portability in addition to machine independence was an important consideration.

Finally, the usefulness of SNOBOL3 as a research tool for exploring the language ideas that led to SNOBOL4 provided motivation for an implementation that would make SNOBOL4 a similarly valuable research tool for future language development.

The attributes expected of the implementation, briefly summarized, were generality and flexibility, with emphasis on ease of modification, and machine independence. Attributes which were assigned secondary significance were running speed and size.

THE SNOBOL4 SYSTEM

Structure of the SNOBOL4 System

4.1 APPROACH TO THE IMPLEMENTATION

The most basic decision made in approaching the implementation of SNOBOL4 was the choice of an interpretive system. An interpreter (as opposed to a compiler) has the advantage of being easy to modify and extend. More fundamentally, SNOBOL4 permits the meaning of many language operations to be deferred until execution. SNOBOL4 programs lack much of the information needed by conventional compilers in order to generate executable code. An interpreter easily permits decisions about meaning to be deferred until execution time. Furthermore, an interpreter is an easier form of implementation for a portable, machine-independent system since generation of machine-dependent code during execution is not required. The main disadvantage of an interpreter is its slow running speed.

Actually the SNOBOL4 system is a combination of a simple compiler (translator) and an interpreter. Figure 4.1.1 shows the essentials of the system structure in a schematic way. A translator processes the source-language program, analyzing it and generating corresponding prefix code. This prefix code is not the executable machine code that a compiler produces, but rather a canonical

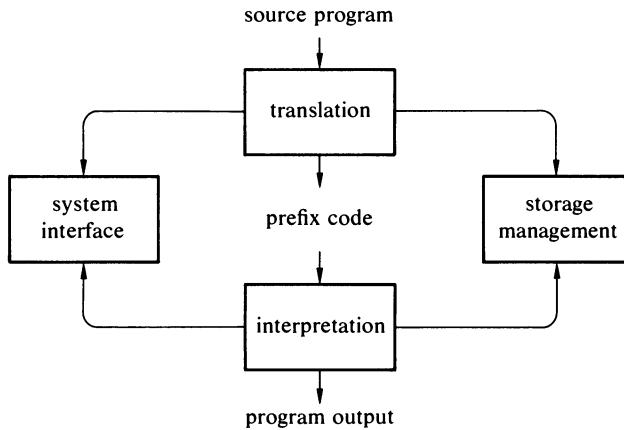


Figure 4.1.1
Organization of the SNOBOL4 system.

representation of the source-language program using pointers and flags. The prefix code is in a form that can be efficiently interpreted during execution. When translation of the source program is complete, the interpreter takes over and executes the prefix code. A storage manager is used both by the translator and interpreter. Allocation of space, storage of strings, and restructuring of storage are done by the storage manager as necessary. The system interface performs input and output (using FORTRAN routines), handles program interrupts, and so forth.

4.2 SYSTEM ORGANIZATION

4.2.1 Procedure Structure

The SNOBOL4 system consists of a collection of procedures that perform the various operations required to translate and execute SNOBOL4 programs. These procedures can be roughly divided into three groups as mentioned above: translation, interpretation, and storage management.

The translator contains a main procedure, ANALYZ, that analyzes the input stream according to the various statement formats. ANALYZ calls other procedures that translate expressions, elements, and so forth. There is a procedure, FORTXT, that converts a series of input lines (including continuations, comments, and control lines) into a text stream for the translator. Another procedure converts the trees generated during translation into prefix code, and so on.

The interpretive section consists of several groups of procedures. A control group performs the interpretation of code and directs program flow. One group

corresponds to the built-in functions and operations of the SNOBOL4 language (SIZE and @ are examples). Another group builds patterns, and still another performs pattern matching. Input, output, and tracing procedures are also part of the interpretive section.

The storage management section contains procedures to allocate strings and structures as required, to perform symbol table lookup, and to reorganize storage as necessary or when requested.

The actual organization is not really as clearly defined as Figure 4.1.1 indicates. There are a few procedures that fall into none of the categories and others that belong to more than one. More significant is the fact that translation can occur

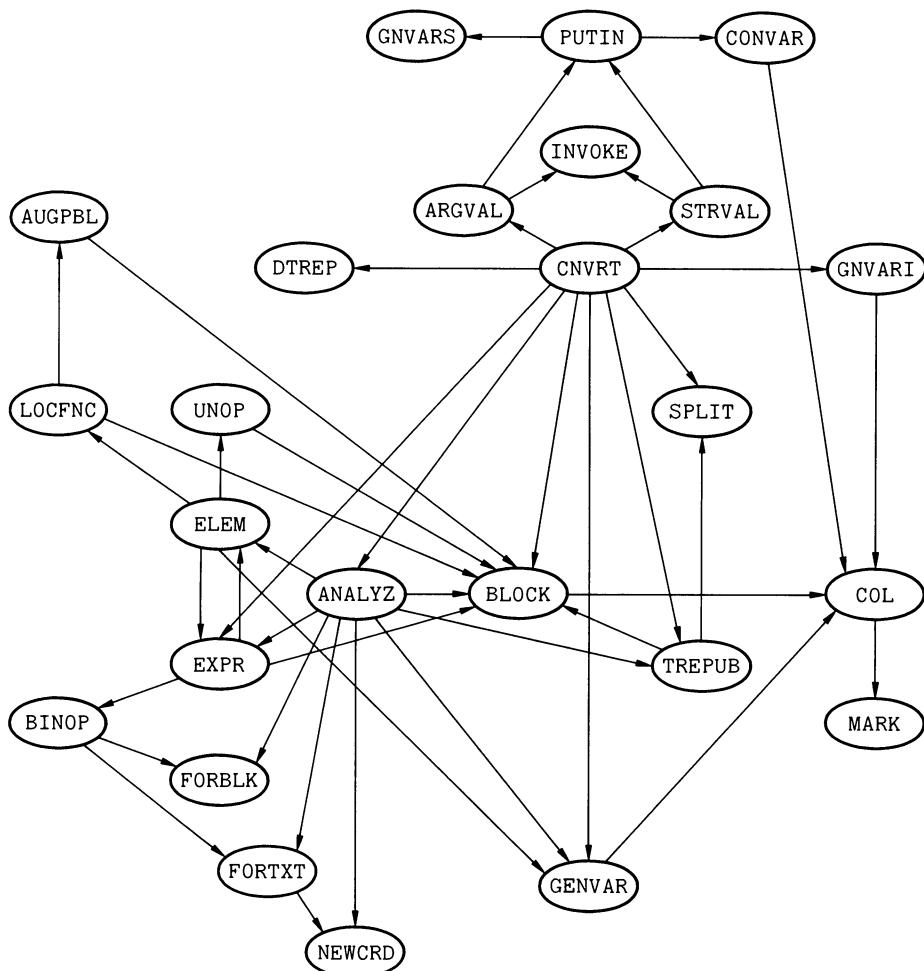


Figure 4.2.1
Typical procedure relationships.

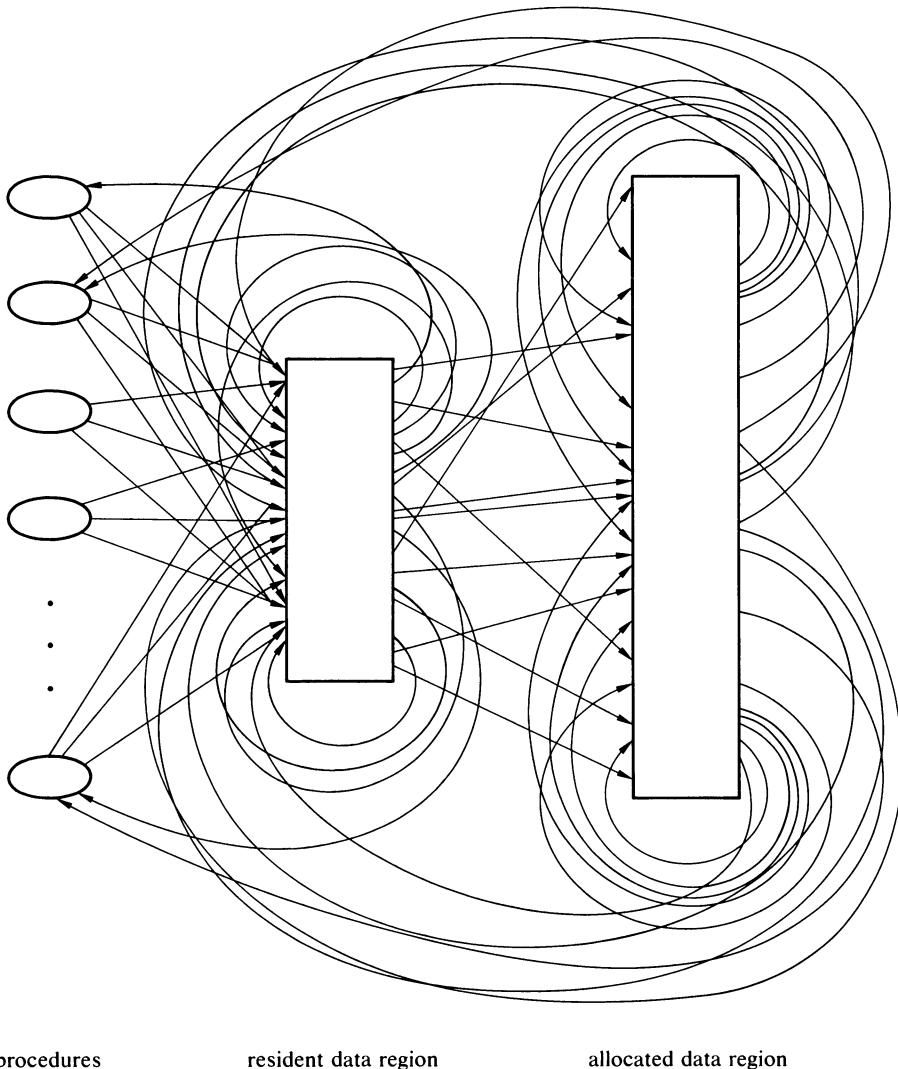


Figure 4.2.2
The relationships among data.

during interpretation (by conversion of strings to CODE, for example). The system is actually just a collection of procedures, not segregated in any significant way, and Figure 4.1.1 should not be taken too literally.

To illustrate the interrelation of procedures, consider what happens when the built-in function CONVERT is executed. The interpretation of CONVERT results in the invocation of a procedure for the built-in function, which, in turn, calls on the interpreter to evaluate its arguments. If conversion to CODE is specified, the translator is necessary. Translation calls on other procedures, including allo-

cation procedures to obtain space for the resulting code. Depending on the string being translated and the state of the SNOBOL4 system, as many as twenty-three different procedures may be called, some several times. This collection of procedures is illustrated in Figure 4.2.1.

There are procedures in Figure 4.2.1 that belong in each of the categories described above. ANALYZ, ELEM, EXPR, and so forth are translator procedures. CNVRT, ARGVAL, INVOKE, and so forth are interpreter procedures. BLOCK, COL, MARK, and so forth are storage management procedures. Note that an interpreter procedure, CNVRT, calls a translator procedure, ANALYZ.

All procedures have the same structure, and each is autonomous and self-contained. To accommodate the essentially recursive nature of the SNOBOL4 language, all calls are recursive. Procedures are written to save necessary values before calling other procedures. A stack, SYSSTK, provides a push-down list for recursion and temporary storage.

4.2.2 Data Organization

Data used by the SNOBOL4 system is divided into two general categories: *resident data* and *allocated data*.

The resident data region is part of the SNOBOL4 system proper and is accessed directly by name from the procedures. Resident data includes, among other things, SYSSTK, various constants, scratch data, error messages, and the like.

The allocated data region consists of a large block of storage handled by storage management procedures. All data created by the source-language program, including variables and structures, are contained in the allocated data region.

Figure 4.2.2 illustrates in a general way the relationship of the procedures, the resident data region, and the allocated data region. Arrows indicate pointers from one piece of data to another. Three facts deserve attention:

- (1) Both data regions contain pointers to themselves and to each other.
- (2) Procedures reference allocated data only indirectly through resident data.
- (3) Pointers to procedures may be contained in both data regions.

Data Representation

Data representation is an important part of most programs—particularly so in SNOBOL4. To a large extent, the choice of a data representation influences the structure of the system. In many cases, data contains in it the essence of algorithms, in the sense that once the format of the data is determined, the techniques for manipulating it are obvious.

5.1 BASIC DATA UNITS

In SNOBOL4, with its lack of declarations, freedom to assign any type of data to any variable, and so forth, the representation of source-language data is at the same time difficult and simple. There is a uniform representation for *all* source-language data objects, known as the *descriptor*.

5.1.1 Descriptors

The descriptor can be thought of as the basic “word” of the SNOBOL4 system. A descriptor has three fields: a V field, an F field, and a T field. Descriptors are illustrated diagrammatically in Figure 5.1.1. The V field represents the datum.



Figure 5.1.1
The layout of a descriptor.

This representation has two forms: the datum itself, or a pointer to a structure corresponding to the datum. Examples of the first case are integers and real numbers. Strings, arrays, patterns, and so forth are examples of the second.

The T field contains an integer that identifies the (source-language) data type. Each type has a unique integer code. For convenience, these codes are referred to by letters, using the initials of the corresponding data types (S for string, I for integer, and so on). Some integer codes are assigned to the built-in data types. Another group of integers is set aside for programmer-defined data types. A third group is reserved for external functions.

The F field contains bits—flags that identify certain properties of the descriptor. For source-language objects, the flag of interest is the *A* flag that identifies a pointer to the allocated data region. The data-type code in the T field essentially determines whether the V field is a pointer or not, but the *A* flag conveniently divides all descriptors into two classes. The representation of the integer 5 is shown in Figure 5.1.2. A pattern is represented by the descriptor shown in Figure 5.1.3. The arrow indicates a pointer to the structure for the pattern itself.

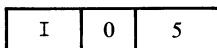


Figure 5.1.2
The integer 5.



Figure 5.1.3
A descriptor pointing to a pattern.

The use of the descriptor is not confined to the representation of source-language data objects but is used extensively throughout the SNOBOL4 system for representing internal data as well. For internal purposes, the fields of the descriptor have various uses different from those described above. The V field typically contains signed integers and addresses; the T field, unsigned integers (representing the sizes of objects, for example); and the F field, various bits that differentiate descriptors according to their status or use. The ways in which the fields are used are described in detail in the sections that follow. It is useful, however, to have a general idea of the nature and capacity of the fields of a descriptor.

The V field must be capable of containing any signed integer or real number that is representable in the source language. For example, the V field must be large enough to contain the maximum integer that can occur in a SNOBOL4 program. Furthermore, the V field must be capable of containing any pointer (generally any machine address). The F field contains several flags. Actually, six are used. The T field must be capable of holding unsigned integers as large as the

size of any structure that can be represented in the source language. For example, the size of the T field limits the largest array that can be created.

The way a descriptor is actually configured on any particular machine depends on satisfying the requirements listed above and on the operations that are available for accessing the fields. The configuration for two specific machines is described in Chapter 11.

5.1.2 Qualifiers

The descriptor is adequate for most of the needs of the SNOBOL4 system, but it is not used to describe character strings. For this purpose another data representation, the *qualifier*, is used.

A qualifier consists of two descriptors. One is laid out in the standard fashion described above, with a base pointer to the general location of the string. The second descriptor is laid out differently, containing an O field and an L field corresponding to an offset and a length, respectively.

Qualifiers have the format shown in Figure 5.1.4. Although the qualifier is shown as having five fields, it is actually composed of two descriptors. In the second descriptor, the F field is not used, and the V and T fields are available for the offset and length. The offset, when added to the base pointer, addresses the first character of the string. The length field specifies the number of characters in the string. Thus, substrings are easily specified by an appropriate offset and length within another string. For example, a qualifier for ALGORITHM might be represented as shown in Figure 5.1.5. The substring OR is shown in Figure 5.1.6.

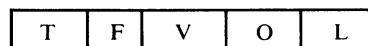


Figure 5.1.4
The layout of a qualifier.

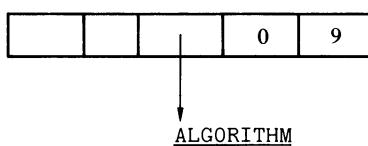


Figure 5.1.5
A qualifier for ALGORITHM.

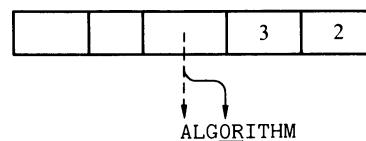


Figure 5.1.6
A qualifier for the substring OR.

5.2 DATA STRUCTURES

5.2.1 Natural Variables

Natural variables are structures containing strings. Natural variables are constructed by storage management procedures when strings are created and occur only in the allocated data region. Each natural variable consists of a title, three descriptors related to the string's status as a variable, and then the string of characters itself. The characters are compacted into descriptor-sized units, with unused space in the last descriptor if necessary. As a result, a natural variable occupies a number of descriptor positions, although the space occupied by the string itself is not laid out in fields. Figure 5.2.1 illustrates the natural variable corresponding to the string HYPNOTHERAPY.

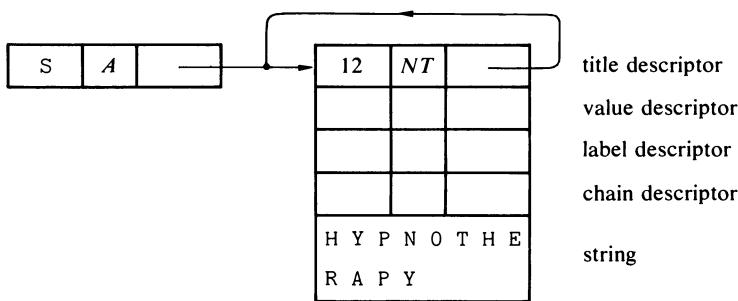


Figure 5.2.1
A natural variable.

The first descriptor in a natural variable is a title, identifying its top. The F field contains two flags. The *T* flag identifies the descriptor as a title descriptor and distinguishes it from all other types of descriptors. The *N* flag identifies the structure as a natural variable. The *T* field contains the number of characters in the string. The *V* field contains a self-pointer, i.e., the address of the descriptor in which it occurs. This seemingly curious pointer is used during storage regeneration (see Chapter 9). In subsequent figures, self-pointers are indicated by the symbol *. In Figure 5.2.1 it is assumed that eight characters fit into the space occupied by a descriptor (actually this number varies from machine to machine). Therefore, two descriptors are required to hold the string HYPNOTHERAPY, and four character positions are not used.

A string, as a source-language data object, is represented by a descriptor pointing to the corresponding natural variable, as illustrated in Figure 5.2.1. Pointers to natural variables are so frequent that a special notation is used, in which a natural variable is represented by the string enclosed in braces. Figure 5.2.2 illustrates this notation for the string HUNTER.

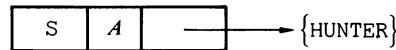


Figure 5.2.2
A pointer to the natural variable HUNTER.

The value descriptor in Figure 5.2.1 is the place where the value of the natural variable is stored. Since all source-language data objects are represented by descriptors, any type of value fits into this space. Figure 5.2.3 illustrates the situation if HUNTER has the value 5. Notice that while the pointer to the variable is a pointer to the title descriptor, the value of the variable is located at one descriptor beyond this location.

Thus, an assignment statement such as

HUNTER = 'WOLF'

simply overwrites the value descriptor of HUNTER with a pointer to WOLF, as illustrated in Figure 5.2.4. Note the use of the abbreviated notation for a pointer to a natural variable. The use of the label and chain descriptors is described in later sections.

To refer to the string in a natural variable, a qualifier is used. Figure 5.2.5 illustrates the relationship between a natural variable and a qualifier. Note that

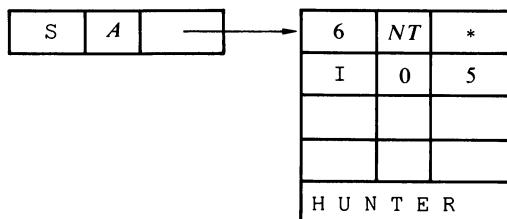


Figure 5.2.3
HUNTER with the value 5.

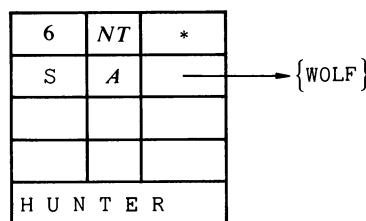


Figure 5.2.4
HUNTER = 'WOLF'.

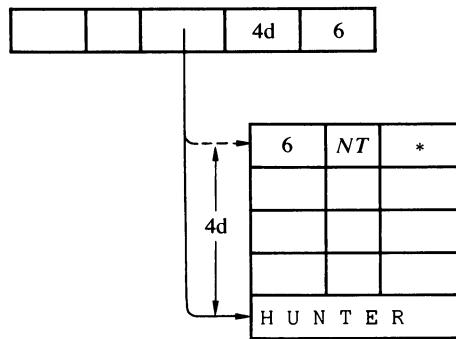


Figure 5.2.5
A qualifier and a natural variable.

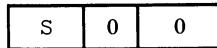


Figure 5.2.6
The null string.

the qualifier points to the title of the natural variable, and an offset of four descriptors indicates the beginning of the string.

The null string, which is not considered to be a natural variable, is represented by a descriptor with a data type S, but a zero V field. Figure 5.2.6 illustrates this canonical representation of the null string.

5.2.2 Blocks

Many source-language data structures consist of blocks of descriptors. Patterns, arrays, and tables are examples.

A block, like a natural variable, has a title descriptor, identifying its top. A block of descriptors is shown in Figure 5.2.7. The T field of the title contains the size of the block (five descriptors in this case, with the title not being included in the size). The F field contains a T flag but not an N flag.

Blocks are also used for internal data. The most frequent use is for *pair blocks*, which are blocks of descriptors logically arranged in pairs. The descriptor pairs are used to associate one piece of information with another. The members of the pairs are referred to as the A and B descriptors, respectively. Figure 5.2.8 illustrates the structure of a pair block containing n pairs. Nothing distinguishes a pair block from any other block except the way that it is used.

A typical application of a pair block is the data-type pair block, DTPBL, which associates the internal integer codes for data types with the data-type names.

5d	<i>T</i>	*	title descriptor

Figure 5.2.7
A block of descriptors.

	2nd	<i>T</i>	*	
A				pair 1
B				
A				pair 2
B				
.	•	•	•	pair n
.	•	•	•	
.	•	•	•	
A				
B				

Figure 5.2.8
A pair block.

Figure 5.2.9 illustrates the structure of DTPBL, assuming there are k pairs. To locate an item in a pair block, a linear search is performed, starting at the top of the block. By convention, a completely zero B descriptor identifies an unused A-B pair which may be used for adding a new entry. For example, the function DATA creates a new data type and adds a new entry to DTPBL. When there are no empty pairs in a pair block, a new, larger block is allocated and the old block copied into it.

Since data-type codes are integers, these integers might be used to index a table, a simpler and more efficient method than the use of a pair block as described above. However, the DATA function permits the creation of a large number of new codes, and external functions may use a wide variety of data-type

	2kd	T	*
A	.	.	.
B	I	0	0
	S	A	
	.	.	.
	.	.	.
	.	.	.
A	N	0	0
B	S	A	
	.	.	.
	.	.	.
	.	.	.
A	T	0	0
B	S	A	
	.	.	.
	.	.	.
	.	.	.

The diagram illustrates the structure of DTPBL (Data Type Pair Block List). It consists of a vertical stack of four pairs of tables, labeled A and B. Each pair of tables has three columns: 2kd, T, and *. The first pair (top) has three rows of dots. The second pair (labeled {INTEGER}) has three rows: I, 0, 0; S, A, and three rows of dots. The third pair (labeled {NAME}) has three rows: N, 0, 0; S, A, and three rows of dots. The fourth pair (labeled {TABLE}) has three rows: T, 0, 0; S, A, and three rows of dots.

Figure 5.2.9
The structure of DTPBL.

codes. Since DTPBL is accessed infrequently (only when the name associated with a data-type code is required), a pair block provides a simpler and more general way of making the desired associations.

Most pair blocks are contained in the resident data region. When it is necessary to create a new pair block, space is obtained in the allocated data region.

Interpretation

It may seem unusual to describe interpretation before translation since translation necessarily precedes interpretation in the processing of a SNOBOL4 program. On the other hand, the translator makes little sense until the interpreter and its data structures are understood.

6.1 PREFIX CODE

The translator converts the source-language program into a suitable internal format. In most programming languages, the internal format is machine code, which is subsequently executed. Because of the nature of the SNOBOL4 language and the goals of the SNOBOL4 project, machine code is not generated by the translator. Rather, an intermediate form is used, part way between the source-language text stream and actual machine code. This intermediate form

is interpreted rather than being directly executed. Binary operators are in prefix, rather than infix, position. The typical infix notation

$$X = Y * -\text{SIZE}(Z)$$

has an equivalent prefix form

$$=_2 X *_2 Y -_1 \text{SIZE}_1 Z$$

where the subscripts indicate the number of operands ($_1$ and $_2$ are different operations, for example). In SNOBOL4, prefix code is arranged as a series of code descriptors, one for each item. There are two kinds of code descriptors: function and operand. Function descriptors have an *F* flag to distinguish them from operand descriptors. The *V* field of a function descriptor points (indirectly) to a procedure for the appropriate function. The *T* field indicates the number of operands that follow. Operand descriptors are in the form used for source-language data, as described in the previous sections. The expression above, in internal prefix form, is illustrated in Figure 6.1.1. To avoid confusion between the notation for built-in functions and the corresponding internal procedures, internal procedures are given in lower case. Thus, the internal procedure for SIZE is size₁, and so forth.

Although functions and operators differ syntactically, they are the same semantically and have the same type of representation in prefix code. Every source-language construction has a prefix representation. There are, in addition, certain

.	.	.
.	.	.
.	.	.
2	<i>F</i>	
S	<i>A</i>	
2	<i>F</i>	
S	<i>A</i>	
1	<i>F</i>	
1	<i>F</i>	
S	<i>A</i>	
.	.	.
.	.	.
.	.	.

The diagram illustrates the mapping of code descriptors to their meanings. Arrows point from specific entries in the table to symbols or identifiers:

- From the entry "2 F" to the symbol $=_2$.
- From the entry "S A" to the set $\{x\}$.
- From the entry "2 F" to the symbol $*_2$.
- From the entry "S A" to the set $\{y\}$.
- From the entry "1 F" to the symbol $-_1$.
- From the entry "1 F" to the identifier size₁.
- From the entry "S A" to the set $\{z\}$.

Figure 6.1.1
An example of prefix code.

other functions inserted by the translator that do not correspond directly to anything in the source-language program. For example, every statement begins with an initialization procedure that performs minor bookkeeping chores. A corresponding function descriptor is inserted by the translator at the beginning of each statement.

6.2 PROCEDURE LINKAGE

Function descriptors in prefix code are used by the interpreter to access procedures needed to execute the source-language program. These procedures, corresponding directly to source-language operations, are called *function procedures*. The procedures for $=_2$, $*_{2,-1}$, and SIZE_1 in the previous section are examples.

A level of indirectness is included in the linkage between the function descriptor and its associated function procedure. Consider the statement

$$X = \text{POS}(N) \mid \text{POS}(M) \mid \text{POS}(P)$$

which has the prefix form

$$=2X \mid_2 \text{POS}_1N \mid_2 \text{POS}_1M \text{ POS}_1P$$

Figure 6.2.1 illustrates the procedure linkage from the prefix code to the function procedures. The first descriptor in the link descriptor pair points to the function procedure and is referred to simply as the link descriptor. The second descriptor of the pair is used only for special types of functions, such as programmer-defined functions, and is called the definition descriptor. The T field of the link descriptor contains the number of arguments expected by the function procedure. This is usually the same as the number of arguments which occur in the prefix code. Note that there is only one link descriptor for each function, although several function descriptors may point to the same link descriptor. For simplicity, link descriptors are omitted from most diagrams. The notation used in Figure 6.1.1 is typical. For example, size_1 is really a notation for the link descriptor to the function procedure for SIZE_1 . For convenience, the function procedure itself is usually referred to as size_1 .

The level of indirectness provided by the link descriptor permits convenient and uniform definition and redefinition of functions. Definition and redefinition (which amount to the same thing) occur as a result of executing DATA, DEFINE, LOAD, OPSYN, and UNLOAD. A link descriptor is the unique connection between the function descriptor and the function procedure actually executed. Changing a link descriptor changes the procedure executed for all references to that link. Suppose, for example, the following statement is executed.

$$\text{OPSYN}('POS', 'TAB')$$

The execution of OPSYN changes the link descriptor pointing to POS to point to the procedure for TAB. As a result, all function descriptors that formerly linked

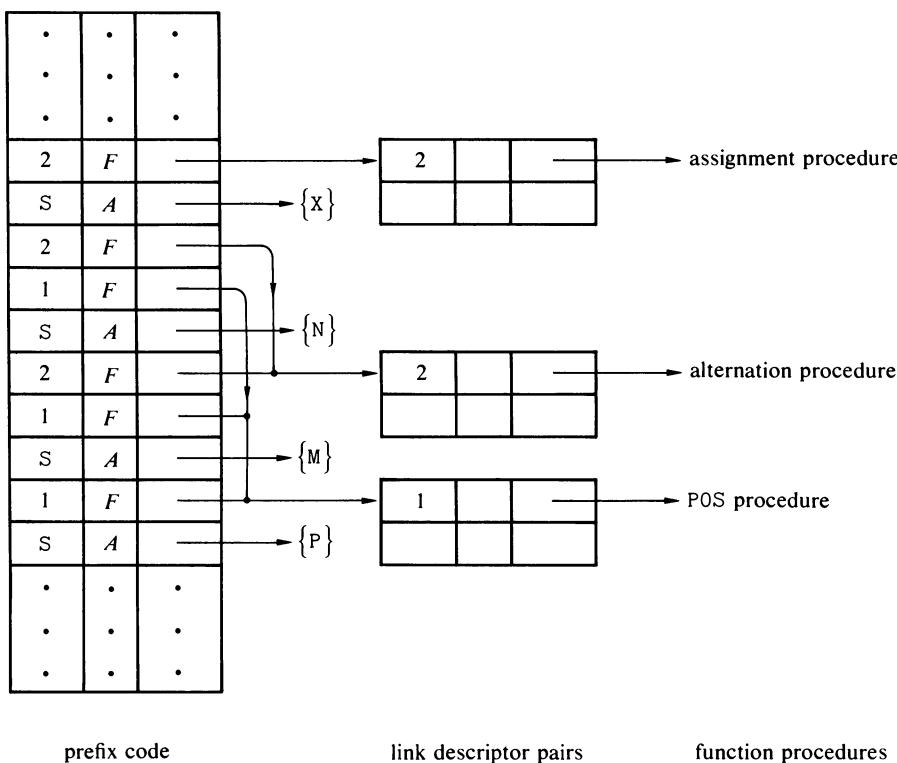


Figure 6.2.1
An example of procedure linkage.

to the POS procedure now link to the TAB procedure, and POS is redefined throughout the program. Figure 6.2.2 illustrates the change effected for the linkage illustrated in Figure 6.2.1.

In order to change a link descriptor as illustrated in the example above, the location of the link descriptor must be known. There is a function pair block, FNCPBL, that contains pointers to all function link descriptors. For each function there is an A-B pair. The A descriptor points to the link descriptor, and the B descriptor points to the name of the function. Figure 6.2.3 illustrates the structure of FNCPBL.

FNCPBL is in the resident data region and contains pointers to all link descriptors for built-in functions. Empty pairs are included for the definition of new functions. If these pairs are used up, a larger block is allocated in the allocated data region and the old block copied into the new block.

In order for OPSYN to redefine POS, the B descriptors in FNCPBL are searched, locating the pairs for POS and TAB. The descriptor pointed to by the A descriptor for TAB is merely copied into the descriptor pointed to by the A descriptor for POS.

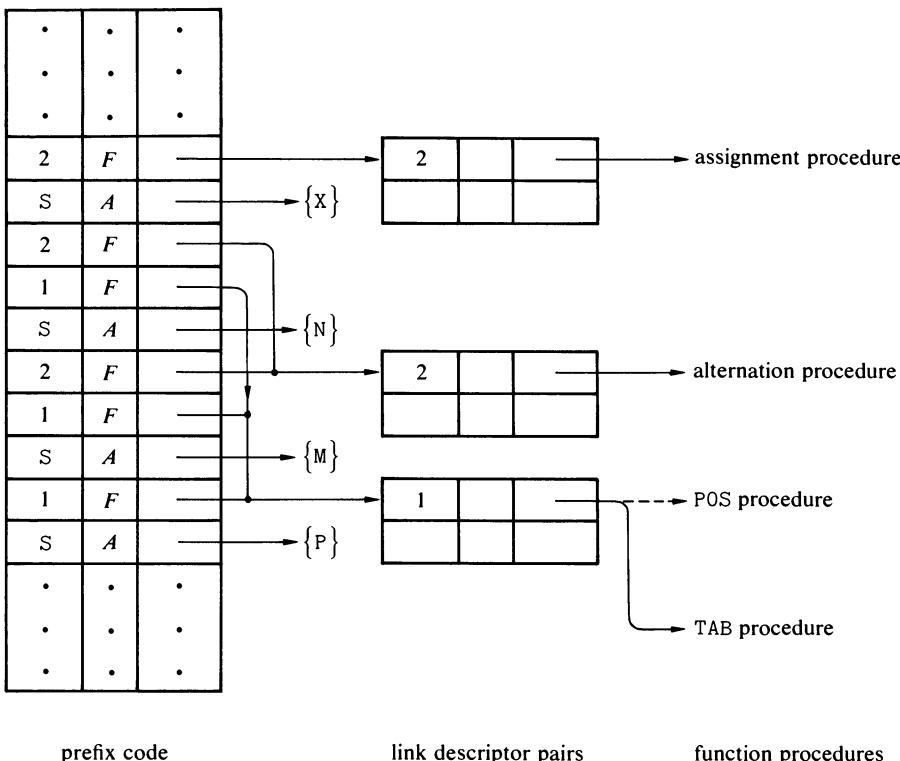


Figure 6.2.2
The result of OPSYNing POS to TAB.

The mechanism described above is used for all function definitions. UNLOAD, for example, changes a link descriptor to point to an error procedure. If an unloaded function is executed, this procedure indicates the execution of an “undefined” function. Changes made by other function-defining procedures are described in the sections that follow.

There are similar link descriptors for operators (such as $|_2$). Link descriptors for operators are located in a different fashion (see Chapter 7).

6.3 THE BASIC INTERPRETIVE PROCESS

Given the format of the prefix code and the procedure linkage described in the preceding sections, interpretation is a straightforward process.

There is a basic interpretive procedure, INTERP, that processes the prefix code. The sum of a code base pointer and a code offset indicates the current position in the prefix code. Descriptors of prefix code are processed in succession, updating the code offset. If a descriptor is a function descriptor, control is passed

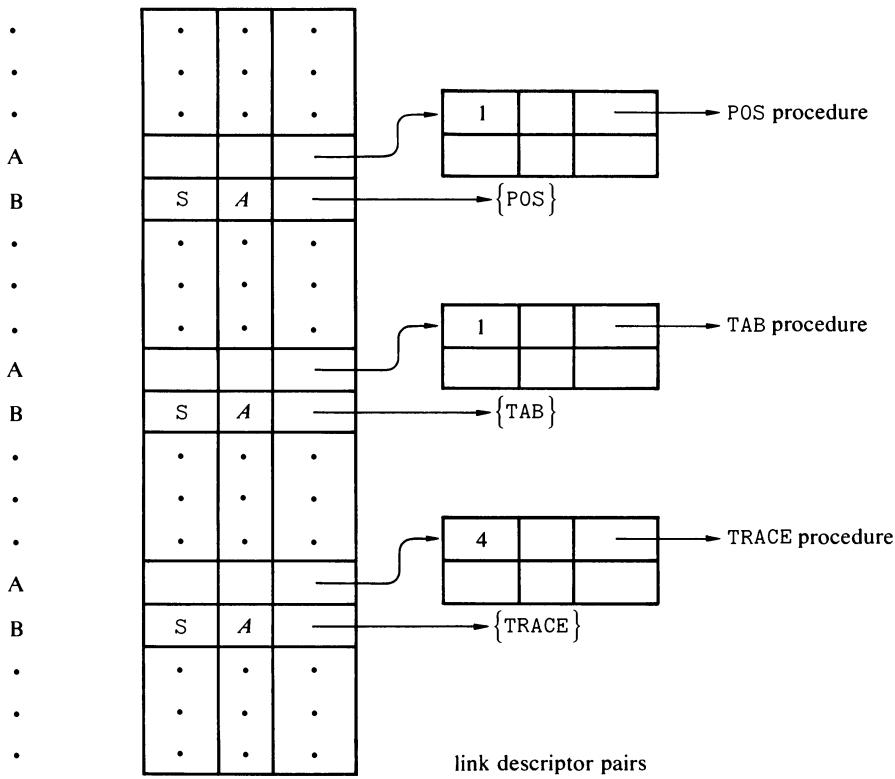


Figure 6.2.3
The structure of FNCPBL.

to the corresponding function procedure through the link descriptor. This is accomplished by a call to an invoking procedure, INVOKE, to which the function descriptor is passed as an argument. INVOKE branches indirectly to the appropriate function procedure. Control then resides in the function procedure, which performs the requested operation. Upon completion, the function procedure returns, relinquishing control to INTERP. INTERP continues with the next descriptor of the prefix code, and so on.

6.4 FUNCTION PROCEDURES

6.4.1 Argument Evaluation

A function procedure gains control with the code base pointer and code offset indicating the location of the arguments for the function in the prefix code. The function procedure evaluates its arguments, behaving much like INTERP, accessing successive descriptors of the prefix code. If a descriptor is an operand

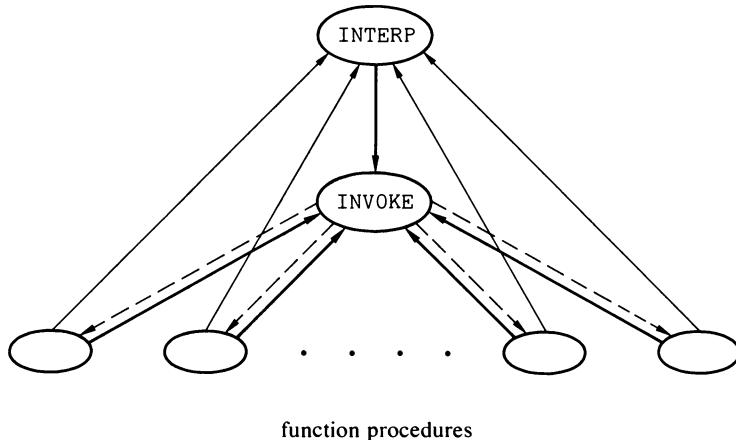


Figure 6.4.1
The relationship of procedures during interpretation.

descriptor, it is the desired argument. If it is a function descriptor, the evaluation of the argument requires a computation that involves invoking other function procedures. In this case, `INVOKE` or its equivalent is called to evaluate the argument. Some function procedures call common argument evaluation procedures instead of processing the prefix code themselves. The choice depends on the frequency of execution of the function procedure and whether data-type checking and conversion are required.

Figure 6.4.1 illustrates the relationships among the procedures involved. The heavy arrows indicate calls. The dashed arrows indicate indirect branches through link descriptors. The thin arrows indicate returns. `INVOKE` is a distributing mechanism serving as the entry to many different function procedures. Function procedures are accessed through `INVOKE`, and function procedures may call `INVOKE`. Thus, `INVOKE` is a focus for recursion.

6.4.2 Data-Type Checking and Conversion

The variety of data types in SNOBOL4 presents three problems.

- (1) Data types must be checked to detect possible errors in the source program.
- (2) In many cases, conversion of data types is required. Conversions of integers to strings and vice versa are most typical.
- (3) Several operators are polymorphous and perform different operations, depending on the data types of the operands.

The first two problems are frequently handled by using argument evaluation procedures that centralize data-type checking and conversion. A typical proce-

dure is INTVAL, which is called when an integer argument is required. INTVAL uses INVOKE to obtain the value from the prefix code. The data type of the result is checked. If the data type is INTEGER, the value is returned. If it is STRING, an attempt is made to convert this string to an integer. If this conversion can be performed, the resulting integer is returned as the value. If the value is neither an integer nor a string that can be converted to an integer, INTVAL transfers control to an error routine, indicating an illegal data type.

Polymorphous operators select appropriate subprocedures, depending on the data types of the operands. In many cases, data-type conversion of the operands is required. The data type of the value returned depends on the data type of the operands. The possible complexity is illustrated by the operation of concatenation. Concatenation basically performs two types of operations: concatenation of two strings to produce another string or concatenation of two patterns to produce another pattern. Figure 6.4.2 illustrates the possible situations.

The entries in the body of the table indicate the data types returned for various operand data types. If the two operands are REAL, for example, they are converted to strings and the strings concatenated to produce a string as indicated. In some cases, there are progressive conversions. For example, in the concatenation of a real number with a pattern, the real number is first converted to a string. Then the string is converted to a pattern, which is combined with the second argument to form a larger pattern.

Concatenation is further complicated by special handling of the null string. If either argument of concatenation is null, the other argument is returned as the value, regardless of its data type. If this were not done, the statement

X = LT(N,M) ARRAY(5)

would specify concatenation of a null string returned by LT with an array. Ordinarily, concatenation of a string with an array is an error. Exceptions are made for null strings so that predicates can be used without interfering with data manipulation.

		right operand									
		S	I	R	P	A	T	N	E	C	D
left operand	S	S	S	S	P				P		
	I	S	S	S	P				P		
	R	S	S	S	P				P		
	P	P	P	P	P				P		
	A										
	T										
	N										
	E	P	P	P	P				P		
	C										
	D										

Figure 6.4.2
Data types in concatenation.

6.4.3 Signaling Failure

As indicated in the previous section, function procedures evaluate their own arguments, rather than having their arguments pre-evaluated by the interpreter. Consider the statement

$$X = LT(SIZE(S), M) Y$$

The equivalent prefix form is

$$=_2 X \mid |_2 LT_2 SIZE_1 S \ M \ Y$$

where $\mid |_2$ is used to represent concatenation, since the SNOBOL4 language has no explicit symbol for concatenation. The sequence of function procedure invocation is

$$\begin{array}{c} =_2 \\ | |_2 \\ \quad LT_2 \\ \quad \quad SIZE_1 \end{array}$$

where indentation indicates level of call.

Several source-language operations may fail, depending on conditions. The predicate LT in the example above is typical. If LT fails, this failure must be signaled in such a fashion that further evaluation ceases and the statement fails. This is accomplished through a mechanism which permits procedures to signal on return. If LT fails, it returns, signaling failure. This failure signal occurs while concatenation is in the process of evaluating its first argument. Upon receiving this failure signal, concatenation ceases processing and also returns, signaling failure. This cascade of failure continues until the assignment procedure signals failure to INTERP. INTERP recognizes a failure signal as statement failure.

Successive returns, with failure signals, are necessary because when failure occurs, several recursive calls may have been made with subsequent storage of information on SYSSTK. This sequence of calls must be unraveled and SYSSTK restored to its state when the first call was made from INTERP.

6.4.4 Names and Values

Some source-language operations produce variables; others produce only values. Thus, an array reference produces a variable, but concatenation only produces a value. All variables have values, of course. Given a variable, its value is always available.

Conversely, some operations require variables while others need only values. Assignment is typical. The statement

$$A(I) = SIZE(I)$$

is an example. The left side of the assignment (the first argument of $=_2$) must be a variable. The right side need only be a value.

The operations that produce variables cannot be completely determined during translation. For example,

$$F(X) = 2$$

may or may not be a valid operation, depending on the definition of F when the statement is executed. Since the translator cannot resolve this problem, it must be handled during interpretation. This is accomplished by return signaling. A function procedure that computes a variable signals this fact on return. If a value is computed, that fact is signaled instead. These two signals are referred to as *return by name* and *return by value*, respectively.

When assignment invokes procedures to evaluate its first argument, it expects a return by name. If the return is by value, control is transferred to an appropriate error routine. Thus, in the statement

$$A(I) = \text{SIZE}(I)$$

the first operand returns by name and the second by value. On the other hand,

$$\text{SIZE}(I) = A(I)$$

is detected as an error (assuming SIZE has not been redefined). In a statement such as

$$A(I) = A(J)$$

evaluation of the second operand results in a return by name. The value is then obtained from the variable $A(J)$ and assigned to the variable $A(I)$.

In general, there are three return signals from function procedures: failure, name, and value. All function procedures that compute variables return these variables, signaling return by name, leaving it to the procedure that invoked them to use the variable or obtain its value. Failure is generally propagated back, regardless of whether a name or a value was expected. An exception is the negation operator, which converts a failure signal to a return of the null string by value.

The difference between name and value is illustrated by the following statements:

$$X = Y$$

and

$$X = 'Y'$$

The first statement produces the prefix code shown in Figure 6.4.3. Evaluation of the second argument of $=_2$ produces the variable Y whose *value* is assigned to the variable X . The second statement is another matter. Here Y , not the value of Y , is assigned to X . This is accomplished by a literal procedure, lit_1 , corresponding to the quotation marks. Figure 6.4.4 illustrates the prefix code. When

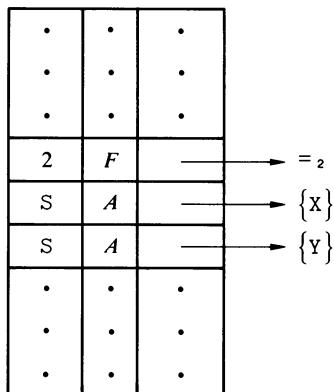


Figure 6.4.3
The statement $X = Y$.

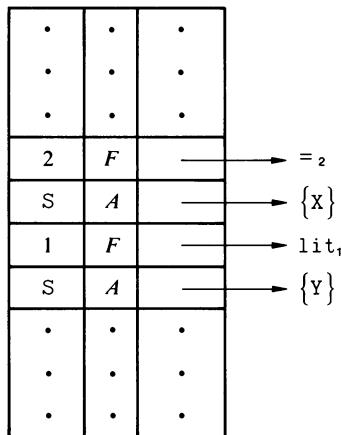


Figure 6.4.4
The statement $X = 'Y'$.

the second argument of $=_2$ is evaluated, the literal procedure is called. The literal procedure simply returns its argument by value, and the desired assignment is made.

The unary name operator, $._1$, is essentially the same as lit_1 , except $._1$ may have an array reference or function call as its argument.

Exercises

6.4.1 What kinds of processes may be invoked as a result of evaluation of an argument in a function procedure?

6.4.2 What is the difference between the prefix code for '10' and 10?

6.5 CONTROL OF PROGRAM FLOW

6.5.1 Sequential Execution

Source-language statements are executed in sequence unless failure, the call of a programmer-defined function, or a program goto intervenes. Within a statement, expressions are evaluated sequentially as determined by the prefix form.

This sequential execution occurs naturally during interpretation by processing successive descriptors of prefix code. As described in previous sections, there is a code base pointer and a code offset. As descriptors of prefix code are processed, the offset is incremented. This progression through the code begins with INTERP for each statement and is carried on by function procedures (such as =₂) and, where appropriate, argument evaluation procedures (such as INTERVAL).

6.5.2 Labels and Gotos

Gotos frequently intervene in the sequence of interpretation. A label identifies a location in the program. Internally, a label corresponds to a position in the prefix code. The association between a label and the corresponding code position is contained in the label descriptor of the natural variable for the label. Consider the following section of a program.

```
: (HEAD)
.
.
.
HEAD    OUTPUT    =    SUM
```

Figure 6.5.1 illustrates the association between the label HEAD and the location of its statement in the prefix code. Note the CODE data type in the label descriptor of HEAD. When the transfer to HEAD occurs, procedure goto₁ sets the code base pointer to the position given in the label descriptor of HEAD and resets the code offset to zero. The label descriptor points to the descriptor above the code for the statement labeled HEAD. The first step of interpretation is to increment the offset and to fetch a descriptor of the prefix code. Thus, the statement initialization procedure, init₁, is executed. Its argument contains the statement number, n, and a code offset, f, to use if the statement fails. The procedure init₁ stores these values, and interpretation continues.

To illustrate the use of the failure offset, consider the statement

```
TRYR    OUTPUT    =    INPUT    :S(START)
```

If INPUT successfully reads a record, the program transfers to START. Otherwise, the next statement is executed. Figure 6.5.2 illustrates the prefix code for this statement. If INPUT is successful, goto₁ is flowed into, and the transfer to

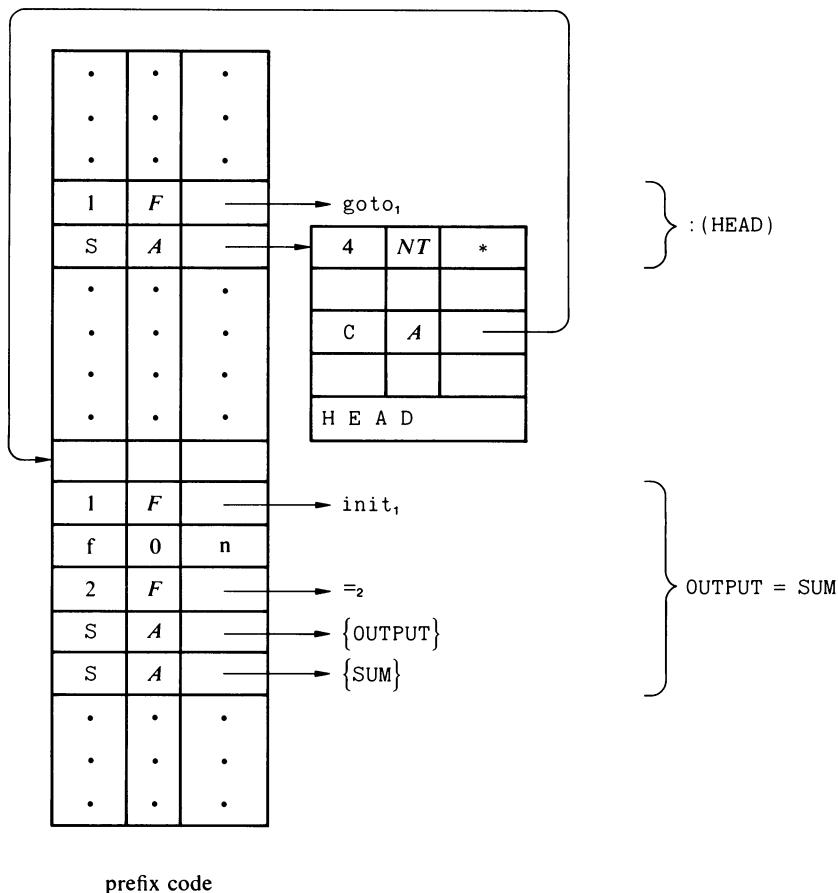


Figure 6.5.1
The label HEAD.

START takes place. If INPUT fails because of an end-of-file, this goto must be by-passed. This is accomplished by resetting the code offset from the failure offset stored by the initialization procedure (7d in this case). Interpretation continues at the new position, incrementing the offset and executing the next statement.

In summary, the position in the prefix code is always the sum of the code base pointer and the code offset. The offset is continually incremented during interpretation. If a statement fails, the offset is set to an appropriate new value where interpretation continues. In case of a transfer, the code base pointer is set to a new location, and the offset is reset to zero. The code base pointer is only reset when a label is encountered or a transfer occurs. Consider the statements

```
PRINT  OUTPUT    =    SUM
TRYR   OUTPUT    =    INPUT      : S(START)
```

.	.	.
.	.	.
.	.	.
1	<i>F</i>	
7d	0	<i>n</i>
2	<i>F</i>	
S	<i>A</i>	
S	<i>A</i>	
1	<i>F</i>	
S	<i>A</i>	
.	.	.
.	.	.
.	.	.

Diagram illustrating a statement with a success goto. A vertical double-headed arrow on the left indicates a height of $7d$. Arrows point from specific memory locations to their corresponding meanings:

- $1 \quad F$ → init_1
- $7d \quad 0 \quad n$ → $=_2$
- $S \quad A$ → $\{\text{OUTPUT}\}$
- $S \quad A$ → $\{\text{INPUT}\}$
- $1 \quad F$ → goto_1
- $S \quad A$ → $\{\text{START}\}$

Figure 6.5.2
A statement with a success goto.

.	.	.
.	.	.
.	.	.
2	<i>F</i>	
S	<i>A</i>	
S	<i>A</i>	
0	<i>F</i>	
1	<i>F</i>	
7d	0	<i>n</i>
2	<i>F</i>	
S	<i>A</i>	
S	<i>A</i>	
.	.	.
.	.	.
.	.	.

Diagram illustrating an instance of the basing procedure. A vertical double-headed arrow on the left indicates a height of $7d$. Arrows point from specific memory locations to their corresponding meanings:

- $2 \quad F$ → $=_2$
- $S \quad A$ → $\{\text{OUTPUT}\}$
- $S \quad A$ → $\{\text{SUM}\}$
- $0 \quad F$ → base_0
- $1 \quad F$ → init_1
- $7d \quad 0 \quad n$ → $=_2$
- $S \quad A$ → $\{\text{OUTPUT}\}$
- $S \quad A$ → $\{\text{INPUT}\}$

Figure 6.5.3
An instance of the basing procedure.

The statement labeled TRYR can be entered by a transfer or by program flow from above. In either case, the code base pointer must be the same when TRYR is executed. If the entry is accomplished by a transfer, the code base pointer is set to a new value, and the offset is zeroed. If, on the other hand, the statement labeled PRINT is executed, the code base pointer and the code offset must be corrected before flowing into TRYR. This is accomplished by executing a simple basing procedure, `base0`. Figure 6.5.3 illustrates the prefix code. The basing procedure, which has no argument, simply increments the current code base pointer by the current code offset and then zeros the offset. The result is the same as if a transfer had been effected.

Translation

The translator converts the source-language program into prefix code of the type described in the previous chapter. Translation involves analysis (parsing) of source-language statements, generation of prefix code, and creation of related structures, such as natural variables, procedure linkages, label definitions, and so forth.

7.1 SYNTAX

Appendix A contains a complete formal syntax of a SNOBOL4 statement. This description is relatively uniform in the sense that the same notation is used for all syntactic types and there is no obvious hierarchy, except that the definitions are generally based on preceding, simpler definitions, which finally culminate in the definition of a statement. It is more natural to think of three levels of structure: (1) statement structure, (2) element and expression structure, and (3) token structure.

There are various types of statements which consist of components such as subjects and patterns. These components, in turn, consist of elements, such as identifiers and literals, and expressions which are built up out of elements. Underlying this structure is the “fine structure” of the individual tokens, such as the characters which compose identifiers and operators. The implementation of the translator is based on these three levels of structure.

7.1.1 Statements

A statement has the general form

label rule goto

If the first character of a statement is not a blank, the characters up to the first blank or semicolon constitute the label. The rule is next. The goto field, if present, is separated from the rule by a colon, and so on.

There are three main types of rules:

$$\begin{array}{lll} \text{subject} & = & \text{object} \\ \text{subject} & & \text{pattern} \\ \text{subject} & \text{pattern} & = \text{object} \end{array}$$

The first two types can be considered special cases of the third. The components of a rule are governed by positional as well as structural factors. Thus, in the statement

L1 X Y Z = U V W :S(L2)

X is the subject by virtue of its position as the first nonblank construction following the label. The pattern is an expression consisting of the concatenation of the two elements Y and Z. The object is also an expression, consisting of the concatenation of U, V, and W. Because of the positional and syntactic definition of a subject,

X Y Z

and

X (Y Z)

are equivalent, but in

(X Y) Z

the subject is the concatenation of X and Y whereas the pattern is Z.

The rule forms have their historical sources in the extremely simple syntax of the first version of SNOBOL. These forms are atypical of more recent programming languages and lack the structural uniformity that permits straightforward parsing techniques. Blanks stand for different operations in different positions. In one place, blanks signify pattern matching and, in another, concat-

tenation. Nevertheless, it is a straightforward, if inelegant, process to convert the rule forms into their prefix equivalents.

The procedure ANALYZ handles syntax at this level. ANALYZ knows that a label begins with the first character of a statement, that the subject is next, and so on.

7.1.2 Elements and Expressions

The word *element* is used to describe self-delimiting components such as literals, identifiers, and parenthesized expressions. An element is similar to a “primary” commonly used in describing programming languages. The word *expression* is used to describe binary operations as well as elements. As is usual in syntactic descriptions, the definitions of element and expression are mutually recursive. Thus,

- (1) An identifier is an element.
- (2) A unary operator applied to an element is an element.
- (3) An element is an expression.
- (4) An expression enclosed in parentheses is an element.
- (5) A binary operation on expressions is an expression.

And so on. The reader who prefers a complete formal description should refer to Appendix A. Examples of elements are

```
A
'A'
-32
F(A + B)
(SIZE(X) * 2)
```

All elements are also expressions. Some expressions which are not also elements are

```
A + B
SIZE(X) * 2
'A' | 'B' | 'C'
```

All components of a rule can be expressions except the subject, which must be an element. This restriction is a consequence of the dual use of blanks, mentioned above, and is used to resolve the inherent ambiguity of

X Y Z

as

X (Y Z)

Two procedures, ELEM and EXPR, analyze elements and expressions, respectively. ELEM is called, for example, to analyze the subject of a statement. A call to EXPR occurs after the subject has been analyzed and a pattern is expected, and so on.

7.1.3 Tokens

Syntax tables are used by a process, STREAM, which consumes input text. These syntax tables describe the token structure and the role of individual characters. Generally speaking, syntax tables are used to define those parts of the syntax that can be recognized by a finite-state machine [16]. A special notation is used for defining the syntax tables. Each table has entries which may have several parts:

- (1) A FOR part, which designates a class of characters for which a particular action is to be taken.
- (2) A GOTO part, which specifies the table to be used for the next character of the input text stream. CONTIN is used as an abbreviation when the same table is to be used again.
- (3) A PUT part, which specifies information to be returned. This information usually takes the form of an identification of the particular type of token discovered. The information returned by PUT is placed in the V field of a global descriptor, STYPE.

In addition there are several terminal actions that stop analysis with various indications:

- (1) STOP, indicating analysis is to stop with the last character of the input text stream to be included in the token identified.
- (2) STOPSH, indicating the last character is not to be included in the token identified.
- (3) ERROR, indicating the detection of a syntactic error.

A typical table is ELEMNT, used to start the analysis of an element. This table has the definition

```
BEGIN ELEMNT
  FOR(DIGIT) PUT(INTCOD) GOTO(INTGER)
  FOR(LETTER) PUT(IDCOD) GOTO(IDENTF)
  FOR(SQUOTE) PUT(LITCOD) GOTO(SQLIT)
  FOR(DQUOTE) PUT(LITCOD) GOTO(DQLIT)
  FOR(LP) PUT(PRNCOD) STOP
  ELSE ERROR
END ELEMNT
```

DIGIT, LETTER, SQUOTE, DQUOTE, and LP are names of classes of characters (LP stands for “left parenthesis”). INTCOD, IDCOD, LITCOD, and PRNCOD are syntactic type codes that distinguish the type of element: integer, identifier, literal, or parenthesized expression. INTGER, IDENTF, SQLIT, and DQLIT are other tables used for the remaining analysis, depending on the character encountered by ELEMNT. STOP and ERROR are termination conditions, mentioned above.

Token analysis involves applying STREAM to the input text. In a sense, this is like feeding symbols into a finite-state machine, in which each state corresponds to a table, and each input symbol causes a transition to another (possibly the same) state. A transition may cause the output of a piece of information which may be thought of as a symbol in a different alphabet. STOP, STOPSH, and ERROR correspond to special terminal states.

If ELEMNT is applied to the string 'HARMONY', the first character causes the output of a literal identification code (LITCOD) and transfer to SQLIT for continued analysis. SQLIT has the definition

```
BEGIN SQLIT
    FOR(SQUOTE) STOP
    ELSE CONTIN
END SQLIT
```

SQLIT continues processing the string, eventually stopping at the terminating quotation mark. A state diagram for the process is shown in Figure 7.1.1. Arrows indicate transitions. The input and output symbols are shown separated by slashes next to the arrows. Dashed lines indicate transitions to states not shown in this figure. Complete diagrams for INTGER, IDENTF, and DQLIT are given in Appendix B.

The major advantage of the use of syntax tables for token definition is generality, ease of description, and ease of modification. Embodying a good part of the syntax of SNOBOL4 in such tables makes it possible to change the meaning of tokens over a wide range of possibilities without modifying the translator itself. Similarly, the character class names are machine-independent, but each class is given an appropriate definition for each machine, independent of the rest of the implementation.

Exercises

- 7.1.1 Suggest a modification to the syntax of SNOBOL4 that would permit the subject of a statement to be an expression.
- 7.1.2 The table ELEMNT has an entry for the left parenthesis, but none for the left bracket. Why?
- 7.1.3 What is the first thing to be looked for if an element is expected?
- 7.1.4 Why is it necessary to have two tables for the two types of quoted literals?

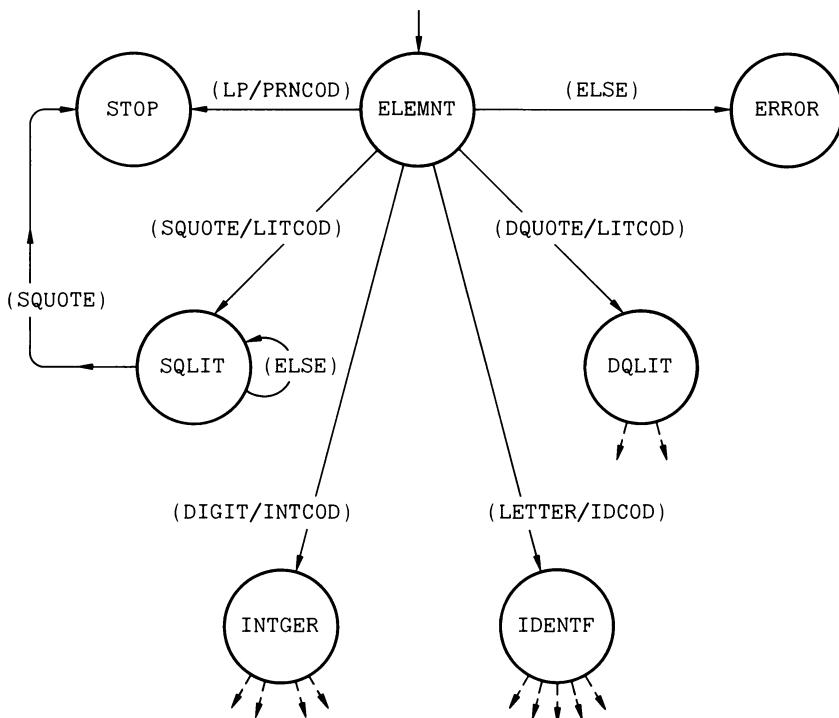


Figure 7.1.1
Transitions from **ELEMNT**.

7.2 THE INPUT TEXT STREAM

Source-language input to the translator consists of line images, i.e., string segments of fixed length. Usually each line is a statement. The continuation convention permits a statement to extend over several lines. Conversely, several statements may appear on one line, separated by semicolons. Conversion to CODE, on the other hand, presents the translator with a string of arbitrary length. Examples of these cases are illustrated by the following lines:

```

OUTPUT      =  'THE TOTAL VALUE IS '
+          X ' WHERE THE MAXIMA AND MINIMA ARE '
+          MAX ' AND ' MIN 'RESPECTIVELY.'
+          :(NEXTCASE)

INIT      X =1; Y = 1; Z = 0; A =; B =; &STLIMIT = &STLIMIT +1000

READ      TEXT      =  TEXT INPUT ';'      :S(READ)
PROG      =  CONVERT(TEXT,'CODE')
  
```

To accommodate these various cases in a uniform manner, the translator is organized to operate on a continuous stream of text. A global qualifier, TEXTQL, always points to the current position in the input stream. Once a construction has been analyzed, the procedure FORTXT is called. FORTXT uses STREAM to update TEXTQL to point to the next significant (nonblank) character in the input stream. If the current line image is exhausted, FORTXT reads another line and establishes TEXTQL appropriately. FORTXT also prints lines that are read in. Continuation lines are recognized by FORTXT so that a continued statement appears to be part of a continuous stream to the rest of the translation procedures. New lines are read only as needed, and there is no limit to the number of continuation lines that are allowed in a statement. FORTXT also handles comment and control lines and, in general, makes source-language statements appear to be a continuous stream of characters to the analyzing routines. If conversion to CODE is being performed, FORTXT signals the end of translation when the end of the string is reached, instead of trying to read another line. The rest of the translator is unaware of the difference between translating source-language programs and converting strings to CODE.

7.3 THE TRANSLATION PROCESS

When translation begins, a large block is allocated for prefix code. As translation proceeds, code descriptors are created and inserted in this block. A code base pointer and a code offset are used during translation to address locations in much the same way as they are used during interpretation. Code descriptors come from various sources. Some code descriptors are constructed by ANALYZ. Others are created by EXPR and ELEM and inserted into code trees, which are subsequently copied into the prefix code block. Most code descriptors originate from STREAM as a result of specifications in syntax tables.

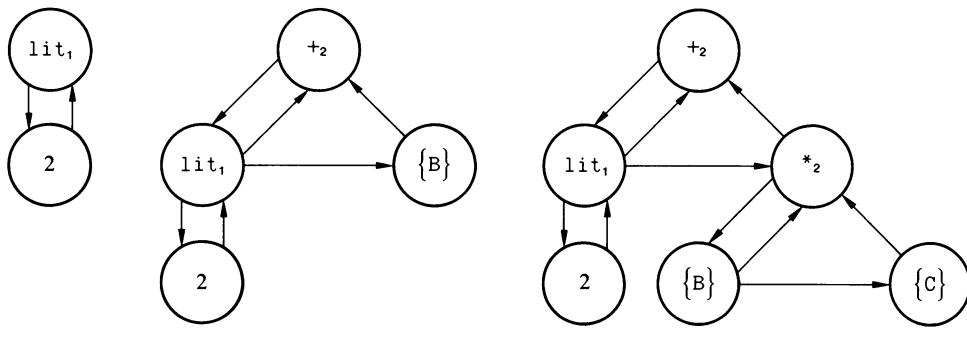
7.3.1 Code Trees

Code trees are transient representations of expressions and elements, built up as the relationship between operators and their operands is determined. Consider the expression

2 + B * C

Figure 7.3.1 shows the development of the code tree as this expression is analyzed. In the second step, the expression 2 + B has been analyzed. When * is encountered, the tree is modified to account for the fact that * has higher precedence than +. Thus, code trees hold parts of an expression until the rest can be analyzed and placed in the proper position.

The following terminology is used to describe trees. Trees consist of *nodes*,



first step

second step

final tree

Figure 7.3.1
A code tree.

4d	T	*	
			pointer to father
			pointer to left son
			pointer to right sibling
			code descriptor

Figure 7.3.2
The structure of a node.

indicated by circles. The top node is the *root*. An arrow downward points to a *leftmost son*. An arrow upward points to the *father* of a node. Sons are siblings arranged left to right. An arrow points horizontally to a *right sibling*. Every node contains a code descriptor, which is depicted in diagrams as the contents of the node. In subsequent diagrams only arrows from fathers to sons are shown. The remaining relationships can be determined by the positions of the nodes.

Nodes of code trees are implemented as blocks of descriptors. Each node consists of four descriptors, as shown in Figure 7.3.2. As trees are constructed, the T field of the code descriptor of a node is incremented whenever a son is added to the node. Thus, the T field contains the number of sons belonging to the node. The internal representation of the tree illustrated in Figure 7.3.1 is shown in Figure 7.3.3.

7.3.2 Elements

ELEM may encounter a variety of constructions: (1) unary operators applied to elements, (2) identifiers, (3) quoted literals, (4) integer and real literals, (5) parenthesized expressions, (6) function calls, and (7) array and table references.

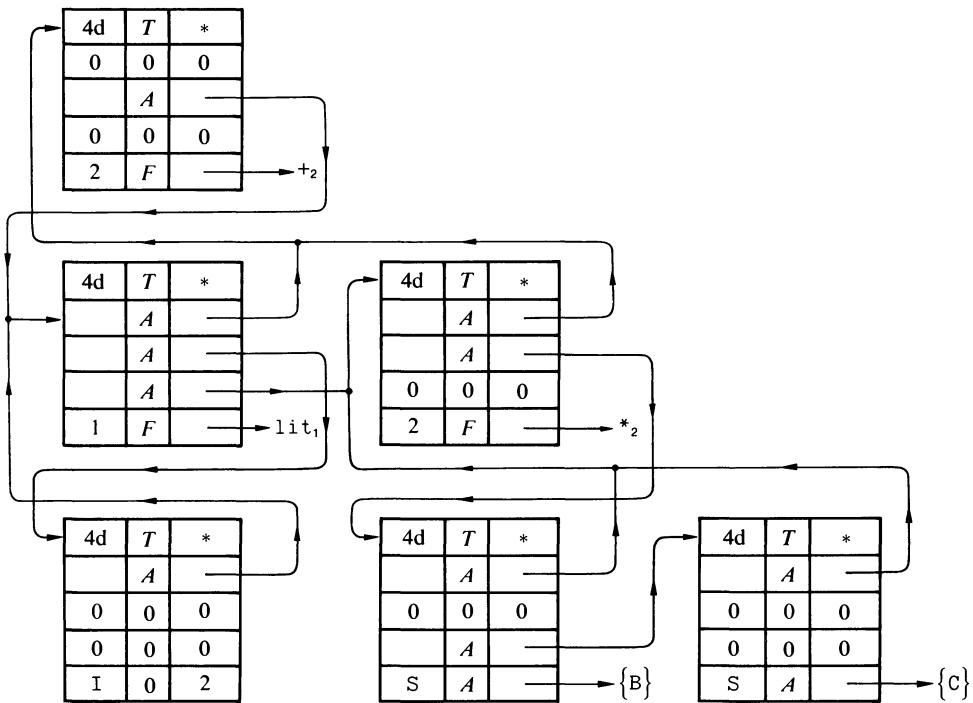


Figure 7.3.3
Implementation of a code tree.

An element may begin with a unary operator. Unary operators are identified by STREAM, using the syntax table UNOPS which has the definition

BEGIN UNOPS

```

FOR(PLUS) PUT(UPLSOP) GOTO(NBLANK)
FOR(MINUS) PUT(UMNSOP) GOTO(NBLANK)
FOR(DOT) PUT(UDOTOP) GOTO(NBLANK)
FOR(DOLLAR) PUT(UDOLOP) GOTO(NBLANK)
FOR(STAR) PUT(USTROP) GOTO(NBLANK)
FOR(SLASH) PUT(USLHOP) GOTO(NBLANK)
FOR(AT) PUT(UATOP) GOTO(NBLANK)
FOR(POUND) PUT(UPNDOP) GOTO(NBLANK)
FOR(PERCENT) PUT(UPCTOP) GOTO(NBLANK)
FOR(EXCLAM) PUT(UXLMOP) GOTO(NBLANK)
FOR(BAR) PUT(UBAROP) GOTO(NBLANK)
FOR(AND) PUT(UANDOP) GOTO(NBLANK)
FOR(NOT) PUT(UNOTOP) GOTO(NBLANK)
FOR(QUES) PUT(UQUEOP) GOTO(NBLANK)
ELSE ERROR

```

END UNOPS

	.	.	.
	.	.	.
	.	.	.
UANDOP	1	0	
	0	0	0
UMNSOP	1	0	
	0	0	0
UNOTOP	1	0	
	0	0	0
	.	.	.
	.	.	.
	.	.	.

Figure 7.3.4
A section of unary operators in OPERBL.

NBLANK is a syntax table which assures that the character following the operator is not a blank. The result returned by UNOPS is not a syntax type code, but rather the location of an entry in an operator block, OPERBL. Entries for unary operators in OPERBL are link descriptors, similar to those for functions pointed to by entries in FNCPBL. Thus, for the operator $-_1$, the value returned is UMNSOP, the location in OPERBL of the unary minus operator. Figure 7.3.4 illustrates a section of OPERBL.

The pointer to a link descriptor returned by STREAM becomes the code descriptor of a root node for the expression. There are no entries for operators in FNCPBL since there is no way to define new operator names.

Other types of elements are determined using STREAM and the table ELEMNT. The type of element found is indicated by a syntax type code returned by STREAM.

An identifier is the simplest case. A natural variable for the string consumed by STREAM is returned in a tree consisting of a single node. Although there is really no need to construct a node for an identifier, this is done to obtain generality so that ELEM always returns a tree. Quoted literals, integer literals, and real literals all produce the same type of code tree. The root is a node pointing to lit₁, with the literal value as its son. Trees for some simple elements are shown in Figure 7.3.5.

When a left parenthesis is encountered, EXPR is called. When EXPR encounters the closing right parenthesis, it returns a tree for the parenthesized expression.

A function call is indicated by an identifier followed by a left parenthesis. FNCPBL is searched, comparing the B descriptors with the natural variable for the identifier. If the identifier is found, the function already has a definition, and a pointer to the link descriptor is inserted in the code descriptor of a node. If

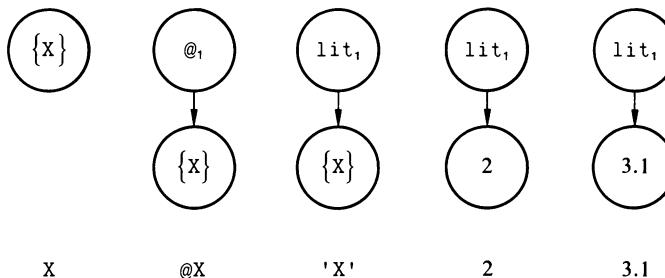
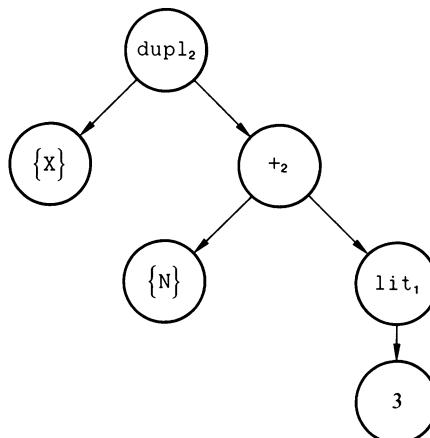


Figure 7.3.5
Trees for simple elements.

there is no definition, a function pair is added to FNCPBL for the new function, and a link descriptor pair is created. Since in this case the function is undefined, the link descriptor of the new pair points to the error routine which handles calls to undefined functions. This pointer is subsequently overwritten during execution when the function is defined (see Section 8.2). Next, EXPR is called to analyze the arguments of the function. EXPR returns a code tree whenever it encounters a comma or right parenthesis, indicating the end of an argument. The trees returned by EXPR become siblings as sons of the function node. When EXPR returns because of a comma, EXPR is called again to get the next argument. When EXPR returns because of a right parenthesis, the tree is complete, and ELEM returns after adding the last sibling tree. Figure 7.3.6 illustrates the tree for a function call. If trailing arguments are omitted in the call of a built-in function, null strings are provided by the translator to fill out the prefix code. The number of arguments expected is given in the T field of the link descriptor.



DUPL(X, N + 3)

Figure 7.3.6
A tree for a function call.

An array or table reference is treated as a call to a built-in function ITEM. Therefore,

$A(I, J)$

is translated as if it were

ITEM(A, I, J)

The first argument of ITEM is the array or table, and the remaining arguments are the indices. ITEM performs the necessary computation to locate the array or table entry. In the analysis of references, the final argument is terminated by a right bracket.

7.3.3 Expressions

Expressions consist of elements separated by binary operators. EXPR obtains elements (by calling ELEM) and binary operators (by applying STREAM) alternately.

As mentioned in Chapter 2, binary operators are ranked according to their precedence, which determines the binding of operands. Furthermore, some operators associate to the left and others to the right. A complete list of binary operators showing their precedence and associativity is shown in Figure 7.3.7. The precedences shown are relative, not absolute.

<u>symbol</u>	<u>definition</u>	<u>associativity</u>	<u>precedence</u>
\sqcap	none	right	12
$?$	none	left	12
$$$	immediate value assignment	left	11
$.$	conditional value assignment	left	11
$!**$	exponentiation	right	10
$\%$	none	left	9
$*$	multiplication	left	8
$/$	division	left	7
$\#$	none	left	6
$+$	addition	left	5
$-$	subtraction	left	5
$@$	none	left	4
blank	concatenation	left	3
$ $	alternation	left	2
$&$	none	left	1

Figure 7.3.7
Binary operators.

Thus, the expressions

$A + B * C$
 $A * B + C$
 $A - B - C$
 $A ! B ! C$

are equivalent to

$$\begin{aligned} A + (B * C) \\ (A * B) + C \\ (A - B) - C \\ A ! (B ! C) \end{aligned}$$

respectively. The code trees for these four expressions are shown in Figure 7.3.8.

Binary operators are identified by STREAM using the syntax table BINOPS, which is similar to UNOPS. The result returned by BINOPS is the location of an entry in OPERBL. Entries for binary operators consist of the usual link descriptor pair, as they do for functions and unary operators, and an additional precedence descriptor. For the operator $*_2$, the value returned is BSTROP (which stands for “binary star operator”). Figure 7.3.9 illustrates a section of the operator table.

The precedence of an operator is determined by the magnitude of the integer in the T field of its precedence descriptor. Thus, $.2$ has higher precedence than

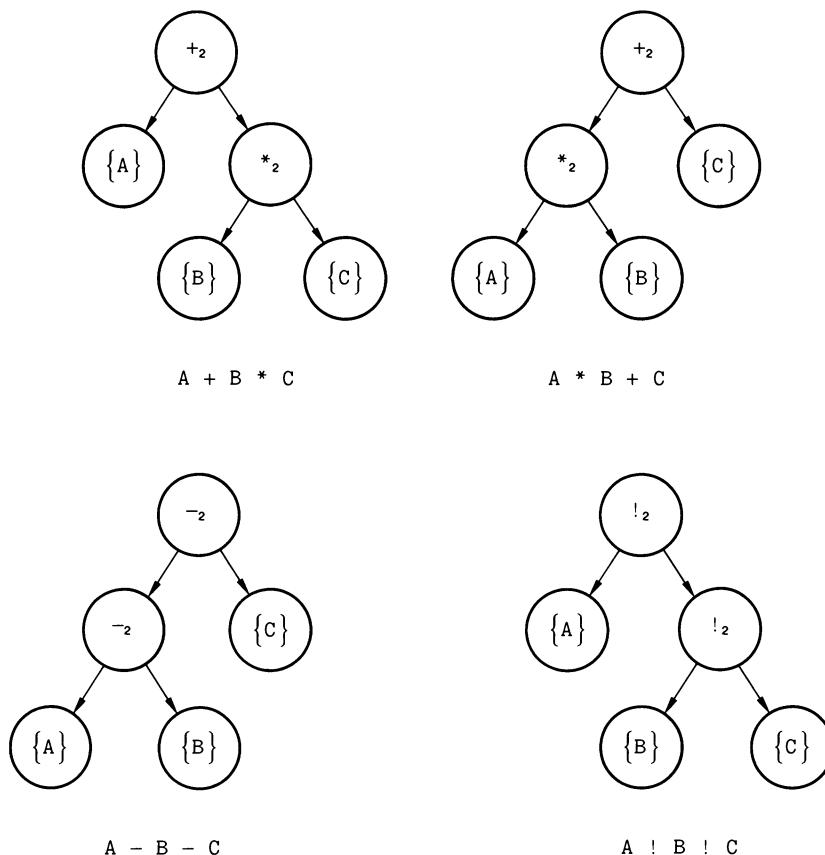


Figure 7.3.8
Code trees for expressions.

BXLMOP	2	0
	0	0
	50	0
BSTROP	2	0
	0	0
	42	0
BDOTOP	2	0
	0	0
	60	0
	•	•
	•	•
	•	•

exponentiation procedure

multiplication procedure

conditional value assignment procedure

Figure 7.3.9
A section of binary operators in OPERBL.

$!_2$, which, in turn, has higher precedence than $*_2$. By convention, an operator associates to the left if the T field of its precedence descriptor is greater than the V field of its precedence descriptor. Otherwise, the operator associates to the right. Thus, $._2$ and $*_2$ associate to the left and $!_2$ associates to the right.

When a binary operator is located by EXPR, the descriptor returned by STREAM (a pointer to the link descriptor) becomes the code descriptor of the operator node. This node is placed in its proper place in the current code tree, using information in the precedence descriptor.

Note that OPERBL is not a pair block, and entries consist of link descriptor pairs (with additional precedence descriptors for binary operators). Link descriptors for operators are located by using STREAM, as contrasted with searching FNCPBL for function link descriptors. In the case of both functions and operators, the code descriptor is a pointer to a link descriptor. Thus, the syntactic difference between functions and operators disappears after translation.

7.3.4 Statements

Translation of a statement consists of four parts: (1) label processing, (2) statement initialization, (3) rule processing, and (4) goto processing. If a statement has a label, a code descriptor for the basing procedure, base_0 , is inserted in the

prefix code, and a pointer to that location is inserted in the label descriptor of the natural variable for the label. These two simple operations associate the location of the labeled statement with the label and assure that statement execution is properly based whether the statement is reached by sequential execution or by a program transfer.

Next, a code descriptor for `init1` is inserted in the prefix code. A statement number counter is incremented and its value inserted into the V field of the argument descriptor. The location of this argument descriptor is stored in a global variable. The T field is left empty until the failure offset can be determined.

Rule analysis is next. The types of rules are exemplified by the following prefix equivalents:

<i>rule form</i>	<i>prefix equivalent</i>
X = Y	$=_2 X \ Y$
X Y	$\text{match}_2 X \ Y$
X Y = Z	$\text{replace}_3 X \ Y \ Z$

ELEM is called to analyze the subject, and the code tree which is returned is saved. If there is a pattern, EXPR is called and the code tree which is returned is also saved. If there is an object, EXPR is called again. By the time rule analysis is completed, the type of statement has been determined, and the appropriate code descriptor, $=_2$, match_2 , or replace_3 , is inserted in the prefix code. The code trees which have been saved are now copied into the prefix code, as described in the next section.

If a goto is present, its type is determined by using STREAM. EXPR is called to analyze the gotos. In the case of an unconditional goto, ANALYZ inserts `goto1` in the prefix code and then copies the code tree for the goto into the prefix code. There are similar cases for other goto configurations.

After analysis of the goto field, the failure offset for the statement is known and is inserted in the T field of the argument of `init1`.

7.3.5 Obtaining Prefix Code from Code Trees

Code trees are only an intermediate representation used during translation. Prefix code is obtained by “left-listing” a tree and abstracting the code descriptors in the proper order. The order is given by the following rules:

- (1) Start with the root.
- (2) Copy the code descriptor of the current node.
- (3) If the current node has a left son, move to the left son and go to rule (2).
- (4) Otherwise, if the node has a right sibling, move to the right sibling and go to rule (2).

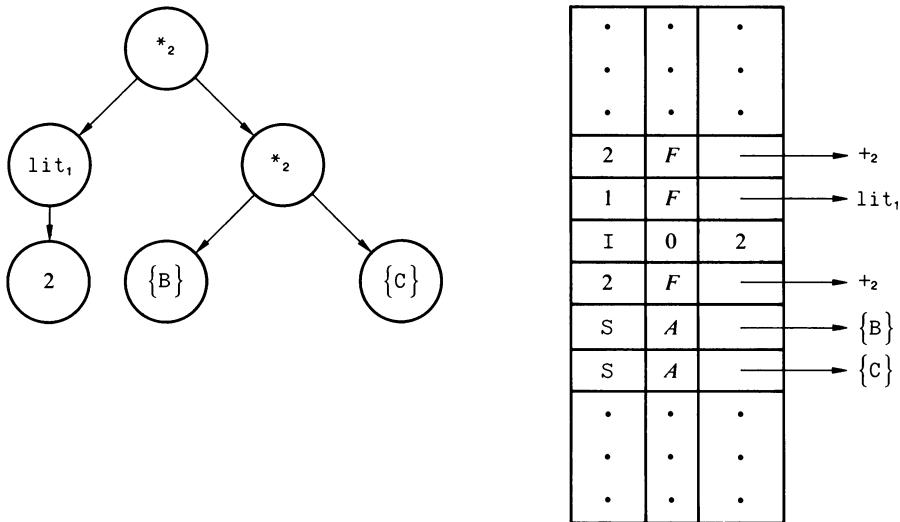


Figure 7.3.10
Prefix code resulting from a code tree.

- (5) Otherwise, if the node has a father, move to the father and go to rule (4).
- (6) Otherwise, the listing is complete.

A left-listing of the code tree given in Figure 7.3.1 produces the prefix code shown in Figure 7.3.10.

7.3.6 Errors

Syntactic errors may be discovered in a variety of contexts. Most errors are signaled by STREAM as the result of ERROR conditions specified in syntax tables. A previously used label is detected when a label is encountered that already has a nonzero label descriptor. An incomplete expression (for example, a missing closing parenthesis) is determined by context. ELEM and EXPR call each other recursively, and when an error is detected by one of these procedures, an identifying error code is posted, and failure is signaled. This failure signal causes the procedure that called it to return, also signaling failure. Finally, ANALYZ receives the failure signal.

When an error is detected by ANALYZ, a considerable portion of the statement may already have been analyzed and the corresponding prefix code already written. ANALYZ resets its code offset to the beginning of the statement and writes a code descriptor for error, over init. If that statement is subsequently executed, error, transfers to an error routine. ANALYZ also prints an error message in the program listing, indicating the location in the input stream where the error was detected.

Exercises

- 7.3.1** Draw code trees for the following elements:

```
*@A(B(I))  
F(X + 2,SIZE(Y),F('3',2,DUPL(7,X)))  
IDENT(X)
```

- 7.3.2** Write the algorithm for building trees for expressions, taking the precedence and associativity of operators into account.
- 7.3.3** An expression may be null. What is returned in this case?
- 7.3.4** Why is EXPR (rather than ELEM) called to analyze gotos?
- 7.3.5** There are two degenerate rule forms not discussed in the section above: the null rule and the rule consisting only of a subject. How might these cases be handled by ANALYZ?
- 7.3.6** Write an algorithm for translating rules, taking all types of rules into account.
- 7.3.7** The syntax of SNOBOL4 permits the object of an assignment or replacement statement to be omitted. For example,

X =

assigns the null string to X. Describe the prefix code generated in such cases.

- 7.3.8** Left-list the code trees obtained in solving Exercise 7.3.1.
- 7.3.9** Write an algorithm for constructing code trees from prefix code. Hence, show that there is a one-to-one correspondence between code trees and prefix code.
- 7.3.10** The interpreter could be written to operate on code trees instead of prefix code. What disadvantages would this have?

Implementation of Representative Features

8.1 STRINGS AND VARIABLES

Strings and operations on strings are the most important features of the SNOBOL4 language. Not surprisingly, strings also have a major influence on the implementation. The large number of strings typically created during program execution and freedom from limitations on length create allocation problems. More significant is the fact that every nonnull string may be used as a variable. In implementing SNOBOL4, the decision was made to treat strings uniformly as variables, whether or not they are ever actually used that way.

8.1.1 Natural Variables

Whenever a string is constructed by a source-language operation, a natural variable is used to represent that string. Natural variables are unique. That is, there is exactly one natural variable for each distinct string. Put another way, there is only one copy of each different string in the allocated data region.

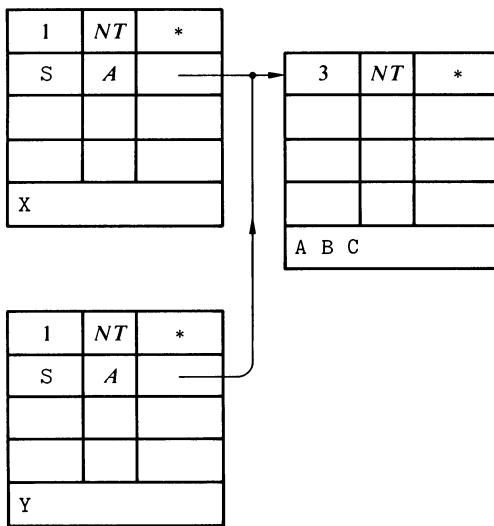


Figure 8.1.1
Typical string values.

Because strings are uniquely represented by natural variables, a descriptor that points to a natural variable is a unique representation of the string. That is, all pointers to the same natural variable are identical. To illustrate this situation, consider the statements

```
X = 'AB' 'C'  
Y = 'A' 'BC'
```

As a result of executing these statements, both X and Y have the same value, ABC. Figure 8.1.1 illustrates the data structures. The important point is that the value descriptors of X and Y are *identical* and point to the same natural variable, although the values are constructed in different ways.

8.1.2 The Table of Natural Variables

In a sense, natural variables (strings) are the symbols of SNOBOL4. Natural variables are organized in a symbol table. Whenever a string is created, this symbol table is searched for that string. If the string already exists, a descriptor pointing to its natural variable is returned. If the string does not exist, a natural variable containing it is added to the symbol table. This operation is performed frequently, and the symbol table is organized for efficient look-up. The organization is based on the concept of hash addressing [17–19]. Instead of using the conventional technique which computes an address in a table, a hash function is used to separate strings into categories. Within each category, natural vari-

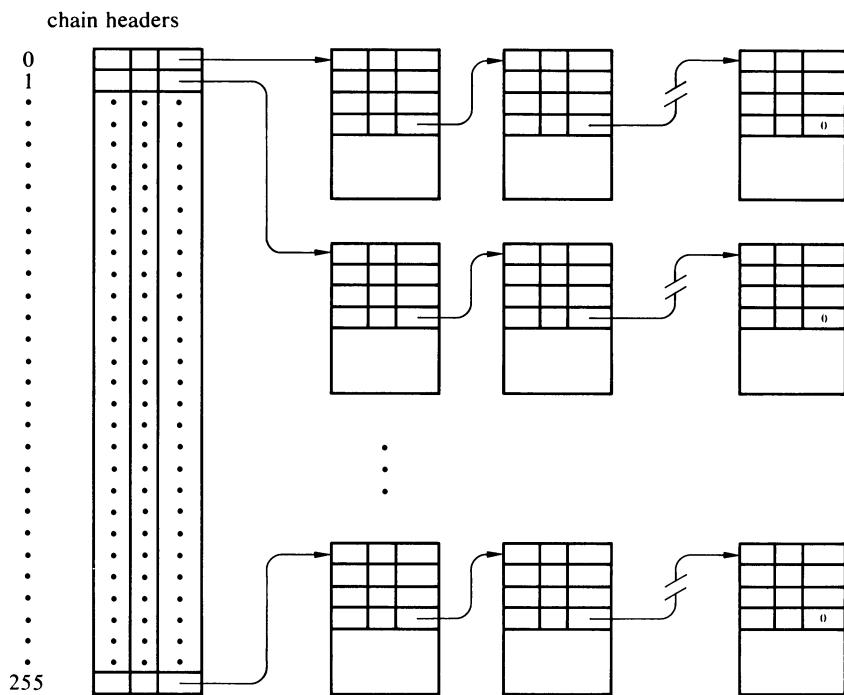


Figure 8.1.2
The table of natural variables.

ables are linked together to form a chain which is ordered by a second hash function. Figure 8.1.2 illustrates schematically the table of natural variables.

There are 256 chains of natural variables. The chain on which a natural variable is placed and its position on the chain are determined by hash computations performed on the characters of the string. The characters, as encoded in the internal machine representation, are treated as numbers. Two hash functions compute a chain offset, C, and an order number, N:

$$\begin{aligned} C &= h_1(\text{string}) \quad (0 \leq C \leq 255d) \\ N &= h_2(\text{string}) \end{aligned}$$

The detailed definitions of h_1 and h_2 and the permissible range of N are machine-dependent. The V field of the chain descriptor in a natural variable is used to point to the next natural variable on its chain. The V field of the last variable on a chain is zero. The order number N is stored in the T field of the chain descriptor. When a string is created, C and N are computed. The chain designated by C is searched for the string by comparing N with successive order numbers. If a string with order number N is found, the strings themselves must be compared since two different strings may have the same order number. Since variables on a chain are ordered by increasing order number, the search stops when an order number exceeding N is found. If the string is found, a pointer to its

natural variable is returned. If the string is not found, space is allocated for a new natural variable, which is then linked into the chain at the appropriate spot. When a new natural variable is created, it is given the null string as value and a zero label descriptor. Figure 8.1.3 illustrates a section of a typical chain in more detail.

Ideally h_1 should evenly divide strings among the chains (to provide a flat distribution), independent of the characters of its string arguments, whatever the

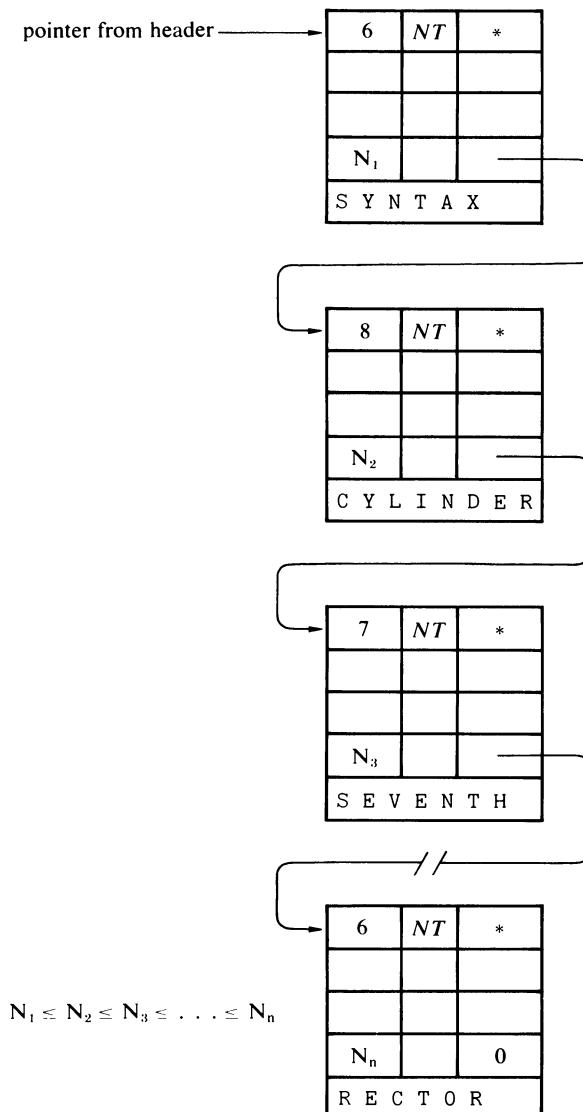


Figure 8.1.3
A chain of natural variables.

regularity and correlation there may be among strings. Similarly, h_2 ideally would produce different values of N for all different strings. Within the constraints given, neither goal is possible. The extent to which h_1 and h_2 approach these goals is a measure of their quality and significantly affects the running speed of many SNOBOL4 programs. (See Sections 11.1.4 and 11.2.4 for specific instances of h_1 and h_2 for two different machines.)

It is important to realize that all strings created by source-language operations are represented by natural variables. Thus, concatenation of two strings produces a third, separate string. This third string is represented by a natural variable into which the first two strings are copied in succession. Similarly, value assignment as a result of pattern matching produces substrings. These substrings are copied into separate natural variables and, hence, have no relation to the string from which they were obtained. In particular, there is never any sharing of strings by source-language data objects.

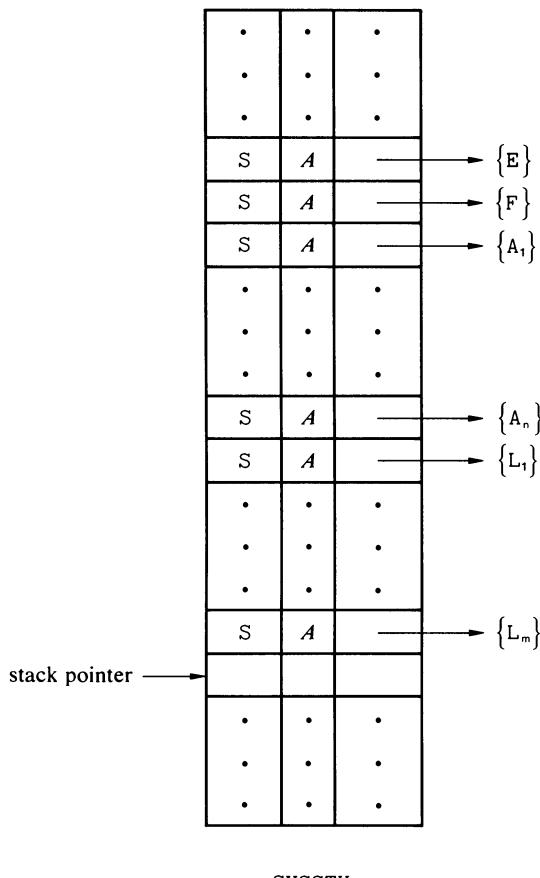
Exercises

- 8.1.1** The number of chains, 256, was selected on the basis of experience. Why is a power of two convenient? What are the advantages and disadvantages of a greater or fewer number of chains?
- 8.1.2** Write an algorithm for searching a chain of natural variables.
- 8.1.3** The fact that the null string is not a natural variable is a result of language design, not an implementation constraint. Discuss what would be involved in allowing the null string to be a natural variable.
- 8.1.4** One of the most significant decisions in the implementation of SNOBOL4 was to have all strings represented by natural variables. Discuss the implications of this decision.
- 8.1.5** Discuss alternatives to representing all strings as natural variables.
- 8.1.6** During program initialization, natural variables are created for function names, keywords, and so forth. Why not build these natural variables into the resident data region instead?
- 8.1.7** What properties do natural variables have that other variables do not?

8.2 DEFINED FUNCTIONS

8.2.1 Definition

The definition of a function involves analysis of the function prototype, updating FNCPBL to reflect the new definition, and creation of a DEFINE block containing information describing the function. Execution of `DEFINE(P, E)` proceeds as follows. Suppose `P` is of the form `F(A1, ..., An)L1, ..., Lm`.



SYSSTK

Figure 8.2.1
Result of analyzing a function prototype.

- (1) P is analyzed to get the function name F.
- (2) FNCPBL is searched to locate a pointer to the link descriptor pair for F. If F is not found among the B descriptors in FNCPBL, it is added and a link descriptor pair created. (If a call to F appears in the source program, the translator will have added F to FNCPBL and created the link descriptor pair for it. See Section 7.3.2.)
- (3) If the entry point name E is not null, it is pushed onto SYSSTK. Otherwise, the default entry F is pushed.
- (4) The function name F is pushed onto SYSSTK.
- (5) Analysis of P continues, identifying the arguments A₁, ..., A_n, pushing them onto SYSSTK in order.
- (6) When the arguments are exhausted, analysis of P continues for locals L₁, ..., L_m, which are also pushed onto SYSSTK in order. Figure 8.2.1 illustrates SYSSTK after the analysis of P.

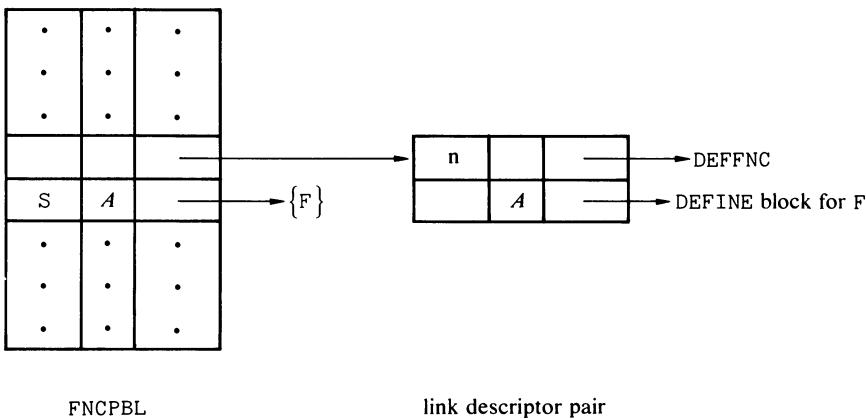


Figure 8.2.2
Result of defining F.

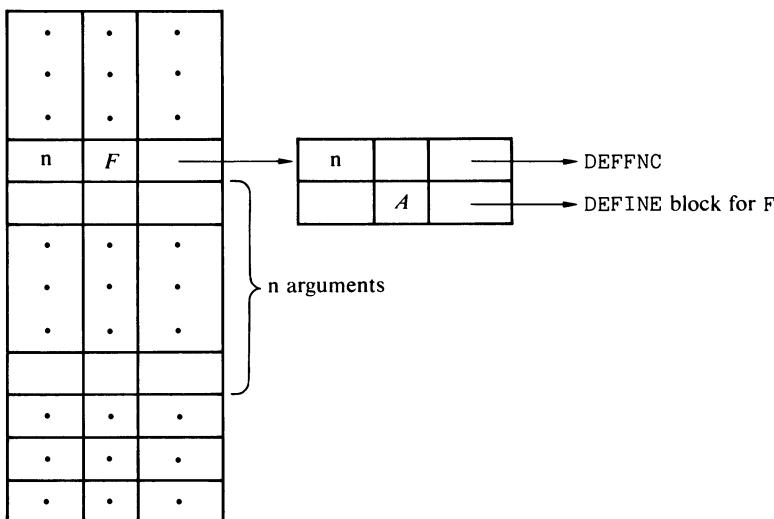


Figure 8.2.3
A call of F.

(7) A DEFINE block of $k = n + m + 2$ descriptors is allocated, and the values saved on SYSSTK are copied into this block. The DEFINE block has the same structure as illustrated in Figure 8.2.1.

(8) The link descriptor for F is set up with a T field of n (corresponding to the n arguments) and a pointer to DEFFNC, the procedure that handles all defined function calls.

(9) The definition descriptor, which follows the link descriptor, is set up to point to the DEFINE block for F .

Figure 8.2.2 illustrates the resulting connections. Figure 8.2.3 illustrates how a call to F appears in the prefix code. Note that the prefix code for such a call is identical in form to the prefix code for the call of a built-in function. Although definition descriptors are not required for built-in functions, they are provided so that built-in functions can be redefined using the method described above.

8.2.2 Execution

The execution of a defined function involves evaluation of the arguments given in the call, saving of the current values of the formal arguments and local variables, passing of new values, and setting locals to null. Necessary state information must then be saved and control passed to the first statement of the function. When the function returns, values are restored and the appropriate value returned for the function call. These operations are carried out by DEFFNC, which handles all calls of programmer-defined functions. The details follow.

- (1) A null string is pushed onto SYSSTK, corresponding to the value assigned to the function name F when the function is called.
- (2) The arguments of the call are evaluated and pushed onto SYSSTK in order. If too few arguments are given in the call, null strings are pushed for the omitted trailing arguments. If too many arguments are provided, the extra arguments are evaluated but ignored. Figure 8.2.4 illustrates SYSSTK after evaluating the arguments.

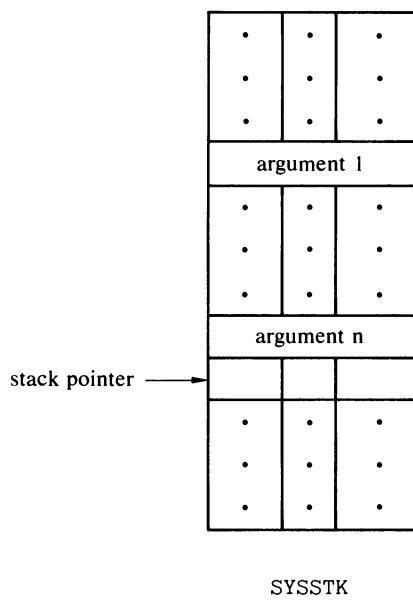


Figure 8.2.4
SYSSTK after evaluating arguments.

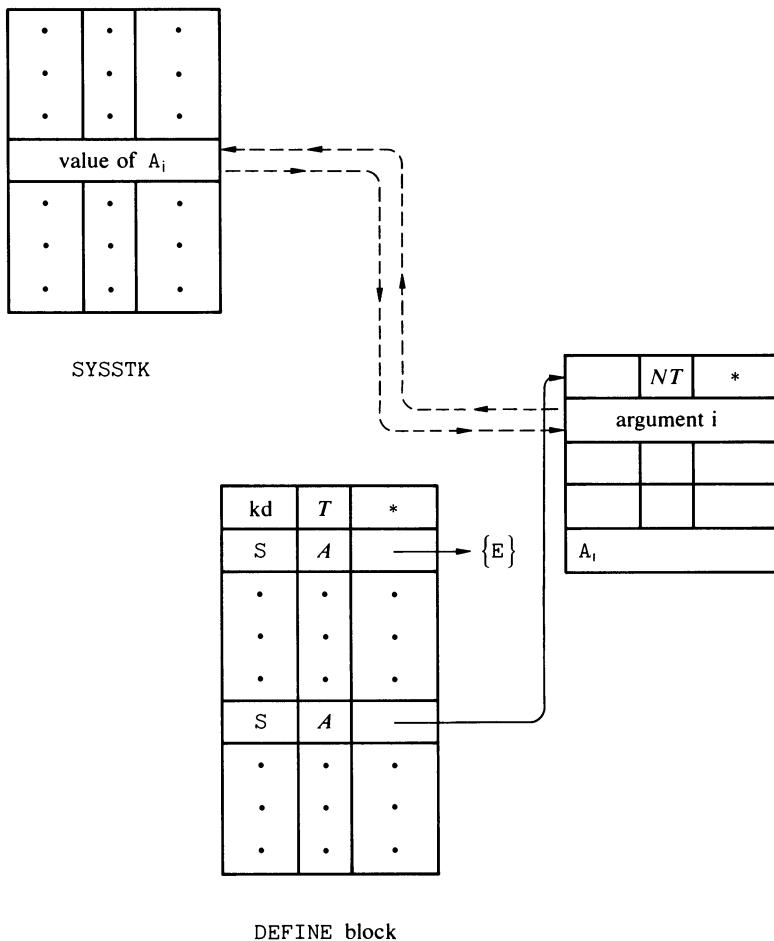


Figure 8.2.5
Passing the value of A_i .

(3) Using the DEFINE block for F, the values on SYSSTK are exchanged with the current values of F, A_1, \dots, A_n . Figure 8.2.5 illustrates the result of the exchange for A_i . In this way the values of F, A_1, \dots, A_n are preserved on SYSSTK during the call of F so that they can be restored when the function returns.

(4) Using the DEFINE block, the current values of L_1, \dots, L_m are pushed onto SYSSTK, and null strings are assigned to L_1, \dots, L_m .

(5) The state of the SNOBOL4 system is pushed onto SYSSTK. State information consists of various descriptors, including the code base pointer and offset, the location of the DEFINE block for F, and so on.

(6) A new code base pointer and offset are established corresponding to the entry point specified by E as given in the DEFINE block.

(7) INTERP is called. At this point, execution begins with the first statement in the function.

(8) When a transfer to RETURN, FRETURN, or NRETURN occurs, a corresponding return is signaled to INTERP, which itself returns, restoring control to DEFFNC.

(9) Upon return from INTERP, the system state descriptors previously saved on SYSSTK are restored.

(10) The current value of F, which by convention is the value returned by the call, is saved temporarily.

(11) The former values of $A_1, \dots, A_n, L_1, \dots, L_m$ are restored from SYSSTK in reverse order.

(12) If RETURN was signaled, the value saved in step 10 is returned by value.

(13) If FRETURN was signaled, failure return is signaled.

(14) If NRETURN was signaled, the value saved in step 10 is returned by name.

Exercises

8.2.1 The DEFINE block does not contain the value of n, the number of arguments. How can one tell where the arguments end and the local variables begin?

8.2.2 Describe the DEFINE block resulting from the prototype F().

8.2.3 What limits the number of arguments and locals that may be given in a prototype?

8.2.4 What happens if the name of the function also appears as an argument in the prototype (e.g., F(F))? As a local variable? What happens if the same argument appears more than once?

8.2.5 Why are the values of arguments and local variables restored in reverse order on return from function execution?

8.3 DEFINED DATA TYPES

8.3.1 Representation of Defined Objects

A defined data object is simply a block of descriptors, one for each field. For example, the statement

```
DATA('COMPLEX(R,I)')
```

defines a type COMPLEX with two fields, R and I. Execution of

```
POSITION = COMPLEX(1.1,2.2)
```

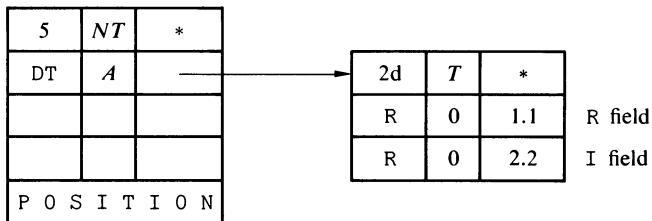


Figure 8.3.1
A COMPLEX object.

results in the structure shown in Figure 8.3.1. COMPLEX creates a block of two descriptors with values 1.1 and 2.2, respectively. The source-language representation of this object is a pointer with some integer data-type code DT.

The following sections describe what happens when the DATA function is executed and how defined objects are created and referenced.

8.3.2 Definition

Data-type definition involves the analysis of the data prototype, definition of the object-creation function, addition of the new data type, creation of a DATA block describing the new type, definition of the field functions, and creation of FIELD blocks for them. Suppose the data prototype P has the form $D(F_1, \dots, F_n)$. Execution of DATA proceeds as follows.

- (1) The prototype P is analyzed to get the data-type name D.
- (2) The link descriptor for D is located. (See Section 8.2.1 for details.)
- (3) A new data-type code DT is generated by incrementing a defined data-type integer.
- (4) The code DT and the name D are added as a pair to DTPBL.
- (5) DT and D are pushed onto SYSSTK.
- (6) Analysis of P continues, identifying the field functions F_1, \dots, F_n . The function names are pushed onto SYSSTK and the corresponding field functions defined. (See the next paragraph for a discussion of field function definition.)
- (7) A DATA block of $k = n + 2$ descriptors is allocated, and the values saved on SYSSTK are copied into the block.
- (8) The link descriptor for D is set up with a T field of n (corresponding to the n fields) and a pointer to DEFDAT, the procedure that handles the creation of all

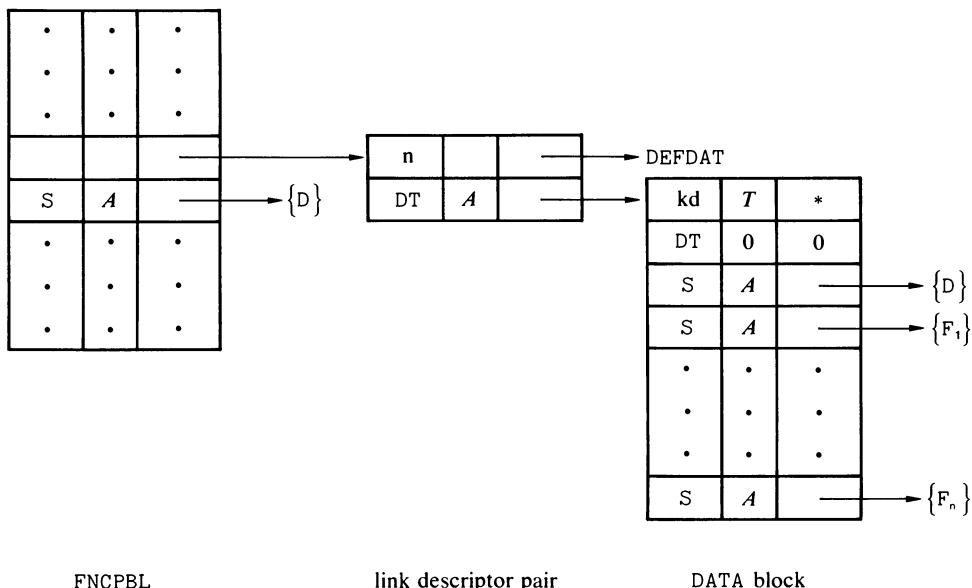


Figure 8.3.2
The result of defining a data type.

defined data objects. The definition descriptor is set up to point to the DATA block. Figure 8.3.2 illustrates the resulting connections.

The function procedure FIELD handles all field references. The definition of F_1, \dots, F_n mentioned in step 6 above requires linking each of these functions to FIELD and setting up FIELD pair blocks describing the field positions for each. For $1 \leq i \leq n$, the procedure is as follows.

- (1) The link descriptor for F_i is located, or created if it does not exist.
- (2) If F_i is not linked to FIELD, that link is set up. A two-descriptor FIELD pair block is allocated. The data-type code DT is put in the T field of the first descriptor, and a field offset of $(i - 1)d$ is put in the V field of the second descriptor. The difference of one descriptor takes into account the fact that the location of the variable is one descriptor above its value. A pointer to the FIELD pair block is inserted in the definition descriptor. Figure 8.3.3 illustrates the connection and the FIELD block.
- (3) If F already links to FIELD, F is already a field function defined on other data types. In this case, a block that is two descriptors larger than the current FIELD block is allocated. The old FIELD block is copied at the beginning of the new one, and the last two descriptors are filled in as described in step 2. Figure 8.3.4 illustrates the FIELD block for a field function defined on m different data types DT_1, \dots, DT_m with offsets o_1, \dots, o_m .

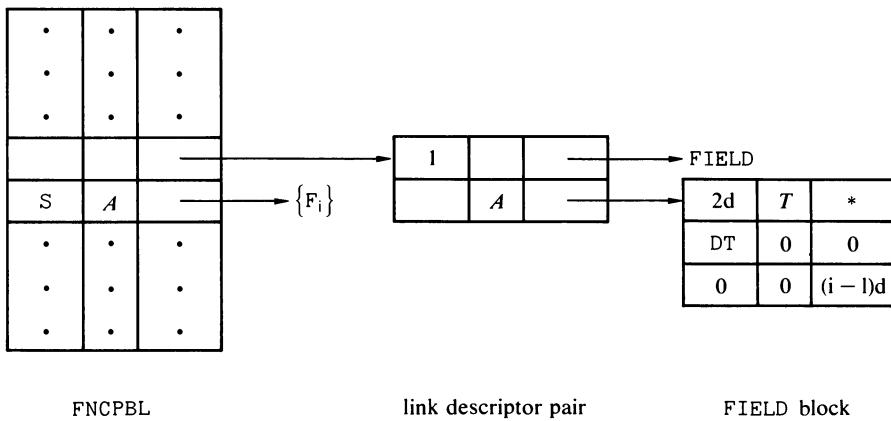


Figure 8.3.3
The result of a field definition.

2md	T	*
DT ₁	0	0
0	0	o ₁
•	•	•
•	•	•
•	•	•
DT _m	0	0
0	0	o _m

Figure 8.3.4
The FIELD block for a field on several types.

8.3.3 Object Creation

When the object-creation function D is called, the function procedure DEFDAT is invoked through the linkages described in the preceding section. DEFDAT proceeds as follows.

- (1) The arguments of the call are evaluated and pushed onto SYSSTK. Missing arguments are added as null strings if necessary. Extra arguments are evaluated but ignored, as in the case for defined functions.
- (2) A block of n descriptors is allocated.
- (3) The arguments pushed onto SYSSTK in step 1 are copied into the block.
- (4) A pointer to the block, with data-type code DT, is returned by value. (Refer to Figure 8.3.1 for an example of the result.)

8.3.4 Field References

When a field reference function is called, the function procedure FIELD is invoked through the linkage described in Section 8.3.2. FIELD proceeds as follows.

- (1) The argument is evaluated, locating a pointer to the defined data object.
- (2) The FIELD pair block is searched for an A descriptor whose T field is the same as the T field of the argument obtained in step 1.
- (3) The V field of the corresponding B descriptor is added to the V field of the pointer obtained in step 1.
- (4) The integer code N for the data type NAME is inserted in the T field of the pointer.
- (5) The resulting descriptor is returned by name.

Referring to the example given in Section 8.3.1, consider the result of the statement

$$I(POSITION) = 3.0$$

Figure 8.3.5 illustrates the value returned as a result of evaluating $I(POSITION)$. Note that the variable $I(POSITION)$ points to the descriptor above the value of that variable. The value of a variable is always one descriptor beyond the location of the variable, whether the variable is a natural variable, a field reference, or whatever. Figure 8.3.6 illustrates the effect of the assignment of 3.0 to this variable.

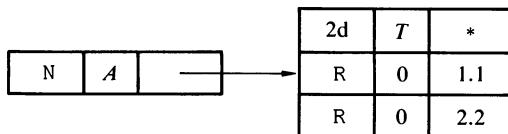


Figure 8.3.5
Evaluation of $I(POSITION)$.

$2d$	T	$*$
R	0	1.1
R	0	3.0

Figure 8.3.6
Result of assignment to $I(POSITION)$.

8.4 ARRAYS

8.4.1 Creation

An array is created by execution of `ARRAY(P, V)`. The prototype `P` has the general form

$$l_1:h_1, l_2:h_2, \dots, l_n:h_n$$

in which $l_i:h_i$ describes the i th dimension, where l_i is the low bound and h_i is the high bound. If the lower bound is omitted, it is assumed to be 1. Thus, the prototypes `10` and `1:10` are equivalent. `ARRAY` analyzes such a prototype and creates a block of descriptors for the array. This size of the i th dimension is

$$s_i = h_i - l_i + 1$$

The total number of elements (variables) in the array is

$$S = \prod_{i=1}^n s_i$$

In addition to the `S` descriptors for the `S` array variables, there are $n + 2$ descriptors of heading information. Figure 8.4.1 illustrates the representation of an

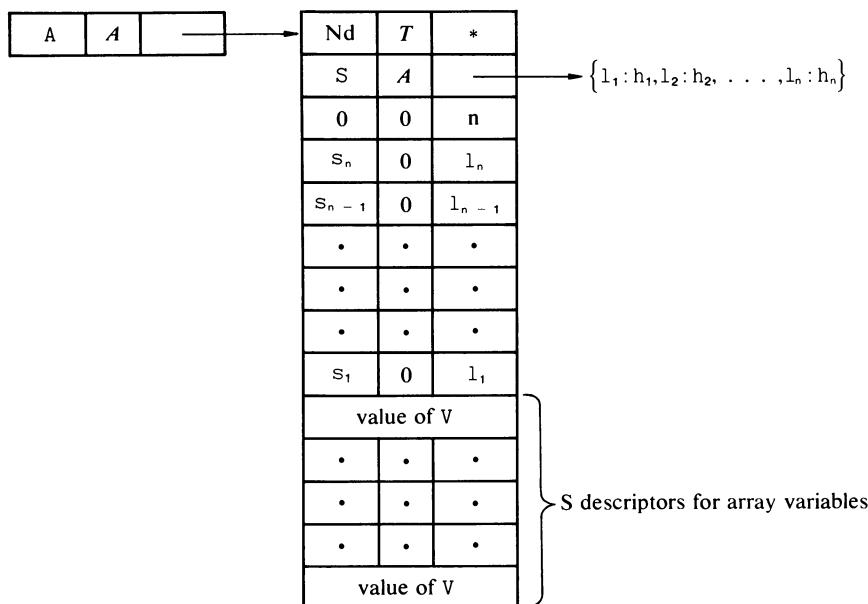


Figure 8.4.1
The structure of an array.

array consisting of $N = S + n + 2$ descriptors. The first descriptor is the prototype of the array, as given in the call of `ARRAY(P, V)`. This prototype is used by the `PROTOTYPE` function and in trace and dump printout. The second descriptor contains the number of dimensions. The next n descriptors describe the n dimensions. They appear in reverse order for ease of computation, as described in the next section. The S descriptors comprising the array variables follow. The value of V is filled in for each, initializing the values of these variables. Note that the `ARRAY` function returns a descriptor pointing to the block.

8.4.2 Access

Suppose M points to an array at some location L , as described in the previous section. Consider an array reference $M(I_1, \dots, I_n)$. The indices are evaluated and pushed onto `SYSSTK`. For each dimension, i , an offset o_i is defined to be $o_i = I_i - l_i$. A pointer, L' , to the desired (I_1, \dots, I_n) th variable is computed by the following formula:

$$L' = L + n + 2 + (o_1 + s_2(o_2 + s_3(o_3 + \dots + s_n o_n)))$$

Note that the innermost computation is $s_n o_n$, which is convenient since I_n is the last value pushed onto `SYSSTK`, and s_n and l_n are first encountered in the array block. In the process of performing this computation, each index I_i is checked against l_i and s_i to assure that it is within bounds. If an index is out of bounds, failure is signaled.

As an example, consider the following statements:

```
M = ARRAY( '-1:1,2' )
M<0,1> = 3
```

Figure 8.4.2 illustrates the result of executing the first statement. The initial value of all variables is the null string since the second argument of `ARRAY` is omitted. Evaluation of $M<0,1>$ produces the pointer shown in Figure 8.4.3. The change in value as the result of assignment is also shown. Note the `NAME` data type in the pointer to the variable $M<0,1>$.

Exercises

8.4.1 The built-in function `PROTOTYPE` returns the prototype of an array.

This prototype could be reconstructed from the dimension information stored in the array. What reasons are there for including a pointer to the prototype as well?

8.4.2 What limits the maximum number of dimensions an array can have?

8.4.3 What happens if a programmer creates a two-dimensional array but refers to it as if it had only one dimension?

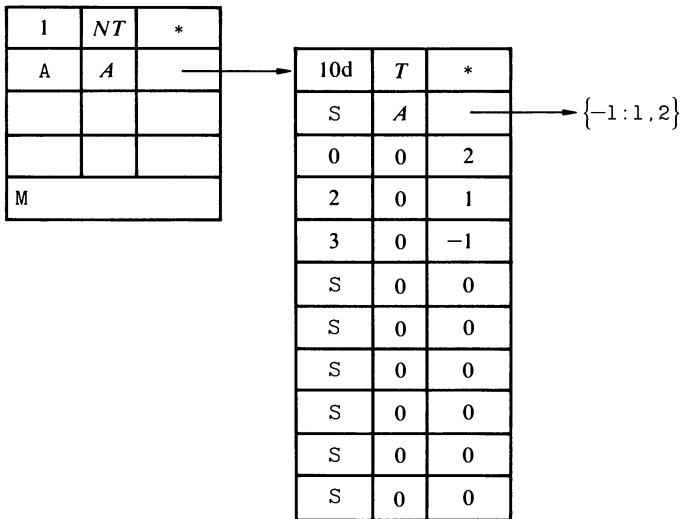


Figure 8.4.2
ARRAY('-1:1,2').

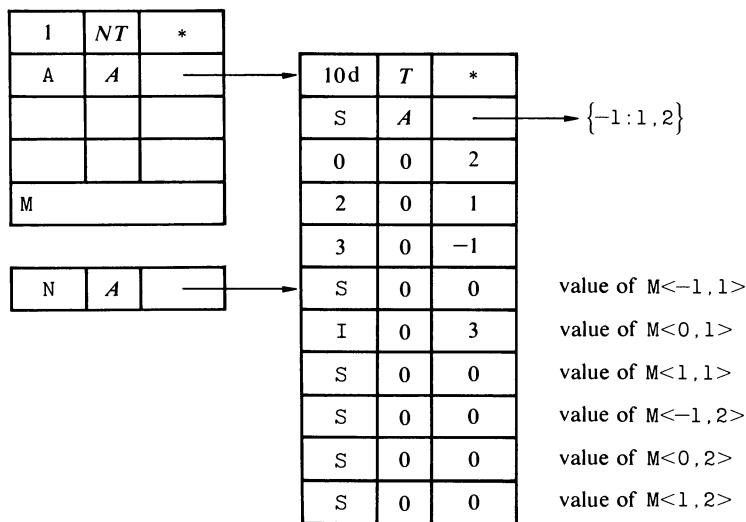


Figure 8.4.3
The array reference M<0,1>.

8.5 TABLES

Tables are implemented as pair blocks in which the values and their indices form A-B pairs. When a table pair block becomes full, another pair block is linked onto the end of it. This process is repeated as often as necessary. TABLE(M, N) creates a table with M pairs in the primary pair block. Additional secondary pair blocks have N pairs. When a pair block is created for a table, the A-B pairs are initialized

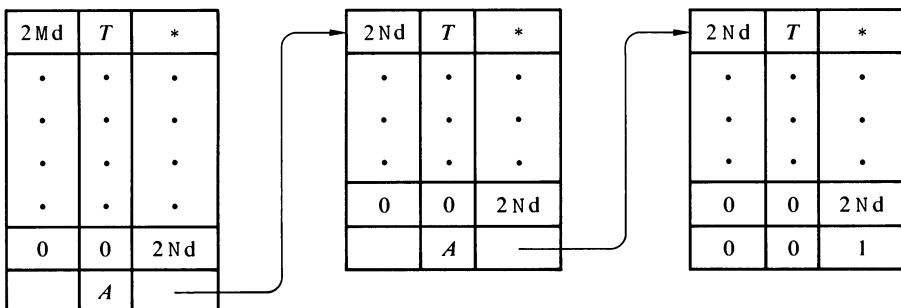


Figure 8.5.1
A table with two extents.

with null strings for the A (value) descriptor and zero for the B (index) descriptor. The zero B descriptor corresponds to an empty pair and cannot be confused with any source-language data object since all data-type codes are greater than zero. Figure 8.5.1 illustrates a table having two secondary extents. Note that the last pair of each block is used for linking to the next. The size of the next extent is stored in the V field of the A descriptor of the last pair. By convention, the V field of the last descriptor of the last block contains a 1 to indicate the end of the chain of pair blocks. The integer 1 is used rather than a 0 to avoid confusion with empty pairs.

When a table is referenced, the B descriptors of the pair blocks are searched for the index descriptor. If the index is found, a pointer to the descriptor above the pair is returned by name.

If the index is not found, a search is made for a zero B descriptor, corresponding to an empty reference. If an empty reference is found, the index is inserted in the B descriptor, and a pointer to the descriptor above the pair is returned by name. Such new references automatically have null values since all A descriptors are initially set to the null string.

If there is no empty descriptor, a new pair block is allocated and linked onto the last pair block. The first pair in this new block is necessarily empty, and it is used as described in the preceding paragraph.

Thus, simply by referring to a table with an index, a table entry is created if it does not already exist. Suppose Q is a table. The result of executing the statement

`Q('MAIN') = 'SERVER'`

is shown in Figure 8.5.2. The result of evaluating the left side of the assignment statement is the descriptor pointing above the pair as shown. The assignment procedure inserts the pointer to SERVER.

Exercises

8.5.1 How can a table reference appearing in a program be distinguished from an array reference?

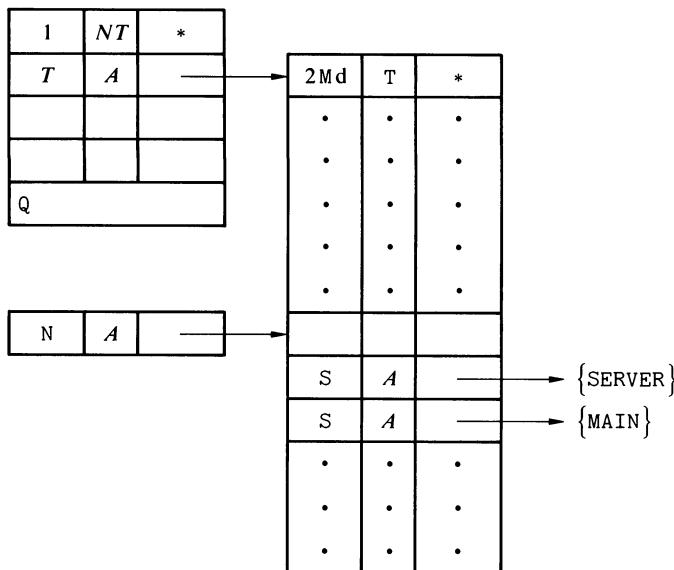


Figure 8.5.2
Assignment to a table reference.

8.5.2 Explain the difference between $X(1)$ and $X('1')$ when X is an array and when X is a table.

8.5.3 What can be said about the order of entries in a table?

8.5.4 Ordinarily, if an empty pair is not available in a pair block, a new, larger pair block is allocated and the old block copied into it. Explain why this would be undesirable for tables.

8.5.5 Zero is used in the V field of many descriptors to indicate the end of a chain or similar condition. Tables use the integer 1 to indicate the end of a pair block. What assumption is made in using such conventions?

8.5.6 One problem that arises in the use of tables is the fact that an entry is created merely by referencing it, whether or not a value is assigned. In some applications this causes tables to become very large, containing many references that are useless. Explain the source of this problem, and discuss ways of avoiding it.

8.5.7 One solution to the problem mentioned in Exercise 8.5.6 is to provide a means of “freezing” a table so that no new entries are created. Explain how this might be accomplished.

8.5.8 The linear structure chosen to implement tables causes the look-up process to be slow for large tables. Discuss better approaches to the implementation of tables.

8.6 KEYWORDS

Keywords are kept in two pair blocks, PKYPBL for protected keywords and UKYPBL for unprotected keywords. For each keyword there is an A-B pair. The A descriptor is the value of the keyword, and the B descriptor points to the name of the keyword. The values of the keywords are at locations known to the SNO-BOL4 system. For example, the value of &STLIMIT is at the location STLVAL, an A descriptor in UKYPBL. Figure 8.6.1 illustrates the structure of UKYPBL.

If a keyword is found in UKYPBL, a pointer to the descriptor above the A-B pair is returned by name. The result of executing

```
&ANCHOR = 1
```

is shown in Figure 8.6.2. The K in the T field stands for the keyword data type. This special data type, which is similar in intent to the NAME data type, is used for identifying keywords in assignments. Only integers can be assigned to keywords, and since the values of keywords are checked internally as well as externally, numeral strings must be converted to integers automatically before assignment to a keyword is made. Thus,

```
&ANCHOR = '1'
```

has the same result as the previous statement, even though the value specified is a string. In the data-type pair block, both K and N data-type codes have the

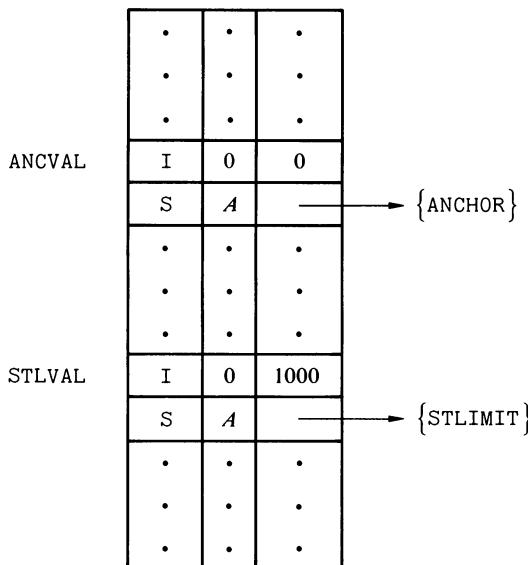


Figure 8.6.1
The structure of UKYPBL.

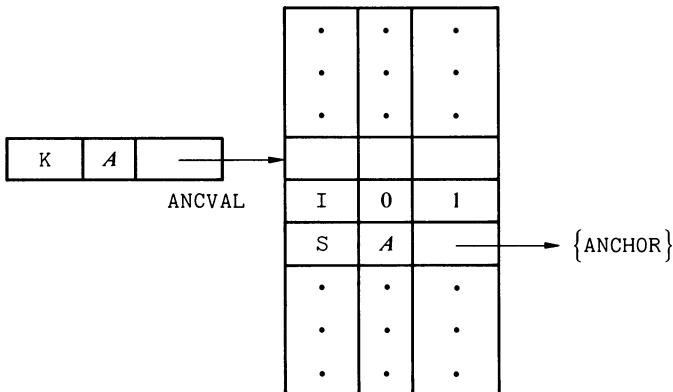


Figure 8.6.2
Assignment to &ANCHOR.

name NAME associated with them. Consequently, they are indistinguishable at the source-language level.

If a keyword is found in PKYPBL, the A descriptor is returned by value. Since only a value is signaled, the value of a protected keyword cannot be changed by a source-language operation.

8.7 PATTERNS AND PATTERN MATCHING

Pattern matching is a large, complicated subject, whether discussed as a source-language feature or as an implementation consideration. Pattern matching remains something of a mystery to many programmers, and the way in which it is implemented is usually not as obvious to the experienced person as the implementation techniques used for other features. Pattern matching is a very high-level facility and constitutes a language of its own within SNOBOL4.

Part of the apparent complexity of pattern matching is a result of the large number of built-in functions and operators related to the pattern-matching process. Further intricacy results from special features, such as variables associated with pattern components, unevaluated expressions, and heuristics. Underlying pattern matching, there is an essentially simple structure whose implementation is reasonably straightforward.

The SNOBOL4 programmer often fails to grasp a very important aspect of patterns and pattern matching: the distinction between pattern construction and pattern matching. While the programmer can get along with only a hazy feeling for this distinction, understanding the implementation requires a thorough appreciation of the two processes and how they relate to each other.

Patterns are data objects. Some patterns, such as the value of ARB, are built into the SNOBOL4 system and are part of the resident data region. Others are

constructed during execution as a result of the various pattern-constructing operations. For example, in the statement

```
BIT = '0' | '1'
```

the alternation operator creates a pattern which is assigned to the variable BIT. Similarly, in

```
TERT = LEN(3)
```

the built-in function LEN constructs a pattern and assigns it to TERT. On the other hand,

```
STRING BIT
```

is a pattern-matching statement for examining the subject STRING for the pattern which is the value of BIT. SNOBOL4 permits patterns to be constructed in the pattern field of a rule. Thus,

```
STRING '0' | '1'
```

first constructs a pattern which is the alternation of two strings and *then* performs the match. The pattern constructed by this alternation is the same as the one constructed above in the assignment to BIT. The process of pattern construction and the process of pattern matching are as distinct as if two statements were executed. A pattern constructed in the pattern field is transient since it is not assigned to any variable and is used only during the execution of the statement in which it is constructed.

8.7.1 Pattern Construction

As implemented in SNOBOL4, patterns are very similar to prefix code. Constructing a pattern is equivalent to translation, and pattern matching is performed interpretively. The procedure that interprets patterns is different from the procedure that interprets statements, but they are functionally similar. Patterns contain function descriptors which link to matching procedures that perform the actual matching.

Alternation and concatenation of patterns produce structural relationships between pattern components. These relationships govern the order in which components are matched, depending on the success or failure in matching individual components. The analog in program interpretation is the order of execution depending on success or failure of statement evaluation. In Section 2.8.1 the relationships between pattern components are shown in diagrams in which the alternate-subsequent structure is explicated. Each component of a pattern corresponds to a pattern-matching operation and consists of a function and its arguments. The structure of a pattern component is shown in Figure 8.7.1.

The function descriptor is identical in use to the function descriptor in prefix code. The connector descriptor is always the first argument and contains offsets

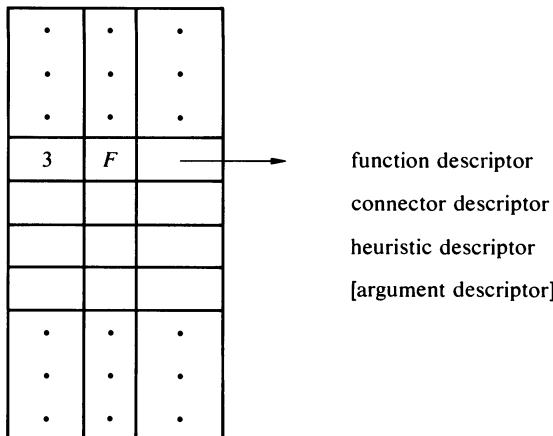


Figure 8.7.1
Structure of a pattern component.

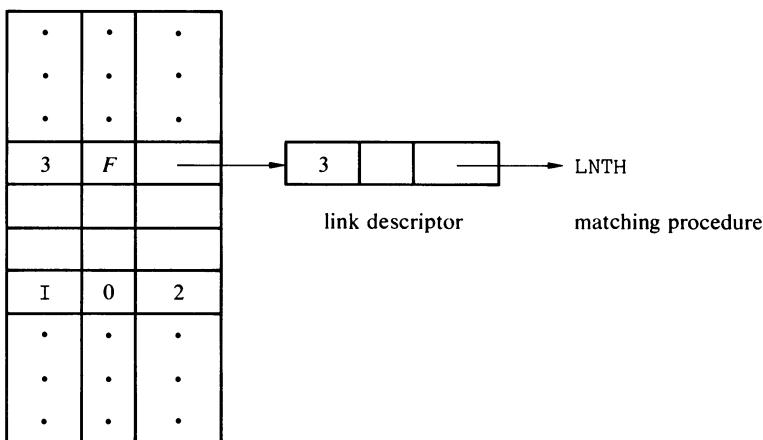


Figure 8.7.2
The pattern component for $\text{LEN}(2)$.

to the alternate and subsequent of the component. The heuristic descriptor contains information which may be used to speed up the matching process. Further discussion of heuristics is deferred to Section 8.7.7. An argument descriptor is provided if required by the matching procedure for the component. If the argument descriptor is not required, the T field of the function descriptor is 2, not 3. Figure 8.7.2 illustrates the pattern component for $\text{LEN}(2)$. This component illustrates an important aspect of the relationship between built-in functions that construct patterns and the matching procedures. LEN constructs a pattern component as shown in Figure 8.7.2 and inserts a function descriptor which points

(indirectly) to a matching procedure LNTH that is used when that component is interpreted during pattern matching. Each pattern-constructing operation has a corresponding matching procedure used during matching. To emphasize this pair relationship and to simplify notation, matching procedures are indicated by prefixing the letter m to the name of the pattern-constructing operation (in lower case if it is a function). The following list illustrates the notation for some pattern-constructing procedures and their corresponding matching procedures.

<i>function procedure</i>	<i>matching procedure</i>
len ₁	mlen ₃
tab ₁	mtab ₃
arbno ₁	marbno ₃
\$ ₂	m\$ ₃
@ ₁	m@ ₃

Thus, LEN(2) invokes len₁, which constructs a component whose prefix representation is

mlen₃c h 2

Where c and h indicate the connector and heuristic arguments. Figure 8.7.3 illustrates this convention in the pattern component for POS(0). Notation for other types of matching procedures is introduced as necessary.

The process of constructing a pattern component is straightforward and similar for all pattern-constructing operations. The appropriate function descriptor is inserted in a block. The argument (if any) of the pattern-constructing operation is evaluated and its data type checked, with data-type conversion being performed as required. The resulting argument descriptor is inserted in the last descriptor of the component.

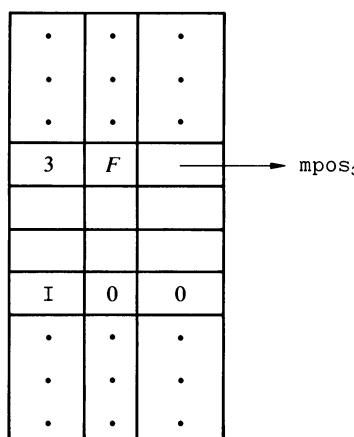


Figure 8.7.3
The pattern component for POS(0).

The remainder of pattern construction involves combining components according to the alternate-subsequent structure in which the components appear. This combining operation is done by the operations of alternation and concatenation.

When two components are combined as alternates (by the alternation operator) or as subsequents (by the concatenation operator), copies of the components are made in a larger block, and an offset in the connector descriptor of the first component is used to indicate the position of the second component relative to the head of the pattern. The V field of the connector descriptor gives the offset of the alternate, and the T field gives the offset of the subsequent. A zero field in a connector descriptor indicates the absence of an alternate or a subsequent. Figure 8.7.4 illustrates the pattern for

$\text{TAB}(4) \mid \text{LEN}(2)$

Note that the offset indicates the descriptor above the alternate component. This kind of offset is the same as a code offset and allows for the fact that the offset is incremented before a descriptor is fetched for interpretation. Figure 8.7.5 illustrates the pattern for

$\text{POS}(0) (\text{TAB}(4) \mid \text{LEN}(2))$

Three points should be observed:

- (1) Alternation and concatenation combine existing pattern components but do not create new ones.
- (2) Patterns are copied and combined by alternation and concatenation to form larger patterns. Since a pattern may be used in the construction of more than one larger pattern, it cannot simply be destroyed when it is used.
- (3) Connector descriptors are adjusted to reflect offsets from the title of the new block.

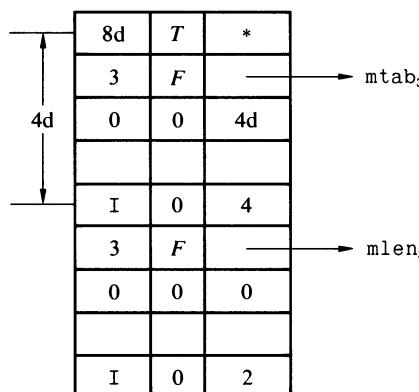


Figure 8.7.4
The pattern for $\text{TAB}(4) \mid \text{LEN}(2)$.

12d	T	*	
3	F		→ $mpos_3$
4d	0	0	
I	0	0	
3	F		→ $mtab_3$
0	0	8d	
I	0	4	
3	F		→ $mlen_3$
0	0	0	
I	0	2	

Figure 8.7.5
The pattern for $\text{POS}(0) \ (\text{TAB}(4)|\text{LEN}(2))$.

8.7.3 The Pattern-Matching Algorithm

The two function procedures match_2 and replace_3 (see Section 7.3.4) perform pattern matching by calling SCNR. SCNR controls the pattern-matching process as described in Section 2.8 and interprets the patterns described above. SCNR has two arguments: the subject string and the pattern. A pattern base descriptor and a pattern offset descriptor are used to determine the current location in the pattern. Matching procedures are executed as a result. The basic approach is to attempt to match subsequents, advancing the cursor when a match is successful. There are three possible signals from a matching procedure: match success, match failure, and match abort (from ABORT). When a match fails, an alternate is attempted. The connector descriptors are used to set the pattern offset for subsequents and alternates. When no alternate exists, the scanner backs up, seeking an alternate for a previously matched component and restoring the cursor to its position prior to that match.

This process requires a separate pattern stack, PATSTK, on which information about successful matches is kept in case the scanner has to back up. Connector and cursor position pairs are saved on PATSTK.

The essence of the pattern-matching algorithm follows.

- (1) Initialize PATSTK by pushing two zero descriptors.
- (2) Set up the pattern base to point to the title of the pattern and zero the pattern offset. Set the cursor position to zero.
- (3) If the connector descriptor of the current component has a nonzero alternate, push the connector descriptor and the cursor position onto PATSTK.

(4) Transfer control to the matching procedure. If match success is signaled, go to step 5. If match failure is signaled, go to step 6. If match abort is signaled, exit from SCNR, signaling failure.

(5) If the current component has a subsequent, set the pattern offset to that subsequent and go to step 3. Otherwise, pattern matching is complete, and success is signaled by SCNR.

(6) Pop the last connector descriptor and cursor position descriptor from PATSTK, and set the code offset from the alternate. If the alternate is nonzero, go to step 3. Otherwise, pattern matching fails, and SCNR signals failure.

SCNR is actually somewhat more complicated than indicated by the algorithm above. Some pattern features and the heuristics require special handling. For example, in the unanchored mode, if the pattern fails to match with an initial cursor position of zero, the cursor position is advanced one character, the process is repeated, and so on.

8.7.4 Matching Procedures

Matching procedures perform the actual matching and advance the cursor position. There are many matching procedures. Some correspond to built-in patterns, such as ARB and BAL, while others correspond to patterns constructed by built-in functions such as LEN and ARBNO. Details of the matching procedures depend on the matching involved. For example, `mlen3` first checks the remaining length of the subject string against its argument. If enough characters remain, the cursor is incremented by the integer value of the argument, and match success is signaled. If there are not enough characters remaining, match failure is signaled. `mtab3` and `mrtab3` move the cursor in a similar manner. `mpos3` and `mrpos3` merely check the cursor position against their arguments, signaling match success or failure as appropriate.

The four matching procedures `many3`, `mbreak3`, `mnotany3`, and `mspan3` modify the syntax table MATCH. MATCH is then used to perform pattern matching in the same way the translator uses syntax tables to analyze the input stream. For example, `BREAK(':, ')` sets up MATCH as if it had the definition

```
BEGIN MATCH
  FOR(':,') STOPSH
  ELSE CONTIN
END MATCH
```

while `ANY('AEIOU')` sets up MATCH as if it had the definition

```
BEGIN MATCH
  FOR('AEIOU') STOP
  ELSE ERROR
END MATCH
```

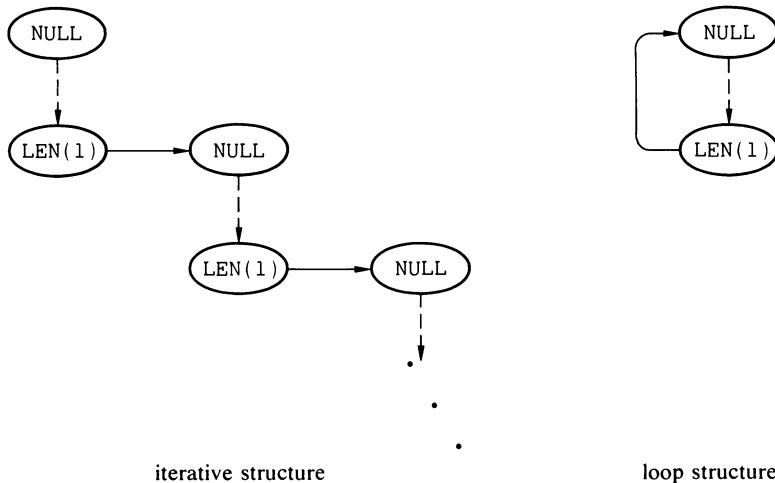


Figure 8.7.6
Hypothetical structure of ARB.

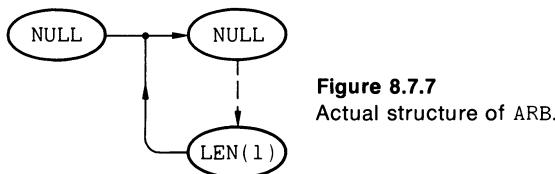


Figure 8.7.7
Actual structure of ARB.

If STOP or STOPSH terminates the streaming process, the cursor position is advanced as indicated, and match success is signaled. If ERROR stops the streaming process, match failure is signaled. If the subject string is exhausted without a stop signal, the result depends on the matching process. `mbreak3` signals match failure, while `mspan3` sets the cursor position past the end of the string and signals match success.

The matching procedure for ARB illustrates a case in which there is an implicit alternative in case the scanner backs up. ARB initially matches the null string. If an alternate match is necessary, ARB advances the cursor one character. Thus, ARB could be described as

$$\text{ARB} = \text{NULL} \mid \text{LEN}(1) \ * \text{ARB}$$

A recursive definition is not necessary. A possible iterative structure and equivalent pattern loop are illustrated in Figure 8.7.6. However, ARB itself might have an alternate, as in the pattern

$$P = (\text{ARB} \mid \text{'DOG'}) \text{ 'COAT'}$$

The structure in Figure 8.7.6 provides no place for the alternate. Therefore, an additional component is required. Figure 8.7.7 illustrates the actual structure

9d	T	*	
2	F		→ mnull ₂
3d	0	0	
2	F		→ mnull ₂
0	0	6d	
2	F		→ mfarb ₂
0	0	6d	

Figure 8.7.8
The pattern for ARB.

of ARB, and Figure 8.7.8 illustrates the pattern. The matching procedure `mnull2` simply signals match success. The matching procedure `mfarb2` advances the cursor one character and signals match success. Note that these components do not require a third argument.

8.7.5 Pattern-Matching Assignment

The binary operators `$` and `.` require extra components to bracket the pattern with which they are associated. The immediate assignment operator, `$`, is the simplest. The cursor position is saved before calling the associated matching procedure. If the matching procedure signals match success, the string matched between the original cursor position and the updated cursor position is converted into a natural variable. This value is then assigned to the variable associated with the component that it matched.

Conditional value assignment is more complicated because assignment is not made until the entire pattern match succeeds. There is a conditional assignment list on which a qualifier and its associated variable are saved for each successful match of a component that has a conditional value assignment. The qualifier gives the substring matched. If the entire pattern match succeeds, the conditional assignment list is then processed, creating natural variables from the qualifiers and assigning the resulting values to the associated variables.

A component with a conditional value assignment may successfully match, but be backed over at a later time because of the subsequent failure of another component. This requires removal of the corresponding entry on the conditional assignment list. Removal is performed by a special alternate for the conditional value assignment matching procedure which is executed in case the scanner has to back up.

The matching procedure for the unary cursor position operator $m@_3$, is quite simple. A descriptor is constructed with the cursor position in the V field and the integer data-type code I in the T field. This descriptor is inserted in the value descriptor of the associated variable. The cursor position is not changed, and match success is signaled.

8.7.6 Unevaluated Expressions

Unevaluated expressions used in pattern matching provide one of the most powerful features of the SNOBOL4 language and yet add comparatively little to the difficulty of the implementation. There are two contexts in which an unevaluated expression may occur in a pattern: as a pattern or as the argument of a pattern-constructor function or operator.

The first context occurs when an unevaluated expression is the entire pattern or appears as an operand of concatenation or alternation. Thus,

$$P1 = *F(X) \text{ LEN}(3)$$

constructs a pattern in which the first component is not evaluated until P1 is used in pattern matching. Figure 8.7.9 shows the pattern for P1.

Any argument of any pattern-constructor function or operator may also be an unevaluated expression. A typical case is

$$P2 = ' (' \text{ LEN}(*N) ') '$$

which constructs a pattern in which the argument of mlen_3 is not evaluated until encountered in pattern matching. A similar instance is

$$P3 = \text{ANY}('0123456789') \$ N \text{ LEN}(*N)$$

8d	T	*	
3	F		→ m^*_3
4d	0	0	
E	A		→ location of call to F(X) in prefix code
3	F		→ mlen_3
0	0	0	
I	0	3	

Figure 8.7.9
The pattern $*F(X) \text{ LEN}(3)$.

in which the value of the argument used by m_{len_3} is set as a result of matching the first component. A more unusual, but possible, case is

$$P4 = TAB(M) \$ *A(I)$$

in which the variable to be assigned to the string matched by $TAB(M)$ is an unevaluated expression, determined during pattern matching by the values of A and I .

As a pattern component, an unevaluated expression merely invokes the matching procedure m^*_3 , which evaluates its argument and matches for the resulting pattern. In this context, the evaluation of an unevaluated expression is much like the call of a programmer-defined function. The state of the system is saved, and control is passed to a procedure that calls `INVOKE` to interpret the prefix code at the location of the expression. Any process may be invoked as a result. This is easy to see by considering the unevaluated expression $*F(X)$, which results in the execution of $F(X)$. In turn, $F(X)$ may execute any section of program. In particular, evaluation of an unevaluated expression may result in pattern matching so that all scanner state information must be saved during the process. In essence, when an unevaluated expression is encountered, pattern matching is suspended. The result of evaluation is a pattern. To match for this pattern, the state of the system is saved, and `SCNR` is called with the remainder of the subject string and the newly created pattern as arguments. If `SCNR` signals failure, $*m_3$ restores the state of the system and signals match failure. If `SCNR` signals success, $*m_3$ restores the state of the system, updates the cursor position to include the string matched by `SCNR`, and signals match success.

When an unevaluated expression is encountered as an argument of a matching procedure, the procedure evaluates the argument and uses the result in matching. The expected data type of the result depends on the procedure involved, and data-type conversion is performed as necessary. Thus, m_{len_3} expects an integer while m_{any_3} expects a string. Again, the state of the system must be saved during evaluation.

8.7.7 Heuristics

One of the most troublesome problems in both the design and implementation of SNOBOL4 has been the subject of the heuristics used to improve the efficiency of pattern matching. The heuristics used in SNOBOL4 are a carryover from SNOBOL3 and are based on the simple observation that length considerations can be used to avoid futile attempts to match. The simplest case is exemplified by the rule

$$'ABC' LEN(4)$$

which obviously cannot succeed. Considered in more generality, as pattern matching progresses and the cursor advances, the remaining length of the sub-

ject string that can be matched by subsequent components decreases. If a situation is reached where there are not enough characters left to satisfy any possible path of subsequents, it is futile to continue matching. The fewest number of characters which can be matched by the subsequent of a component is called the residual of that component. In the pattern

$$\text{TAB}(2) (\text{LEN}(3) 'X' | 'AB' \text{ LEN}(1))$$

the residual of TAB(2) is 3, the residual of LEN(3) is 1, and the residual of AB is also 1. It is futile to attempt a match for the subsequent of TAB(2) if the cursor is less than three characters from the end of the subject string. That condition occurs for the pattern above if the subject string is less than five characters long.

Relatively simple information about lengths can be used to avoid futile attempts to match. Pattern matching is inherently subject to combinatorial explosion, particularly in futile contexts. These observations led to the addition of heuristics to the relatively simple algorithm given in Section 8.7.3.

When patterns are constructed, length information is placed in the heuristic descriptors. Each component has associated with it a minimum match length. Some typical minimum match lengths are illustrated by the following table.

<i>component</i>	<i>minimum match length</i>
LEN(5)	5
TAB(2)	0
SPAN('XY')	1
ARBNO(P)	0
BAL	1
ARB	0
LEN(*N)	0

The minimum match length of LEN(*N) is 0 because the value that N has during pattern matching is not known when the pattern is constructed. N may be 0 when pattern matching takes place.

The total length of subject string required, starting at any component, is the sum of the minimum match length of the component and the residual of the component. This total is kept in the V field of the heuristic descriptor, and the residual is kept in the T field. When a component is first constructed, it has a zero residual and a total which is its minimum match length. When a component gets a subsequent (by concatenation), the total of its subsequent is added to its total and residual. When a component gets an alternate (by alternation), its residual becomes the minimum of the totals of the alternate and all of its alternates. Figure 8.7.10 illustrates the structure for the pattern

$$P = (\text{SPAN}('*') | \text{LEN}(2) \text{ SPAN}('+')) (\text{SPAN}('-') | \text{BREAK}('/'))$$

During pattern matching, the length information in the heuristic descriptors is used to avoid futile matches. Failure because of insufficient remaining char-

20d	<i>T</i>	*	
3	<i>F</i>		→ mspan_3
12d	0	4d	
0	0	1	
S	<i>A</i>		→ { * }
3	<i>F</i>		→ mlen_3
8d	0	0	
1	0	3	
I	0	2	
3	<i>F</i>		→ mspan_3
12d	0	0	
0	0	1	
S	<i>A</i>		→ { + }
3	<i>F</i>		→ mspan_3
0	0	16d	
0	0	1	
S	<i>A</i>		→ { - }
3	<i>F</i>		→ mbreak_3
0	0	0	
0	0	0	
S	<i>A</i>		→ { / }

Figure 8.7.10
A pattern showing totals and residuals.

acters in the subject string is a different kind of failure than failure to match because desired characters are not found. The two types of failure are called length failure and match failure, respectively, and have correspondingly different return signals. The two kinds of failures are distinguished because length failure can often be transmitted backward to avoid futile attempts to try alternative matches for previously matched components. This situation is illustrated by an attempt to match DOGSHATEPIGS with the pattern

'DOG' ARB 'CAT'

DOG matches and advances the cursor accordingly. ARB first matches the null string but does not advance the cursor. CAT signals match failure because the characters are not found. On backup, ARB advances the cursor one character, and another attempt is made to match CAT. Again CAT signals match failure.

This process continues until only two characters of the subject string remain. At this point, CAT signals length failure. ARB in turn transmits this condition by signaling length failure since advancing the cursor cannot possibly result in a match for a component to the right which has failed for reasons of length. DOG similarly signals length failure and the entire pattern match fails, avoiding futile attempts at further matching. Most pattern components transmit length failure backward.

Unevaluated expressions introduce the problem of left recursion. Consider the pattern

$$P = *P 'A' | 'B'$$

Intuitively, this pattern should match strings of the form B, BA, BAA, and so forth. However, if no special mechanism is introduced, an attempt to match P results in another attempt to match P, and so on until the recursive plunge is halted by stack overflow. This problem is caused by the fact that the first alternative never fails, and, hence, the second alternative never has a chance to match. This situation can be avoided by using length information. P must match at least one character since both alternatives must match either an A or a B. Consequently, when $*P 'A'$ is encountered, at least two characters are required. If the subject string is simply B, length failure results, and the second alternative successfully matches.

The mechanism for matching unevaluated expressions uses the residual. When the scanner is called recursively in matching for the unevaluated expression, the subject string given to the scanner is shortened by the residual required for the remainder of the pattern. Suppose the subject string is BAAA. The scanner is called with the successive subjects

BAAA
BAA
BA
B

The final subject, B, causes the first alternative of P to fail. The second alternative matches, and on successive returns the As are matched until finally the entire subject string is successfully matched.

A more difficult problem is introduced by the fact that the minimum matching length of a component is not always known when the component is constructed. Thus, in

$$P = *P \text{ LEN}(*N) | 'B'$$

the total length required by the first alternative may be zero. The residual would not shorten the subject string in this case. This problem has been avoided by making the assumption that every unevaluated expression matches at least one character. Thus, no zero residuals result from unevaluated expressions. An unpleasant by-product of this assumption is that the heuristics predict that a pattern such as

$*X *Y *Z$

must match at least three characters. Suppose the value of X is A and that Y and Z have null values. Then,

```
'AB' *X *Y *Z
```

fails although it should, at least intuitively, succeed. A way around this problem is provided by the keyword &FULLSCAN which can be used to bypass all heuristics.

The heuristics were originally introduced in an attempt to improve the efficiency of the scanner. Until immediate value assignment and unevaluated expressions were introduced into the language, the heuristics were transparent to the user in the sense that their presence did not affect the semantics of the language. When immediate value assignment was introduced, the length heuristics became evident because avoiding futile matches means not performing operations that may have noticeable side effects. Unevaluated expressions have further confused the matter to the point where it may be extremely difficult to determine exactly what a complicated pattern will do. In fact, in the final analysis, pattern-matching language features can be described as “whatever it is that the scanner does.” A more exhaustive treatment of pattern matching in SNOBOL4 is given in a paper by Gimpel [20].

Exercises

8.7.1 What is the difference between

```
ANY('0123456789')
ANY('1234567890')
ANY(1234567890)
ANY(0123456789)
```

8.7.2 An alternative to modifying MATCH during execution of many₃, mbreak₃, mnotany₃, and mspan₃ is to construct tables when ANY, BREAK, NOTANY, and SPAN are executed and use these tables as arguments to the matching procedures. Discuss the advantages and disadvantages of this alternative.

8.7.3 Under what circumstances might m@₃ signal failure?

8.7.4 What is the difference between LEN(*N) and *LEN(N)?

8.7.5 What is the result of executing

```
P = *¬ARB
```

What is the result of using P in pattern matching?

8.7.6 What is the result of matching for *.A(I)?

8.7.7 When may the operand of @₁ be a value instead of a variable?

- 8.7.8** Consider a pattern P which involves no recursion. $*P$ matches the same strings as P. If P is a complicated pattern, and is used as a component of another pattern Q, there is an advantage to using $*P$ instead of P in the construction of Q. Explain.

- 8.7.9** BAL does not transmit length failure backwards. Why?

- 8.7.10** Do the patterns

$$P = *P \text{ 'A'} \mid \text{'B'}$$

and

$$Q = \text{'B'} \mid *Q \text{ 'A'}$$

always match the same strings?

- 8.7.11** Show that there are pattern-matching situations in which the heuristics make an arbitrarily large improvement in speed.

- 8.7.12** Experience shows that many programs actually run faster if the heuristics are not used. Explain this behavior.

- 8.7.13** As indicated in the preceding section, heuristics have been a troublesome aspect of the implementation. One of the arguments for retaining the heuristics is based on the combinatorial properties of futile matching, which may cause certain programs to run extremely slowly. Design a language feature that avoids this problem without using heuristics.

8.8 TRANSLATION DURING EXECUTION

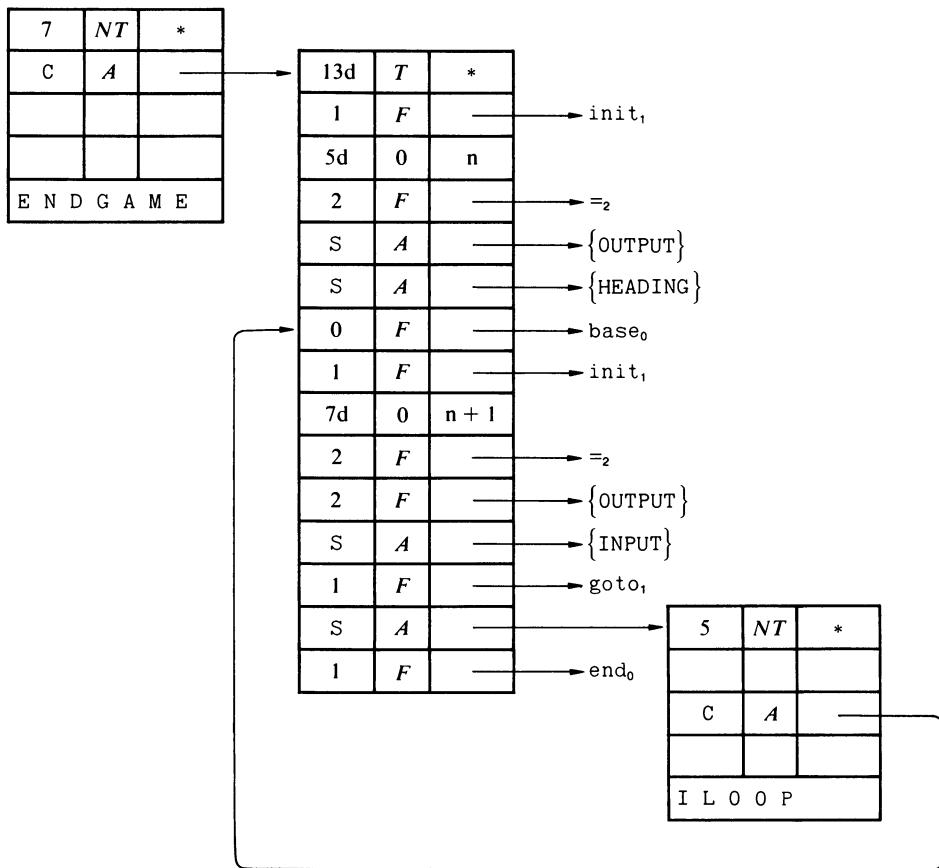
Translation during execution occurs when the argument of EVAL is a string or when a string is converted to CODE. For EVAL, EXPR is called to analyze the argument. The resulting code tree is written into a block of descriptors as prefix code, and INVOKE is called with the code base pointer pointing to this block.

For conversion to CODE, the situation is slightly more complicated. ANALYZ is called to translate the string of statements. The result is a pointer to a block containing the prefix code. Consider execution of the statements

```
ENDIO = ' OUTPUT = HEADING ; ILOOP OUTPUT = INPUT :S(ILOOP)'
ENDGAME = CODE(ENDIO) :<ENDGAME>
```

These statements convert a string into SNOBOL4 statements and execute the result. The result of executing CODE is the same as if the following program had been translated.

```
OUTPUT = HEADING
ILOOP OUTPUT = INPUT :S(ILOOP)
END
```

**Figure 8.8.1**

A result of conversion to CODE.

The value of ENDGAME becomes a pointer to a block of prefix code as illustrated in Figure 8.8.1. The value of ENDGAME is the same kind of pointer that appears in the label descriptor of a natural variable which is used as a label (ILoop, for example).

The direct goto to ENDGAME is a simple matter. The code base pointer is set from the value of ENDGAME, and control is returned to INTERP. The only difference between a label goto and a direct goto is the place from which the code base pointer is obtained.

The last descriptor in the code block points to the end procedure. When the statement labeled ILoop fails, normal program termination occurs just as if the end statement of a program had been encountered. This termination is provided to prevent the interpreter from accidentally flowing off the end of the code block.

The ease with which translation is accomplished during execution is an excellent example of the generality of the SNOBOL4 implementation. Of particular interest is the use of the same data structures for internal use and source-language objects, which makes representation of the CODE data type trivial. This generality was in no way contrived. The representation of prefix code as a block of descriptors existed long before the idea for conversion to CODE was conceived. A facility for translation during execution had long been a goal of SNOBOL language design. Such a feature was planned for the first SNOBOL language but not implemented. It is interesting to note that the existence of prefix code blocks in the implementation suggested the form for a language feature.

8.9 INPUT AND OUTPUT

As noted in Chapter 2, input and output operate through associations made with variables. The two cases are similar. There is an input pair block, INPBL, and an output pair block, OUTPBL. An association consists of an A-B pair in which the B descriptor points to the associated variable and the A descriptor points to a block that contains information about the association. Figure 8.9.1 illustrates the built-in association for INPUT. As illustrated, the first descriptor in the block that is pointed to is the FORTRAN unit number. The second descriptor is the input record length.

Figure 8.9.2 illustrates the built-in association for PUNCH. Note the pointer to the format string.

The INPUT and OUTPUT functions make associations of the kinds illustrated above. They search their respective pair blocks in case there is already an association for the given variable. If an existing association is found, it is altered as

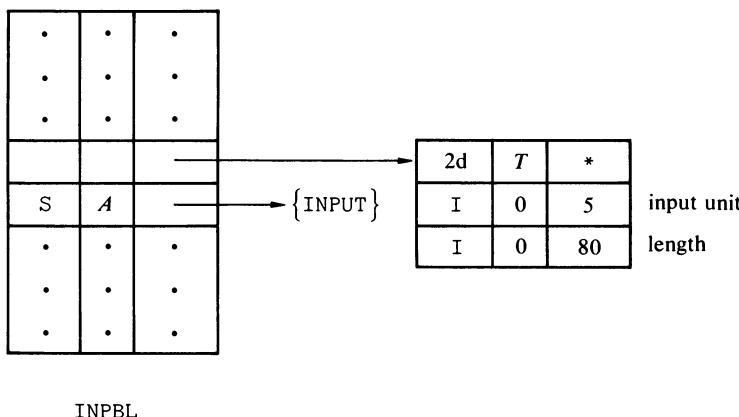
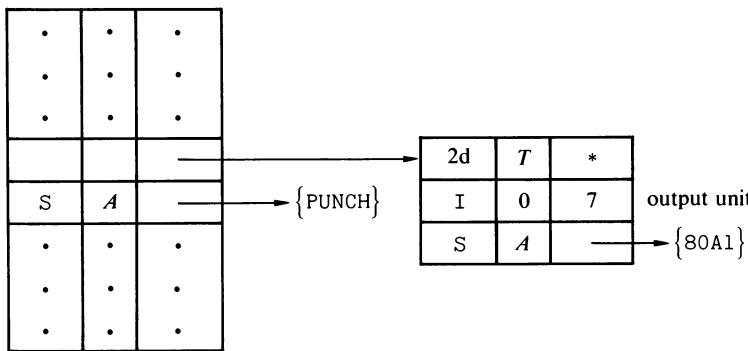


Figure 8.9.1
The association for INPUT.



OUTPBL

Figure 8.9.2
The association for PUNCH.

specified. If an existing association is not found, a new association pair and descriptive information are added. Note that since the B descriptor of the association pair points to the variable, an association can be made for any kind of variable, whether it is a natural variable or some other kind.

Determining when input and output should be performed requires searching the pair blocks described above in appropriate situations. For input, every time the value of a variable is fetched, INPBL is checked to see if that variable has an input association. If it does, a procedure is called to read a record of the specified length from the specified unit. The resulting string is made the value of the input-associated variable, and the value is returned to the fetching process.

Output is similar, except that OUTPBL is checked whenever a value is assigned to a variable. The format specified for an output association is a string. Formats that appear explicitly in FORTRAN source programs are processed by the FORTRAN compiler and converted into a more convenient internal representation. FORTRAN IV permits unprocessed formats in arrays which are usually read into by FORTRAN source programs. The SNOBOL4 format as provided to the output routines is in the unprocessed form.

The use of pair blocks for input and output associations is expensive in terms of running speed. For natural variables, flags stored in the natural variable itself could be used to achieve greater efficiency. However the use of pair blocks permits any type of variable to be input- or output-associated since only the location of the variable appears in the pair block.

Exercises

8.9.1 What would be entailed in replacing FORTRAN unit numbers by file names?

8.9.2 Discuss the advantages and disadvantages of using FORTRAN formats.

- 8.9.3** Describe how flags in natural variables could be used for I/O associations.
- 8.9.4** Explain why it would be impractical to use I/O flags for other types of variables.

8.10 TRACING

Tracing is similar to output in many respects. Value tracing, for example, prints a line whenever a value is assigned to a traced variable. Tracing is complicated by the fact that there are several types of tracing and by the facilities for programmer-defined trace procedures. The variety of trace types is handled by a trace-type pair block, TTPBL, as illustrated in Figure 8.10.1. Thus, each type of trace has its own structure. These structures are all similar. Figure 8.10.2 illustrates the value trace structure. The indirect linkage from the TTPBL to the value trace pair block VTRPBL is followed by a pointer to a link descriptor for valtr₂, the default procedure used for printing value trace messages.

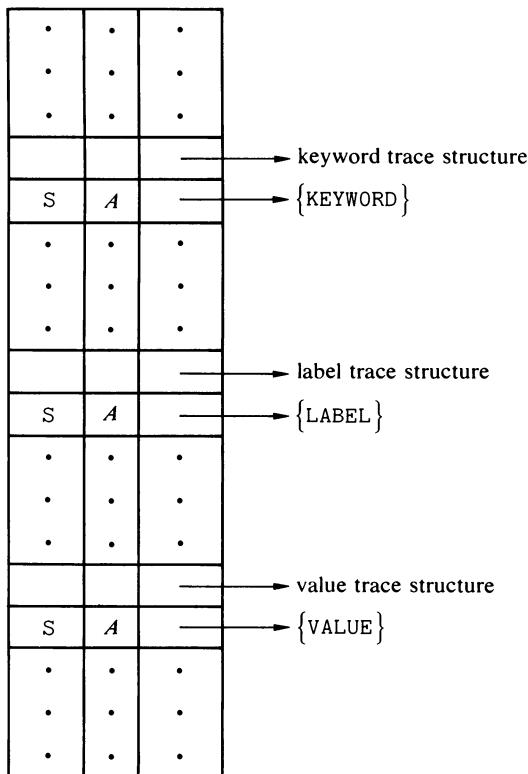


Figure 8.10.1
The structure of TTPBL.

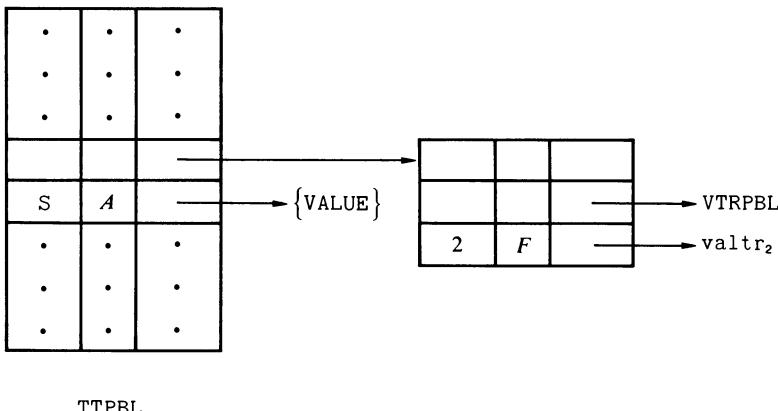


Figure 8.10.2
Value trace structure.

Value trace associations modify VTRPBL. If the statement

```
TRACE( 'SUM' , 'VALUE' )
```

is executed, the second argument of TRACE is compared with the B descriptors in TTPBL. As a result, the value trace structure is located. Figure 8.10.3 illustrates the modification to the VTRPBL and the structure added as a result. The code block is used to print a trace message whenever a value is assigned to SUM. The situations in which this can occur are the same as those for an output association. If a value is assigned to SUM, the code base pointer is set to point to the code block illustrated in Figure 8.10.3. Thus, valtr₂ is a function procedure accessed in the same way that a procedure for a built-in function is accessed. valtr₂ expects two arguments, the variable name and the tag used for print identification. Here the variable name is SUM and the tag is null. The unary name operator, .₁, causes these arguments to be treated as literals.

The reason for this apparently unnecessary subterfuge becomes clear when programmer-defined trace procedures are considered. Suppose the statements

```
DEFINE( 'F(N,T)' )
TRACE( 'SUM' , 'VALUE' , 'TYPE1' , 'F' )
```

are executed. Figure 8.10.4 illustrates the structure that results. The only difference in the structure is that the first descriptor of the code block links to the programmer-defined function F rather than to valtr₂. Thus, using a function procedure for tracing makes it trivial to insert a programmer-defined function in its place. The trick is in constructing a block of code that is executed when tracing occurs, rather than simply calling a procedure. In the example above, when a value is assigned to SUM, F is called with the two arguments SUM and TYPE1.

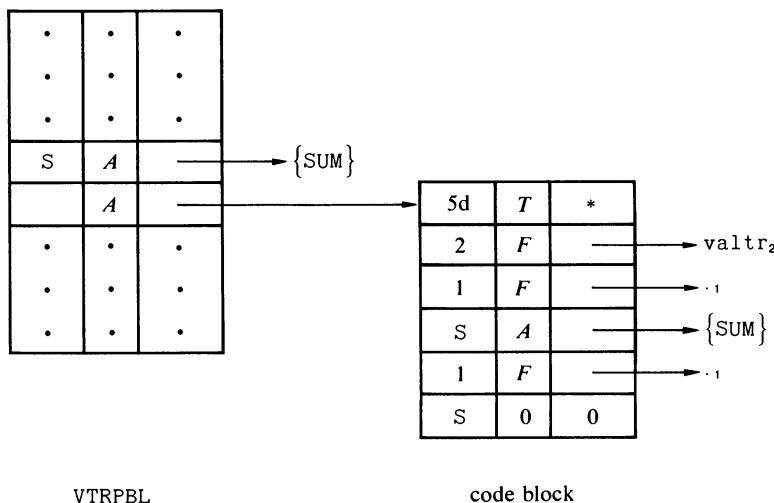


Figure 8.10.3
Structure resulting from a trace association.

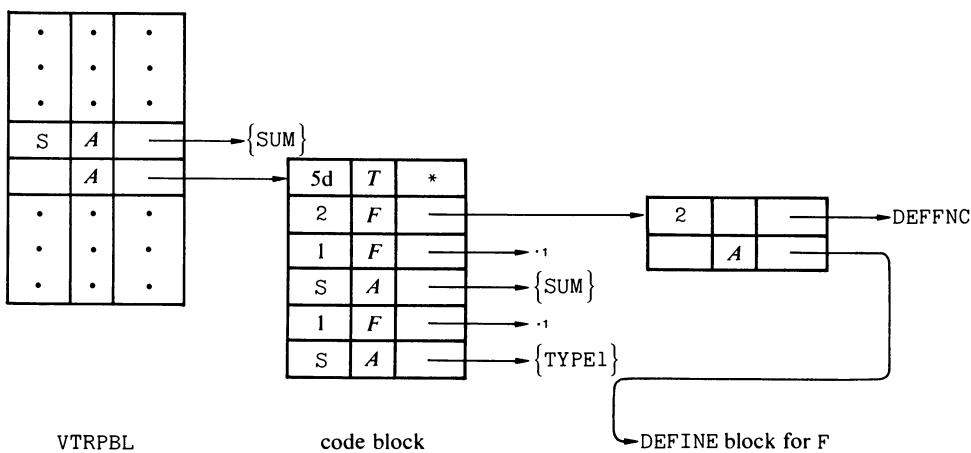


Figure 8.10.4
Structure for a programmer-defined trace procedure.

The other types of tracing are handled in the same way. The difference is in the default trace procedures and in the places where checking is done for traced actions.

Exercise

- 8.10.1** What might happen if a programmer-defined trace procedure has more than two arguments? Explain how to avoid this problem.

8.11 ERRORS AND ERROR HANDLING

Severe errors which prevent continued execution originate partly from language features and partly from the way SNOBOL4 is implemented. Executing more statements than allowed by &STLIMIT is a severe error because &STLIMIT is provided to limit statement execution. On the other hand, overflow of SYSSTK is a severe error because SYSSTK is fixed in size, and no general reasonable way has been devised for continuing execution after overflow.

In the case of a severe error, a diagnostic message is printed, and execution is terminated. Summary information is then provided as in the case of normal termination.

There are many types of errors which prevent the performance of a particular operation but do not prevent continued execution of the program. Three examples are: (1) attempting to perform an arithmetic operation on a nonarithmetic object, (2) referencing a nonexistent keyword, and (3) calling an undefined function. If such a nonsevere error occurs, program execution ordinarily terminates in the same way that it does for severe errors. The value of &ERRLIMIT is checked before terminating, however, and if this value is greater than zero, the error routine signals failure instead of terminating. The effect is as if the function procedure in which the error occurred had signaled failure. Since all function procedures and argument evaluation procedures have the capacity for signaling failure, program execution continues properly.

Besides decrementing a positive value of &ERRLIMIT when a nonsevere error occurs, an integer code is assigned to &ERRTYPE so that the program can examine the type of error. If &ERRTYPE is being traced as a keyword, the state of the SNOBOL4 system, consisting of relevant descriptors and qualifiers, is saved on SYSSTK in much the same manner as is done in the call of a programmer-defined function. When the trace procedure returns, the system state is restored and failure is signaled.

Nonsevere errors that occur during pattern matching require slightly different handling since the return signals from matching procedures are different from those of function procedures. Most nonsevere errors that occur during pattern matching are treated as match failure (corresponding to the built-in pattern FAIL), and match failure is signaled. Overflow of PATSTK, however, prevents continued pattern matching but not the continued operation of the SNOBOL4 system itself. Therefore, overflow of PATSTK is treated as failure of the entire pattern match (corresponding to the built-in pattern ABORT), and match abort is signaled.

Exercises

- 8.11.1 Discuss the effects of returning control to the program even in the case of severe errors.
- 8.11.2 Why must the system state be saved when &ERRTYPE is traced? Discuss what the system state consists of in this case.

8.11.3 A transfer to an undefined label is a severe error. Why not treat such an error as nonsevere, going on to the next statement if &ERRLIMIT is greater than zero?

8.11.4 Suggest a language feature that would permit transfer to an undefined label to be treated as nonsevere.

8.12 OTHER OPERATIONS

Not all of the many operations which go together to make up the SNOBOL4 system can be described within the confines of a book of reasonable size. Four unary operators which are important, yet simple to implement, complete this chapter.

8.12.1 Indirect Referencing

The uniform treatment of all variables in SNOBOL4 makes indirect referencing a fast and trivial operation. The value of any variable, whether it is a natural variable, an array reference, a table reference, a field reference, or an unprotected keyword, is in the descriptor following the location of the variable. The one-descriptor offset is a consequence of the title descriptor at the top of natural variables and blocks. Figure 8.12.1 illustrates this relationship between a variable and its value.

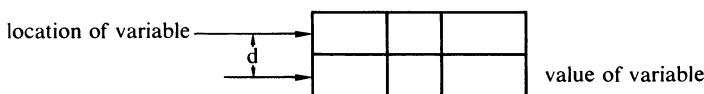


Figure 8.12.1
A variable and its value.

An indirect reference in the source language is essentially an indirect address reference internally. In performing an indirect reference, the value is fetched. Since the result of an indirect reference must itself be a variable, the data type of the value is checked. Three data types are permissible: STRING, NAME, and INTEGER. The first two data types identify a value that can be returned by name directly. If INTEGER is encountered, the integer is converted to a string by constructing a natural variable which is returned by name. This conversion is a consequence of automatic integer-to-string conversion where required by context.

8.12.2 The Unary Name Operator

The unary name operator is the inverse of indirect referencing and is semantically equivalent to a literal. The unary name operator simply fetches the next descriptor of prefix code. If the descriptor is an argument descriptor, the argument

is a variable since prefix code is generated by name. In this case, the argument is returned by value. If the argument is a function descriptor, the function is evaluated by calling INVOKE. If INVOKE returns by name, the name returned is returned by value. If INVOKE returns by value, control is transferred to an error routine since the argument of the unary name operator must be a variable.

8.12.3 The Unevaluated Expression Operator

Unevaluated expressions are usually used in conjunction with pattern matching. The creation of unevaluated expressions by execution of *, is not related to pattern matching, however.

When *, is executed, the current code base and offset are added together to determine the current location in the prefix code. The result is inserted in the V field of a descriptor, and the data-type code E is inserted in the T field. The operand of the unevaluated expression operator is not executed. Rather the code offset is advanced by using the procedure CODSKP to skip over the operand of the unevaluated expression. After advancing the code offset, the descriptor pointing to the operand is returned by value. Figure 8.12.2 illustrates the result of executing the statement

X = *SIZE(A)

SIZE(A) is not immediately executed, but the value returned points to the code to be executed when the expression is subsequently evaluated.

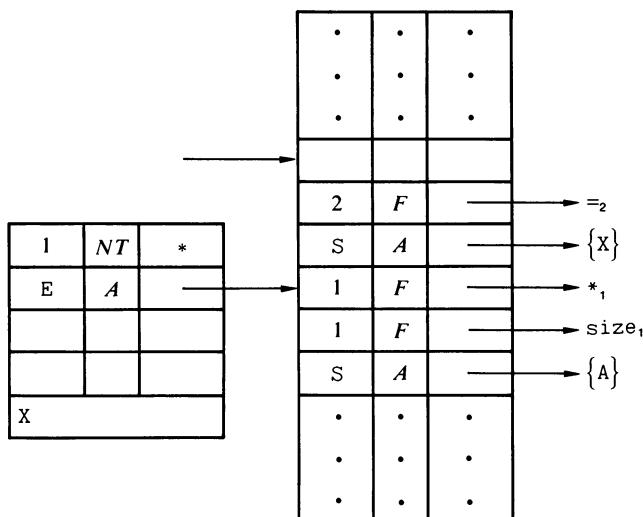


Figure 8.12.2
An unevaluated expression.

8.12.4 Negation

Negation is an infrequently used but interesting operation. The unary negation operator, \neg_1 , saves the current code base and offset and then evaluates its argument. If this evaluation returns by name or value, \neg_1 signals failure. On the other hand, if the evaluation of the argument fails, the original code base and offset are restored. The procedure CODSKP is then called to skip over the argument of \neg_1 . The procedure \neg_1 then returns the null string by value.

Exercises

8.12.1 Discuss the result of executing the statements

```
$2      =      'MARK'  
$'2'    =      'MARK'  
$'2.7'  =      'MARK'
```

8.12.2 What is the result of evaluating the expressions

```
$.$.$.X  
$$X  
.X
```

8.12.3 What is the result of executing the statements

```
$ * X    =      3  
$ \ * X   =      3  
\ $ * X   =      3
```

8.12.4 Discuss the implementation of CODSKP.

8.12.5 Why is it necessary to skip over the argument of \neg_1 if evaluation of its argument fails? Why is it not necessary to skip over the argument if evaluation succeeds?

Storage Management

Storage management is, to a large extent, functionally independent of the rest of the SNOBOL4 system. Storage management procedures are called to allocate blocks of descriptors, to look up strings in the table of natural variables (entering new natural variables as necessary), and to provide space for the formation of strings which may produce new natural variables. Storage regeneration is performed automatically when insufficient space remains to satisfy an allocation request.

9.1 ALLOCATION

From the point of view of the procedures that call storage management procedures, the organization of the allocated data region is largely irrelevant. In fact many different organizations would be possible without modification of the rest of the SNOBOL4 system. The organization of the allocated data region in fact is quite simple, and allocation is a straightforward process. Figure 9.1.1 illustrates the structure of the allocated data region.

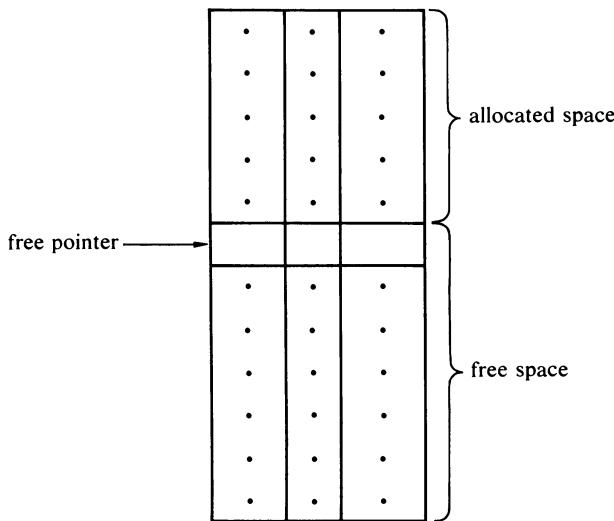


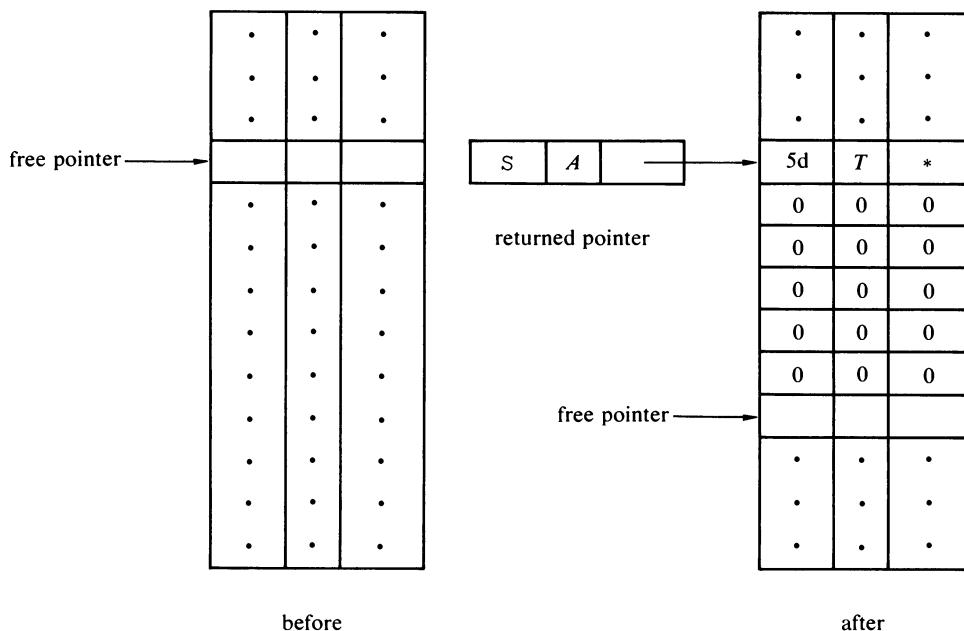
Figure 9.1.1
The allocated data region.

The allocated data region is simply a large block of descriptors. An allocated data region typically contains about 10,000 descriptors. Initially, the free pointer points to the top of the region. When an allocation request is made, the space requested is provided starting at the position of the free pointer, and the free pointer is moved down past the newly allocated space to point to the next available descriptor. Thus, all allocated space is above the free pointer, and all free space is below it.

Two types of objects are allocated: blocks and natural variables. When a block of descriptors is allocated, a title descriptor is provided and the block is zeroed. Figure 9.1.2 illustrates the allocated data region before and after allocation of a block of five descriptors. Note that six descriptors are required because of the title. The value returned to the procedure which requested the block is a pointer to the title (the original value of the free pointer), as shown. The *A* flag in the F field identifies a pointer to the allocated data region.

Allocation of space for a natural variable is done only after it is discovered that the string is not in the table of natural variables and the place on the chain for the new variable has been determined. Figure 9.1.3 illustrates the allocated data region before and after allocation of the natural variable for HUNTER. When a new natural variable is allocated, it is given the null string as its initial value and a zero label descriptor. The new natural variable is linked into the appropriate chain, and its order number, N, is inserted in the chain descriptor, as indicated. The value returned is a pointer to the title.

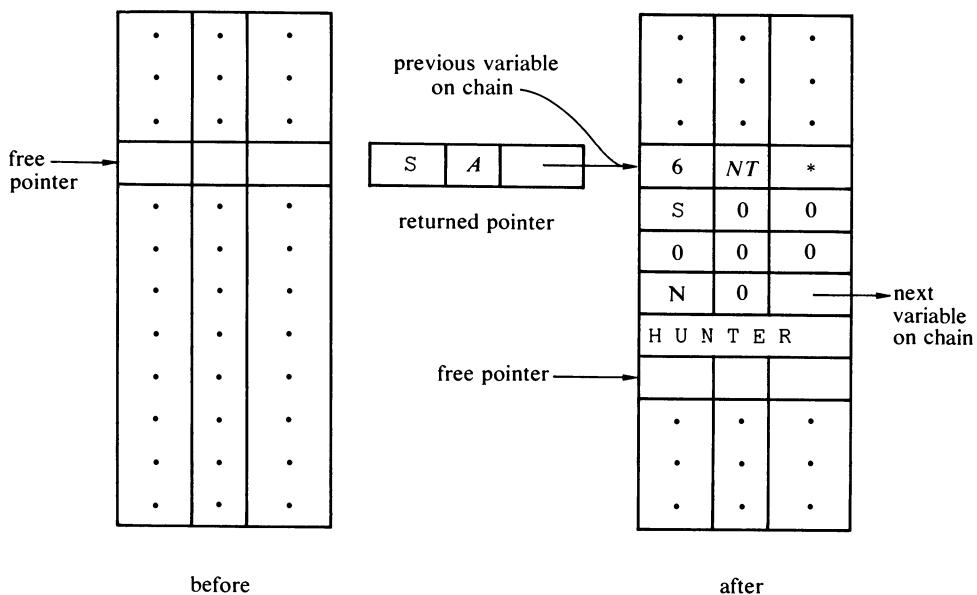
Allocation of blocks and natural variables may occur in any order, depending on the particular SNOBOL4 program that is executed. Blocks and natural variables are physically intermixed in the allocated data region. Both blocks and



before

after

Figure 9.1.2
Allocation of a block of five descriptors.



before

after

Figure 9.1.3
Allocation of the natural variable for HUNTER.

natural variables are allocated in multiples of descriptors, but the two types of structures are differentiated by flags in their titles. Because of the method of allocation, allocated objects appear physically in the same order in which they are created chronologically. The chains of natural variables constitute a logical structure of links within the allocated data region.

Exercises

- 9.1.1 What can be said about the physical order of natural variables on a chain?
- 9.1.2 Of what use is the correspondence of the physical and chronological ordering of the allocated objects?

9.2 STORAGE REGENERATION

Allocation of blocks and natural variables usually continues throughout the course of program execution. It is, therefore, inevitable that, regardless of the size of the allocated data region, it will eventually be exhausted during the execution of some programs. When this happens, there are usually some objects in the allocated data region that are no longer needed. Program execution can continue if the space occupied by these unneeded objects is reclaimed and storage is reorganized in such a way that allocation of new objects can resume.

The situation described above leads logically to a discussion of the techniques used for storage regeneration. In the first place, objects in the allocated data region are divided into two classes: those that must be kept and those that may be deleted. This is accomplished by marking objects that must be kept to distinguish them from the remaining objects. In the second place, in order for allocation to resume after storage regeneration, the objects kept must be above the free pointer and the free space below it. To accomplish this, the marked objects are relocated upward to fill in the space made available by the unmarked objects.

9.2.1 The Marking Process

How can those objects that may be needed after storage regeneration be distinguished from those that will not? Superficially, the criterion is simple: Any object that can possibly be referenced after storage regeneration must be saved. This is begging the question, however.

In fact, the SNOBOL4 system is written so that all objects that can be referenced may be determined in a systematic manner. In Chapter 4 the relationships among data in the resident and allocated data regions were discussed in a general

way. The important point is that objects in the allocated data region are referenced only indirectly through the resident data region. Furthermore, the resident data region is organized so that only certain blocks, called *basic blocks*, contain pointers to the allocated data region. Typical basic blocks are SYSSTK, DTPBL, PKYPBL, and UKYPBL. There is a list of all basic blocks. Figure 9.2.1 illustrates the relationships schematically.

All objects that can be referenced can be determined by starting with the list of basic blocks. Furthermore, pointers to the objects in the allocated data region are distinguishable by *A* flags in their F fields. The algorithm for marking is based on the following recursive definition.

- (1) Any object in the allocated data region that is pointed to from a basic block is marked.
- (2) Any object in the allocated data region that is pointed to from a marked block is marked.

The second statement takes care of references that could result from pointers from one allocated object to another. Marking an object consists of adding an *M* flag to the F field of the object's title. Marking is done by a recursive procedure that examines objects for pointers to allocated objects, starting with the basic blocks. If an *A* flag is detected, the title of the object it points to is examined. This requires locating the title since not all pointers into the allocated data region point to titles. If the descriptor pointed to does not have a *T* flag, the title is the first descriptor above the descriptor pointed to which has a *T* flag in its F field. There are two possible situations: (1) the title is not marked (does not have an *M* flag in its F field), and (2) the title is already marked.

If the title is not marked, it is marked by adding an *M* flag. The marking process then continues recursively, suspending examination of the current object and starting on the newly located object. If the title is already marked, the object has been (or is being) examined. Therefore, a marked block is ignored.

The following example illustrates the marking process for an extremely simple situation in which there is only one basic block and only six blocks in the allocated data region. This example is contrived for the purpose of illustration only. Real storage configurations are much more complicated. Figure 9.2.2 illustrates the situation before marking begins.

The marking procedure, MARK, starts with pointer A from the basic block into the allocated data region. Since block 2 has not been marked, it is now marked. MARK pushes the position in the basic block on SYSSTK and begins examination of block 2. The first descriptor in block 2 is the pointer B to block 1. Block 1 is marked, and again MARK pushes the current position and begins examination of block 1. The first descriptor in block 1 is the pointer C to block 2. Block 2 is already marked, however, so pointer C initiates no new processing. Note that processing of block 2 is in suspension because of the processing of block 1. The next descriptor in block 1 is the pointer D to block 1 itself. Block 1 has been

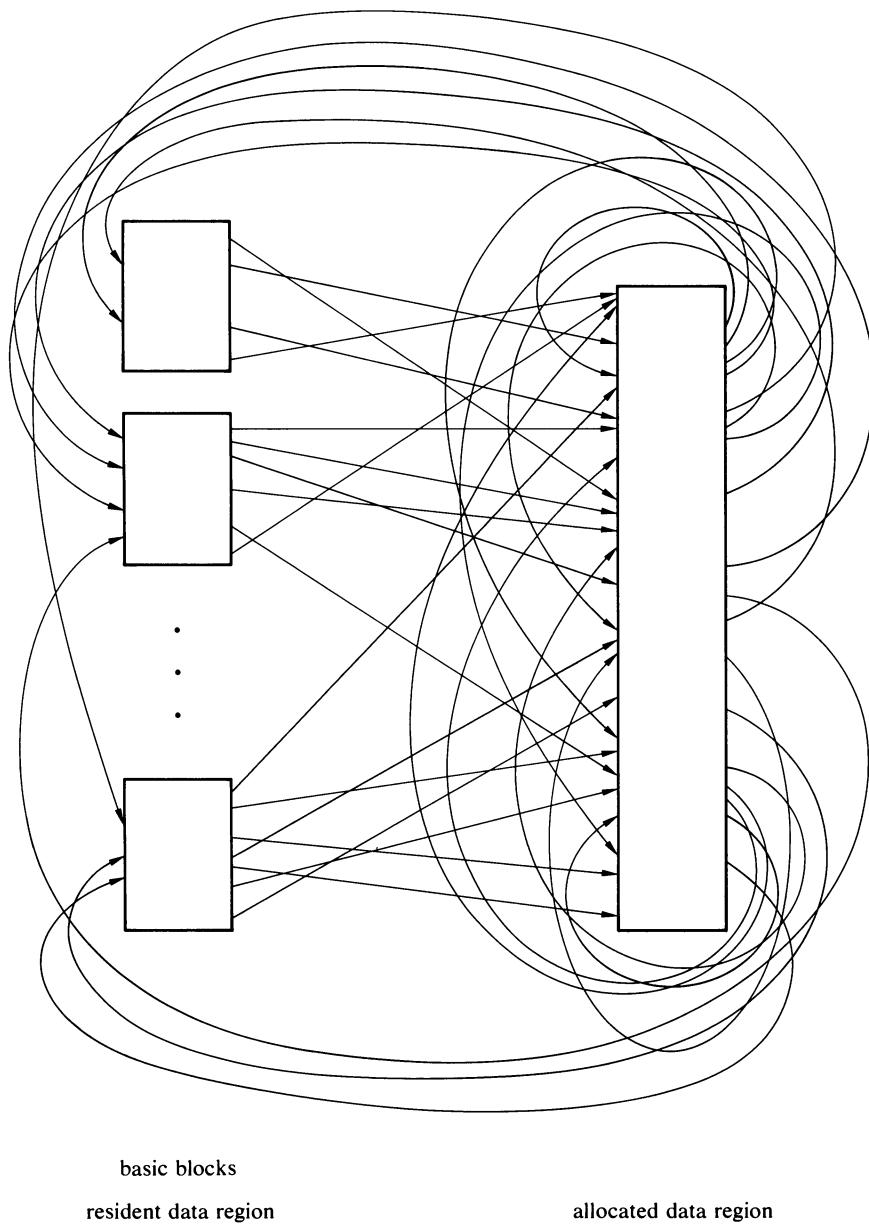


Figure 9.2.1
Pointers to allocated data.

previously marked; therefore, no new processing is initiated. The third descriptor in block 1 does not point to the allocated data region. Processing of block 1 is, therefore, complete, and the marking procedure restores the pointer to block 2 from SYSSTK. Processing of block 2 resumes with the second descriptor. This descriptor is not a pointer. Processing of block 2 is complete, and the marking procedure restores the previous position from SYSSTK. Processing resumes with

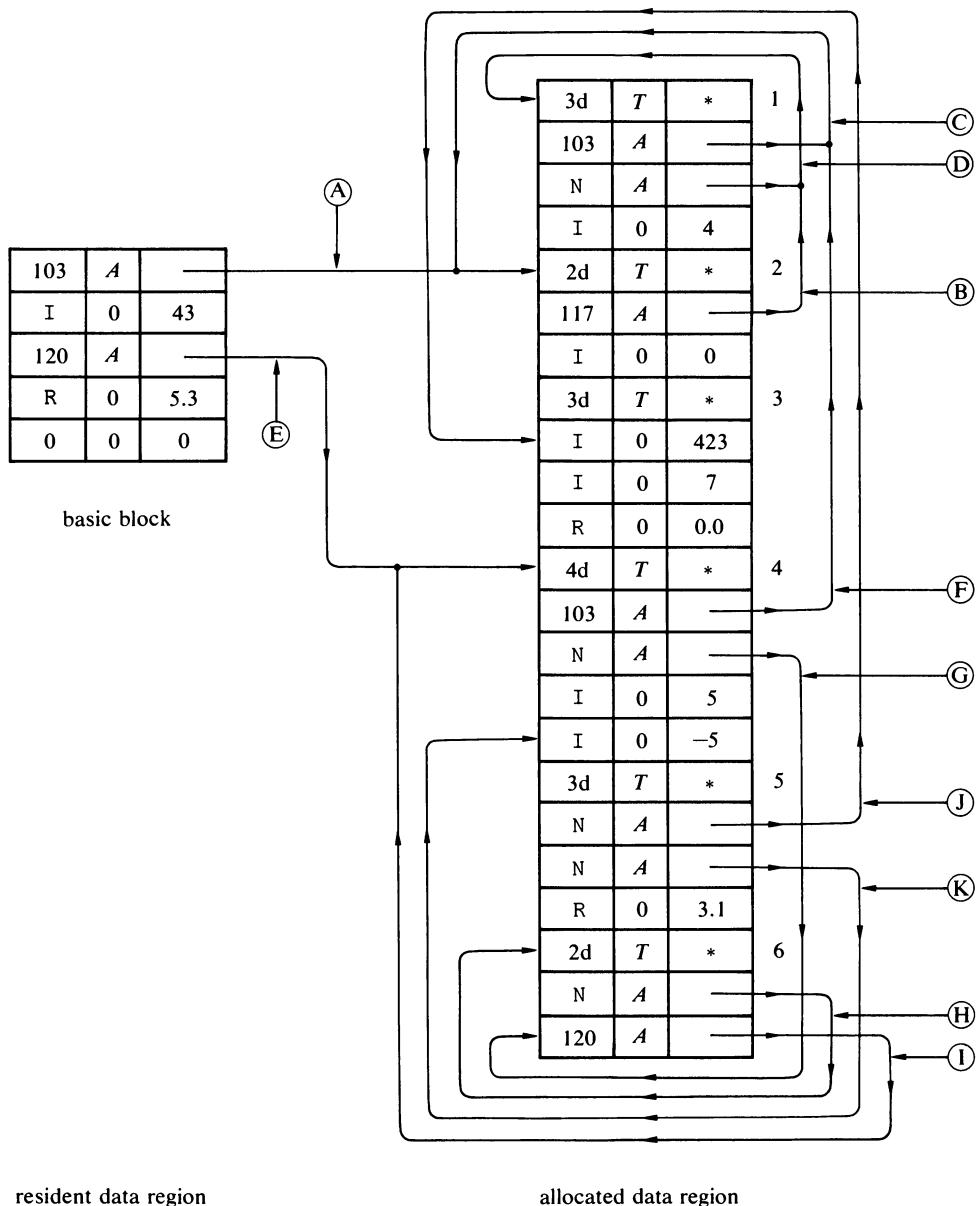


Figure 9.2.2
Storage before marking.

the basic block. The second descriptor in the basic block is not a pointer to the allocated data region, but the third, pointer **E**, points to block 4. Block 4 is marked, and MARK again pushes the current position in the basic block and examines block 4. Pointer **F** points to a marked block. Pointer **G** is processed next. Pointer **G** does not point to a title, but the title is located by examining preceding de-

scriptors until the title of block 6 is located. Block 6 is marked and processed by MARK. Pointers H and I initiate no new processing. The marking process restores the position in block 4, which similarly initiates no new processing. Restoring again, the basic block is completed. The results of marking are shown in Figure 9.2.3. The pointers are suppressed to simplify the figure. Blocks 1, 2, 4, and 6 are marked, and blocks 3 and 5 are not. Even though block 5 contains a pointer J to block 3, neither block can be referenced since block 5 cannot be referenced.

The marking algorithm given at the beginning of this section omits one category

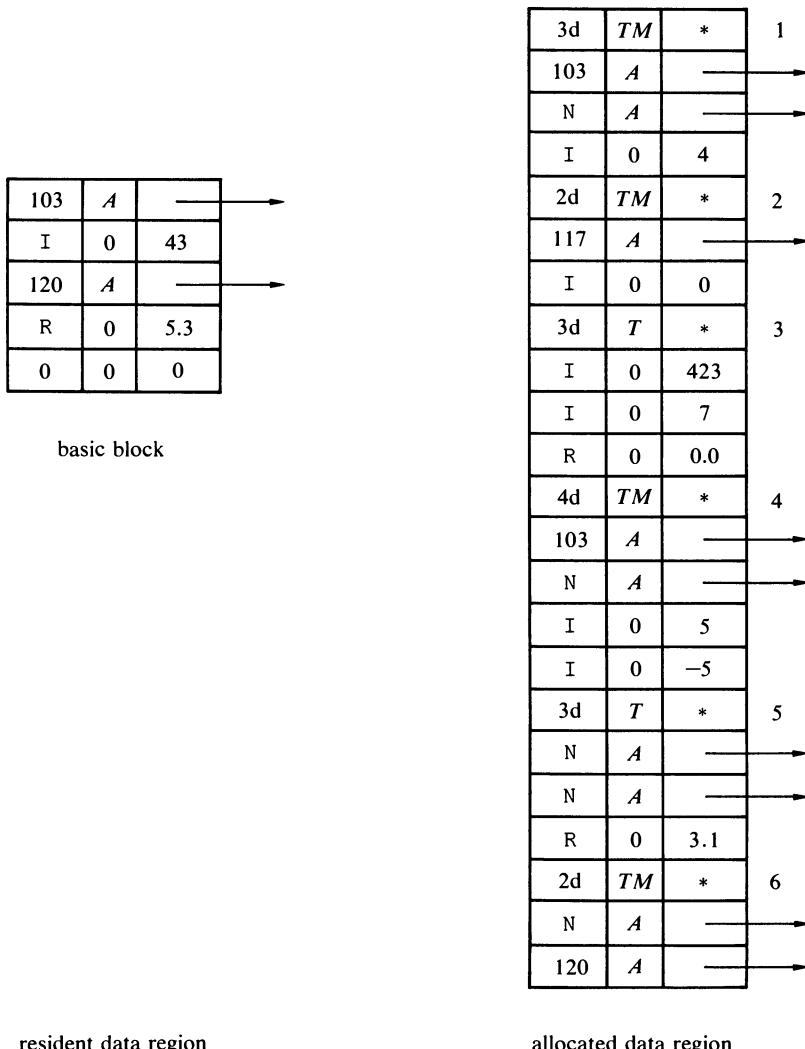


Figure 9.2.3
Storage after marking.

of object that can be referenced without any explicit pointers leading from the basic blocks. Consider the following program segment.

```

N      =      1
$( 'SUM' N)    10
.
.
.
N      =      1
OUTPUT   =      $( 'SUM' N)

```

The variable SUM1 is created by a concatenation and a value assigned to it by an indirect reference. There is no explicit reference to SUM1. Suppose storage regeneration is required between the time the value is first assigned to SUM1 and the time the value is to be printed. When this storage regeneration occurs, there may be no pointer to the natural variable SUM1 anywhere in either data region. The marking process described above would not mark the natural variable for SUM1, and if nothing else were done, SUM1 (and its value) would be destroyed by storage regeneration. SUM1 would be recreated by the output statement, but its value would then be null.

To take care of such problems, a second marking phase is required to examine the natural variables. Any natural variable that has not already been marked, but which has a nonnull value or nonzero label descriptor, is marked. If the value points to an object in the allocated data region, MARK saves the current position and examines that object, and so on. A nonzero label descriptor is treated similarly.

9.2.2 Relocation

After the allocated data region has been marked, the marked blocks and unmarked blocks are usually interspersed. The allocation process requires that free space be below the free pointer. The next step, therefore, is to relocate the marked blocks to the top of the allocated data region so that allocation can resume. Three passes are used in the relocation process: (1) title adjustment, (2) pointer adjustment, and (3) block movement.

Title adjustment consists of adjusting the V fields of marked titles so that they point to where the marked objects will go when they are moved. Title adjustment is accomplished by a linear pass from the top to the bottom of the allocated data region. Whenever an unmarked object is encountered, the space it occupies is taken into account. Whenever a marked object is encountered, the V field of its title is adjusted to point to the next available location in the allocated data region. Figure 9.2.4 illustrates the results of title adjustment applied to the previous example. The contents of blocks 3 and 5 are omitted for clarity. Blocks 1 and 2 cannot be moved up because there are no unmarked blocks above them. Since

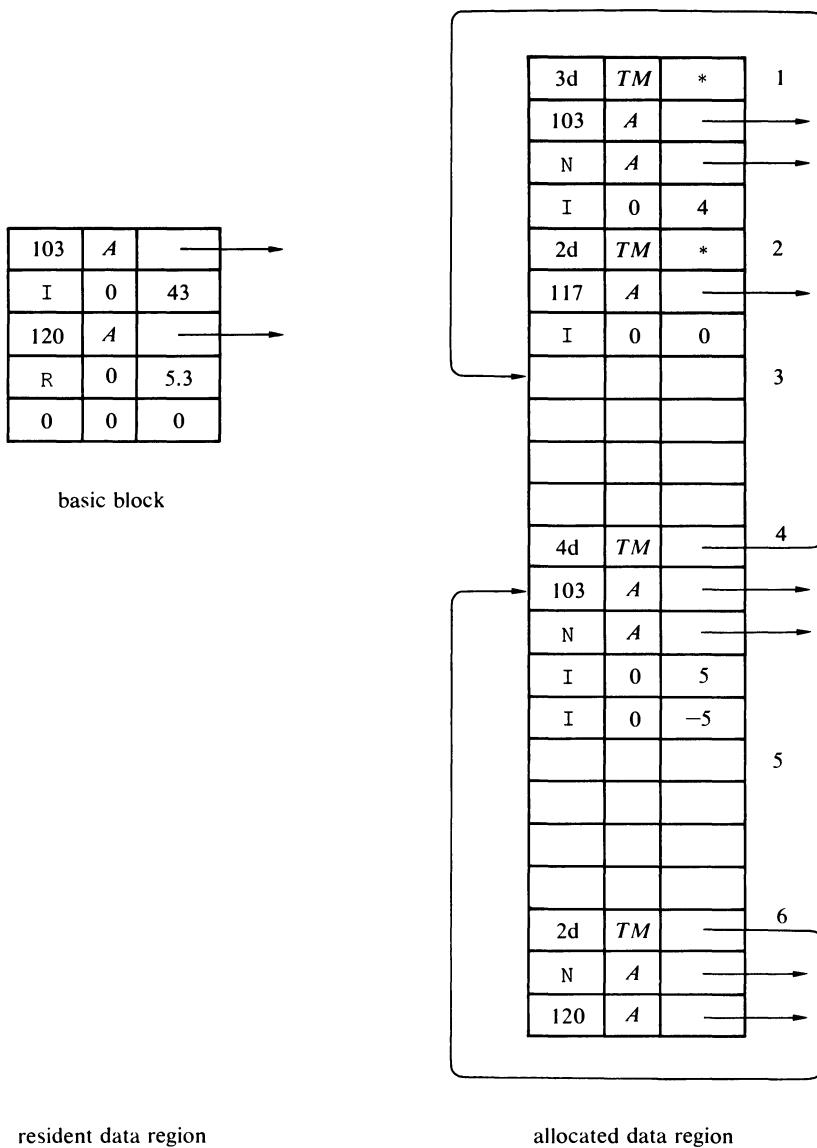


Figure 9.2.4
Results of title adjustment.

block 3 is not marked, block 4 will move up to occupy the position formerly held by block 3. Block 5 leaves additional room, and block 6 will move up to follow block 4.

V fields of title descriptors are ordinarily self-pointers. During the relocation process, however, the V fields point to where they will be moved. When they are moved, they become self-pointers again.

Pointer adjustment is performed next so that pointers to the allocated data region will be correct when the objects they point to are moved. This adjustment is performed for pointers in the basic blocks and in the marked blocks in the allocated data region. Each pointer to a marked block is adjusted using the adjusted pointer in the title of the block it points to. Figure 9.2.5 illustrates the results

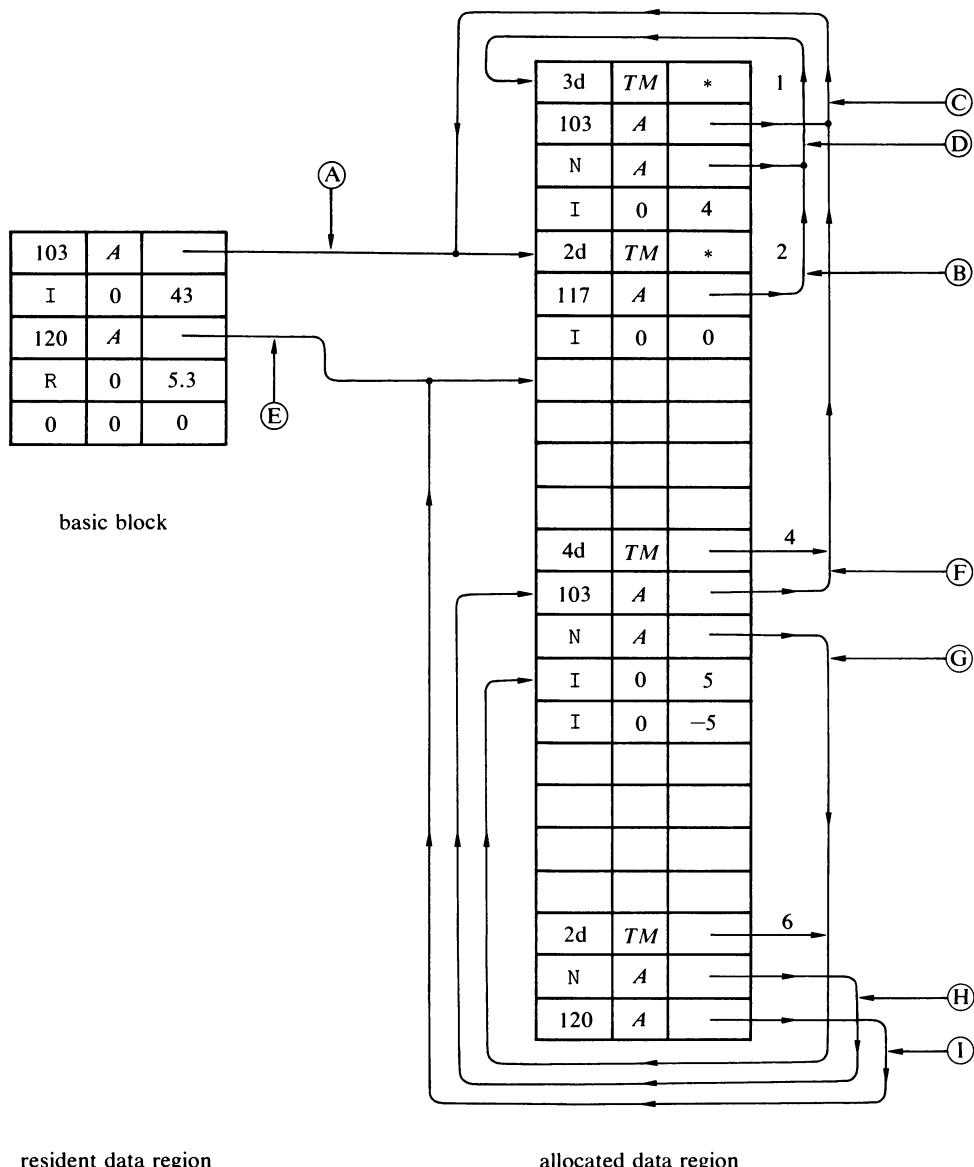


Figure 9.2.5
Results of pointer adjustment.

of pointer adjustment on the previous example. Adjusted title pointers are suppressed to simplify the figure. Pointers A, B, C, D, and F require no adjustment since the blocks they point to are not to be moved. Pointers E, G, H, and I are adjusted according to the V fields of the titles of the blocks which they point to. Pointer G does not point to a title, but the title of the object it points to is located as before, and pointer G is adjusted according to its offset from this title.

Finally, the marked objects are moved to the locations pointed to by the V fields of their titles. Figure 9.2.6 illustrates the final configuration, retaining the original block numbering for reference. Note that the blocks have been unmarked by removing the *M* flags.

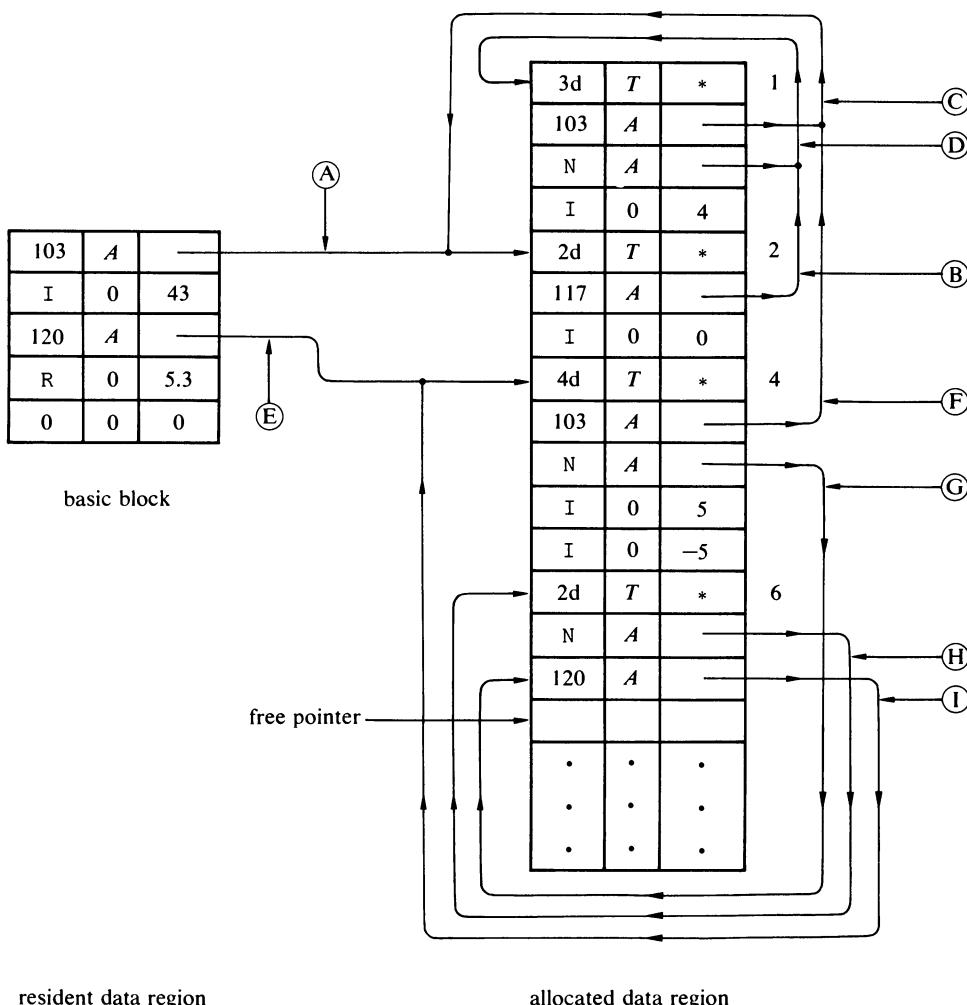


Figure 9.2.6
Results of storage regeneration.

Again natural variables require additional processing. The chain descriptors must be adjusted appropriately. Often there are natural variables in a chain that are not marked and, therefore, do not have to be saved. Not only must the chain pointers be adjusted because of title adjustment, but the unmarked natural variables must be by-passed on the new chain.

Exercises

- 9.2.1 “Garbage collection” is frequently used to describe the process of reorganizing allocated data. Explain why that term is not appropriate for the process described above.
- 9.2.2 Pointers that do not point to titles are called *interior pointers*. What is the source of interior pointers? Can interior pointers appear both in basic blocks and in the allocated data region itself?
- 9.2.3 Discuss the differences in handling blocks and natural variables in the marking process.
- 9.2.4 Why must a pointer never point directly into a string of characters?
- 9.2.5 Write a detailed algorithm for the marking process. Consider both the phase starting with the list of basic blocks and the processing of the table of natural variables.
- 9.2.6 Why is it unnecessary to mark a natural variable which has a null value and a zero label descriptor, even though it may subsequently be referenced?
- 9.2.7 The allocated data region is in a known area. Why are *A* flags rather than address bounds used to locate pointers into the allocated data region?
- 9.2.8 Discuss the problems inherent in using SYSSTK for temporary storage during marking.
- 9.2.9 Another way of keeping track of objects that must be saved during storage regeneration is to keep “use counts.” In such a scheme, a field in an allocated object is incremented every time a pointer to it is created and decremented every time a pointer to it is deleted. Therefore, when storage regeneration is required, all objects that have to be saved have nonzero use counts and marking is unnecessary. Discuss the relative merits of marking as opposed to use counting in general and in the case of SNOBOL4 in particular.
- 9.2.10 Write a detailed algorithm for title adjustment.

- 9.2.11** Relocation as described above preserves the property of having all allocated space above the free pointer and all free space below it. What other property of the allocated data region is preserved?
- 9.2.12** Objects that are always needed tend to move to the top of the allocated data region as a result of storage regeneration. Consequently, there are usually many marked blocks before the first unmarked block. What use can be made of this situation?
- 9.2.13** Write a detailed algorithm for pointer adjustment. Consider both blocks and natural variables.
- 9.2.14** In what order must blocks be moved in the final phase of relocation?
- 9.2.15** What would happen if objects were not unmarked at the end of storage regeneration?
- 9.2.16** A suggested alternative to relocating all marked objects upward is to move blocks only as necessary to fill in holes left by unmarked blocks. Evaluate this suggestion.
- 9.2.17** The SNOBOL4 system may run more slowly if the size of the allocated data region is increased. Explain this curious behavior.

part



THE MACRO IMPLEMENTATION

The Macro Language

Previous chapters describe the workings of the SNOBOL4 system. The structure of this implementation is machine-independent. The question of how such a system is actually implemented in a machine-independent and portable way remains. The obvious answer is that the implementation itself is written in a machine-independent language.

Individuals who are not familiar with the way SNOBOL4 is implemented usually assume that it is written in assembly language. Those who have heard of its machine independence generally assume it is written in FORTRAN. PL/I is sometimes supposed since PL/I is a high-level language whose operations are more suitable to such an implementation.

Clearly machine independence and portability rule out assembly language in its basic form. At the time the implementation of SNOBOL4 was undertaken, FORTRAN was considered but rejected because it is poorly suited to performing the string and list manipulations that are the heart of SNOBOL4. There are versions of FORTRAN that have such capabilities, but these extensions vary from machine to machine and lack the desired independence. PL/I was also considered but rejected. The reasons for this rejection are somewhat different.

At the time the implementation was started, there was no viable implementation of PL/I. Neither was there a likelihood of compatible versions of PL/I for most large-scale scientific machines. Other languages (such as COBOL, ALGOL, and LISP) were considered but rejected for similar reasons.

10.1 THE SNOBOL4 IMPLEMENTATION LANGUAGE

What alternative remained? The answer seemed to be to develop a machine-independent language suitable for implementing SNOBOL4. Such a language, to be useful, would have to be considerably simpler than SNOBOL4 itself since the implementation language has to be implemented for each machine on which SNOBOL4 is desired. Thus, the idea of the SNOBOL4 Implementation Language (SIL) evolved. Again, there were choices. SIL could be a procedural language, such as ALGOL. Another alternative was an assembly-like language. The latter choice was made since most machines have assembly languages with macro facilities that permit the definition of new operations [21-23]. Taking advantage of existing macro assemblers, therefore, makes the implementation of SIL considerably easier.

A set of operations appropriate for implementing SNOBOL4 was then developed. SNOBOL4 was written in SIL. Implementing SIL for a particular machine then amounted to writing the necessary macro definitions. When that was done, SNOBOL4, written in SIL, was assembled, and a SNOBOL4 implementation for that machine resulted.

It sounds simple. The ideas *is* simple, but its reduction to practice is not. The SIL operations necessary to implement SNOBOL4 have to be derived. These operations must be machine-independent over a wide range of machine architectures. SNOBOL4 has to be written in SIL in such a way that it can be assembled by a variety of assemblers without major revisions. Since SNOBOL4 is a large language with a wide range of capabilities, many SIL operations are necessary, and a few require considerable effort to implement.

Nonetheless, this idea has been reduced to practice and has resulted in working implementations of SNOBOL4 for most third-generation, large-scale scientific machines. Chapter 11 describes in some detail the implementation of SIL on two machines. Chapter 12 discusses some experiences encountered in the implementation of SIL on various machines. First, however, SIL must be described.

10.2 A DESCRIPTION OF SIL

In designing SIL, many compromises were necessary. Often conceptual elegance had to be sacrificed to the reality of existing machine capabilities. An attempt was made to limit the number of different macros to make the job of implementing SIL less formidable. Attempting to keep the number of macros small resulted in

some obvious asymmetries. An operation was included only if it was actually needed, not to make the set of operations logically complete or symmetrical. There was also a conflict between a small set of SIL operations and efficiency. Often a section of program could be written to run more efficiently if a new macro were added for a special case. Here the goal of efficiency was weighed against the penalty of increasing the size of SIL. Rarely was the situation clear-cut, and often the decision could not be justified on an objective basis. How are small differences in execution speed measured against additional difficulties in implementation? At what point would SIL become so cumbersome that its implementation would be impractical or unattractive on a particular machine? At what point would the inefficiency of SNOBOL4, aggravated by concessions to SIL, become so expensive that SNOBOL4 would not be attractive to a user?

SIL consists of about 130 different macro operations. These macros reference data contained in the resident data region. There is no concept of a machine register as such. SIL performs storage-to-storage operations. Storage is assumed to consist of descriptor-sized units as described in Chapter 5.

Fortunately for the implementor, most SIL macros are simple or even trivial. Several are troublesome. A few involve interfacing system facilities such as input and output. Several are optional in the sense that their implementation is not essential to the running of SNOBOL4. Failure to implement an optional SIL macro simply disables certain SNOBOL4 language features.

SIL is too large to describe in detail. Instead, general structure and representative macros are given. A complete list of SIL macros, arranged by function, is given in Appendix C.

SIL macros have the following format, where the numbers refer to card columns:

1	8	16	36
[loc]	oper	[operand list]	[comments]

The location name, beginning in column 1, is optional and is used if the macro is referenced. In the descriptions that follow, the location name is omitted unless it is necessary. The operation, beginning in column 8, is mandatory. The operand list begins in column 16. Most macros have operands which are separated by commas. Operands that are modified appear first in the operand list. Sometimes a particular operand in a list is omitted. An operand may itself be a list, in which case the list is enclosed in parentheses. In addition to the provision for comments starting in column 36, an entire line may be a comment. This is indicated by a * in column 1 of the line. The following line illustrates a typical macro.

GETX RCALL X, AUGPBL, (Y, Z) Augment pair block

In this example, the procedure AUGPBL is called and returns a value in X. Y and Z are two arguments passed to AUGPBL. The third operand is a list, containing two items, Y and Z.

Mnemonic conventions, followed in most cases for naming the SIL macros,

will be helpful in understanding the following sections. Usually, the first three or four characters of the name describe the type of operation. ADD, CVT (for “convert”), and MOV (for “move”) are examples. Letters following the operation usually specify the data on which the operation is performed. Thus, ADDVV adds V fields and MOVDD moves a descriptor from one place to another. The letter C indicates a constant. Thus, MOVTC moves a constant into the T field of a descriptor. Where different data types are involved, the usual abbreviation for the data type is used. Thus, CVTIR performs a data-type conversion, creating an integer from a real number. Because of the vocabulary and limited number of characters available, there are conflicts. The letter I, for example, is sometimes used to indicate indirect addressing as well as the INTEGER data type. In several cases, the entire name describes the operation. BRANCH and RCALL are examples.

10.2.1 Data Assembly Macros

The resident data region consists of various descriptors, qualifiers, and related data. This region is constructed by SIL macros that only assemble data. They are not executed during the running of SNOBOL4.

A descriptor is assembled by the macro DESCR. The format of DESCR is

DESCR T, F, V

T, F, and V correspond to the values assembled for the T, F, and V fields, respectively. A typical descriptor is

NULSTR DESCR S, 0, 0

which is the canonical representation of the null string. QUAL, which assembles a qualifier, is similar. Its form is

QUAL T, F, V, O, L

There are four other data assembly macros. BUFFER assembles a buffer of blank characters. DBLOCK assembles a block of contiguous descriptors. FORMAT assembles a string of characters for a FORTRAN format. STRING assembles a given string and qualifier for it.

10.2.2 Data Movement Macros

It is frequently necessary to move (copy) a descriptor from one place to another. The macro that does this is MOVDD, which has the form

MOVDD D₁, D₂

When MOVDD is executed, the entire descriptor at location D₂ is moved to D₁, replacing the former value at D₁. The contents of D₂ are not changed. MOVQQ is a similar macro for moving qualifiers.

Sometimes only one field needs to be moved. An example is the macro MOVT, which moves the T field from one descriptor to the T field of another. More interesting cases arise when one type of field is moved to another type of field. Two examples are MOVVT and MOVT. The first moves a T field from one descriptor into the V field of another. The second performs the converse operation. MOVVT is straightforward since the T field always fits into the V field. The converse is not true. Since the V field is larger than the T field, the meaning of MOVT must be precisely described. In fact, MOVT is used only when the V field contains an unsigned integer small enough to fit into a T field. Thus, the low order part of the V field can be moved into the T field.

Much data is referred to indirectly. To move such data, there are macros that permit a level of indirect addressing. GETD and PUTD are examples. GETD ("get descriptor") has the form

GETD D_1, D_2, D_3

D_1 is the target as in MOVDD. D_2 and D_3 are descriptors of the form

D_2	DESCR	T_2, F_2, V_2
D_3	DESCR	T_3, F_3, V_3

GETD adds the values of V_2 and V_3 to get the location of a descriptor that is then moved to the target position D_1 . Figure 10.2.1 illustrates a typical situation in which V_2 is a pointer and V_3 is an offset. GETDC is the same as GETD except that

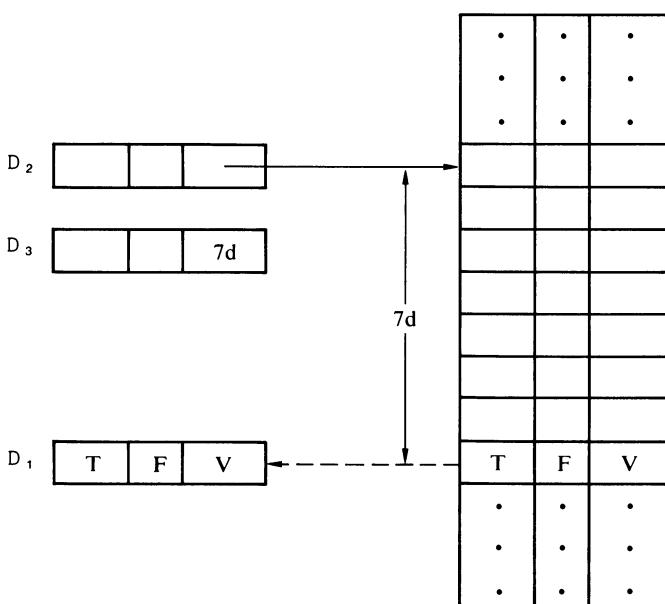


Figure 10.2.1
An example of GETD.

the offset is given by a constant instead of V_3 . PUTD is similar to GETD, except that D_3 is moved to a target location given by $V_1 + V_3$. TRANDC is a combination of GETDC and PUTDC in which constant offsets are provided for determining both source and target locations.

There are a number of similar operations for moving individual fields, both directly and with various forms of indirect addressing. There are many SIL operations that move fields from one place to another, with various kinds of indirect addressing. Such operations are basically easy to implement.

10.2.3 Arithmetic Operations

Several operations perform integer and real arithmetic as well as address computation. Among the simplest are those that increment or decrement fields. INCVC is typical. The form of the macro is

INCVC D, C

where the V field of D is incremented by the constant C. DECVC is the converse operation.

Another class of operations is exemplified by ADDVV, which adds two V fields and puts the result in a third. The form is

ADDVV D₁, D₂, D₃, F, S

in which the V field of D₁ is replaced by the sum of the V fields of D₂ and D₃. ADDVV illustrates a convention about the flow of control in SIL. If $V_1 + V_2$ is too large to fit into a V field, ADDVV fails. The V field of D₁ is not modified, and control branches to the location F. If the operation is successful, the branch is to S. If either F or S is explicitly omitted, control passes to the next macro in line if the corresponding situation occurs. ADDRVV has the same format, but the V fields contain real numbers, and real addition is performed. There are complete sets of operations for both integer and real arithmetic.

10.2.4 Data Comparison Macros

The need to test the value of data arises frequently. A typical macro compares two descriptors to see if they are equal. EQLDD performs this operation and has the form

EQLDD D₁, D₂, NE, EQ

D₁ and D₂ are the two descriptors that are compared. The exits NE and EQ are the names of two locations. If D₁ and D₂ are identical, execution of EQLDD causes transfer of control to the location EQ. Otherwise, control is transferred to NE. If

either NE or EQ is explicitly omitted in the call of EQLDD, control is not transferred for that condition but rather passes to the next macro in line. The following sequence of macros illustrates a typical situation.

```
EQLDD    X, Y, ,BYPASS
MOVDD    X, Z
BYPASS   MOVDD   Y, W
```

This sequence moves Z into X unless the descriptors X and Y are the same, in which case the value of X is unchanged. In either case, W is moved into Y. There are a number of similar macros. For example, EQLTT compares two T fields, EQLVC compares a V field to a constant, EQLTC compares a T field to a constant, and so on.

Another typical operation compares the arithmetic value of a V field with a constant. The macro CMPVC ("compare V field to a constant") has the form

```
CMPVC    D, C, GT, EQ, LT
```

The value of the V field of D is compared to C. The three exits indicate locations to which control is transferred, depending on the result of the comparison. If the value of the V field is greater than C, control is transferred to GT, and so on. As in the former macro, any of the exits can be explicitly omitted, indicating control is to pass to the next macro in line. For example, execution of

```
CMPVC    X, 0, POSIT
```

results in transfer to POSIT if the V field of X is greater than zero.

CMPSS compares strings pointed to by two qualifiers. The comparison is lexical. The form of the macro is

```
CMPSS    Q1, Q2, GT, EQ, LT
```

with the usual exit conventions. CMPSS is used for various internal comparisons of strings and by the built-in predicate LGT.

10.2.5 Flag Macros

Operations on F fields of descriptors are similar to those on other fields, except that the F field is treated as a string of disjoint bits, referred to as flags. For example,

```
ADDF    D, FLG
```

inserts the flag FLG in the F field of D but does not disturb other flags in that field. Similarly, CLRF clears the flag FLG. The macro

```
TESTF   D, FLG, F, S
```

branches to S if the flag FLG is present, and to F otherwise.

For each of these operations there is a corresponding indirect operation. Thus,

ADDFI D , FLG

inserts the flag FLG in the F field of the descriptor pointed to by the V field of D.

10.2.6 Operations on Blocks of Descriptors

There are several macros that operate on blocks of descriptors. CLRBLK ("clear block") is an example. The format is

CLRBLK D₁ , D₂

The V field of D₂ contains a length, specifying how much space is to be cleared. The V field of D₁ points to the first descriptor to be cleared. CPYBLK copies a block of descriptors from one place to another.

LOCAD and LOCBD are macros for locating descriptors in pair blocks. LOCAD has the format

LOCAD D₁ , D₂ , D₃ , F , S

which searches the block pointed to by the V field of D₂ for a descriptor that has the same value as D₃. The A positions of the A-B pairs are searched. If a matching A descriptor is found, a pointer to the descriptor above the A-B pair is placed in the V field of D₁ and control branches to S. If the descriptor is not found, control passes to F. If S or F is omitted, control passes to the next macro for that case. LOCBD is similar, except that B descriptors are searched. Figure 10.2.2 illustrates a typical result of using LOCAD.

10.2.7 Type Conversion Macros

Explicit conversion of data types is performed by the function CONVERT. Implicit conversion occurs in many contexts, particularly between STRING and INTEGER. Conversions between STRING, INTEGER, and REAL are performed by a group of macros with a similar format. A typical macro is CVTIS ("convert to integer from string") which has the format

CVTIS D , Q , F , S

The string qualified by Q is converted to an integer, which is placed in the V field of D. The F field of D is cleared, and the T field is set to the integer data-type code I. Figure 10.2.3 illustrates a typical application of CVTIS. Note that the string may have a sign and leading zeros. The conversion fails if the string does not represent an integer or if the integer it represents is too large to fit into a V field. In this case, D is not changed, and a branch to F is made. If the conversion is successful, a branch to S is made. CVTSI, the inverse operation, always succeeds and, hence, does not have branch point options.

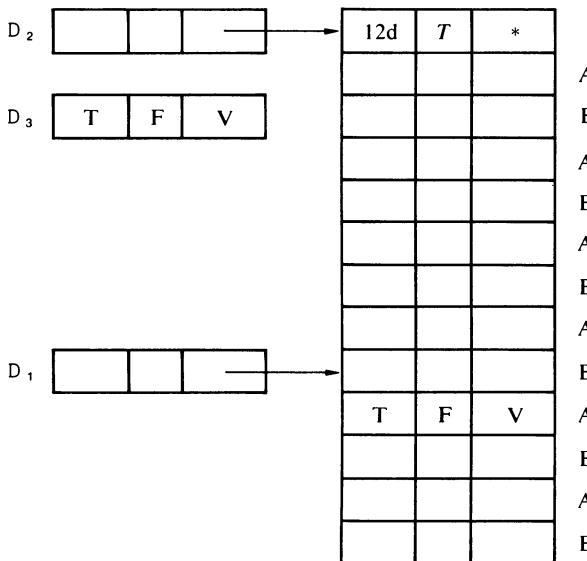


Figure 10.2.2
An application of LOCAD.

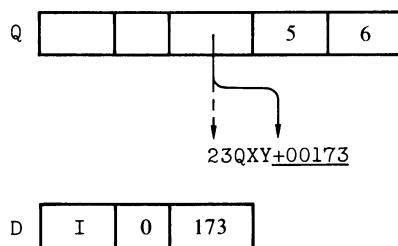


Figure 10.2.3
An application of CVTIS.

10.2.8 String Manipulation and Syntax Table Macros

Several macros operate on qualifiers and their associated strings. Concatenation is performed by APDQQ, which appends strings pointed to by two qualifiers. APDQQ has the format

APDQQ Q_1, Q_2

A copy of the string pointed to by Q_2 is appended to the string pointed to by Q_1 . APDQQ is usually applied to strings in the allocated data region, copying into space provided by the storage management procedures. TRIMQQ shortens the length of a specifier as appropriate to delete trailing blanks.

COMQNV constructs a qualifier for a string in a natural variable, as illustrated in Section 5.2. DELQC deletes a character from the head of a string by incrementing the offset and decrementing the length. COMSQ computes a qualifier for a substring between two other strings, located as the result of pattern matching.

String analysis is performed by STREAM, which has the form

```
STREAM Q1, Q2, T, E, R, S
```

Q₂ is analyzed using the syntax table specified by T. This analysis accepts an initial part, which becomes Q₁ and leaves a remainder, Q₂.

The three exits are:

- (1) E, if ERROR is signaled by the syntax tables
- (2) R, in case STREAM runs off the end of the string without a signal from the syntax tables
- (3) S, if normal termination results from a signal of STOP or STOPSH by the syntax tables

The last value specified by a PUT is returned in the V field of the global descriptor STYPE.

As an example, consider the procedure FORTXT, described in Chapter 7. This procedure uses STREAM to advance a pointer to the next nonblank character in the input text. The table used is FORWRD, which has the definition

```
BEGIN FORWRD
  FOR(BLANK) CONTIN
  FOR(EQUAL) PUT(EQCOD) STOP
  FOR(RP) PUT(RPCOD) STOP
  FOR(RB) PUT(RBCOD) STOP
  FOR(COMMA) PUT(CMACOD) STOP
  FOR(COLON) PUT(CLNCOD) STOP
  FOR(SEMI) PUT(EOSCOD) STOP
  ELSE PUT(NBKCOD) STOPSH
END FORWRD
```

A macro call is

```
STREAM XQL, TEXTQL, FORWRD, , , FORRD
```

where TEXTQL points to the input text. Figures 10.2.4 and 10.2.5 show the situation before and after executing this macro in a typical situation. TEXTQL is advanced four characters, consuming the three blanks after the subject and the equal sign. The four characters consumed are pointed to by XQL. EQCOD is inserted in the V field of STYPE, indicating the type of character that stopped STREAM.

STREAM is used mainly by the translator and in the analysis of prototype arguments of ARRAY, DATA, DEFINE, and LOAD. The only syntax table modified during execution is MATCH, used by the pattern-matching procedures for ANY, NOTANY,

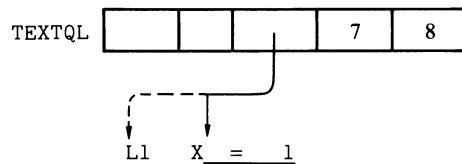
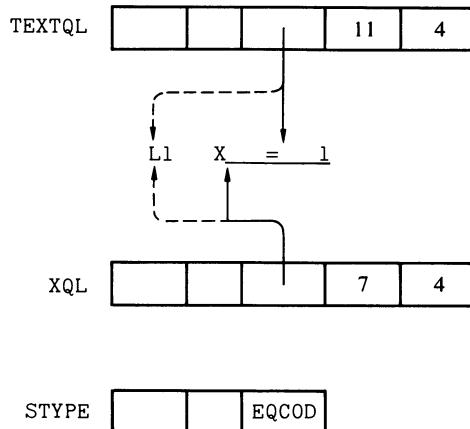


Figure 10.2.4
TEXTQL before executing STREAM.



STYPE [] EQCOD

Figure 10.2.5
The results of applying STREAM.

BREAK, and SPAN. There are two macros used to modify this table: CLEART and SETUPT. CLEART has the form

CLEART T,K

where T is the table and K is a condition which is one of the following: CONTIN, ERROR, STOP, or STOPSH. These conditions correspond to the meanings given in the syntax table description. As a result of executing CLEART, all entries in T are set to correspond to the given condition. SETUPT is then executed to establish conditions (STOP, STOPSH, or ERROR) for selected characters. The form of SETUPT is

SETUPT T,K,Q

which inserts the condition given by K for all characters in the string pointed to by Q.

10.2.9 Stack Management, Recursion, and Procedures

All calls are recursive. SYSSTK in the resident data region provides the necessary storage. This stack, consisting of a block of descriptors, is used by several macros. There is a pointer to the current stack position. Descriptors are pushed onto and popped from SYSSTK by the macros PUSHD and POPD. The form of PUSHD is

PUSHD (D₁, ..., D_n)

which copies the values of the descriptors D₁, ..., D_n onto SYSSTK. Figure 10.2.6 illustrates SYSSTK before and after execution of PUSHD.

POPD has the same form as PUSHD. The inverse of the previous operation is

POPD (D_n, ..., D₁)

which copies the value of the top n descriptors from SYSSTK into D_n, ..., D₁. The descriptors are listed in reverse order to emphasize that restoration occurs from the top of SYSSTK backward. After executing POPD, a SYSSTK pointer is repositioned accordingly. Although former values remain on SYSSTK, they are not accessible. Only information below the stack pointer is available. PUSHQ and POPQ are corresponding operations for saving and restoring qualifiers. Since a

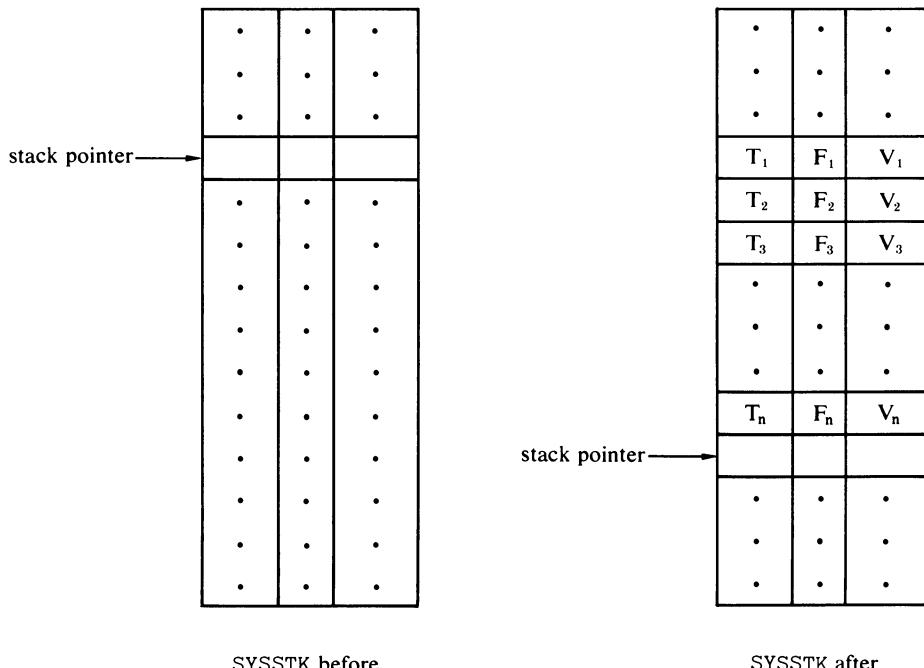


Figure 10.2.6
Results of PUSHD.

qualifier is composed of two descriptors, pushing a qualifier amounts to pushing two descriptors. Only descriptor-sized quantities are put on SYSSTK.

Recursive calls are made by RCALL which has the form

```
RCALL D, P, (D1, ..., Dn), (R1, ..., Rm)
```

This macro calls the procedure P, which is to return a value in the descriptor D. D₁, ..., D_n are argument descriptors placed on SYSSTK for the use of P. R₁, ..., R_m are locations to which transfer is made upon return from P, which signals the appropriate exit by means of an integer code. D may be omitted, indicating no value is returned by P. There may be no argument descriptors. If any of R₁, ..., R_m are omitted, control passes to the next instruction following the RCALL if the corresponding integer is signaled. In the simplest case, a call has the form

```
RCALL , P
```

which simply calls P and then continues.

When an RCALL is executed, several changes are made to SYSSTK. Some machine-dependent state information (such as registers) may have to be saved. The location of the code to select one of R₁, ..., R_m is saved. The location of D must be noted in some way. Finally, D_n, ..., D₁ are pushed onto SYSSTK. The descriptors are pushed in reverse order so that the called procedure can pop them in the same order as they appear in the RCALL. To facilitate handling of recursive calls and returns, two stack pointers are used. In addition to the current stack pointer, there is an old stack pointer which points to the position of SYSSTK from which the previous call was made. Whenever a call is made, the old stack pointer is saved on SYSSTK, the old stack pointer becomes the current stack pointer, and a new current stack pointer is established beyond the information pushed by the call. Figure 10.2.7 illustrates SYSSTK before and after a call.

Finally, transfer is made to P. P is a procedure entry point, identified by the PROC macro, of the form

```
P PROC [primary name]
```

The optional primary name is used if P is an entry point within another procedure.

As a result of RCALL, execution starts at P with SYSSTK modified as indicated. P pops arguments off SYSSTK as necessary and performs whatever computation is required. When the computation is complete, P returns by executing the macro RRETURN, which has the form

```
RRETURN X, C
```

RRETURN returns the value of the descriptor X, placing it in D as indicated in the RCALL macro. C is a constant that signals which exit is to be taken from RCALL. X may be omitted if no value is returned. After storing the returned value, RRETURN restores SYSSTK to its condition before the call. In the process, the current stack pointer is restored from the old stack pointer, the old stack pointer is restored from SYSSTK, and any machine-dependent state information that was saved is

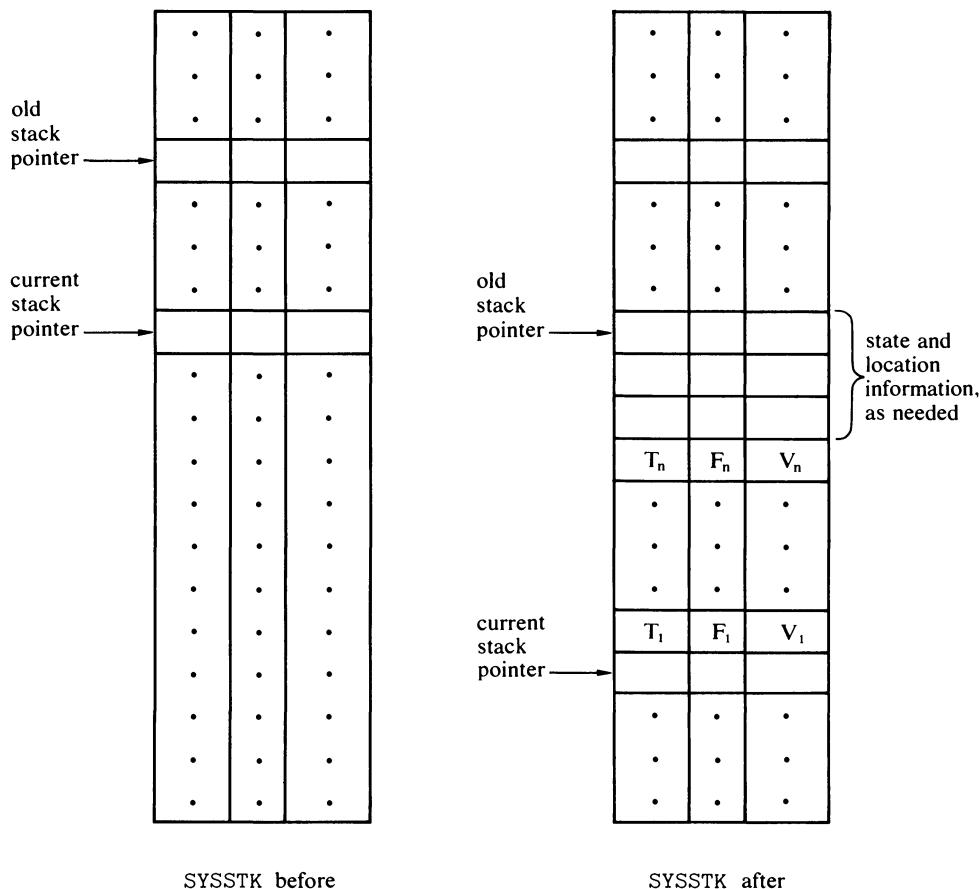


Figure 10.2.7
Modifications to **SYSSTK** by **RCALL**.

restored. Control is then transferred to the specified branch point. A typical return is

FAIL RRETURN ,1

which selects exit **R₁** from the call, but returns no value. (The implementation of **RCALL** and **RRETURN** on two machines is described in some detail in Chapter 11.)

The macro **INITST** initializes the current stack pointer and the old stack pointer. **INITST** is used to reinitialize **SYSSTK** at program termination in order to avoid stack overflow while printing diagnostic information. The macro **STKPTR** stores the current stack pointer in the **V** field of a descriptor. In most cases, **SYSSTK** is manipulated as a push-down list in a last-in, first-out (lifo) manner. Some procedures, however, make use of the known location of information on **SYSSTK**, using pointers provided by **STKPTR**.

10.2.10 Flow of Control

RCALL and RRETURN illustrate ways in which control is transferred to and from procedures. Within a procedure there are various methods of branching. Previous sections have indicated the use of branch points to alter the flow of control, depending on the success or failure of an operation, the result of a test, and so on. One of a number of branches can be selected using the macro BRANLV which has the form

```
BRANLV D, (B1, ..., Bn)
```

The V field of D contains an integer i, $1 \leq i \leq n$. Branch point B_i is selected. If the specified branch point is explicitly omitted, control passes to the next macro in line.

Finally, there are two forms of unconditional transfer, BRANCH and BRANIN. BRANCH has the form

```
BRANCH L
```

and causes an unconditional branch to location L. BRANIN is an indirect branch with the form

```
BRANIN D
```

which causes an unconditional branch to the location indirectly pointed to by the V field of D. D is a link descriptor, and the location is always an entry point of a procedure. Figure 10.2.8 illustrates BRANIN. BRANIN is used by the interpreter to access function and matching procedures through link descriptors pointed to from prefix code.

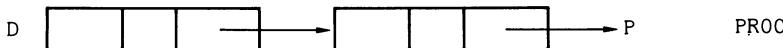


Figure 10.2.8
The BRANIN operation.

10.2.11 Pattern Building and Matching Macros

Three macros are used in building patterns. MAKPAT is the most basic, constructing a pattern component from a given function descriptor, argument descriptor, and heuristic information. CPYPAT copies two patterns in separate blocks into a larger block, relocating the offsets in the second pattern to account for the displacement from the title of the new block. CONALT connects one pattern as the alternate of another. Since the first pattern may already have an alternate, CONALT links through the alternates of the first pattern, examining connector descriptors until a zero is found, indicating the absence of an alternate. An offset to the new alternate is inserted in this field.

Two macros are used for special purposes during pattern matching. COMBAL performs the pattern matching required for the built-in pattern BAL, determining the length of the shortest string that is balanced with respect to parentheses. This requires counting left and right parentheses. COMBAL may fail. CMPPOS performs a calculation based on the cursor position and the residual to determine whether length failure should be signaled.

The macro COMMML ("compute minimum match length") is used during pattern building to determine the minimum match length when constructing heuristic descriptors. COMMML is also used at the beginning of pattern matching to determine whether the subject string is long enough to satisfy the minimum match length of the entire pattern. This is necessary since the minimum match length stored in the heuristic descriptor of the first component of a pattern applies only to its subsequents.

10.2.12 Storage Management Macros

COMNVZ is used by the procedure that constructs natural variables. COMNVZ, given a qualifier for a string, computes the size of a block required for a natural variable for that string. The size is four descriptors in addition to however many descriptor-sized units are necessary to store the string itself.

COMSZ is used by storage regeneration procedures to determine the size of a structure. COMSZ is given a pointer to the title of a structure that may be a block or a natural variable. If the title contains an *N* flag, indicating a natural variable, COMSZ uses the length of the string in the T field of the title to determine how many descriptors there are in the natural variable. See COMNVZ above. If the title does not contain an *N* flag, the size in the T field gives the number of descriptors. COMSZ is used, among other things, during linear passes through the allocated data region, where it is necessary to move from title to title.

LOCTTL is used to locate the title of a block. The argument given to LOCTTL is a pointer into the allocated data region. This argument may point directly to the title which can be determined by a *T* flag in the descriptor pointed to. On the other hand, the argument may be an interior pointer. In this case, LOCTTL decrements the pointer until the title descriptor is located. In the process, the offset of the interior pointer from the title is determined for use in relocating the pointer to a new location. RELPTR performs the relocation of a pointer, as described in Chapter 9, using the offset computed by LOCTTL.

10.2.13 System Interface Macros

To obtain as much machine independence as possible, system interface macros are simple and few in number. INITEX is the first macro executed when SNO-BOL4 is called. INITEX performs whatever machine-dependent initialization is required. Establishing registers and setting up the allocated data region are

typical functions of INITEX. TERMEX is the last operation executed and is responsible for returning to the operating system.

Input and output are performed by three macros. The macro

```
READQ    Q, D, EOF, ERR, S
```

reads a string into the area pointed to by the qualifier Q. D points to an input block which contains unit and record length information. The three exits specify locations for transfer on end-of-file, error, and successful read, respectively. The macro

```
OUTPUT   D, F, (D1, D2, ..., Dn)
```

performs FORTRAN-style output. The list of items given by D₁, D₂, ...D_n is put out according to the format F. D contains the unit number for output. The macro

```
PRINTQ   D1, D2, Q
```

prints the string given by Q. D₂ points to an output block which contains unit number and format information. D₁ is available for a return code to be supplied by the output routine, but this feature is not used at present.

Two macros are used to get the time and date. The macro

```
TIME     D
```

places the time in milliseconds in the V field of D. The macro

```
DATE     Q
```

obtains the date as an eight-character string of the form mm:dd:yy and sets up Q to point to this string.

Three macros handle external functions. The macro

```
LOAD     D, Q1, Q2, F, S
```

loads the external function whose name is given by Q₁ from the library given by Q₂. The LOAD macro places the entry point address of the loaded function in the V field of D. F and S are transfer locations corresponding to failure and success of the loading operation. Transfer to F might occur if, for example, the named function is not found in the specified library. To access an external function, the macro

```
LINK     D1, D2, D3, D4, F, S
```

is executed. D₂ points to a list of argument descriptors whose length is given in the V field of D₃. The V field of D₄ points to the external function. The T field of D₁ contains the code for the expected data type of the value to be returned, as described in Section 2.3.2. When the external function returns, the value is placed in D₁. Failure is indicated by branching to F. A branch to S indicates successful computation of a value. Finally, the macro

```
UNLOAD   Q
```

unloads the external function whose name is given by Q.

10.2.14 Other Macros

There are several other categories of macros that will be discussed only briefly. Three macros are used specifically for constructing code trees. APDSIB and APDSON append a node as a right sibling and left son, respectively. INSNOD inserts a node above another node.

Assembly control macros include EQLLOC, which defines a location; EQU, which equivalences two symbols; TITLE, which titles the assembly listing; COPY, which inserts a segment of machine-dependent code from an external source; and END, which terminates the assembly.

There are three macros that do not fall into any convenient classification.

One of these is HASH, the macro that computes the hash numbers used to structure the table of natural variables. The format of HASH is

HASH D , Q

where Q points to the string on which the computation is to be made. The resulting chain offset and order number are stored in the V field and T field of D. HASH is one of the most machine-dependent of all the macros. A detailed description of the implementation of HASH on two machines is given in the next chapter.

ORDNVT is used to order the natural variable table for the string dump given at the end of program execution. ORDNVT puts all the natural variables with nonnull values on a single chain in alphabetical order. This operation is quite involved, but the SNOBOL4 system is written so that ORDNVT need not be implemented, as indeed it usually is not. If ORDNVT is not implemented, the string dump is not printed in alphabetical order.

Finally, RPLACE is a macro that performs a character-for-character replacement used to implement a built-in function, REPLACE.

10.3 SNOBOL4 WRITTEN IN SIL

10.3.1 The Structure of the SNOBOL4 Program

The SNOBOL4 program consists of three parts: (1) linkage and definitions of constants, (2) executable code, and (3) the resident data region. Linkage and definitions of constants are divided into three sections: (1) a COPY segment for machine-dependent linkage to subroutines, (2) a COPY segment for machine-dependent definitions of constants, and (3) machine-independent definitions of constants. The subroutines called by SIL depend on decisions by the implementor about what code should be in line and what code should be out of line. Decisions vary according to the machine and the environment and are made on the usual criteria of frequency of execution, number of occurrences, and the amount and complexity of the code required.

Certain constants depend on machine architecture and environmental con-

siderations. Typical machine-dependent constants are DWDTH and QWDTH, the address widths of a descriptor and a qualifier, respectively. Flags used in the F field have a machine-dependent representation. Standard FORTRAN unit numbers are also included among the machine-dependent constants to give the implementor more flexibility. Altogether about a dozen machine-dependent constants must be supplied by the implementor.

Most constant definitions are machine-independent. These include positions in fields of data structures, the sizes of resident data objects (such as SYSSTK), token codes, and data-type codes.

Executable code is arranged in the following manner:

- (1) program initialization code
- (2) translator and interpreter invocation code
- (3) miscellaneous support procedures
- (4) storage management procedures
- (5) translator procedures
- (6) interpreter executive and control procedures
- (7) argument evaluation procedures
- (8) arithmetic procedures
- (9) pattern-construction procedures
- (10) scanner procedures
- (11) matching procedures
- (12) function definition and invocation procedures
- (13) external function procedures
- (14) procedures relating to arrays, tables, and defined data objects
- (15) input and output procedures
- (16) tracing procedures
- (17) other function procedures
- (18) common return signaling code
- (19) program termination code
- (20) error routines

The resident data region contains many data objects. Principal among these are:

- (1) a COPY segment for machine-dependent data
- (2) scratch descriptors and qualifiers
- (3) constants

- (4) initialization data
 - (5) character buffers
 - (6) pair blocks such as DTPBL, FNCPBL, PKYPBL, UKYPBL, INPBL, and OUTPBL
 - (7) trace structures, including TTPBL, VTRPBL, and so forth
 - (8) OPERBL
 - (9) SYSSTK and PATSTK
 - (10) chain headers for the natural variable table
 - (11) strings used in messages
 - (12) formats

Machine-dependent data includes the string of all characters in collating sequence (&ALPHABET), certain characters which have syntactic significance to some assemblers (such as the quotation mark), and whatever data is required by SIL macros for a particular machine.

Data used for initialization consists mainly of strings which are used to create natural variables for the names of keywords, built-in functions, and so forth. This area of resident data is subsequently used as a buffer for the formation of trace and dump messages.

10.3.2 Examples of Procedures Written in SIL

Four short SIL procedures follow. These procedures, although simple, are representative and at the same time illustrate details of the interpretive process. The code that follows is slightly different from the actual code in the SNOBOL4 system. Minor, inessential modifications have been made for tutorial reasons. The descriptions of the procedures are brief, and further study may be helpful in understanding the SNOBOL4 system and SIL.

INTERP is the heart of the interpreter. INTERP is called initially to start program execution and is called recursively when a programmer-defined function is executed. The INTERP procedure is

INTERP PROC			
INCVC	CODOFF, DWDTH	Increment offset	
GETD	XPTR, CODPOS, CODOFF	Get object code descriptor	
TESTF	XPTR, FFLG, INTERP	Test for function	
RCALL	XPTR, INVOKE, XPTR, (, INTERP, INTERP, INT1, INT2, INT3)		
MOVDD	CODOFF, FRTNCL	Set offset for failure	
INCVC	STFVAL, 1	Increment &STFCOUNT	
CMPVC	TRCVAL, 0, , INTERP, INTERP		Check &TRACE

* Check &TRACE

LOCAD	ATPTR, KTRPTR, STFKEY, INTERP	
*		Check for KEYWORD trace
*		through pointer to pair block
RCALL	, TRPHND, ATPTR	Perform trace for &STFCOUNT
BRANCH	INTERP	
INT1	RRTURN , FAIL	Signal FRETURN
INT2	RRTURN , NAME	Signal NRETURN
INT3	RRTURN , VALUE	Signal RETURN

INTERP first increments the code offset (CODOFF) and then fetches a descriptor from the prefix code. If this descriptor contains an *F* flag (FFLG), it is a function descriptor. If the descriptor is not a function descriptor, it is ignored. Otherwise, INVOKE is called to access the function procedure specified. Six exits are possible. Exits 1, 2, and 3 correspond to failure, return by name, and return by value, respectively. Exits 4, 5, and 6 result from FRETURN, NRETURN, and RETURN from a programmer-defined function. In case of failure, CODOFF is set to the position in the prefix code where execution is to continue in case of statement failure. The value of &STFCOUNT is incremented and a trace procedure called if that keyword is being traced. FAIL, NAME, and VALUE are mnemonics for return signals and have the values 1, 2, and 3, respectively. Return by name or value results in continuing interpretation with the next descriptor of prefix code. The three returns from a programmer-defined function call are converted into appropriate signals to DEFFNC.

INVOKE is called from many places in the SNOBOL4 system, including INTERP. INVOKE is the entrance to all function procedures, serving as a distributor to the appropriate location. The INVOKE procedure is

INVOKE	PROC	,	Invocation procedure
	POPD	INCL	Get function descriptor
	GETDC	XPTR, INCL, 0	Get link descriptor
	EQLTT	INCL, XPTR, INVK2	Check argument counts
INVK1	BRANIN	INCL, 0	If equal, branch indirect
INVK2	TESTF	XPTR, VFLG, ARGNER, INVK1	
*			Check for variable number

INVOKE pops its argument, which was originally obtained from prefix code. The argument count given in the prefix code is checked against the argument count given in the procedure descriptor pointed to. In most instances these are the same. If they are not, the number of arguments that actually appear in the prefix code is different from the number expected by the function procedure. Certain procedures such as array references and DEFFNC permit these counts to be different. Such procedures are identified by the *V* flag (VFLG) in the procedure descriptor. If the argument counts agree, or if the *V* flag is present, the function procedure is accessed through BRANIN. If not, control is transferred to an error routine, ARGNER.

TRIM is a simple function procedure accessed through INVOKE. The TRIM procedure is

TRIM	PROC	,	
	RCALL	XPTR,STRVAL,,TRIMF	Get string
	COMQNV	ZQL,XPTR	Get qualifier
	TRIMQQ	ZQL,ZQL	Trim string
	RCALL	ZPTR,GENVAR,ZQLPTR	Generate variable
	RRTURN	ZPTR,VALUE	Return by value
TRIMF	RRTURN	,FAIL	Signal failure

TRIM calls STRVAL (which is shown below) to evaluate its argument as a string. STRVAL operates on the prefix code and has no argument. STRVAL has two exits. Exit 1 signals failure during argument evaluation, causing TRIM to signal failure back to the procedure that invoked it. Exit 2 signals that the argument evaluation was successful, and XPTR points to the result. COMQNV constructs a qualifier for the string, and TRIMQQ modifies this qualifier, decreasing the length as necessary to delete trailing blanks. GENVAR is a procedure that creates a natural variable from a pointer (ZQLPTR) to the trimmed qualifier (ZQL). The resulting value is returned.

STRVAL is a procedure called by function procedures that require string arguments. The STRVAL procedure is

STRVAL	PROC	,	
	INCVC	CODOFF,DWDTH	Increment offset
	GETD	XPTR,CODPOS,CODOFF	Get object code descriptor
	TESTF	XPTR,FFLG,,STRVC	Check for function
STRV1	EQLVC	INVAL,0,,STRV4	Check & INPUT
	LOCBD	ZPTR,INBPTR,XPTR,STRV4	
*			Look for input association
*			through pointer to pair block
	GETDC	ZPTR,ZPTR,DWDTH	Get input association
	RCALL	XPTR,PUTIN,(ZPTR,XPTR),(STRF,STRR)	
*			Perform input
STRV4	GETDC	XPTR,XPTR,DWDTH	Get value
STRV2	EQLTC	XPTR,S,,STRR	Is it STRING?
	EQLTC	XPTR,I,ILLDT,STRI	Is it INTEGER?
STRVC	RCALL	XPTR,INVOKE,XPTR,(STRF,STRV1,STRV2)	
*			Evaluate function
STRI	RCALL	XPTR,GNVAI,XPTR,STRR	
*			Convert integer to natural
*			variable
STRF	RRTURN	,FAIL	Signal failure
STRR	RRTURN	XPTR,RESULT	Return result

STRVAL updates CODOFF and fetches a descriptor. In case of a function, INVOKE

is called from STRVC. If INVOKE signals failure, STRVAL signals failure in return. If INVOKE returns by name, processing continues at STRV1, which checks to see if the input mode is on, and if so whether the variable is input-associated. If it is, PUTIN is called to read a record. If not, the value of the variable is obtained. If the value is a string, it is returned to the calling procedure. If the value is an integer, a corresponding natural variable is obtained by calling GNVARI. If the value is not a string or an integer, control is transferred to an error routine, ILLDT. STRR returns value by exit 2 (RESULT). STRVAL is not a function procedure and has only two exits.

Exercises

10.3.1 What happens as a result of a statement such as

INPUT

which consists of only a subject?

10.3.2 Why is it necessary for INVOKE to compare the argument count in the function descriptor with the argument count in the link descriptor it points to?

10.3.3 What happens if TRIM(2) is executed?

The Implementation of SIL on Two Machines

This chapter discusses some aspects of the implementation of SIL (and hence SNOBOL4) on two types of machines with dissimilar architecture: the IBM 360 and the CDC 6000 series. All the details cannot be given here, but selected topics are covered for each of the machines, showing what reduction to practice amounts to in two situations.

11.1 THE IBM SYSTEM/360 IMPLEMENTATION

The implementation for the IBM 360 is historically interesting since most of the SNOBOL4 and SIL developmental work was done on the 360. A familiarity with the 360 architecture [24], OS assembly language [22], and a rudimentary knowledge of OS [25] is a prerequisite to understanding this material.

11.1.1 Basic Data Units on the IBM 360

The 360 descriptor is a double word (8 bytes). Since the byte is the addressing unit of the 360, DWDTH is 8 and QWDTH is 16. The descriptor is arranged as shown in Figure 11.1.1. The choice of a full word for the V field is a natural one, considering that this corresponds to integers and single-precision reals on the 360. The 360 uses two's-complement arithmetic. Thus, the V field can represent integers in the range -2^{31} to $2^{31} - 1$. Reals have the approximate range 10^{-78} to 10^{75} . All pointers (addresses) fit into this field also, although only the low-order three bytes are used.

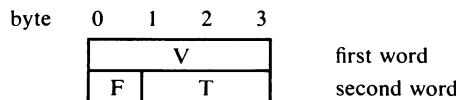


Figure 11.1.1
The IBM 360 descriptor.

The three-byte T field can represent sizes of $2^{24} - 1$ (16,777,215). The T field is used in titles to represent lengths. In natural variables, the string length is given, while in blocks, the size of the block is given. Since the maximum memory size on the main line of the 360 series is 2^{24} bytes, no string can be this long. Similarly the maximum number of (eight-byte) descriptors is $2^{21} - 1$ (2,097,151).

The one-byte F field provides for eight flags, leaving two bit positions for possible future extensions.

The qualifier consists of four words, arranged as shown in Figure 11.1.2. The high-order byte of the L field is never used since the length of strings is limited to $2^{24} - 1$. The F field position of the first descriptor is always zero to prevent the first descriptor from accidentally being mistaken by storage regeneration procedures for a title or a pointer into the allocated data region.

The offset is a character (byte) offset and can be added directly to the pointer in the V field to address the first character of the string. On the IBM 360 there are eight bits per character (256 different characters) and, therefore, eight characters per descriptor for the storage of strings.

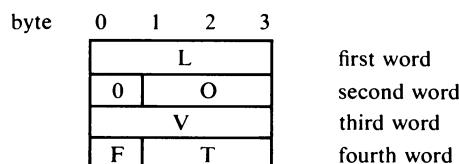


Figure 11.1.2
The IBM 360 qualifier.

Although the 360 is nominally a byte-oriented machine, many operations apply only to words or double words. Since descriptors and qualifiers are composed of double words, little space is wasted by forcing them to begin on double-word boundaries, both in the resident and allocated data regions. Thus, all 360 instructions can be used on these data objects.

11.1.2 Register Usage on the IBM 360

The general-purpose registers are used as follows:

R0	constant 0
R1	scratch
R2	scratch
R3	scratch
R4	scratch
R5	scratch
R6	scratch
R7	scratch
R8	constant 8
R9	resident data region basing
R10	resident data region basing
R11	procedure basing
R12	current stack pointer
R13	old stack pointer
R14	subroutine linkage
R15	subroutine basing and scratch

Registers R0 and R8 remain constant while executing macro-generated code. The constant 0 is useful for clearing fields of descriptors and qualifiers and for comparisons (the V field of the null string is 0, and the end of a natural variable storage chain is indicated by a 0 link).

The constant 8 (DWDTH) is used frequently for incrementing and decrementing pointers as they move sequentially through descriptor locations. The choices of R0 for 0 and R8 for 8 were made for their mnemonic value.

Basing is one of the most troublesome aspects of programming for the 360. The structure of SNOBOL4 written in SIL does little to make the problem easier. Fortunately, no procedure generates too much code for basing by a single register. Within a procedure, R11 is used for basing code and always contains the location of the first instruction of the procedure. (PROC establishes an appropriate USING.) RCALL saves R11 on SYSSTK and sets up the new value for the called procedure. R11 is restored from SYSSTK by RRETURN.

Two registers are required to base the resident data region. R9 and R10, used for this purpose, remain constant while executing macro-generated code. Thus, all directly referenced data are always based. The resident data region is arranged so that the part that has to be based comes first, followed by a part that does not

have to be based. This arrangement is a concession to the 360 architecture made when SNOBOL4 was written in SIL. Common return points and error exits are also based by R9. Hence, these points can be reached by a transfer from any part of the program.

R12 is initially set to point to 12*8 bytes less than the title descriptor of the stack. As R12 is incremented and decremented by stack manipulation operations, it always points 12 descriptors below the actual stack position. This offset permits addressing above and below the current stack position, using only positive displacements as required by the 360. The choice of 12 descriptors for the offset is arbitrary and allows for all required addressing. R12 is modified by RCALL, RRETURN, PUSHD, POPD, PUSHQ, and POPQ. R13 is initially set to zero and then modified by RCALL and RRETURN.

R14 and R15 are used in the conventional ways for linking to and basing subroutines. When R15 is used as a scratch descriptor, it is usually to compute a V field so that the instruction

```
STM      15,0,X
```

can be used to store the V field while at the same time clearing the F and T fields of the descriptor X.

Floating-point registers are frequently used to move data. On most 360 models, the fastest way to move a descriptor from X to Y is

```
LD      0,X
STD     0,Y
```

11.1.3 Examples of 360 Macro Definitions

MOVDD and GETD are two simple macros that illustrate the movement of data. Floating-point instructions are used in MOVDD because they are faster than the more obvious MVC instruction.

```
MACRO
&LOC  MOVDD    &D1,&D2
&LOC  LD       0,&D2
        STD      0,&D1
        MEND
```

GETD illustrates the technique used for indirect address references.

```
MACRO
&LOC  GETD    &D1,&D2,&D3
&LOC  L       1,&D3
        L       2,&D2
        LD      0,0(1,2)
        STD     0,&D1
        MEND
```

INCVC illustrates alternate expansions of code to take advantage of the value of DWDTH in register 8.

```

MACRO
&LOC  INCVC    &D ,&C
&LOC  L         1 ,&D
        AIF      ('&C' EQ 'DWDTH').A
        LA       1 ,&C.(1)
        AGO     . B
.A     AR       1 ,8
.B     ST       1 ,&D
MEND

```

EQLDD illustrates conditional expansions depending on different specifications of branch points.

```

MACRO
&LOC  EQLDD    &D1 ,&D2 ,&F ,&S
&LOC  CLC      &D1.(8) ,&D2
        AIF      (T'&F EQ '0'). A
        BNE     &F
.A     AIF      (T'&S EQ '0'). B
        BE      &S
.B     MEND

```

LOCAD and LOCBP are larger than most macros. These two macros are expanded in line because of the frequency of their use in searching pair blocks. The definition of LOCAD follows. LOCBP is similar.

```

MACRO
&LOC  LOCAD    &D1 ,&D2 ,&D3 ,&F ,&S
&LOC  L         15 ,&D2           set up pointer to pair block
        L         5 ,4(15)
        LA        5 ,0(5,15)
        LR        4 ,8
        SR        5 ,4
        AR        4 ,8
C&SYSNDX CLC    8(8,15) ,&D3      compare descriptors
        BE        D&SYSNDX
        BXLE    15 ,4 ,C&SYSNDX   loop if not equal
        AIF      (T'&F EQ '0'). A
        B         &F
        AGO     . B

```

```

. A      B      A&SYSNDX
. B      ANOP
D&SYSNDX STM      15,0,&D1
              AIF      (T'&S EQ '0').C
              B      &S
              AIF      (T'&F EQ '0').C
A&SYSNDX EQU      *
. C      MEND

```

BRANIN is used to pass control to function and matching procedures. R11 serves both as a transfer register and as a base for the target procedure.

```

MACRO
&LOC  BRANIN  &D
&LOC  L        11,&D
       L        11,0(11)
       BR      11
MEND

```

BRANLV illustrates the generation of a multi-way branch list and the handling of omitted branch points in an argument list.

```

MACRO
&LOC  BRANLV   &D,&BLIST
       LCLA     &N
       LCLA     &M
&LOC  L        1,&D
       SLA      1,2
       B        *+0(1)           branch to list entry
.A    ANOP
&N    SETA     &N+1
       AIF      (T'&BLIST(&N) EQ '0').B
       B        &BLIST(&N)           generate branch
       AGO      .C
.B    B        A&SYSNDX
&M    SETA     &M+1
.C    AIF      (&N LT N'&BLIST).A
       AIF      (&M EQ 0).D
A&SYSNDX EQU      *
.D    MEND

```

PUSHD and POPD make use of assembly control operations to generate the most efficient code for incrementing and decrementing the current stack pointer, depending on the number of arguments involved. In case of stack overflow in PUSHD, control is transferred to an error routine OVER. The definitions are:

MACRO

&LOC	POPD	&DLIST
	LCLA	&NO, &N, &K
&NO	SETA	N'&DLIST
&K	SETA	&NO
	AIF	('&LOC' EQ ''). A
&LOC	EQU	*
. A	AIF	(&NO EQ 0). DONE
&N	SETA	&N+1
	LD	0, 13*8-&N*8(12) move from SYSSTK
	STD	0, &DLIST(&N)
&NO	SETA	&NO-1
	AGO	. A
. DONE	AIF	(&K EQ 1). DONI
	AIF	(&K EQ 2). DONK
	S	12, =A(&K*8) decrement stack pointer
	MEXIT	
. DONI	SR	12, 8
	MEXIT	
. DONK	SR	12, 8
	SR	12, 8
	MEND	

MACRO

&LOC	PUSHD	&DLIST
	LCLA	&NO, &N, &K
&N	SETA	1
&NO	SETA	N'&DLIST
&K	SETA	&NO
	AIF	('&LOC' EQ ''). A
&LOC	EQU	*
. A	AIF	(&NO EQ 0). DONE
	LD	0, &DLIST(&N) move to SYSSTK
	STD	0, 12*8+8*&N. (12)
&NO	SETA	&NO-1
&N	SETA	&N+1
	AGO	. A
. DONE	AIF	(&K EQ 1). DONI
	LA	12, 8*&K. (12) increment stack pointer
	AGO	. DONA
. DONI	AR	12, 8
. DONA	C	12, =A(SYSSTK-12*8+SSTKSZ*8)
	BH	OVER
	MEND	

RCALL is one of the more complex macros in SIL. First, the address of return code to be generated is placed in R14. R11 through R14 are stored on SYSSTK, to save the current procedure base, the current and old stack pointers, and the address of the return code. The old stack pointer is set to the current stack pointer, and the arguments are pushed. The current stack pointer is then incremented to place it past the saved arguments. A test for stack overflow is made at this point. R11 is set up as the base register for the called procedure, and control is transferred to the called procedure. RCALL generates a section of code to perform the assignment to &D, which is subsequently executed by RRETURN. The nature of this code depends on whether or not a value is to be returned. If &D is omitted and no value is to be returned, a fast NOP (SR 0,0) is generated. The second SR is included to align the code that follows. If a value is to be returned, it will be placed in FR0 by RRETURN; therefore, the generated code in this case is an appropriate store. Finally, a sequence of branch instructions is generated for the specified return points.

	MACRO	
&LOC	RCALL	&D ,&P ,&DLIST ,&BLIST
	LCLA	&K ,&L ,&N ,&M ,&NO
&NO	SETA	N'&DLIST
&N	SETA	1
&K	SETA	&NO
&LOC	LA	14,U&SYSNDX
	STM	11,14,13*8(12) save state information
	LR	13,12
. A	AIF	(&NO EQ 0).DONE
	LD	0,&DLIST(&N) put arguments on SYSSTK
	STD	0,14,*8+&NO*8(12)
&NO	SETA	&NO-1
&N	SETA	&N+1
	AGO	.A
. DONE	LA	12,16+&K*8(12) increment stack pointer
	C	12,=A(SYSSTK-12*8+SSTKSZ*8)
	BH	OVER
	L	11,=A(&P) base target procedure
	BR	11 transfer
	AIF	('&D' EQ '').C
U&SYSNDX	STD	0,&D store result
	AGO	.D
. C	ANOP	no operation
U&SYSNDX	SR	0,0
	SR	0,0
. D	AIF	('&BLIST' EQ '').M
&K	SETA	0

```

.H      ANOP
&L      SETA    &L+1
        AIF     (T'&BLIST(&L) EQ '0').F
        B       &BLIST(&L)           generate branch
        AGO    .G
.F      B       V&SYSNDX
&K      SETA    &K+1
.G      AIF     (&L LT N'&BLIST).H
        AIF     (&K EQ 0).M
V&SYSNDX EQU    *
.M      MEND

```

RRTURN works in conjunction with RCALL. If a value is to be returned, it is loaded into FR0. R11 through R14 are restored from SYSSTK, resetting the base for the procedure from which the call was made, resetting the old and current stack pointers, and providing the address of the code generated by RCALL in R14. The instruction pointed to by R14 is executed, and a branch is selected on the basis of the integer code. The definition of RRTURN follows.

```

MACRO
&LOC   RRTURN  &D,&N
        AIF     (T'&LOC EQ '0').C
&LOC   EQU    *
.C      LM      11,14,13*8(13)      restore state information
        AIF     (T'&D EQ '0').B
        LD      0,&D            get descriptor returned
        EX      0,0(14)         execute instruction in RCALL
.B      ANOP
        B       4*&N.(14)        branch to return point
        MEND

```

To illustrate RCALL and RRTURN, consider the following macro calls:

```

RCALL    X,P,(Y,Z),(F,S)
RRTURN  X,1

```

The macro definitions generate the following code for these calls:

```

LA      14,U0001
STM    11,14,13*8(12)
LR     13,12
LD     0,Y
STD    0,14*8+2*8(12)
LD     0,Z
STD    0,14*8+1*8(12)
LA     12,16+2*8(12)
C      12,=A(SYSSTK-12*8+SSTKSZ*8)
BH     OVER

```

	L	11 ,=A(P)
	BR	11
U0001	STD	0 , X
	B	F
	B	S
	LM	11 ,14 ,13*8(13)
	LD	0 , X
	EX	0 ,0(14)
	B	4*1(14)

11.1.4 Computation of Hash Numbers on the 360

The HASH macro computes the chain offsets and order numbers used to divide natural variables into chains and to order them within the chains, as described in Section 8.1.2. The algorithm used on the 360 is based on multiplication [18]. Parts of the string to be hashed are selected, treated like integers, and multiplied together. Selected parts of the result are used for the chain offset and order number. Over a period of time, various frills have been added to improve the quality of the numbers generated. The present algorithm follows.

Two 4-byte (32-bit) integers, I_1 and I_2 , are selected for the multiplication. Suppose the string to be hashed consists of n characters $C_1 \dots C_n$. There are three cases, depending on n .

(a) $1 \leq n \leq 4$: Zeros are added, if necessary, to the right of the string to make up four characters. $I_1 = I_2 = C_1 \dots C_4$.

(b) $5 \leq n \leq 8$: $I_1 = C_1 \dots C_4$, $I_2 = C_{n-3} \dots C_n$.

(c) $n > 8$: The first nonblank character in $C_1 \dots C_n$ is located. Let the index of this character be j . If there is no such character, let $j = 1$. Then $I_1 = C_j C_{j+1} C_{j+2} C_n$ and $I_2 = C_{j+4} \dots C_{j+7}$.

Having selected I_1 and I_2 , the algorithm proceeds as follows:

- (1) I_1 is rotated right 19 bits.
- (2) I_1 and I_2 are multiplied together to get a 64-bit product.
- (3) The product is shifted left 11 bits.
- (4) The high-order 32 bits of the result are taken for the order number.
- (5) The length of the string, n , is shifted left 24 bits and added to the low-order 32 bits.
- (6) The result is shifted right 24 bits.
- (7) Finally, the result is multiplied by 8 (i.e., shifted left 3 bits) to get the chain offset as a multiple of DWDTH.

The order number is a full word and is stored in the combined F and V fields of the chain descriptor. This is a nonstandard use of a descriptor and is possible only because the fields are adjacent and the chain descriptor is never examined by MARK.

In selecting I_1 and I_2 , case (c) deserves explanation. Strings to be hashed are rarely randomly distributed in any sense. Often they display diabolical regularity. In particular, some programs process a large number of fixed-length records (such as assembly-language source cards). Typically, such strings begin and end with blanks. If case (b) is used for such strings, I_1 and I_2 are always the same. Consequently, all these strings are placed on the same chain with the same order number, a disastrous situation. Case (c) is added to avoid this possibility. The particular form of case (c) is convenient and fast on the 360, using a translate-and-test instruction and a syntax table already present for the translator.

Throughout the algorithm, various devices are used in an attempt to improve the quality of the hash numbers. Insertion of the last character in case (c) is included to involve more of the string in the computation. The rotation in step (1) is intended to suppress regularities introduced by the way characters are encoded as bits. The shift in step (3) is designed to make better use of the central part of the product which is, theoretically, more random. Because of the way characters are selected for the multiplication, it is possible for strings of different lengths to produce the same product. The addition of the length to the chain offset is included to separate such strings into different chains.

There is little theoretical basis for the hash algorithm described above. It developed over a period of time, largely as a result of experiment rather than analysis. The many ways in which patterns of string regularity occur in different programs makes the application of analytical techniques difficult. Of course, if the hash algorithm is known, ways can be found to defeat it.

The question of the quality of the generated numbers versus the time it takes to compute them must always be considered. Devices that are fast on the 360 were used where possible. The algorithm given is, therefore, quite machine-dependent.

The actual HASH macro follows. Code is expanded in line since HASH is only used in one place.

MACRO			
&LOC	HASH	&D , &Q	
&LOC	LA	1 , &D	get address of target descriptor
	LR	6 , 1	
	LM	2 , 4 , &Q	get relevant part of qualifier
	LR	7 , 2	save length
	LA	4 , 0(3 , 4)	get address of first character
	C	7 , =F'4'	check for length > four
	BH	A&SYSNDX	if greater, branch
	ST	0 , 4(6)	clear second word of target
	LR	2 , 6	address of first word
	SR	2 , 7	less length
	BCTR	7 , 0	less one for execute

EX	7,B&SYSNDX	move in string
L	5,4(6)	bring back zero-padded string
B	C&SYSNDX	join hash computation
B&SYSNDX	MVC 8(0,2),0(4)	
D&SYSNDX	TRT 0(0,4),0(2)	
E&SYSNDX	MVC 0(0,6),0(1)	
A&SYSNDX	CR 7,8	check for length > 8
	BH G&SYSNDX	if greater, branch
	LA 2,0(7,4)	compute location of last word
	S 2,=F'4'	
	MVC 4(4,6),0(2)	move last word to target
	MVC 0(4,6),0(4)	move first word into target
	L 5,0(6)	bring back string in register
	B C&SYSNDX	join hash computation
G&SYSNDX	S 7,=A(8+1)	subtract 8 plus execute factor
	LR 1,4	set up string for translate
	L 2,=A(FORWRD)	address of stream table
	EX 7,D&SYSNDX	translate off blanks
	MVC 0(8,6),0(1)	move result
	L 5,0(6)	bring string into register
	IC 5,8(4,7)	insert last character
C&SYSNDX	LR 4,5	set up for rotate
	SRDL 4,19	rotate
	M 4,4(6)	multiply strings together
	SLDL 4,11	select center portion
	SLL 7,24	shift length
	ALR 5,7	add into product
	SRL 5,24	shift
	SLL 5,3	multiply by DWDTH
	ST 5,0(6)	store offset
	ST 4,4(6)	store order number
	MEND	

The natural variables shown earlier in Figure 8.1.3 are actual examples from the 360 implementation. HASH computes a chain offset of 112*8 for these strings. The order numbers are:

SYNTAX	-788269302
CYLINDER	266747855
SEVENTH	1236499812
RECTOR	1424976985

Note that the use of a full word permits negative numbers.

11.1.5 Syntax Tables and STREAM on the IBM 360

Syntax tables constitute one of the most machine-dependent aspects of SIL. The problem lies largely in the fact that the number of different characters, as well as their collating sequence, varies from machine to machine. The actual form of the syntax tables is chosen by the implementor of SIL. A syntax table for the 360 consists of a 256-byte translate-and-test (TRT) string and associated instructions. Each position in the table corresponds to one of the 256 possible characters. For each position, the byte determines the action for the corresponding character. X'00' occurs for CONTIN in the table description and is the condition on which the TRT instruction continues processing. Other bytes specify offsets of instructions following the table itself. In general, there are two things to be taken care of when a character is encountered: a PUT and a next table or stop condition. For each such situation, as specified in the syntax table description, there are two instructions following the table. When the TRT is stopped by a nonzero byte, B, the value of B is used to select two instructions to execute, using the EX instruction. The instructions are located at 256-4+B bytes and 256+B bytes from the beginning of the table.

Two typical tables, ELEMNT and INTGER, illustrate these features.

The TRT tables and the associated instructions are generated by a SNOBOL4 program, using the syntax table descriptions as input.

STREAM uses R7 for the value to be PUT in the V field of STYPE and R3 for the address of the next table to be used. To illustrate how these tables are used, consider the qualifier shown in Figure 11.1.3. Suppose that the macro instruction

```
STREAM Q1,Q2,ELEMNT,ERR,RUN,OKAY
```

is executed. On the first access to ELEMNT, the TRT instruction stops with the byte X'04', which is in position X'F1', corresponding to the internal code X'F1' for the digit 1. Thus, the instructions at ELEMNT+256 and ELEMNT+260 are executed, loading R7 with INTCOD and R3 with INTGER. STREAM continues with INTGER. The bytes for positions X'F3' and X'F7' (corresponding to 3 and 7) are zero; thus, the TRT continues. The position X'6E' causes the TRT to stop with a byte offset of X'04'. Execution of the branch to STRMSHRT causes a transfer to the location in STREAM that stops the streaming process without including the character `>`. No second instruction is executed because of the branch. Since the STREAM ended on the STOPSH condition, control is transferred to OKAY. Figure 11.1.4 illustrates the resulting values of the qualifiers Q1 and Q2.

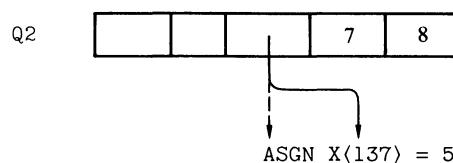


Figure 11.1.3
A qualifier for STREAM.

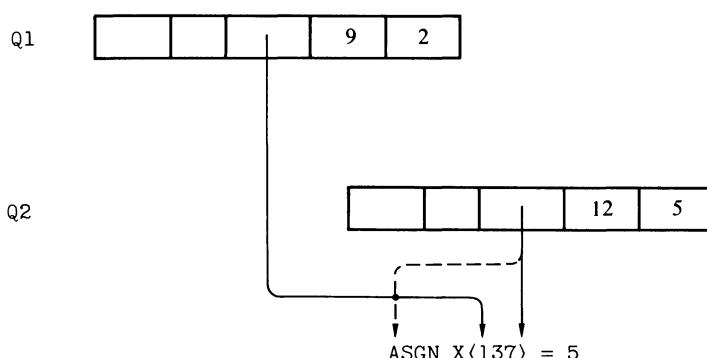


Figure 11.1.4
Result of a STREAM operation.

STREAM occurs in many places in SNOBOL4 and expands into a simple calling sequence to a subroutine. For the example above, the code generated is

```

LA      15,=A(STREAM)
LA      1,Q1
LA      2,Q2
L      3,=A(ELEMNT)
BALR    14,15
B      ERR
B      RUN
B      OKAY

```

The STREAM subroutine follows.

```

USING   STREAM,15
*      on entry, R1 points to Q1, R2 to Q2, and R3 to the table
STREAM   LM      4,7,0(2)          get Q2
          STM    1,2,STTEMP        save qualifier pointers
          STM    4,7,0(1)          initialize Q1
          LTR    4,4              check for null string
          BZ     4(14)           signal run out if null
          LA     1,0(5,6)         compute address of first
*                                character
          BCTR   1,0
          AR     4,1
          BCTR   4,0
          LR     7,0              initialize R7 for STYPE
          B      STRMTEST        join processing loop
STRMLOOP LR     2,0              zero R2 for TRT
          C      5,=F'256'        check length
          BL     STRMSMAL        handle short string elsewhere
          TRT    1(256,1),0(3)    translate 256 bytes
          BNZ    STRMJJOIN       check for CONTIN
          LA     1,256(1)         update pointer to first
*                                character
          B      STRMTEST        continue processing
STRMSMAL EX     5,STRMTRT      translate rest of
                      string
          BZ     STRMOUT         if CONTIN, run out
STRMJOIN AR     2,3              add offset to table address
          EX     0,256-4(2)       execute first instruction
          EX     0,256(2)         execute second instruction
STRMTEST LR     5,4              check for run out
          SR     5,1
          BNM    STRMLOOP        continue processing

```

STRMOUT	L	1,=A(STYPE)	this is run out exit
	ST	7,0(1)	set V field of STYPE
	L	2,STTEMP+4	
	ST	0,0(2)	zero length of Q1
	B	4(14)	signal run out
STRMSHRT	BCTR	1,0	on STOPSH skip last character
STRMSTOP	SR	4,1	
	LA	4,1(4)	
	LA	5,1(1)	
	SR	5,6	
	LM	2,3,STTEMP	
	STM	4,5,0(3)	set length and offset of Q2
	L	3,0(2)	
	SR	3,4	
	ST	3,0(2)	set length of Q1
	L	1,=A(STYPE)	
	ST	7,0(1)	set V field of STYPE
	B	8(14)	signal normal termination
STRMTRT	TRT	1(0,1),0(3)	TRT executed remotely
STTEMP	DC	2F'0'	local storage for qualifiers

11.1.6 Interfacing the Environment on the IBM 360

The 360 version of SNOBOL4 was implemented to run under OS. The interface with OS consists of: (1) gaining control from OS, (2) seizing space for the allocated data region, (3) FORTRAN I/O, (4) loading and unloading external functions, (5) time and date calls, (6) interrupt handling, and (7) returning control to OS on termination.

The interface can be considered in three parts according to the time when the interaction occurs: (1) initialization, (2) execution, and (3) termination.

Initialization is accomplished by the INITEX macro. INITEX executes the OS SAVE macro and bases the program on R11 and the resident data region on R9 and R10. A save area is set up, and the FORTRAN I/O initialization procedure is called. The allocated data region is obtained by seizing space from the user region. There are three parameters that govern the size: (1) H, which establishes the highest value acceptable for the size of the allocated data region; (2) L, giving the lowest value for the size of the allocated data region; and (3) R, giving the minimum amount of space to be reserved for other uses such as I/O buffers and external functions.

These parameters have the default values of 200,000, 35,200, and 10,000 bytes, respectively. The user may specify different values in the PARM field of his EXEC

card. A variable-length GETMAIN is issued to obtain the maximum amount of space between H+R and L+R bytes. R bytes are returned by a FREEMAIN.

Next, a SPIE is issued to give SNOBOL4 control over program interrupts. The program mask is then set to mask interrupts from fixed-point overflow and floating-point significance. Masking fixed-point overflow permits detection of overflow in a source-language statement by testing the condition code. Unfortunately, exponent overflow cannot be masked. FORTRAN I/O routines require significance to be masked.

Finally, INITEX issues an STIMER call which starts task timing and specifies the maximum possible timing interval.

During execution, input and output are done using standard calling sequences to FORTRAN I/O routines. Source-language output uses default formats or formats provided by execution of the built-in function OUTPUT. These formats are strings which are treated by FORTRAN routines as unprocessed formats in arrays. FORTRAN routines take care of opening files when they are first referenced, allocating buffers, and so on.

The time is obtained when needed by issuing a TTIMER call which gives the time remaining from the STIMER call. From this, the elapsed time is computed. If the date is required, it is obtained using the OS TIME macro.

The external function mechanism uses the OS facilities for loading modules from libraries specified in job control statements. The LOAD function calls the OS LOAD macro to bring the desired function into core. The entry point returned by the LOAD macro is used when calling the external function. UNLOAD calls the OS DELETE macro. The responsibility for allocating space and keeping track of use counts for loaded modules is left to OS.

Program interrupts are intercepted by a SPIE routine in the SNOBOL4 system. The SPIE routine stores registers for system-debugging purposes, assures that the program and resident data region are based, and transfers to an error routine in SNOBOL4. This error routine prints a message and processes SNOBOL4 program termination in the usual way. If the SPIE routine is reentered, a user ABEND is issued. Most program interrupts result from errors in the SNOBOL4 system. Exceptions are exponent overflow and underflow, which may occur as a result of source-language operations.

Program execution is terminated by the SIL macro TERMEX, which calls the FORTRAN I/O exit routine to close the files and return to the operating system.

11.1.7 Implementation Problems on the IBM 360

Problems and difficulties experienced in implementing SIL occur for the same general reasons on most machines: (1) hardware architecture, (2) implementation software (assemblers and utilities), (3) FORTRAN support routines, and (4) system facilities and interface.

Some problems are simply annoyances to the implementor of SIL. Others affect the speed and size of the resulting SNOBOL4 system. Some have direct effects on SNOBOL4 programmers. Basing is probably the most vexing problem encountered in implementing SNOBOL4 written in SIL. Although SNOBOL4 as written in SIL is designed to be machine-independent, several concessions to 360 basing were made in structuring the system. The major problem is the large amount of common data in the resident data region that must always be based. The resident data region is, in fact, organized with the based data at the beginning. This organization produces some unlikely divisions and combinations of data that are perplexing to implementors for machines which do not have basing problems.

The character instructions of the 360 are tantalizing to the SNOBOL4 implementor. Unfortunately, these instructions are severely limited in the size of string that can be handled. As a result, there is nearly as much special code required to treat the general case as there would be if the machine had only word-oriented instructions. The IBM 370 line alleviates some of these problems. While the character instructions are generally faster than other instructions which may be used for handling long strings, it is interesting to note that a floating-point load and store is the fastest way of moving a descriptor on most 360 models.

Somewhat bothersome is the fact that some interrupts can be masked, but other functionally similar interrupts cannot. Thus, fixed-point overflow is easily treated as a SNOBOL4 programming error, but floating-point overflow is not.

As an implementor of SIL on the 360, the author has many vivid memories of the 360 implementation software, especially in its early phases. Only one limitation of the OS assembler significantly affected SIL, however. The inability to handle nested argument lists imposed a simplicity of argument structure that would not have been required by other assemblers.

The largest single difficulty encountered on the 360, and the one that has the most residual effect on the SNOBOL4 programmer, is the use of FORTRAN I/O routines. The size of these routines alone is a problem. The relatively slow speed in performing I/O is apparent also. The lack of support for the full range of OS data facilities has caused users serious problems of compatibility with other OS processors. Finally, interfacing SNOBOL4, FORTRAN, and OS has always been a problem. A typical example is the difficulty that arises when an error occurs, giving the SNOBOL4 user a FORTRAN diagnostic which is usually meaningless to him. It probably was a mistake to borrow an I/O system from a processor that is convinced of its autonomy.

The interface with OS is bothersome mainly because SNOBOL4 cannot regain control in case of certain kinds of errors. Storage management is an example. If there is not enough room to load an external function or to provide a buffer for a newly opened file, OS terminates the run, leaving SNOBOL4 no opportunity to regain control or even provide a reasonable error message. Job cancellation is a similar problem, but, in addition, buffers are not even flushed. While con-

tinuing improvement of OS has alleviated many of these problems, the SNOBOL4 users may still be cut unceremoniously by the operating system with little information about the cause of the difficulty.

Exercises

11.1.1 Write a definition for the LOCBD macro.

11.1.2 Write a definition for MOVTV.

11.1.3 Write a definition for PUTD.

11.1.4 Expand the macro call

BRANLV D, (L1, L2, , L3)

11.1.5 RCALL saves R11 through R14 on SYSSTK. Explain what this amounts to in terms of the descriptor structure of SYSSTK.

11.2 THE CDC 6000 SERIES IMPLEMENTATION

The implementation for the CDC 6000 series machines (hereafter referred to as the CDC 6000) was the first undertaken after the IBM 360 and was developed parallel to the 360 implementation as SIL evolved. A familiarity with the CDC 6000 architecture [26], COMPASS assembly language [23], and the SCOPE operating system [27] is prerequisite to a full understanding of the material that follows.

11.2.1 Basic Data Units on the CDC 6000

The 6000 descriptor is one word (60 bits). Since the word is the addressing unit, DWDTH is 1 and QWDTH is 2. The general format of a descriptor is shown in Figure 11.2.1.

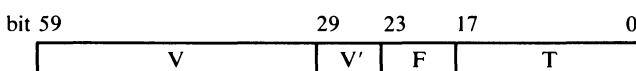


Figure 11.2.1
The CDC 6000 descriptor.

The V field must be capable of containing a relocatable address. The SCOPE loader requires relocatable address fields, which are 18 bits wide, to end on bit 30, 15, or 0. Integers are stored in one's-complement form in the V field and can be used in arithmetic, including address arithmetic, without shifting. The range of

value is from $-2^{29} + 1$ to $2^{29} - 1$. All pointers (addresses) fit into this field also, although only the low-order 18 bits can be used, and seldom are more than 15 or 16 actually used.

Real numbers are stored in a 36-bit field obtained by adding a 6-bit extension, designated the V' field, to the V field. Figure 11.2.2 illustrates the descriptor for a real number.

bit 59	23	17	0
real number	0	R	

Figure 11.2.2
The CDC 6000 descriptor for real numbers.

The real value is a standard CDC 6000 floating-point number with the fraction part rounded to 24 bits. Because binary rather than hexadecimal normalization is used, precision is slightly better than the IBM 360 single-precision floating point. Reals have a magnitude range of approximately 10^{-294} to 10^{322} .

The 18-bit T field can represent values from $-2^{17} + 1$ to $2^{17} - 1$, in one's-complement form. Normally, only the positive values are used, but the choice of an 18-bit width is natural because of the "set" instructions of the CDC 6000 which perform 18-bit integer arithmetic and 18-bit field isolation. Since the maximum number of central memory words available on the CDC 6000 is 2^{17} , this is not a restriction on the number of descriptors in a data object. There are 6 bits per character (64 different characters) and, therefore, 10 characters per descriptor for the storage of strings. As a result, there may be enough memory to create strings longer than the maximum allowed of $2^{17} - 1$ (131,071), but this is not a significant restriction.

The remaining 6-bit F field accommodates the flags. Additional bits could be taken from the V' field at the expense of real precision if necessary.

The qualifier consists of two words arranged as shown in Figure 11.2.3. The L field, as indicated above, restricts the length of strings to $2^{17} - 1$ characters. The F field in the first descriptor is always zero to prevent the first descriptor from accidentally being mistaken for a pointer into the allocated data region.

The offset consists of two parts, a word offset (WO) and a bit offset (BO). The word offset can be added directly to the pointer in the V field to address the word containing the first character of the string. The bit offset is the number of

bit 59	47	29	23	17	0	
BO	WO		0	L		first word
0	V	0	F	T		second word

Figure 11.2.3
The CDC 6000 qualifier.

bits the word must be rotated left to obtain the first character of the string in the high-order position of a register. The bit and word offset are computed from the character offset when a qualifier is assembled or computed during execution.

The following section of code indicates how the bit and word offsets are used to access the first character of a string.

SA1	Q	GET QUALIFIER, 1ST WORD
SA2	A1+B1	. 2ND WORD
IX2	X1+X2	COMPUTE STARTING WORD ADDRESS
AX2	30	POSITION IT
SA3	X2	PICK UP FIRST WORD
AX2	18	POSITION BIT OFFSET
SB2	X2	SET IT UP AS SHIFT COUNTER
LX3	B2,X3	ROTATE CHARACTER TO HIGH BITS

At this point, register X1 has the length in the low-order 18 bits where it can be isolated with a set-register instruction. X2 and B2 both contain the number of bits of the word “used up” (if subtracted from 60, the number remaining is obtained). X3 contains the first character of the string, left-adjusted. From one through ten characters of the string are in X3.

11.2.2 Register Usage on the CDC 6000

The registers are used as follows:

B0	constant 0
B1	constant 1
B2	indexing and branching scratch
B3	indexing and branching scratch
B4	indexing and branching scratch
B5	indexing and branching scratch
B6	constant 30
B7	current stack pointer
X0	high-order, 30-bit (halfword) mask
X1	operand scratch (may be fetched from memory)
X2	operand scratch (may be fetched from memory)
X3	operand scratch (may be fetched from memory)
X4	operand scratch (may be fetched from memory)
X5	operand scratch (may be fetched from memory)
X6	operand scratch (may be stored in memory)
X7	operand scratch (may be stored in memory)
A0	scratch
A1	fetching address for X1

A2	fetching address for X2
A3	fetching address for X3
A4	fetching address for X4
A5	fetching address for X5
A6	storing address for X6
A7	storing address for X7

Register B0 is constrained to be zero by the hardware. This fact is used by many machine instructions. The constant 1 is used frequently for incrementing and decrementing pointers as they move sequentially through descriptor locations. The choice of B1 was made for its mnemonic value.

The constant 30 in B6 is used frequently in moving a quantity from one X register to another and, simultaneously, in shifting and rotating it. The instruction

LX2 B6 , X1

for example, moves the contents of register X1 to register X2, at the same time transposing the two halves of a word by left rotation.

The halfword mask in X0 is used for obtaining either half of a word in one X register from another. The instruction

BX2 X0 * X1

fetches the high-order 30 bits (V field of a descriptor) from X1 into X2 while clearing the low-order bits (F and T fields). Similarly,

BX3 -X0 * X1

uses the complement of the mask to fetch the low-order 30 bits into X3 while clearing the high-order bits. Further discussion of register usage is contained in Section 11.2.7.

Basing is virtually no problem on the CDC 6000. Each program is allocated space as a contiguous block of memory, with a reference address (RA) base register and a field length (FL) register set by the operating system. All addressing in the program appears to be absolute, but, in fact, it is relative to the value of the RA register. The return jump (RJ) instruction is used for most subprogram linkage, with occasional branching done using the B registers.

The stack pointer, B7, is initially set to point to the first descriptor of SYSSTK. B7 is modified by RCALL, RRETURN, PUSHD, POPD, PUSHQ, and POPQ. A word named OLDSTACK is used to store the pointer before updating by RCALL and RRETURN.

11.2.3 Examples of CDC 6000 Macro Definitions

The following CDC 6000 macro definitions, the same as given for the IBM 360, are typical. For the most part, they are self-explanatory. Several macros invoke other macros, which are also included.

MOVDD moves a descriptor by loading it into an X register, moving it to another register, and storing it from that register. The move is required because loads can be made only into X1, X2, X3, X4, or X5, and stores can be made only from X6 or X7.

```
MOVDD MACRO D1,D2
        SA1      D2
        BX6      X1
        SA6      D1
        ENDM
```

GETD shows the use of the shift with count in the register (AXn B6,Xm) to obtain the V field.

```
GETD MACRO D1,D2,D3
        SA1      D2
        SA2      D3
        AX3      B6,X1
        AX4      B6,X2
        IX5      X3+X4
        SA1      X5
        BX6      X1
        SA6      D1
        ENDM
```

INCVC demonstrates the SETREG macro, used to set an 18-bit value in a register either from a constant (the case shown) or from a constant and the contents of a register.

```
INCVC MACRO D,C
        SA1      D
        SETREG  X2,C
        LX3      B6,X2
        IX6      X1+X3
        SA6      A1
        ENDM
```

** SETREG (USED BY SEVERAL OTHER MACROS)
 * SET REGISTER REG1 TO CONSTANT VALUE VAL OR TO THE CONTENTS
 * OF REGISTER REG2 (IF SPECIFIED) PLUS VAL.

```
SETREG MACRO REG1,VAL,REG2
Q     IFC    EQ,/REG2//
P     IF     DEF,VAL
      IFEQ   VAL,0,2
      S>REG1 B0
Q     IFNE
      IFEQ   VAL,1,2
      S>REG1 B1
```

```

Q      IFNE
      IFEQ      VAL,2,2
      S>REG1   B1+B1
Q      IFNE
      IFEQ      VAL,-1,2
      S>REG1   B0-B1
Q      IFNE
P      ENDIF
      S>REG1   VAL
Q      ENDIF
R      IFC      NE,/REG2///
S      IF      DEF,VAL
      IFEQ      VAL,0,2
      S>REG1   REG2+B0
R      IFNE
      IFEQ      VAL,1,2
      S>REG1   REG2+B1
R      IFNE
S      ENDIF
      S>REG1   REG2+VAL
      ENDIF
      ENDM

```

EQLDD illustrates conditional branching, based on the contents of an X register. The macro TESTX generates the appropriate sequence of conditional instructions, depending on the branches specified.

```

EQLDD    MACRO    D1,D2,F,S
          SA1      D1
          SA2      D2
          BX3      X1-X2
          TESTX    3,F,S,F
          ENDM

```

```

**      TESTX (USED BY SEVERAL OTHER MACROS)
*
TEST REGISTER X.I AND BRANCH IF GREATER, EQUAL, OR LESS
*
THAN ZERO. FALL THROUGH IF APPROPRIATE CONDITION IS NOT
*
SPECIFIED.
TESTX    MACRO    I,G,E,L
          LOCAL    Z
TGL     IFC      EQ,/G/L/
          IFC      NE,/G//,1
          NZ      X.I,G

```

```

        IFC      NE,/E//,1
        ZR      X.I,E
TGL    ENDIF
TGLX   IFC      NE,/G/L/
        IFC      NE,/E//,1
        ZR      X.I,E
        IFC      EQ,/E//,1
        ZR      X.I,Z
        IFC      NE,/L//,1
        NG      X.I,L
        IFC      NE,/G//,1
        PL      X.I,G
        IFC      EQ,/E//,1
Z      BSS      0
TGLX   ENDIF
        ENDM

```

LOCAD and LOCBD both use a macro LOCAUBD, which calls the subroutine LOCAUBD to perform the action. The system implementor can move the code in line if he desires to make the time/space trade-off using more space. LOCBD varies from LOCAD only in that register B3 is set to 2 instead of 1 before LOCAUBD is called.

```

LOCAD   MACRO    D1,D2,D3,F,S
        LOCAUBD  D1,D2,D3,F,S,B1
        ENDM

**      LOCAUBD (USED BY MACROS LOCAD AND LOCBD)
*
LOCATE A OR B. REGISTER B3 CONTAINS 1 IF A OR 2 IF B.
LOCAUBD MACRO    D1,D2,D3,F,S,N
        LOCAL    Z
        SA2     D2
        SA3     D3
        SB3     N
        RJ      LOCAUBD
        IFC     NE,/F//,1
        ZR      B2,F
        IFC     EQ,/F//,1
        ZR      B2,Z
        SA6     D1
        IFC     NE,/S//,1
        EQ      S
        IFC     EQ,/F//,1
Z      BSS      0
        ENDM

```

```
**
** LOCAUBD (ROUTINE CALLED BY MACRO LOCAUBD)
*
* ENTRY      X2 CONTAINS D2
*             X3 CONTAINS D3
*             B2 = 1 IF CHECK IS OF A FIELDS
*                   2 IF CHECK IS OF B FIELDS
*
* EXIT       B2 = 0 IF FAILURE TO LOCATE
*             NON-ZERO IF ENTRY IS LOCATED
*             X6 CONTAINS D1 IF REQUEST IS LOCATED
*
* ENTRY POINT IS LOCAUBD
*
LOCAUBDM SX1      A1-B3          MATCH
             LX3      B6,X1          SET UP RESULT
             BX4      -X0*X2
             BX6      X3+X4          FALL THROUGH TO EXIT
LOCAUBD  BSSZ     1              ENTRY/EXIT
             AX4      B6,X2          GET ADDRESS FROM D2
             SB2      B1+B1          SET UP CONSTANT 2
             SA1      X4              GET ITEM REFERENCED BY D2
             SB4      X1              LIMIT
             SA1      A1+B3          PICK UP FIRST ENTRY
             SB5      B1              COUNTER, SET TO 1
LOCAUBDL BX5      X1-X3          CHECK FOR FLAG
             SB5      B5+B2          ADJUST COUNTER TO NEXT VALUE
             ZR      X5,LOCAPM        FOUND MATCH
             SA1      A1+B2          PICK UP NEXT VALUE
             LT      B5,B4,LOCAUBDL   LOOP IF NO MATCH AND NOT DONE
LOCAUBDF SB2      B0              FAIL TO MATCH
             EQ      LOCAUBD         RETURN
```

BRANIN passes control to function and matching procedures by loading B2 indirectly from the descriptor and using it for a branching address.

BRANIN	MACRO	D
	SA1	D
	AX2	B6,X1
	SA3	X2
	AX4	B6,X3
	SB2	X4
	JP	B2
	ENDM	

BRANLV illustrates multiple branching and the use of recursive macro calls to generate the list of branch points.

The macro GOLIST used by BRANLV to generate the list depends in part on the SIL labels not exceeding six characters.

```

BRANLV  MACRO      D,LOCN
        LOCAL      LIST,EXIT
        SA1       D
        AX2       B6,X1
        SA3       X2+LIST-1
        SB2       X3
        JP        B2
LIST     BSS        0
        GOLIST    EXIT,LOCN,ENDLIST
+
VFD      60/EXIT
EXIT    BSS        0
        ENDM

GOLIST  MACRO      EXIT,A,B,C,D,E,F,G,H,I,J,K,L
GOLST   IFC        NE,/A/ENDLIST/
        IFC        NE,/A//,1
+
VFD      60/A
        IFC        EQ,/A//,1
+
VFD      60/EXIT
        GOLIST    EXIT,B,C,D,E,F,G,H,I,J,K,L,ENDLIST
GOLST   ENDIF
        ENDM

```

POPD and PUSHD again use recursion on lists in order to generate the appropriate code. In PUSHD, the descriptor being pushed is checked to see if it is the constant zero descriptor ZEROCL, and, if so, a constant zero is generated in the register rather than fetching it from memory.

```

POPD    MACRO      A
        POP1      A
        SB7       A1-B1
        ENDM

POP1    MACRO      A,B,C,D,E,F,G,H
        SA1       B7
        BX6       X1
        SA6       A
        IFC        NE,/B//,1
        POP2      B,C,D,E,F,G,H
        ENDM

```

POP2	MACRO	A,B,C,D,E,F,G
	SA1	A1-B1
	BX6	X1
	SA6	A
	IFC	NE,/B//,1
	POP2	B,C,D,E,F,G
	ENDM	
PUSHD	MACRO	A
	PUSH1	A
	SB2	SYSSTK+SSTKSZ
	SB7	A6
	GE	B7,B2,OVER
	ENDM	
PUSH1	MACRO	A,B,C,D,E,F,G,H
	IFC	EQ,/A/ZEROCL/,1
	BX6	X6-X6
	IFC	NE,/A/ZEROCL/,2
	SA1	A
	BX6	X1
	SA6	B7+B1
	IFC	NE,/B//,1
	PUSH2	B,C,D,E,F,G,H
	ENDM	
PUSH2	MACRO	A,B,C,D,E,F,G
	IFC	EQ,/A/ZEROCL/,1
	BX6	X6-X6
	IFC	NE,/A/ZEROCL/,2
	SA1	A
	BX6	X1
	SA6	A6+B1
	IFC	NE,/B//,1
	PUSH2	B,C,D,E,F,G
	ENDM	

As in the 360 implementation, RCALL is one of the most complex macros, using three additional macros to generate argument and return lists. First, the old stack pointer (OLDSTACK) is pushed onto SYSSTK and the current stack pointer (register B7) is made the old stack pointer. Then a descriptor is pushed onto SYSSTK containing the return address in the T field and the desired result address in the V field. If no result is desired, the V field is zero. Then the arguments are pushed onto SYSSTK in reverse order, using macros RCARGS and RCARG. A branch is then made to the desired procedure. The macro RCRTNS

generates an array of returns, pointing either to the desired branch or to the next instruction. One additional address is provided for the fall-through to the next instruction, as in BRANLV.

RCALL	MACRO	RESULT , PROC , ARGS , RTNS
	LOCAL	EXIT , RETURNS
	SA1	OLDSTACK
	SX7	B7
	SA7	A1
	SX6	X1
	SA6	B7+B1
	IFC	EQ , /RESULT// , 1
	SX6	RETURNS
	IFC	NE , /RESULT// , 2
	SA1	RETURNS-1
	BX6	X1
	SA6	A6+B1
	IFC	NE , /ARGS// , 1
	RCARGS	ARGS
	SB7	A6
	SB2	SYSSTK+SSTKSZ-10
	GE	B7 , B2 , OVER
	EQ	PROC
	IFC	NE , /RESULT// , 1
+	VFD	30/RESULT , 30/RETURNS
RETURNS	BSS	0
	IFC	NE , /RTNS// , 1
	RCRTNS	EXIT , RTNS , ENDLIST
+	VFD	60/EXIT
EXIT	BSS	0
	ENDM	
RCARGS	MACRO	A , B , C , D , E , F , G , H
	IFC	NE , /H// , 1
	RCARG	H
	IFC	NE , /G// , 1
	RCARG	G
	IFC	NE , /F// , 1
	RCARG	F
	IFC	NE , /E// , 1
	RCARG	E
	IFC	NE , /D// , 1
	RCARG	D
	IFC	NE , /C// , 1
	RCARG	C

```

IFC      NE,/B//,1
RCARG    B
IFC      NE,/A//,1
RCARG    A
ENDM

RCARG    MACRO   Q
          SA1     Q
          BX6     X1
          SA6     A6+B1
ENDM

RCRTNS   MACRO   EXIT,A,B,C,D,E,F,G,H
          IFC     NE,/A//,1
+
          VFD     60/A
          IFC     EQ,/A//,1
+
          VFD     60/EXIT
          IFC     NE,/B/ENDLIST/,1
          RCRTNS  EXIT,B,C,D,E,F,G,H,ENDLIST
          ENDM

```

RRTURN is used in conjunction with RCALL to return to the calling operation with a possible result and a branching location. The current stack pointer is re-loaded from the old stack pointer which is re-loaded from SYSSTK. Then the return word, containing the addresses of the desired result and the return list, is loaded for SYSSTK. If a result is available and desired, it is moved to the result location. Then the appropriate branching address is obtained and a return is made.

```

RRTURN   MACRO   RESULT,N
          LOCAL   NORESULT
          SA1    OLDSTACK
          SA2    X1+B1
          SB7    X1
          SA3    A2+B1
          SX7    X2
          SA7    A1
          IFC    NE,/RESULT///
          AX4    B6,X3
          ZR     X4,NORESULT
          SA1    RESULT
          BX6    X1
          SA6    X4

```

```

NORESULT BSS      0
ENDIF
SETREG    A1,N-1,X3
SB2       X1
JP        B2
ENDM

```

These macros are illustrated by the following calls:

```

RCALL      X,P,(Y,Z),(F,S)
RRETURN    X,l

```

The code generated is:

```

SA1      OLDSTACK
SX7      B7
SA7      A1
SX6      X1
SA6      B7+B1
SA1      ↑↓000002-1
BX6      X1
SA6      A6+B1
SA1      Z
BX6      X1
SA6      A6+B1
SA1      Y
BX6      X1
SA6      A6+B1
SB7      A6
SB2      SYSSTK+SSTKSZ-10
GE       B7,B2,OVER
EQ       P
+       VFD      30/X,30/↑↓000002
↑↓000002 BSS      0
+       VFD      60/F
+       VFD      60/S
+       VFD      60/↑↓000001
↑↓000001 BSS      0

SA1      OLDSTACK
SA2      X1+B1
SB7      X1
SA3      A2+B1
SX7      X2
SA7      A1
AX4      B6,X3
ZR       X4,↑↓000003

```

SA1	X
BX6	X1
SA6	X4
↑↓000003 BSS	0
S→A1	X3+B0
SB2	X1
JP	B2

11.2.4 Computation of Hash Numbers on the CDC 6000

The CDC implementation of the HASH operation uses a slightly less sophisticated procedure than the IBM 360. This simpler procedure can be used in part because of the smaller character set. A higher proportion of the total character set is likely to appear within a string on the CDC 6000. The chain offset is computed from the initial characters of the string. The order number is computed from the length and initial characters.

For a one-character string, the chain offset is four times the binary value of the character, and the order number is zero. For strings of two or more characters, the chain offset is four times the binary value of the first character plus the value of the high-order two bits of the second character.

The order number is computed from the first ten characters of the string, filled out with binary zero values if the length is less than ten. The first five characters are exclusive-ored with the second five characters. The result is converted to a normalized floating-point value which is multiplied by a scatter constant. Bits 30 through 37 of the result are used as the low-order bits of the value, and the length as the high-order bits. The low-order 17 bits are extracted for the order number. The result is returned in register X7 in descriptor format.

The HASH macro and procedure follow

```
**      HASH
HASH    MACRO      D , Q
        SA1       Q
        SA2       A1+B1
        RJ        HASH
        SA6       D
        ENDM

        TITLE     HASH -- COMPUTE HASH NUMBER PAIR FOR STRING
*
*      ENTRY     X1 AND X2 CONTAIN QUALIFIER FOR STRING
*
*      EXIT      X6 CONTAINS A DESCRIPTOR WITH N1 AND N2 IN THE
*                  VALUE AND TAG FIELDS RESPECTIVELY
*
```

HASH	BSSZ	1	ENTRY/EXIT
	SB2	X1	LENGTH OF STRING
	IX2	X1+X2	COMPUTE STARTING WORD ADDRESS
	AX2	B6,X2	GET ADDRESS
	SA1	X2	PICK UP FIRST WORD OF TEXT
	AX2	18	POSITION BIT OFFSET
	EQ	B2,B1,HASHONEC	ONE CHARACTER STRING
	SB5	10	CHARACTERS PER WORD
	ZR	X2,HASHA	TEXT IS NOT OFFSET FROM START
	SB3	X2	GET BIT OFFSET
	SA2	A1+B1	PICK UP (POSSIBLE) 2ND WORD
	MX3	1	FORM MASK WORD
	SB4	B3-B1	.
	AX3	B4,X3	.
	BX1	-X3*X1	GET REMAINING CHARS FROM WORD
	BX2	X3*X2	AND CHARS FROM NEXT WORD
	BX1	X1+X2	PUT THEM TOGETHER
	LX1	B3,X1	1ST CHAR TO TOP OF WORD
HASHA	GE	B2,B5,HASHB	CHECK IF FULL WORD IS USED
	SA2	HASHMASK+B2	NO - MASK OFF UNUSED CHARS
	BX1	-X2*X1	.
HASHB	MX2	8	FORM 8-BIT MASK
	BX7	X1*X2	N1=CHAR1*4 + 2 BITS OF CHAR2
	LX7	38	POSITION TO ADDRESS FIELD
	LX2	8	POSITION MASK
	BX3	X0*X1	GET TOP HALF OF TEXT WORD
	BX5	-X0*X1	GET BOTTOM HALF OF TEXT WORD
	AX4	B6,X5	SHIFT TOP HALF TO BOTTOM POSN
	BX6	X4-X3	EXCLUSIVE OR THE PARTS
	SA1	HASHSCAT	PICK UP SCATTER CONSTANT
	PX3	X6	FLOAT TEXT RESULT
	NX4	X3	NORMALIZE IT
	FX5	X4*X1	MULTIPLY BY SCATTER CONSTANT
	SX4	B2	GET LENGTH
	AX5	30	PICK SOME HIGH ORDER BITS
	MX1	43	FORM RESULT MASK
	LX4	8	LENGTH TO HIGHER POSITION
	BX5	X2*X5	EXTRACT HASH BITS
	IX4	X4+X5	LENGTH*2**8 + HASH BITS
	BX2	-X1*X4	USE ONLY LOW-ORDER 17 BITS
	BX6	X2+X7	FORM RESULT
	EQ	HASH	RETURN

```

*
*      ONE CHARACTER STRING -- HANDLE SPECIALLY
*
HASHONEC ZR      X2 , HASHONED      NO BIT OFFSET
                  SB3           X2           SHIFT CHAR TO HIGH ORDER BITS
                  LX1           B3 , X1       .
HASHONED MX2     6               FORM CHARACTER MASK (6 BIT)
                  BX6           X1 * X2    N1=CHAR*4, N2=0
                  LX6           38           POSITION N1
                  EQ            HASH          RETURN
*
HASHSCAT DATA   20005252101001001001B SCATTER CONSTANT
*
HASHMASK DATA   77777777777777777777B END-OF-WORD MASKS
                  DATA          00777777777777777777B
                  DATA          00007777777777777777B
                  DATA          00000077777777777777B
                  DATA          00000000777777777777B
                  DATA          00000000007777777777B
                  DATA          00000000000077777777B
                  DATA          00000000000000777777B
                  DATA          0000000000000000777777B
                  DATA          0000000000000000007777B

```

11.2.5 Syntax Tables and STREAM on the CDC 6000

A syntax table for the CDC 6000 consists of a 64-word table of executable code, with one word corresponding to each character in the character set. The binary value of the character being examined is used as an offset from the beginning of the table. The word is plugged into the code stream and executed. The instructions set a new value in register X7 if a PUT is specified, and a new table pointer in register B5 if a GOTO is specified. STOP, STOPSH, CONTIN, and ERROR conditions are indicated by a branching instruction in the code word. If no PUT is specified for a character, a branch to the CONTIN processor is placed in the lower half of the instruction word.

The tables are constructed from macro instructions generated by a compiler (written in SNOBOL4), which is included in the tape distributed to all installations. This permits the installation to readily define a new set of syntax tables. This frequently occurs if I/O devices are used which do not have the same characters as the standard set. Communications terminals are typical.

Listings of the syntax macro and two tables for the standard character set follow.

```
**      SYNVEC (USED IN SYNTAX TABLE CONSTRUCTION)
*      GENERATE ONE WORD OF CODE AS A SYNTAX TABLE (VECTOR) ENTRY
      MACRO    SYNVEC ,LABEL ,NEXT ,PUT
      IFC      NE ,/LABEL// ,2
      ENTRY    LABEL
LABEL   BSS      0
+
      BSS      0
      IFC      NE ,/PUT/0/ ,1
      SX7      PUT
      IFC      EQ ,/NEXT/STOP/ ,2
      EQ       STRMSTP
      IFNE
      IFC      EQ ,/NEXT/STOPSH/ ,2
      EQ       STRMSHT
      IFNE
      IFC      EQ ,/NEXT/CONTIN/ ,2
      EQ       STRMNXT
      IFNE
      IFC      EQ ,/NEXT/ERROR/ ,2
      EQ       STRMERR
      IFNE
      SB5      NEXT
      IFEQ    $ ,29 ,1
      EQ       STRMNXT
      ENDIF
      ENDM

*  BEGIN ELEMNT
*  FOR(DIGIT) PUT(INTCOD) GOTO(INTGER)
*  FOR(LETTER) PUT(IDCOD) GOTO(IDENTF)
*  FOR(SQUOTE) PUT(LITCOD) GOTO(SQLIT)
*  FOR(DQUOTE) PUT(LITCOD) GOTO(DQLIT)
*  FOR(LP) PUT(PRNCOD) STOP
*  ELSE ERROR
*  END ELEMNT
*
```

ELEMNT	SYNVEC	ERROR,0	BINARY 00
	SYNVEC	IDENTF, IDCOD	A
	SYNVEC	IDENTF, IDCOD	B
	SYNVEC	IDENTF, IDCOD	C
	SYNVEC	IDENTF, IDCOD	D
	SYNVEC	IDENTF, IDCOD	E
	SYNVEC	IDENTF, IDCOD	F
	SYNVEC	IDENTF, IDCOD	G
	SYNVEC	IDENTF, IDCOD	H
	SYNVEC	IDENTF, IDCOD	I
	SYNVEC	IDENTF, IDCOD	J
	SYNVEC	IDENTF, IDCOD	K
	SYNVEC	IDENTF, IDCOD	L
	SYNVEC	IDENTF, IDCOD	M
	SYNVEC	IDENTF, IDCOD	N
	SYNVEC	IDENTF, IDCOD	O
	SYNVEC	IDENTF, IDCOD	P
	SYNVEC	IDENTF, IDCOD	Q
	SYNVEC	IDENTF, IDCOD	R
	SYNVEC	IDENTF, IDCOD	S
	SYNVEC	IDENTF, IDCOD	T
	SYNVEC	IDENTF, IDCOD	U
	SYNVEC	IDENTF, IDCOD	V
	SYNVEC	IDENTF, IDCOD	W
	SYNVEC	IDENTF, IDCOD	X
	SYNVEC	IDENTF, IDCOD	Y
	SYNVEC	IDENTF, IDCOD	Z
	SYNVEC	INTGER, INTCOD	0
	SYNVEC	INTGER, INTCOD	1
	SYNVEC	INTGER, INTCOD	2
	SYNVEC	INTGER, INTCOD	3
	SYNVEC	INTGER, INTCOD	4
	SYNVEC	INTGER, INTCOD	5
	SYNVEC	INTGER, INTCOD	6
	SYNVEC	INTGER, INTCOD	7
	SYNVEC	INTGER, INTCOD	8
	SYNVEC	INTGER, INTCOD	9

SYNVEC	ERROR, 0	PLUS
SYNVEC	ERROR, 0	MINUS
SYNVEC	ERROR, 0	ASTERISK
SYNVEC	ERROR, 0	SLASH
SYNVEC	STOP, PRNCOD	OPEN PARENTHESIS
SYNVEC	ERROR, 0	CLOSE PARENTHESIS
SYNVEC	ERROR, 0	CURRENCY SYMBOL
SYNVEC	ERROR, 0	EQUAL
SYNVEC	ERROR, 0	SPACE
SYNVEC	ERROR, 0	COMMA
SYNVEC	ERROR, 0	PERIOD
SYNVEC	ERROR, 0	EQUIVALENCE
SYNVEC	ERROR, 0	OPEN BRACKET
SYNVEC	ERROR, 0	CLOSE BRACKET
SYNVEC	ERROR, 0	COLON
SYNVEC	DQLIT, LITCOD	NOT EQUAL
SYNVEC	ERROR, 0	RIGHT ARROW
SYNVEC	ERROR, 0	OR
SYNVEC	ERROR, 0	AND
SYNVEC	SQLIT, LITCOD	UP ARROW
SYNVEC	ERROR, 0	DOWN ARROW
SYNVEC	ERROR, 0	LESS
SYNVEC	ERROR, 0	GREATER
SYNVEC	ERROR, 0	LESS OR EQUAL
SYNVEC	ERROR, 0	GREATER OR EQUAL
SYNVEC	ERROR, 0	NOT
SYNVEC	ERROR, 0	SEMICOLON

```

* BEGIN INTGER
* FOR(DIGIT) CONTIN
* FOR(TERM) STOPSH
* FOR(DOT) PUT(RLCOD) GOTO(REAL)
* ELSE ERROR
* END INTGER
*
```

INTGER	SYNVEC	STOPSH, 0	BINARY 00
	SYNVEC	ERROR, 0	A
	SYNVEC	ERROR, 0	B
	SYNVEC	ERROR, 0	C
	SYNVEC	ERROR, 0	D
	SYNVEC	ERROR, 0	E
	SYNVEC	ERROR, 0	F

SYNVEC	ERROR, 0	G
SYNVEC	ERROR, 0	H
SYNVEC	ERROR, 0	I
SYNVEC	ERROR, 0	J
SYNVEC	ERROR, 0	K
SYNVEC	ERROR, 0	L
SYNVEC	ERROR, 0	M
SYNVEC	ERROR, 0	N
SYNVEC	ERROR, 0	O
SYNVEC	ERROR, 0	P
SYNVEC	ERROR, 0	Q
SYNVEC	ERROR, 0	R
SYNVEC	ERROR, 0	S
SYNVEC	ERROR, 0	T
SYNVEC	ERROR, 0	U
SYNVEC	ERROR, 0	V
SYNVEC	ERROR, 0	W
SYNVEC	ERROR, 0	X
SYNVEC	ERROR, 0	Y
SYNVEC	ERROR, 0	Z
SYNVEC	CONTIN, 0	0
SYNVEC	CONTIN, 0	1
SYNVEC	CONTIN, 0	2
SYNVEC	CONTIN, 0	3
SYNVEC	CONTIN, 0	4
SYNVEC	CONTIN, 0	5
SYNVEC	CONTIN, 0	6
SYNVEC	CONTIN, 0	7
SYNVEC	CONTIN, 0	8
SYNVEC	CONTIN, 0	9
SYNVEC	ERROR, 0	PLUS
SYNVEC	ERROR, 0	MINUS
SYNVEC	ERROR, 0	ASTERISK
SYNVEC	ERROR, 0	SLASH
SYNVEC	ERROR, 0	OPEN PARENTHESIS
SYNVEC	STOPSH, 0	CLOSE PARENTHESIS
SYNVEC	ERROR, 0	CURRENCY SYMBOL
SYNVEC	ERROR, 0	EQUAL
SYNVEC	STOPSH, 0	SPACE
SYNVEC	STOPSH, 0	COMMA

SYNVEC	REAL , RLCOD	PERIOD
SYNVEC	ERROR , 0	EQUIVALENCE
SYNVEC	ERROR , 0	OPEN BRACKET
SYNVEC	STOPSH , 0	CLOSE BRACKET
SYNVEC	ERROR , 0	COLON
SYNVEC	ERROR , 0	NOT EQUAL
SYNVEC	ERROR , 0	RIGHT ARROW
SYNVEC	ERROR , 0	OR
SYNVEC	ERROR , 0	AND
SYNVEC	ERROR , 0	UP ARROW
SYNVEC	ERROR , 0	DOWN ARROW
SYNVEC	ERROR , 0	LESS
SYNVEC	STOPSH , 0	GREATER
SYNVEC	ERROR , 0	LESS OR EQUAL
SYNVEC	ERROR , 0	GREATER OR EQUAL
SYNVEC	ERROR , 0	NOT
SYNVEC	STOPSH , 0	SEMICOLON

The tables are used by the routine STREAM called by the macro STREAM. The routine uses X7 for the value to be PUT into the V field of STYPE, and B5 for the address of the next table to be used. Using the same example as that used for the IBM 360, STREAM is entered with the string 137) and the table ELEMNT. On entry to the routine, X1 and X2 contain the qualifier for the string, and B5 points to the initial table (ELEMNT). X7 is initialized to zero. On the first table access, the value of the character 1 (octal 34) is used as a table offset. The appropriate word from the table,

SYNVEC INTGER, INTCOD

is moved into the code stream so that the instructions

SX7	INTCOD
SB5	INTGER

are executed.

The next two iterations use the table INTGER with offset values of 36 octal (3) and 42 octal (7). These have the syntax table entry

SYNVEC CONTIN, 0

which is a branch to the next-character routine:

EQ STRMNXT

The fourth time through, the offset into the table is 73 octal (), which has the entry

SYNVEC STOPSH, 0

This is a branch to STRMSHRT, the routine for stopping short of a character. The results are computed, and control is returned to the main program.

The STREAM macro and routine follow.

```
**      STREAM
STREAM MACRO    Q1 , Q2 , T , E , R , S
              SA1      Q2
              SB5      T
              SA2      A1+B1
              BX6      X2
              SA6      Q1+B1
              RJ       STREAM
              SA6      Q1
              SA7      Q2
              IFC      NE , /S// , 1
              GT       B2 , B0 , S
              IFC      NE , /R// , 1
              EQ       B2 , B0 , R
              IFC      NE , /E// , 1
              LT       B2 , B0 , E
              ENDM

          TITLE     STREAM -- SCAN STRING FOR NEXT TOKEN
*
*          ENTRY     X1 AND X2 CONTAIN Q2
*                      B5 POINTS TO FIRST TABLE
*
*          EXIT      STYPE HAS BEEN SET
*                      X6 CONTAINS LENGTH AND OFFSET OF Q1
*                      X7 CONTAINS LENGTH AND OFFSET OF Q2
*                      B2 = 1 FOR SUCCESS, 0 FOR RUNOUT, -1 FOR ERROR
*
*          ENTRY POINT IS STREAM
*
```

STREAMX	SX2	FNC	SET FLAG FIELD
	LX7	12	START POSITIONING STYPE
	BX7	X7+X2	MERGE THEM
	LX7	18	MOVE BOTH TO FINAL POSITION
	SA7	STYPE	SAVE IN STYPE
	BX7	X1	GET LENGTH AND OFFSET OF Q2
*			
*	FALL THRU TO EXIT		
*			
STREAM	DATA	0	ENTRY/EXIT
	SB2	X1	LENGTH OF STRING
	BX7	X7-X7	STYPE VALUE INITIALIZATION
	LE	B2 , B0 , STRMRUN	NULL STRING – RUNOUT
	IX2	X1+X2	ADJUST ADDRESS
	AX3	B6 , X2	GET ADDRESS OF START
	SA2	X3	PICK UP FIRST WORD OF TEXT
	AX3	18	GET BIT OFFSET
	SB3	X3	PICK IT UP
	LX2	B3 , X2	POSITION TEXT
	SB4	60	BITS LEFT IN WORD
	SB3	B4-B3	.
	MX3	54	CHARACTER MASK
	SB4	6	BITS PER CHARACTER
STRMNXT	EQ	B2 , B0 , STRMRUN	RUNOUT
	GT	B3 , B0 , STRMNXTA	STILL TEXT IN X2
	SA2	A2+1	GET NEXT WORD OF TEXT
	SB3	60	BITS LEFT IN WORD
STRMNXTA	LX2	6	POSITION CHARACTER
	SB2	B2-B1	CHARACTERS LEFT IN STRING
	BX4	-X3*X2	GET CHARACTER
	SA4	X4+B5	PICK UP TABLE ENTRY
	SB3	B3-B4	BITS LEFT IN WORD
	BX6	X4	GET TABLE ENTRY
	SA6	STRMXTXT	SAVE EXECUTABLE TEXT
	RJ	STRMXEQ	GO EXECUTE TEXT
	EQ	INTR10	SYSTEM ERROR IF RETURN
*			
*	EXECUTE CODE WHICH HAS BEEN PLUGGED IN STRMXTXT		
*			
*	USING THE RJ INSTRUCTION TO REACH THIS CODE VOIDS		
*	THE HARDWARE INSTRUCTION STACK		
*			

STRMXEQ	DATA	0	ENTRY POINT
STRMXTXT	DATA	0	PLUGGED BEFORE ENTRY
+	EQ	STRMNXT	IN CASE OF FALL THROUGH
*			
*		RUNOUT (END OF TEXT ENCOUNTERED BEFORE TERMINATION)	
*			
STRMRUN	BX6	X1	GET Q1
	BX1	X0*X1	GET Q2
	EQ	STREAMX	GO TO EXIT ROUTINE
*			
*		ERROR (CHARACTER IN TEXT IS NOT VALID FOR MATCH)	
*			
STRMERR	SX7	FNC	RETURN STYPE = 0
	BX6	X1	Q1
	LX7	18	POSITION STYPE FLAGS
	SA7	STYPE	SAVE IT
	SB2	-B1	SET FLAG FOR ERROR
	BX7	X6	QUAL2 REMAINS THE SAME
	EQ	STREAM	GO EXIT
*			
*		STOPSH (STOP SHORT OF CHARACTER WE ARE LOOKING AT)	
*			
STRMSHT	SB2	B2+1	ADJUST CHARACTER COUNT
*			
*		STOP (STOP ON CHARACTER WE ARE LOOKING AT)	
*			
STRMSTP	SB3	X1	L
	SX2	B3-B2	L-(L-J)=J
	BX1	X1*X0	0,0
	BX6	X1+X2	0,J FOR QUAL1
	AX1	30	WORD OFFSET
	SB3	X1	
	PX4	X2	CONVERT J TO FLOATING
	SA3	=10.	CHARACTERS PER WORD
	AX1	18	BIT OFFSET
	FX4	X4/X3	GET NUMBER OF WORDS
	UX4	B5,X4	.
	LX4	B5,X4	.

SX5	X4+B3	GET NEW WORD OFFSET
LX5	30	POSITION WORD OFFSET
IX3	X4+X4	WORDS*2
LX4	3	WORDS*8
IX3	X3+X4	WORDS*10=CHARACTERS
IX2	X2-X3	CHARACTERS OFFSET
IX3	X2+X2	CHAR*2
LX2	2	CHAR*4
IX1	X1+X2	NEW BIT OFFSET
IX1	X1+X3	.
SX2	B2	L-J
SA3	BITOFF	OFFSET ADJUSTMENT CONSTANT
BX5	X5+X2	WORD OFFSET, L-J
LX1	48	POSITION BIT OFFSET
BX1	X1+X5	FORM DESCRIPTOR
IX3	X1-X3	TEST IT
SB2	B1	SET FLAG FOR SUCCESS
NG	X3, STREAMX	DESCRIPTOR IS OK
BX1	X3	NO --- ADJUST IT
EQ	STREAMX	NOW GO EXIT
*		
BITOFF	VFD	6/0,24/-4000000B,30/0 CONSTANT TO ADJUST BIT AND WORD OFFSETS SIMULTANEOUSLY
*		

11.2.6 Interfacing the Environment on the CDC 6000

The 6000 version of SNOBOL4 is implemented to run under either of the two commonly used operating systems, SCOPE or MACE. The interface is designed to make SNOBOL4 look like a FORTRAN program. The interface performs the following operations: (1) initial entry from the operating system; (2) FORTRAN input and output; (3) time, date, and clock calls; (4) some interrupt handling, depending on the operating system used; and (5) returning control to the operating system on termination. Logically, the interface is arranged similarly to the 360: initialization, execution, and termination.

The main system entry point, SNOBOL4, is entered from the loader with the job field length in register A0. The system routine Q8NTRY is called to initialize the file environment tables (FET) for all files to be used. Then the SNOBOL4 system is called as a subroutine.

The first operation in SNOBOL, INITEX, sets up the allocated data region. This region starts at the beginning of blank common, and its end is determined by the field length saved from the loader. The free pointer is initialized to point to the beginning of the region.

During execution, input and output are done using modified versions of the FORTRAN I/O routines INPUTC and OUTPTC. Modifications consist only of deleting unneeded numeric conversions so that standard FORTRAN I/O routines can be used as well. The standard calling sequences are employed, using assembled-in formats for OUTPUT and user-provided or default formats for string printing. FORTRAN routines take care of file management. Output strings are unpacked to one character per word so that nA1 format specifications can be used as on the 360.

The FORTRAN routines REWINM and ENDFIL are used for the built-in functions REWIND and ENDFILE. The BACKSPACE function is not included in the distributed implementation, although it can be added by individual installations.

The CP elapsed time, time-of-day clock reading, and date are derived from calls using the system macros TIME, CLOCK, and DATE. The built-in function CLOCK, an extension of SNOBOL4 for the 6000, returns an eight-character string of the form hh:mm:ss.

In case of an error detected by one of the FORTRAN routines, a modified version of the FORTRAN SYSTEM routine transfers to SNOBOL4 to print execution termination messages, a string dump if requested, and accounting information. It then returns to SYSTEM to terminate the job. Program execution is terminated by TERMEX which calls the routine END, which, in turn, closes all files.

External function linkage is performed by a routine called XLINK. This is an optional routine, used in a relocatable version of SNOBOL rather than the absolute core-image version ordinarily used. The system loader loads the linkage routine and all other routines it calls during execution. Dynamic loading at execution time is an option of individual locations since loader facilities vary from installation to installation.

11.2.7 Implementation Problems on the CDC 6000

The major problem on the 6000 implementation is the difference between the architecture of the 6000 and the SIL language. The 6000 is almost exclusively a register-to-register computer, but SIL is a memory-to-memory language. Therefore, each SIL operation must fetch one or more operands from memory and operate on them in registers. The word is the smallest unit which may be addressed so that the words must often be divided into subfields before performing operations. As indicated in the section on register usage, the B registers and X registers can be used to isolate a low-order 18-bit field from an X register. Fortunately, shifting is a fast operation, independent of the amount of the shift.

There are no character string operations at all although by convention the characters themselves are 6-bit quantities and are stored internally in "display code." This code provides reasonably nice properties for string scanning. The letters, for example, have numeric values 1 through 26.

Descriptor design was a matter of compromises imposed by the language and the operating system. The system loader requires a word containing two relo-

cutable address fields to have these addresses on halfword boundaries. Furthermore, isolation of a field for testing by using a “set” instruction can be performed only on the right-hand 18 bits of a word. With these constraints, the descriptor containing a 30-bit value field, a 12-bit flag field, and an 18-bit tag field was developed, working in order from left to right. A primary reason for placing the value on the left is that for integer operations, an arithmetic right shift of 30 bits produces a 60-bit integer value. It means, however, an extra shift instruction when addresses are involved.

The 30 bits are insufficient for real numbers, however, because this only leaves 18 bits for the fraction after the exponent and sign bits are used. A smaller exponent field was considered, but this would have required unpacking and repacking. Instead, it was decided to extend the value field 6 bits to the right, specifying that these bits are not to be used for other descriptor quantities.

Similar considerations were used in designing the qualifier. It was decided to compute a word and bit offset from the character offset when a qualifier is constructed rather than when it is used, because a qualifier is referenced more often than it is tested. The bit offset can be extracted either by shifting the first word of the qualifier right 48 bits (as is done in most places in SNOBOL4) or by using the unpack instruction (UXi Bj,Xk), and appropriately biasing the offset when it is stored or extracted.

Integer-to-string conversion is a problem because there are no conversion instructions. Therefore, the conversion process is iterative and complex. A number of techniques have been employed for performing the conversions, and, at this writing, experimentation is continuing toward developing and optimizing the algorithms.

Floating-point arithmetic was somewhat of a problem to implement, primarily because the 6000 has only a single length floating-point value: a 60-bit word with a 48-bit fraction. As a compromise between the hardware and the SNOBOL4 requirement that real numbers be stored in a descriptor, the high-order 24 bits are retained. Three macros were devised to handle conversion between a real number in a descriptor and a floating-point value: FLGREAL, which sets up registers required by the other operations; GETREAL, which converts a descriptor containing a real (in a register) to a floating-point value; and PUTREAL, which converts a floating-point value to a descriptor real number and stores it.

Interfacing the system was a problem that was generally ignored by making the SNOBOL4 system appear to the operating system to be a FORTRAN program. There are many variations of the basic operating system at different installations; thus, a subset common to all of them is used. The one facility required in SNOBOL4, which is not universally available, is dynamic loading at execution time, required by the routines LOAD and UNLOAD used for external functions. The problem was solved by requiring the user to specify in a table any routines that are to be used at the time the system is loaded. This forces their loading and provides information to the interface as to their location.

Differences between the two kinds of central processors which make up the 6000 series caused a few compromises to be made in the code generated by the

macros. While it is possible to make certain assembly options, this would have complicated the implementation and would not necessarily have produced noticeably improved execution speeds. The primary reason for the differences is that the 6600-type CP, with its various functional units, can perform several operations in parallel if properly programmed, while the 6400-type CP executes them serially. As an example of the type of coding differences this might produce, the problem of moving the contents of a register pair (perhaps containing the qualifier) from X1 and X2 to X6 and X7 is best solved on the 6400-type CP by the instructions:

```
BX6      X1  (move register X1 to X6)
BX7      X2  (move register X2 to X7)
```

both of which use five minor cycles (0.1 microsecond), requiring a total of 1.0 microsecond. On the 6600-type CP, both these operations take place in one functional unit and cannot be overlapped. Each operation requires three minor cycles, for a total of 0.6 microsecond. However, if the code is changed to

```
BX6      X1
LX7      B0,X2 (left shift register X2 into X7,
                  with a shift count of zero)
```

the operations are fully overlappable on the 6600-type CP, producing a minimum timing of 0.3 microsecond. The left-shift instruction requires six minor cycles on the 6400-type, raising the timing to 1.1 microsecond. In general, where such compromises were required, they were optimized in favor of the 6600-type CP code.

Exercises

11.2.1 Write a definition for LOCBD.

11.2.2 Write a definition for MOVTV.

11.2.3 Write a definition for PUTD.

11.2.4 Expand the macro call

```
BRANLV D,(L1,L2,,L3)
```

11.3 COMPARISON OF THE IBM AND CDC IMPLEMENTATIONS

Similarities and differences between the two implementations are evident in the preceding sections. The 360 has more character-handling facilities, which makes string manipulation routines shorter and faster. The longer word on the 6000 permits a somewhat more compact data representation. Difficulties in interfacing the environment are similar.

A comparison of the difficulty in implementing the two systems would be in-

teresting. Unfortunately, such a comparison is too intangible to be useful. If nothing else, the fact that SIL and SNOBOL4 were developed for the most part on the 360 inevitably biases comparisons with implementations on other machines. The overall results can be compared, however. The main features of interest are speed and size.

Comparing two machines of such dissimilar architecture is a tricky business. Differences may be more a matter of the quality of macro definitions than the appropriateness of SIL to the machine's architecture. Therefore, the following figures should be considered only for their most general content. Comparisons are based on an IBM 360 Model 65 and a CDC 6500. The timings for the 65 are representative of a typical configuration [28].

At the most elementary level, the following timings are indicative of the speeds of the two machines. The times given are in microseconds.

<i>operation</i>	<i>360/65</i>	<i>6500</i>	<i>ratio</i>
load register from memory	1.30	1.20	1.08
load register from register	0.65	0.50	1.30
add registers	0.65	0.60	1.08
branch	1.20	1.30	0.92

As can be seen from the figures above, the two machines have quite similar timings for elementary operations. However, the operations required to perform required computations vary considerably because of different architecture. Simple SIL operations produce the following comparisons.

<i>operation</i>	<i>360/65</i>	<i>6500</i>	<i>ratio</i>
MOVDD	2.66	2.70	0.98
GETD	5.26	6.90	0.76
EQLDD	8.50	4.70	1.81
BRANIN	3.80	5.40	0.70

The timing on the EQLDD assumes the success branch is taken. The advantage of the 6500 on EQLDD is the speed of the comparison. The disadvantage of the 6500 in GETD and BRANIN is due to the computation required to isolate V fields.

More complicated operations, such as character manipulation, are better compared in the context of SNOBOL4. Using the constants

```
X10      =  DUPL('X',10)
X1000    =  DUPL('X',1000)
```

the times required to execute the following statements can be compared.

DUPL(X10,1)	1
DUPL(X1000,1)	2
X1000 'Z'	3

The first statement moves 10 characters (a convenient length for the 6500), and the second moves 1000. The third statement, operating in the unanchored mode, performs a large number of character comparisons before failing.

The timings in milliseconds, excluding statement overhead, but including interpretive overhead, are

<i>statement</i>	<i>360/65</i>	<i>6500</i>	<i>ratio</i>
1	0.49	0.76	0.65
2	1.36	1.89	0.72
3	41.10	71.35	0.58

Finally, meaningful comparisons can only be made on complete programs. Several production programs were selected in the following categories:

- A: largely string construction
- B: largely pattern matching
- C: largely numerical computation
- D: largely list processing

The average statement execution times for these programs are:

<i>category</i>	<i>360/65</i>	<i>6500</i>	<i>ratio</i>
A	1.54	4.65	0.34
B	1.46	3.12	0.47
C	1.03	1.76	0.59
D	0.98	1.45	0.67

Size measurements are similar. The following table shows the approximate sizes of the two systems in terms of descriptor-sized units: words on the 6000 and double words on the 360. The total size given is for all program and data exclusive of the allocated data region. The resident data regions, consisting mostly of descriptors, are essentially the same size. If the results are normalized to the bit level, considering the fact that the 360 descriptor requires 64 bits while the 6000 descriptor requires only 60, the ratios are slightly larger.

	<i>360/65</i>	<i>6500</i>	<i>ratio</i>
total	14,300	19,000	0.75
syntax tables	890	1,600	0.56
I/O routines	1,730	1,300	1.33

The difference in the space required for syntax tables illustrates the advantage of the 360 in dealing with characters. Although the 360 character set is four times the size of the 6000 character set, significantly less space is required. There are, of course, other ways of implementing the syntax tables to use less space on both machines. Nevertheless, machine architecture governed the methods used.

part **IV**

RETROSPECT AND PROSPECT

History of the SNOBOL4 Project

The preceding chapters of this book have discussed the background of the SNOBOL4 project, language features, implementation goals, the structure of the system, and the macro language used to achieve a portable and machine-independent implementation. Evolving together, the language, the system that implements it, and SIL interacted with each other to produce the results described. A brief look at the history of the SNOBOL4 project follows. This history puts a time scale on the development and discusses the interactions between various aspects of the project.

12.1 EVOLUTION OF THE LANGUAGE

SNOBOL4 is largely an extension of earlier SNOBOL languages. There is no fixed point in time where SNOBOL4 can be said to have originated. SNOBOL3 was completed in mid-1964. For a period, language development took the form of extensions to SNOBOL3. Gradually, the need for a new language became apparent. The demand for more powerful pattern-matching facilities was most obvious. For example, SNOBOL3 provided no way to describe alternatives in

pattern matching. Many suggestions for new pattern-matching features were made, but these could not be realized simply by extending SNOBOL3. Patterns were part of the syntax of SNOBOL3. Each type of pattern had its own notation. To overcome this limitation, the idea of a pattern as a data object was developed. Such patterns could be constructed during program execution and operated upon like other data objects, thus eliminating the need for special syntactic devices.

Much of the language development that followed the completion of SNOBOL3 was devoted to adding data types for structures such as trees and linked lists. Experience with these extensions showed that users desired a wide variety of such data types, and no matter how many might be built into a language, others would be requested. Again, proliferation and complexity were the prospect. The idea of programmer-defined data types was introduced to allow programmers to extend the language, building whatever data structures they liked. Finally, the need for subscripted variables (arrays) was evident.

During the fall of 1965, various experiments were performed with the ideas mentioned above in mind. These experiments were conducted in SNOBOL3, using its external function mechanism extensively. The results were informative and encouraging.

Early in 1966 the decision was made to implement SNOBOL4. The new language was to include all the features of SNOBOL3, as well as extended pattern matching, programmer-defined data types, and arrays. Other language features were expected to develop as the implementation proceeded.

12.1.1 Evolution of Pattern Matching

The first problem was to incorporate the pattern-matching facilities of SNOBOL3 into the new language. This was accomplished by replacing the syntactic notation for arbitrary and balanced strings by the built-in patterns ARB and BAL. The fixed-length string feature was replaced by the pattern-constructing function LEN. Concatenation followed naturally, changing a syntactic notation that formerly fixed the pattern during translation into an operation that constructed a pattern during execution. Alternation was an obvious extension. Value assignment became a binary operator, completing the picture. From there, new pattern-constructing functions followed. ANY, BREAK, NOTANY, SPAN, TAB, RTAB, and ARBNO were the earliest.

One perplexing problem was how to accomplish the SNOBOL3 back-referencing feature in SNOBOL4. This feature permitted reference to strings matched by components earlier in the same pattern. At first, this feature was discarded altogether. A partial replacement was provided by deferred pattern definition in which $*$, could be applied to a variable. The value of that variable was then determined during pattern matching rather than at the time the pattern was constructed. Although the operand of $*$, could only be a variable, it permitted the construction of recursive patterns.

All the features described above, as well as additional built-in patterns and pattern-constructing functions, were present in the preliminary versions of SNOBOL4 released in the last half of 1967.

Language development continued. Immediate value assignment added in late 1967 gave the dynamic capability that finally provided the equivalent of back-referencing in SNOBOL3. The breakthrough came in early 1968 when unevaluated expressions replaced deferred pattern definition. Unevaluated expressions greatly generalized the pattern-matching facility, permitting any expression to be executed during pattern matching. In mid-1968, the cursor position operator was added, completing the pattern-matching facilities.

12.1.2 Evolution of Other Features

Since SNOBOL4 lacks declarations, the concept of an array as a data object arose naturally. Array references were first implemented using the same syntactic notation, parentheses, as is used for function calls. This turned out to be a rather bad idea in practice since programmers tended to use the same identifier for a defined function and an array. Thus, what was intended to be an array reference turned out to be a call of a function, often resulting in unintended recursion and other mysterious effects. Brackets were introduced to solve this problem.

Programmer-defined data types were implemented early in the project, and changed little during its development. Tracing was implemented in mid-1967 while preliminary versions of SNOBOL4 were still being distributed. A variety of other features appeared in late 1967, due in part to the stimulus of responses from the first users of the preliminary versions. These features included keywords, formatted output, real numbers, the negation operator, the unary name operator, and the CODE data type with its associated facilities.

Tables were a late addition to the language, implemented in mid-1969 and appearing in Version 3. Tables were motivated by the need for associative facilities beyond those provided by indirect referencing. Among the last features to be added were error control in mid-1969 and operator definition in late 1969.

12.1.3 Data Types

One way of viewing the development of the SNOBOL4 language is through the development of its many data types. The concept of many built-in data types evolved slowly. SNOBOL3 had only one data type, STRING. The additional pattern-matching facilities introduced PATTERN. ARRAY was the third data type. Programmer-defined data types introduced the facility for creating new data types during execution. The flurry of activity in late 1967 resulted in the data types REAL, CODE, and NAME. Unevaluated expressions added EXPRESSION in early 1968. Interestingly, INTEGER was not added until 1968 although integers

were distinguished from strings internally from the earliest implementation. The reason for this curiosity has its origins in SNOBOL3, where there were no integers as such. All SNOBOL3 arithmetic was performed by converting numeral strings to integers, performing the operation, and converting the result back into a string of numerals. While SNOBOL4 distinguished integers and strings internally, automatic data-type conversion made the two types essentially indistinguishable to the programmer until the data type INTEGER was introduced. The implementation of external functions in mid-1968 added another facility whereby data types could be created during execution, although such data types could only be differentiated by the external functions that operated on them. Tables added the last data type in mid-1969.

Figure 12.1.1 summarizes the major language developments. The circles at the left indicate releases of different versions of the SNOBOL4 system. P indicates preliminary releases. In the period between the first preliminary release in mid-1967 and the release of Version 1 in early 1968, many different preliminary sys-

version	feature	data types
1970	(3) operator definition error control tables	SPADRCNIDXT
1969	(2) external functions cursor position operator unevaluated expressions	SPADRCNIDX SPADRCNID
1968	(1) immediate value assignment, real numbers CODE , negation, name operator keywords, formatted output tracing	SPADRCN
1967	(P) most pattern-constructing functions, arrays, deferred evaluation, defined data types SNOBOL3 features, new pattern matching	SPAD SP
1966		

Figure 12.1.1
The development of SNOBOL4 language features.

tems were distributed with a variety of language features. Versions 1, 2, and 3 were distinct releases, each with significantly different language features. Between these releases, corrections were issued, but no new language features were distributed. The dates indicated for new language features refer to the time these features were implemented, not when they were first conceived. The column at the right illustrates the growth of the language in terms of the addition of data types.

12.2 EVOLUTION OF THE IMPLEMENTATION

Prior to 1966, experimental implementations were used to test prospective language features. When implementation was begun in earnest early in 1966, the designers of the language were using an IBM 7094 operating under a home-grown monitor, BESYS7 [21]. At that time, GE 645 computers were anticipated and the MULTICS operating system [29] was under development at Project MAC. The designers did not have reasonable access to the facilities being used by the MULTICS development group and chose instead to do their initial implementation work on the 7094 which was readily accessible and had established software. Nevertheless, it was realized that the 7094 was inadequate for SNOBOL4, and that, in any event, conversion from the 7094 to some other machine would eventually be necessary. As it happened, the “other machine” turned out to be an IBM 360 rather than a GE 645.

The first thought was to convert the existing SNOBOL3 system which was written in BEFAP [21] assembly language for the 7094. This idea was gradually abandoned, and an independently implemented SNOBOL4 system began to take shape.

The earliest work consisted of the design of data structures, the translator, the core of the interpreter, storage management procedures, and pattern matching. Before any other system design was undertaken, data structures were considered in some depth. The concept of the descriptor was developed first and made the basic building block for other structures. Qualifiers were a carry-over from SNOBOL3. The qualifier was originally thought of as a distinct unit rather than as a pair of descriptors, as it is now.

The design of blocks and natural variables followed. Constant effort was made to unify and simplify, avoiding proliferation of other structures. Although there has been some change in the use of data structures, such as pair blocks which were introduced much later, the earliest ideas have persisted through the latest implementation.

The translator was completed first. The technique used deserves mention in passing. The translator was first written in SNOBOL3, using programmer-defined functions extensively. Pattern matching was segregated into one section corresponding to STREAM. Debugging was done in SNOBOL3, and then the SNOBOL3 program was hand-translated into SIL macro operations. High-level

SNOBOL3 functions became procedures such as ANALYZ, ELEM, and EXPR. Lower-level functions became macro operations. The translation into SIL was almost mechanical and took only a few hours. The resulting assembly-language program was running the next day. The translator has changed little from its initial implementation. Modifications were required when brackets were introduced for array references and direct gotos were added to handle the CODE data type.

The structure of the interpreter presented some difficulties. The form of the prefix code, the procedure structure, and the return-signaling mechanism evolved over a period of months and after several false starts. The biggest problems were the handling of failure in source-language operations and distinguishing between names and values. These problems were solved by using return signaling and by structuring the system so that procedures evaluate their own arguments. Most interpreter problems were resolved by mid-1966. Although individual features have been added and the method of accessing function procedures changed somewhat, the basic structure of the interpreter has changed little since then.

Storage management procedures were originally designed on the assumption that SNOBOL4 would operate in a virtual memory environment (the GE 645 operating under MULTICS). The first implementation assumed that when storage regeneration was required, a new region would be obtained and the old allocated data region copied into the new one, collecting available space in the process. This storage regeneration technique had the interesting property of requiring only a single pass. The machine actually used did not have virtual memory capabilities; thus, two regions were set aside and successive storage regenerations moved data from one region to the other, alternately. Eventually the two-region approach was abandoned because of its wastefulness of real memory, and the present scheme was introduced.

Pattern matching presented the greatest problems. One major source of difficulty was the lack of a clear definition of what pattern matching was to do, and, hence, of a basis for an algorithm for an implementation. For this reason, pattern matching as a language feature has been more influenced by the implementation than any other aspect of the language. First, deferred pattern definition and then unevaluated expressions introduced problems of recursion. The heuristics, designed to improve the running speed of pattern matching, proved to be a great bother.

From the beginning, patterns were structures. At first, these structures consisted of separate blocks that were linked together by pointers. When one pattern was used in the construction of another, a copy of the first pattern was made and linked into the larger structure. The process of copying a linked structure was complicated and slow. Therefore, the present form of pattern was chosen, consisting of a single block with offsets indicating relationships among the components. This change involved introducing a new set of pattern-building macros and rewriting the pattern-matching procedures. At first, the descriptors in the

pattern components were not organized like prefix code but rather in an ad hoc fashion specifically designed for pattern matching. A final modification, designed to reduce the number of different internal structures, resulted in patterns that consist of prefix code.

12.3 EVOLUTION OF SIL

While the desirability of a macro language for implementing SNOBOL4 was recognized from the beginning of the project, the idea of a machine-independent implementation language became clear only gradually.

The first approach was to use a mixture of machine operations and macro operations, allowing only those machine operations that had counterparts on most machines, such as loads and stores. At this stage, macros were used mainly as a device to simplify the implementation, making it easier to think about the workings of SNOBOL4. Gradually, the concept of register independence was introduced. Finally, SIL took shape as a language that applied to SNOBOL4 data objects and performed those operations required to implement SNOBOL4.

While early documentation of this aspect of SNOBOL4 is fragmentary, and much has been lost in the fog of human memory, a few remaining facts provide insight into the development of SIL. In mid-1966, some 78 SIL operations had been defined. Many showed their heritage from SNOBOL3 and the influence of the 7094. By late 1966 there were 180 SIL operations, more oriented toward SNOBOL4 operations. Since that time the operations have changed somewhat. Their names have been changed many times in an attempt to achieve better mnemonics and to reflect the changes in terminology used to describe the implementation. The greatest effort has been to reduce the number of operations without increasing the inefficiency inherent in using SIL. The present list of some 130 macros reflects the results of that effort.

The first real test of SIL came in the fall of 1966 when the designers learned that the replacement for the 7094 was to be a 360, not a 645. The designers decided to convert their 7094 program to the 360, using a machine that was available at Princeton University.

At that time, the 7094 version was running, although it still had many bugs and the language design itself was far from complete. With the assistance of two colleagues at Princeton, the designers undertook the conversion, but there were many obstacles. The designers had no prior knowledge of the 360; the Princeton machine was some thirty miles distant and available only at certain times; the software was extremely primitive; and the hardware was unreliable.

The designers soon learned to what extent their concepts of machine independence and portability had been achieved. The problems of basing on a machine like the 360 had never been considered. The character orientation of the 360 pointed out several dependencies in word addressing that had escaped the de-

signers. New SIL macros had to be written and several old ones revised. Nevertheless, despite all the problems, the conversion was accomplished in less than a month. A 360 version was running by the end of 1966, and the 7094 implementation effort was abandoned.

12.4 IMPLEMENTATION OF SNOBOL4 ON OTHER MACHINES

As the development of SNOBOL4 became known outside of Bell Laboratories, several individuals expressed interest in implementing SNOBOL4 on other machines. The designers' previous experience with independent implementations of SNOBOL3 had already contributed to the goal of machine independence and the consequent development of SIL.

In the spring of 1967, the first attempt was undertaken to implement SIL on another machine, the CDC 6600. At that time, there was little or no documentation of SIL, and SNOBOL4 was still under intensive development. SIL was changing daily, and all parts of the system were full of bugs. The designers were hardly prepared for such an effort; they would have preferred delaying this experiment until documentation was available; and they had given some thought to how such a project should be approached. However, the opportunity was there, and the prospective implementor (or victim, as the case may be) was enthusiastic. This first implementation was done by R. S. Gaines at the Institute for Defense Analyses (IDA) at Princeton. The major motivation for the project at IDA was a desire to have the new SNOBOL4 language available on their machine as soon as possible. An interest in the implementation technique was a secondary motivation.

What followed were many telephone calls, a steady flow of information through the mail, a visit or two, and much suffering. The effort was justified when the first 6000 series version of SNOBOL4 began to run, if somewhat falteringly, a few months later. A solid preliminary release appeared a month or two after the corresponding 360 release, and the 6000 series implementation has kept pace ever since.

The pattern has repeated itself many times since. Implementors were always available before the implementation material was ready, and every implementation until the present time has been undertaken while the SNOBOL4 project was in a state of flux. Figure 12.4.1 illustrates the time scale on which such implementations have been accomplished.

All these implementations were done by one or two individuals without much help. In most cases, implementation of SNOBOL4 was a part-time project undertaken with little official support. Many implementations, like the 360, were started with new operating systems and primitive software. In addition, the implementors were faced with the following handicaps:

- (1) There was no documentation of SNOBOL4 internals except the source listing itself.

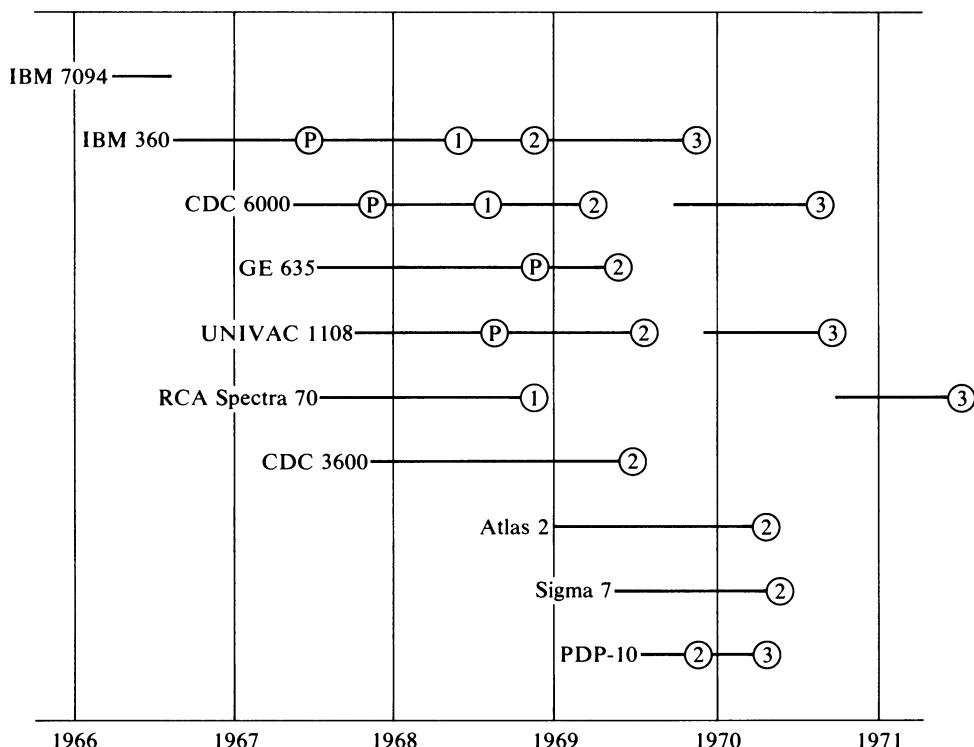


Figure 12.4.1
Implementations of SIL.

- (2) Documentation of SIL was inadequate and sometimes erroneous.
- (3) SNOBOL4 as written in SIL contained significant errors, especially prior to Version 1.
- (4) There were no test procedures for SIL.
- (5) There were no systematic test programs for SNOBOL4.
- (6) There was no documented approach to the process of implementing SNOBOL4 by implementing SIL.

Implementing SNOBOL4 by implementing SIL is a process quite different from most implementations of a programming language. Certain macro definitions have to be written, some of which are simple; others are more complex, requiring substantial subroutines. With luck, some code can be borrowed from other processors or system facilities used. When the macro definitions are complete, the SNOBOL4 source program, written in SIL, can be assembled. The first step in debugging is to correct assembly errors and other obvious errors evident in the macro expansions. When a clean assembly is obtained, SNOBOL4 is ready to test, but of course, it does not work. The implementation of SIL is debugged as

far as possible, but errors remain. Since no test procedures for SIL are provided, except the SNOBOL4 system itself, the task of debugging SIL is left to the implementor. Only a few implementors expended the effort to develop tests for SIL independent of the SNOBOL4 system. Most implementors chose a more direct, but ultimately more difficult approach: trying to debug SIL on the basis of errors that appear in SNOBOL4.

The usual course is something like this: Input and output must work first. At this point, troubles in the translator can be detected. The first problems usually have to do with recursion because of errors in coding RCALL and RRETURN. When these errors are rectified, the translator may appear to work. In most cases, however, the translator fails to stop at the end of the source program. This indicates that the HASH macro is not giving reproducible results, and, consequently, the label END is not recognized. When this is corrected, SNOBOL4 usually goes into interpretation but dies in some unpleasant way because the prefix code is ill formed. This in turn is usually a consequence of errors in coding the macros that construct code trees. When this difficulty is fixed, a good part of the SNOBOL4 language may operate properly. Troubles often arise in executing programmer-defined functions because of more subtle errors in macros dealing with recursion. Pattern matching invariably fails to function properly because of errors in CPYPAT that cause improper relocation of alternate and subsequent fields. When these errors are corrected, programs proceed far enough to require storage regeneration, at which point all kinds of troubles appear. The problem is almost always traceable to stray bits in F fields, causing erroneous sensing of title descriptors. These stray bits may be due to a failure to clear F fields as prescribed in the individual macro operations, or accidentally using F fields of stack descriptors to store system information in RCALL. When all of these problems are solved, SNOBOL4 usually runs with little further difficulty. From the implementor's point of view, this is a mysterious process. After struggling through the definition of the SIL macros, SNOBOL4, somewhat ghostlike, materializes. Then, after days of frustrating debugging at the lowest level, entire language features work flawlessly.

The author has served as an informal consultant at one time or another on all the implementations shown in Figure 12.4.1 and has shared vicariously in the frustrations and successes. In retrospect, it is regrettable that better documentation and an organized scheme for implementing and debugging SIL was not produced to help the implementors. On the other hand, the SNOBOL4 project benefitted immeasurably in its formative stages from the feedback obtained from these early implementations.

12.5 INTERACTION OF LANGUAGE DESIGN AND IMPLEMENTATION

As discussed in Chapter 3, the designers' original intention was to minimize the restraining effect of implementation on language design. Wherever possible, difficulties of implementation were ignored when language features were being

designed. Such an approach can only be an ideal. There are always practical constraints, and a knowledge of their existence inevitably influences the designer. Furthermore, failure to place some practical control on language features would produce an unuseable result. Features do have to be implemented, and inefficiency can be tolerated only to a point. The author does not recall any language feature that was rejected because no way could be found to implement it or because it would be too inefficient to be practical. However, experience with previous SNOBOL languages undoubtedly had its effects in this regard.

An entirely different effect of implementation on language design deserves mention. In several cases, the implementation suggested a language feature. Translation during execution, discussed in Section 8.8, is the most obvious example. Other influences were more subtle. Certainly the concept of keywords was influenced by storing system status information in descriptors in the resident data region and by the facility offered by pair blocks. Tables are simply an extension of an internal mechanism to the external language.

In one case at least, documentation was the source of a language feature. After several attempts had been made to describe how pattern matching works, the concept of the cursor position was introduced in written material. The cursor had, in fact, existed in the implementation, although it had not been thought of in quite that way. A recognition of the importance of the cursor as a concept not only made the implementation easier to understand, but also suggested the cursor position operator. Thus, the programmer was given access to what amounts to an implementation technique.

Evaluation of the Results

Evaluation of a system of the size and complexity of SNOBOL4 cannot be reduced to a checklist to which yes and no answers can be made or point scores assigned. For one thing, there are few bases for comparison. For another, there are many points of view from which an evaluation can be made, and some of these are inherently in opposition to each other. What the implementor considers important is quite different from what the user considers important. For example, the goals of the SNOBOL4 project, as set forth in Chapter 3 were: (1) generality and flexibility, (2) machine independence, and (3) portability. These goals were recognized to be in conflict with two important aspects of the resulting system: size and running speed. One could argue about how well the project met its goals, but one could just as well criticize these goals. This chapter discusses both views and their relationship to each other.

13.1 THE USER'S VIEWPOINT

SNOBOL4 generally gets poor marks for its slow running speed and its large size. The typical user finds SNOBOL4 expensive to run. This may amount simply to an economic burden. In some environments service may be affected as a result, causing lower priority and poorer turn-around time.

Factors that adversely affect the running speed of SNOBOL4 are: (1) the interpretive nature of the system, (2) generality and uniformity in handling language features, (3) the use of SIL as an implementation vehicle, (4) inefficient techniques and algorithms, and (5) bad coding.

The large size of SNOBOL4 is attributable to: (1) the interpretive nature of the SNOBOL4 system, requiring a large amount of code to be resident during execution; and (2) the amount of space required to represent source-language data objects.

There is no way to make an accurate quantitative assessment of the costs of the individual factors listed above. Indeed, several are closely related. However, some general remarks can be made.

The greatest source of inefficiency in SNOBOL4 lies in the interpretive nature of the system. The cost of interpretation, compared with what actually is accomplished, is most noticeable in simple operations such as assignment. In its most basic form, an assignment requires two instructions: a load and a store. For interpretation, some sixty instructions are executed in the 360 implementation. The overhead of interpretation is correspondingly less noticeable in complex operations, such as pattern matching, for which extensive routines are required independent of interpretation. The author's own estimate, admittedly based on intangible considerations, is that in the case of SNOBOL4, interpretation introduces a factor of three or four in the overall running time of a typical program.

Inefficiencies arising from the use of SIL are easier to measure. Here the main problem is redundant code, largely generated because SIL macros refer to locations in the resident data region and have no concept of registers. The problem arises at the boundary between macros. Successive macros may refer to the same location, producing, for example, a redundant fetch simply because there is not a method of communicating that the value needed is already in a register. On the 360, for example, the sequence

```
MOVDD X, Y  
MOVDD Z, Y
```

expands into

LD	0, Y
STD	0, X
LD	0, Y
STD	0, Z

The third instruction is obviously unnecessary. Ideally, some of this redundant code could be avoided by clever macro definitions and by the use of assembler facilities to keep track of global state information. So far as the author knows, no implementor has in fact done anything significant along these lines. The use of SIL also produces poor code because there is no loop control mechanism (such as "do loops"). Consequently, loops are written in SIL by specific arithmetic operations and tests on memory locations, with resulting wasteful code.

A measure of the cost of SIL is available as a by-product of a technique used to improve the code generated in the 360 implementation done at Bell Laboratories. In this implementation, considerable care is given to the code generated by individual macros, but the redundancy of the generated code is ignored during development and debugging stages. When a final production version is desired, a series of steps are undertaken to improve the code. This is done by processing the code produced by the expansion of the SIL macros, using several programs written in SNOBOL4. A number of techniques have been used to obtain successively better code. One technique consisted of applying a program which processes the expanded code, looking for obviously redundant adjacencies such as

ST	X , A
L	X , A

In this case, the second instruction is simply deleted. In other cases, instructions are replaced by equivalent, faster instructions. Another, more sophisticated, program performs a pseudo-simulation of the 360 with respect to a number of frequently referenced locations in the resident data region. In this simulation, descriptors and qualifiers under consideration are given distinct, recognizable values, and each machine operation is simulated to see if it performs a necessary function. For example, before a register is loaded from a location, the contents of the register are compared with the value in storage. If they are the same, the instruction is deleted. If they are not the same, other registers are checked to see if the value can be obtained by a register load instead of a storage reference. This pseudo-simulation is straight-line and makes no attempt to reassign registers. Nevertheless, a substantial amount of code is deleted or improved.

Finally, hand work is undertaken to improve the remaining code. This hand work consists of two efforts. The first effort is improvement of the kind described above, utilizing the additional knowledge and intelligence of the implementor. Some register reassessments are made, and attempts are made to improve the code in loops. Finally, certain frequently executed sections of the code are located by running a variety of SNOBOL4 programs under a core-profile monitor. These sections are replaced by hand-coded versions in which considerable attention is given to execution speed.

The code resulting from all of these modifications is then reassembled to produce a production version of the 360 implementation. Such code improvement is laborious, requiring weeks of effort. It is also hazardous since insidious bugs are easily introduced. It has only been done twice to date, once for Version 2 and once for Version 3.

Timing tests show that SNOBOL4 programs run from 10 to 25 percent faster after code improvement. The profile results indicate that a complete hand coding of the SNOBOL4 system would not improve the running speed by more than an additional 5 percent. (The timing figures given in Chapter 11 were obtained from an unoptimized version.)

Another source of inefficiency is the generality and uniformity in treatment of language features. The cost of generality is very difficult to estimate and depends greatly on the feature in question. The most noticeable culprit is the natural variable. Uniform treatment of strings as natural variables greatly simplifies the implementation of some language features, such as indirect referencing. On the other hand, much unnecessary hash computation, allocation, and subsequent storage regeneration are performed in executing some programs. In some programs this contributes significantly to running time. In others it does not.

Programs are always subject to criticism for inefficient techniques and algorithms. SNOBOL4 is no exception. However much effort goes into design, hindsight suggests improvements. In some cases a technique known to be inefficient is used because of practical considerations such as available time, manpower, and revisions of other parts of the program that a better technique might impose. The implementation of the TABLE data type as a linear structure is an example. The expense of a linear look-up for a table reference may be quite significant for large tables (fortunately the look-up is performed in a machine-language loop, not in a SIL loop). A better choice, using hashing techniques, was not undertaken because of limited manpower and time pressures to make the new feature available. These are the compromises with reality.

Some bad code is present in any system, and again SNOBOL4 is no exception. The cost of bad code is probably an order of magnitude less than the cost of any of the other factors mentioned above. This assessment is based on the experience obtained by improving the code on the 360 system, in which frequently executed sections of code are replaced by hand-coded versions. Most of the improvement that results is easily traceable to replacing redundant code generated by expansions of SIL macros, and relatively little can be attributed to bad coding in SIL. The rest of the program which is not hand-coded is executed so infrequently that bad code in it is an insignificant influence.

The factors that adversely affect the size of the SNOBOL4 system are largely the same as those that affect its running speed. Well over half the SNOBOL4 system consists of data. In the 360 implementation, the breakdown between program and data is roughly: (1) program including I/O routines, 40 percent; (2) resident data region, 20 percent; and (3) allocated data region, 40 percent. The size of the program and resident data region is fixed. The minimum size of the allocated data region required to run a program varies considerably. A smaller allocated data region may result in more storage regenerations and, consequently, slower running speed. The percentages given above are based on an allocated data region designed to run production programs efficiently.

The size of the program itself is increased by: (1) the redundant code generated by SIL macros, (2) the generality of the procedure structure with its large number of recursive calls, and (3) the decision to expand several relatively large macros in line because of their frequency of use.

The size of the program is reduced approximately 10 percent by the code improvement techniques described above. The trade-off of size versus speed in the

choice of in-line code versus subroutines is difficult to assess, and, so far as the author knows, no thorough analytic study of this question has been made.

The amount of space required for resident and allocated data is significantly increased by the generality and machine independence of data structures. There is considerable waste space in many descriptors. The use of pair blocks as a general way of associating information is also wasteful. A larger number of types of basic data units would reduce the size of the resident data region somewhat at the expense of a more complicated program. Syntax tables are also wasteful of space. Incorporating the details of the SNOBOL4 syntax in the program would reduce the size of the resident data region at the expense of machine independence and flexibility in changing the syntax.

Many users are concerned with matters other than size and speed. A discussion of the value of various language features in programming is beyond the scope of this book, but it should be mentioned that some of the penalty paid in size and speed is a consequence of an attitude toward language design that put these factors in a position of lesser importance than is usually the case. Stated another way, SNOBOL4 would be a different language if considerations of efficiency had been paramount.

13.2 THE IMPLEMENTOR'S VIEWPOINT

The person who implements SIL on a new machine is less concerned with execution speed than with the effort involved in getting SNOBOL4 running. Size may be a significant factor, however, if SNOBOL4 does not readily fit into the available memory.

Chapter 12 discussed some of the factors that hamper implementors. These are largely peripheral to the design of the SNOBOL4 system and reflect the fact that much more could have been done to increase portability by providing better documentation and diagnostic tools.

It is extremely difficult to attach a "portability figure" to SNOBOL4. The main concern is usually in terms of man-months required. As the documentation and stability of the SNOBOL4 system have improved, SNOBOL4 has become more portable. Unfortunately, experience with previous implementations is not too helpful in arriving at such an estimate. Some implementations that took a year to accomplish were worked on only occasionally. The author would estimate, however, that a minimum of three man-months is required to accomplish an implementation under reasonably favorable circumstances. The factors that affect the amount of time required fall into two general categories: the machine and the implementor.

For the machine, the following points are important: (1) suitability of the target machine, especially the amount of memory available; (2) the quality of the operating system and software needed for the implementation; and (3) access to the

machine. Success in implementing SNOBOL4 on large machines has led to attempts to implement SNOBOL4 on small machines. The IBM 1130 is an example. SIL is simply not intended for a machine with a limited memory capacity, and such attempts have been unsuccessful to date.

Any person with implementation experience knows how frustrating poor support software can be. In several cases, implementation bugs in assemblers have been brought to light by the large size of the SNOBOL4 system. In one case (which shall remain unidentified), the assembler had to be extensively modified before actual work on SIL could begin.

For the implementor, more factors are important: (1) experience with the machine and its software, (2) previous experience with implementing large programming systems, (3) knowledge of the SNOBOL4 language, (4) knowledge of SNOBOL4 internals, (5) availability of time for a sustained effort, (6) motivation, and (7) ability. Most of these factors are self-explanatory and need little discussion. Motivation and ability are the most important. It is well known that programmer productivity varies enormously. A range of an order of magnitude is common, and two orders of magnitude is not rare.

When asked if his system is portable, a system designer may respond, "Of course—I could put it on machine X in N weeks." He may be able to do this, but only because he has a completeness of knowledge, experience, and motivation beyond anyone else. That is not the point. Portability is not measured by a superhuman effort by its system designer, but rather by the extent to which an implementation can be achieved by an ordinary mortal in an average environment with the minimum of assistance by the system designer.

13.3 THE AUTHOR'S VIEWPOINT

The author views the SNOBOL4 project from a somewhat different perspective than the user or the implementor. Yet he has interacted with both groups and belongs to both groups himself.

The merits and demerits listed above are clear. Summarized, the SNOBOL4 system is relatively large and slow. Its portability has been demonstrated. Several factors remain to be discussed.

Machine independence is not a major concern to most individual users, although it may be very important to a few. SNOBOL4 gets very good marks on this point. Since SNOBOL4 is written in SIL, the logic is machine-independent. In fact, SNOBOL4 is one of the most machine-independent languages over the range of large scientific machines on which it is available. FORTRAN, while far more widely available, suffers from more idiosyncrasies and dialectic differences between machines. Interestingly, machine independence has the effect of imposing de facto standardization. The availability of compatible implementations on most machines goes a long way toward discouraging development of

dialects. Furthermore, SNOBOL4 is relatively free of bugs traceable to a particular implementation. An implementation of SIL must be relatively free of bugs for SNOBOL4 to function properly at all.

Different implementations of SNOBOL4 may differ on the following points: (1) character set, (2) bounds on integers and real numbers, (3) input and output (via FORTRAN), and (4) system-dependent facilities, such as external functions. Most SNOBOL4 programs will run on any machine having SNOBOL4 with only a simple transliteration of characters.

Machine independence and ease of maintenance are related. Corrections to SNOBOL4 are distributed as alters to the SNOBOL4 program written in SIL. Corrections are sent in the SIL language to implementors who simply insert them and reassemble SNOBOL4 to bring their system up to date.

One of the goals of the SNOBOL4 project was to produce a system that could be easily extended. An excellent example of the success of the implementation in this respect is given by SNOBOL4B, a major extension of SNOBOL4 designed and implemented by Gimpel [30]. SNOBOL4B includes a new data type, BLOCK, which is a three-dimensional generalization of a character string. Blocks may be manipulated and combined in various ways and are particularly useful for printing tables and charts. Two dimensions of a block produce a rectangular print image. The third dimension, depth, is used to obtain overstriking.

SNOBOL4B was implemented by adding a few operations to SIL [31], by assigning a new integer code for the BLOCK data type, and by adding necessary procedures to the SNOBOL4 system. SNOBOL4B is thus a superset of SNOBOL4.

Generality and flexibility are vague terms, and yet they are the most important attributes of the SNOBOL4 system. Now that the language design is finished and the implementation complete, the effect of these attributes is hard to see. Yet without them, the language would not have developed as it did.

13.4 AREAS FOR IMPROVEMENT

There are many areas in which the implementation of SNOBOL4 might be improved without departing radically from the approach taken. Some specific difficulties have been mentioned or alluded to in earlier chapters. The major defects, as the author sees them, are listed below.

(1) Input and output should be handled through a machine-dependent interface that does not use FORTRAN I/O. This would permit the utilization of I/O facilities and file structures appropriate to the machine and free SNOBOL4 from having to operate as a subroutine called from a FORTRAN environment.

(2) Strings could well be handled differently. Treating every string as a natural variable has great generality and permits easy and uniform handling of strings in all contexts. However, the penalty paid in execution time and storage is severe.

A better solution would be to have two types of strings: strings as data and strings as variables. Most strings fall into the first category. When a data string is used as a variable, a conversion would be necessary. Section 14.1 describes an implementation of SNOBOL4 in which this approach has been taken.

(3) Pair blocks should not be used for all forms of association. The simple structure and ease of searching pair blocks simplified many implementation problems. In cases where the number of associations is small, pair blocks are an economical and efficient implementation technique. However, in cases where a large number of associations are involved, pair blocks introduce considerable inefficiency. An alternate method, utilizing hashing techniques, should be available.

(4) SIL solved many problems of machine dependence by eliminating the concept of registers and using only storage-to-storage operations. The penalty paid is significant, as illustrated by the discussion of code improvement given in Section 13.1. A better solution might have been to introduce the concept of registers and a notation for distinguishing registers from storage locations. On most machines, at least a few SIL registers could be implemented in machine registers, with significant savings in generated code. To a certain extent, certain frequently referenced variables used by SIL could be treated this way in the present implementation, but modifications would have to be made to the SNOBOL4 system. Storage regeneration procedures presently expect such variables to be in basic blocks, and a change of the type described above would have to take registers into account as well.

(5) Code generated by SIL could be significantly improved by the addition of facilities for iteration control and declarations of local variables. These facilities would permit the use of machine registers for functions that presently require operation on descriptors. In a sense, such a feature is an alternative to the description of registers suggested in item 4 above. The two concepts are not mutually exclusive, however.

(6) The organization of the SNOBOL4 system should be modified so that not all procedures need be resident during the execution of SNOBOL4.

Alternatives

The preceding chapter evaluates the implementation of SNOBOL4, discussing its good points and its bad points. The major problems with the implementation are clearly its size and speed. If it were half its size and ten times as fast, there would be little to complain about.

This chapter considers two alternatives. The first has already been realized. The other is, as of this writing, still speculation. The first solves the problem of size and speed in software. The second promises to solve these problems in hardware.

14.1 A SNOBOL4 COMPILER

All implementations of SNOBOL languages developed at Bell Laboratories have been interpretive. The reasons for this kind of implementation have been discussed in some detail in previous chapters. The novel nature of the languages, which has made their implementations something of a mystery, has led to a mystique about interpreters. There have even been claims that SNOBOL4 can only

be implemented interpretively. Such claims are clearly false. If nothing else, SNOBOL4 programs could be compiled into series of subroutine calls, in which the subroutines would function very similarly to the present system. Such an implementation would be begging the question, however, because it is generally understood that a compiler should produce machine code whose execution more or less directly executes the source language program. Assignment amounts to a load and a store on most machines. Thus, a SNOBOL4 compiler should generate a load and a store for an assignment.

It has long been recognized that a SNOBOL4 compiler is possible, if difficult to implement. The problems have been in how to handle some language features within a compilation context. Even granting that subroutine calls should be used for more complicated operations to prevent the size of the compiled program from being unreasonable, there are still serious problems. Examples are tracing, input and output associations, and conversion to CODE.

Various attempts have been made, but some have succeeded at the expense of omitting certain language features. One implementation, however, has proved that all of SNOBOL4 can be implemented within a compiler framework. This implementation, known as SPITBOL [32] (for SPeedy ImplementaTion of SNOBOL4), was designed and implemented on the IBM 360 by Dewar and Belcher at the Illinois Institute of Technology. After several years of development, SPITBOL was first publicly distributed at the end of 1970. This system has produced very encouraging results. Not only does SPITBOL require less memory than SNOBOL4, making it useful to a wider range of installations, but it is quite fast. Timing tests of SPITBOL and SNOBOL4 show typical execution speed ratios from 3:1 to 10:1, with 8:1 being typical. This speed makes SPITBOL a reasonable tool for production programming and promises to greatly extend the usefulness of the SNOBOL4 language.

SPITBOL is an in-core compiler in the spirit of WATFOR [33]. That is, SPITBOL compiles the source program directly into executable code in its data region. SPITBOL is a true compiler, and machine operations that correspond closely to language operations are used wherever possible. More complicated operations generate calls to subroutines. Consider the statement

$$X = M + N$$

For this statement, SPITBOL generates the following code:

LM	A1 , A1+1 , VAL.M
LM	A2 , A2+1 , VAL.N
BAL	RETURN , \$\$ADD2
STM	A1 , A1+1 , VAL.X

The first two instructions fetch the values of M and N. \$\$ADD2 is a subroutine that performs binary addition. A subroutine call is used since data-type checks and possible conversions on the value of M and N may have to be performed. The result returned by the subroutine is stored in the value of X.

An amusing example of the difference between SNOBOL4 and SPITBOL is provided by the way &STLIMIT is implemented and statement numbers located. In SNOBOL4, each statement number is stored in the prefix code as an argument of init₁. At the beginning of statement execution, init₁ is invoked and stores its statement number argument in a global descriptor in PKYPBL. &STLIMIT (at a known location in UKYPBL) is checked by init₁. If the statement limit has been exceeded, control is transferred to an error routine, and if the current statement number is required by a reference to &STNO, it is located in PKYPBL. Error messages that require the current statement number simply reference a known location in PKYPBL.

SPITBOL, in its quest for efficiency, approaches these matters quite differently. At the beginning of each statement, the following instruction occurs.

AUR	SCNT , SINC
-----	-------------

This instruction increments a floating-point register reserved for statement counting. SCNT is set up so that when &STLIMIT is exceeded, a program interrupt occurs because of floating-point overflow. An interrupt handler detects this interrupt as excessive statement execution. In order to determine the value of &STNO, the compiled program is processed, counting AURs. Similar processing is required to determine the current statement number for error messages. Thus, SPITBOL accomplishes statement initialization, which occurs frequently, in an extremely simple, if machine-dependent, way. Determining the current statement number, which occurs infrequently, is slow and complicated, compared with SNOBOL4.

Data structures used by SPITBOL are in many cases similar to those used by SNOBOL4. For example, SPITBOL has a descriptor for representing source-language data objects. The layout of the SPITBOL descriptor is somewhat different in detail from the SNOBOL4 descriptor, but it serves the same purpose, containing a data-type code and a value or pointer to a structure.

SPITBOL differs markedly from SNOBOL4 in its handling of strings. Strings in SPITBOL are not ordinarily represented by natural variables. A natural variable is created for a string only when the string is used as a variable. This avoids unnecessary hash computations and allocation. More importantly, this handling of strings permits multiple pointers to the parts of a string. Several substrings may be represented within the same string, avoiding the constant creation of copies of strings so frequently required in SNOBOL4.

Since SPITBOL compiles executable code, some features that are easily implemented in SNOBOL4 are more difficult or complex in SPITBOL. Tracing a variable is an example. In SNOBOL4, when TRACE is executed, the location of the variable to be traced is placed on a list. Subsequently, when a value is assigned to a variable, the location of the variable is simply compared with values on the list. In SPITBOL, when TRACE is executed, a flag is inserted in the vari-

able and the entire compiled program is examined, looking for code that assigns values to the traced variable. Whenever such code is found, it is replaced by a branch to a subroutine for performing the trace. Subsequently, a branch to the tracing subroutine occurs whenever the variable is to be assigned a value. The flag in the variable provides the subroutine with the information required to perform the desired trace. Thus, in SPITBOL, execution of TRACE is considerably more complex and time-consuming than in SNOBOL4, but tracing itself is more efficient.

Some features of SNOBOL4 have proved difficult to implement in SPITBOL. The most serious omission in early versions of SPITBOL was the CODE data type and its associated facilities. As of this writing, SPITBOL includes CODE and most other SNOBOL4 features. Lacking are tracing and I/O associations for nonnatural variables, programmer-defined trace procedures for some types of tracing, redefinition of defined operators, assignment to unprotected keywords associated with pattern components, and a few other minor facilities. Nevertheless, SPITBOL is sufficiently compatible with SNOBOL4 so that most programs will run under either implementation without modification.

A thorough description of SPITBOL would occupy more space than a description of SNOBOL4. In any event, SPITBOL is someone else's story to tell. It is a fascinating, well-designed, and cleverly implemented system which the serious student of software is well advised to study.

In size and speed, SNOBOL4 is no match for SPITBOL. SPITBOL, of course, is hardly machine-independent or portable. Heavy use is made of idiosyncrasies of the 360 instruction set, and the coding is very tight. Nevertheless, much of the work that went into the design of SPITBOL is independent of the 360. Data structures, algorithms, and approaches to some of the more difficult language features apply to any machine. Hopefully, SPITBOL will be implemented on other machines in the near future.

14.2 SNOBOL4 MACHINES

Persons interested in SNOBOL have long dreamed of a "SNOBOL machine," in which the language would be executed by a computer designed for that purpose. The enormous discrepancy between conventional hardware and the data structure and operations performed by SNOBOL4 is obvious. Most machines are designed for performing arithmetic operations efficiently. In the past, memory organizations have usually been word-oriented. More recently, machines such as the IBM 360 and 370 have provided more character-oriented architecture, but there is still a long way to go.

Two approaches to a SNOBOL4 machine have been investigated. The first considers the design of SNOBOL4 hardware. The second strives to emulate SNOBOL4 on a machine with microprogramming capabilities.

14.2.1 Design for a SNOBOL4 Machine

The earliest discussion of hardware considerations on a SNOBOL machine known to the author is by Chai and DiGuiseppi [34] in 1965. They were concerned with the original SNOBOL language and investigated the representation of strings in memory, hardware hashing mechanisms, and rudimentary pattern-matching processors. Many of the considerations in SNOBOL apply to SNOBOL4 as well, but the additional data types, list-processing facilities, and much more powerful pattern matching require more elaborate design. Recently, a comprehensive investigation of these problems has been undertaken by Shapiro [35].

The approach of this research is to study the internal data structures and operations of the SNOBOL4 language and to use conventional hardware to obtain a highly efficient design. The internal data and register structures are based, in part, on the implementation of SNOBOL4 in SIL. However, the possibilities provided by hardware designed for SNOBOL4 operations permit more efficient and direct methods.

The concept of the descriptor has been extended so that the way in which fields are laid out depends on the type of descriptor. For example, a single bit distinguishes a string descriptor (which replaces the SIL qualifier) from other descriptors. The remainder of the string descriptor is divided into a base, offset, and length (in a somewhat different format than the SIL qualifier).

If a descriptor is not a string descriptor, a second bit determines whether or not it is a numeric-valued descriptor. Additional structure is provided for other types of data, including code descriptors. The fields of these descriptors are laid out differently, with varying field sizes as appropriate. All values, whether numeric or not, can be loaded into descriptor registers and manipulated there.

A set of machine operations, similar to SIL operations, is defined on these descriptors. These operations have the same syntax as for the field functions on defined data objects in SNOBOL4, although the fields vary in size depending on the type of descriptor. Thus, systems programming can be performed in SNOBOL4 on the SNOBOL4 machine.

The handling of strings is accomplished through character-string registers. When a string descriptor is loaded into a descriptor register, a string register is automatically associated with the descriptor register. The string register contains a limited number of characters of the string specified by the string descriptor, and it can be thought of as a snapshot of the specified string. As the fields of the string descriptor are manipulated, characters are automatically loaded from memory into the string register, replacing positions vacated by shifting.

String registers have several uses. During pattern matching, the subject string may be examined a character at a time. The pattern component may also be in a string register. This may occur in two ways. First, if the pattern element is a normal, ordered string, it can be matched character for character with the subject string in another string register. If the pattern component is an unordered string,

in which the order of the characters does not matter (for example, as an argument of SPAN), the string register contains a linear array of bits used to identify the characters of the alphabet present in the component.

The frequency of conversion between various forms of numeric-valued data has resulted in a design in which internal arithmetic is performed in decimal real format. Integers are considered to be a special form of real number whose value is the same as an integer-valued real. Hardware provides automatic conversion between numbers and numeric-valued strings. Thus, an operation on a numeric-valued descriptor register constructs the corresponding string in a string register, and conversely.

Software support for this SNOBOL4 machine includes a compiler, storage management routines, and subroutines for performing more complicated language functions. The compiler converts the SNOBOL4 source program into machine-language instructions (for most operations) or subroutine calls (in a few cases).

14.2.2 A Microprogrammed SNOBOL4 Machine

Recent advances in microprogramming [36–38] have stimulated a great deal of interest in realizations of programming languages in firmware [39–40]. Writable control store can be configured to emulate operations of a “machine” more suitable to the language than conventional hardware. Several proposals have been made to implement SNOBOL4 in this fashion. The design of an implementation for the MLP-900 [41–42] has been described by Syrett [43]. At least one effort is under way for the Meta 4 machine [44].

SNOBOL4, as implemented in SIL, is suggestive to proponents of a microprogramming approach. Although the designers did not intend it that way, SIL looks like a machine language for a SNOBOL4 machine. From this point of view, the SIL operations are implemented in micro code, at least to a first approximation.

As of this writing, these ideas have not been reduced to practice, and it would be premature to project the results. It should be noted, however, that the implementors of SNOBOL4 did not think in terms of a SNOBOL4 machine, and SIL was not designed to be a machine language. If this approach had been taken, the results would certainly have been somewhat different. The remarks in Section 13.4 about the distinction between fast memory (“registers”) and slow memory (“core”) are relevant in this respect. Certainly, some hierarchy of this kind would be useful, if not essential, in a microprogrammed implementation of SNOBOL4. Furthermore, the design of SIL often required compromises between efficiency and difficulty of implementation. As a result, there is a wide variation in complexity between operations. This is exemplified by an operation such as LOCAD, which could be written in terms of other SIL macros but is a separate macro because of the need to generate more efficient code. Many SIL operations could

be reduced in this way. A microprogrammed implementation might better start from a smaller set of operations on the fields of descriptors and qualifiers. Such a set of operations would likely sacrifice less in a microprogrammed environment and have the virtue of greater uniformity and simplicity.

APPENDICES

The Syntax of SNOBOL4

This appendix describes the syntax of SNOBOL4. A variety of systems are used for syntax description and individual preferences vary. BNF [45] has the longest tradition and is still most frequently used. BNF has the virtue of simplicity but lacks descriptive power. As a result, simple and frequently used types (such as “letter” and “identifier”) are cumbersome and unnatural in BNF. The description system used below reflects the author’s preference for an enriched BNF system, in which primitive definitions and functions are used to make descriptions more concise and straightforward.

It is not the author’s purpose to give a rigorous description of this system here. The syntax that follows should be clear, and the reader familiar with BNF and the rudiments of SNOBOL4 pattern matching should have no trouble grasping the idea.

Informally, the notation is similar to BNF, with certain obvious simplifications and the addition of some primitives. As in BNF and SNOBOL4, concatenation has precedence over alternation. There are seven additions to BNF.

- (1) A * before an expression indicates zero or more repetitions of that expression. (Compare this notation with the pattern-valued function ARBNO.)
- (2) An ! before an expression indicates one or more repetitions of that expression.

- (3) A ? before an expression indicates that the expression is optional.
- (4) Parentheses are used freely for grouping.
- (5) Primitive syntactic types are used where it is necessary to specify characters used by the description system itself. To describe SNOBOL4, the following primitive types are needed:

```

⟨eq⟩ for =
⟨lb⟩ for ⟨
⟨rb⟩ for ⟩
⟨lp⟩ for ⟨
⟨rp⟩ for ⟩
⟨star⟩ for *
⟨bar⟩ for |
⟨ques⟩ for ?
⟨exclam⟩ for !
⟨bl⟩ for the blank character

```

- (6) Primitive syntactic functions used are:

```

⟨any(arg)⟩
⟨span(arg)⟩
⟨break(arg)⟩

```

which are obvious analogs of SNOBOL4 primitives. The arguments of these functions may be any expressions that describe sets of characters.

- (7) Finally, the primitive type ⟨eol⟩ is used to signify the end of a line (such as the end of a card on card input or a carriage return on terminal input).

A.1 Character Syntax

The following type definitions include all the syntactically significant characters that occur in SNOBOL4, besides the primitive types listed above. Standard IBM 360 graphics are given. The internal representation of these IBM 360 characters is given in [24]. On other machines there are different graphics in some cases, especially among the special characters. Some machines lack lower-case letters. Internal representations also vary. Therefore, the type definitions that follow may be taken as a definition of the syntax for the IBM 360.

```

⟨semi⟩=;
⟨colon⟩=:
⟨dot⟩=.
⟨uscore⟩=_
⟨squote⟩='
⟨dquote⟩="

```

```

⟨not⟩=¬
⟨dollar⟩=$
⟨percent⟩=%
⟨slash⟩=/
⟨plus⟩=+
⟨minus⟩=-
⟨at⟩=@
⟨and⟩=&
⟨pound⟩=#"
⟨comma⟩=,
⟨digit⟩=(any(0123456789))
⟨letter⟩=(any( ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz))

```

A.2 Statement Syntax

The following type definitions culminate in the definition of a SNOBOL4 statement. The type names are chosen to correspond, at least roughly, to the terminology used in the text. Redundant definitions (such as ⟨unary⟩) are included for clarity.

```

⟨alphanumeric⟩=⟨digit⟩|⟨letter⟩
⟨break⟩=⟨dot⟩|⟨uscore⟩
⟨identifier⟩=⟨letter⟩*⟨(alphanumeric)|⟨break⟩⟩
⟨ob⟩=*⟨b1⟩
⟨mb⟩=!⟨b1⟩
⟨integer⟩!=⟨digit⟩
⟨real⟩=⟨integer⟩⟨dot⟩?⟨integer⟩
⟨oper⟩=⟨not⟩|⟨dollar⟩|⟨dot⟩|⟨percent⟩|⟨slash⟩|⟨plus⟩|
    ⟨minus⟩|⟨at⟩|⟨and⟩|⟨star⟩|⟨bar⟩|⟨ques⟩|⟨exclam⟩|⟨pound⟩
⟨unary⟩=⟨oper⟩
⟨binary⟩=⟨mb⟩?((⟨oper⟩|⟨star⟩⟨star⟩)⟨mb⟩)
⟨sqlit⟩=⟨quote⟩⟨break(⟨quote⟩)⟩⟨quote⟩
⟨dqlit⟩=⟨dquote⟩⟨break(⟨dquote⟩)⟩⟨dquote⟩
⟨literal⟩=⟨sqlit⟩|⟨dqlit⟩|⟨integer⟩|⟨real⟩
⟨element⟩=*⟨unary⟩(⟨identifier⟩|⟨literal⟩|⟨call⟩|⟨ref⟩|
    ⟨lp⟩⟨expr⟩⟨rp⟩)
⟨operation⟩=⟨element⟩⟨binary⟩(⟨element⟩|⟨expr⟩)
⟨expr⟩=⟨ob⟩?((⟨element⟩|⟨operation⟩)⟨ob⟩)
⟨arglist⟩=⟨expr⟩*(⟨comma⟩⟨expr⟩)
⟨call⟩=⟨identifier⟩⟨lp⟩⟨arglist⟩⟨rp⟩
⟨ref⟩=⟨identifier⟩⟨lb⟩⟨arglist⟩⟨rb⟩
⟨lfield⟩=?⟨(alphanumeric)⟨break(⟨blank⟩|⟨semi⟩)⟩⟩
⟨sfield⟩=⟨mb⟩⟨element⟩
⟨pfield⟩=⟨mb⟩⟨expression⟩

```

```

<ofield>=?(<mb><expression>)
<assign>=<mb><eq>
<goto>=<lp><expr><rp>|<lb><expr><rb>
<gfield>=?(<mb><colon><ob>?(<goto>)|<s><goto>
    ?(<ob><f><goto>)|<f><goto>?(<ob><s><goto>)))
<eos>=<ob>(<semi>|<eol>)
<arule>=<sfield><assign><ofield>
<mrule>=<sfield><pfield>
<rrule>=<sfield><pfield><assign><ofield>
<drule>=?<sfield>
<statement>=<lfield>(<arule>|<mrule>|<rrule>|<drule>)
    <gfield><eos>

```

Note that this description does not account for the *end* statement and is defective since the label END is not excluded by the definition of *<lfield>*. The syntax description system used above does not have the facility for making such exclusions. The definition of an *end* statement is

```
<end>=END?(<mb><lfield>)<eos>
```

A.3 Prototype Syntax

Prototypes of ARRAY, DATA, DEFINE, and LOAD are analyzed during program execution, not during translation. The definitions are:

```

<item>=?<identifier>
<itemlist>=<item>*<(<comma><item>)
<datapproto>=<identifier><lp><itemlist><rp>
<defineproto>=<identifier><lp><itemlist><rp><itemlist>
<loadproto>=<identifier><lp><itemlist><rp><item>
<signedinteger>=?(<plus>|<minus>)<integer>
<dimension>=<signedinteger>?(<colon><signedinteger>)
<arrayproto>=<dimension>*<(<comma><dimension>)

```

Syntax Tables

The form of the syntax table descriptions that follow is discussed in Section 7.1. The arguments of FOR correspond to the syntactic types defined in Appendix A. In addition, TERM corresponds to

$\langle \text{term} \rangle = \langle \text{semi} \rangle | \langle \text{rp} \rangle | \langle \text{rb} \rangle | \langle \text{comma} \rangle | \langle \text{bl} \rangle$

B.1 Syntax Tables Used by the Translator

```
BEGIN BINOPS
FOR(PLUS) PUT(BPLSOP) GOTO(TBLANK)
FOR(MINUS) PUT(BMNSOP) GOTO(TBLANK)
FOR(DOT) PUT(BDOTOP) GOTO(TBLANK)
FOR(DOLLAR) PUT(BDOLOP) GOTO(TBLANK)
FOR(STAR) PUT(BSTROP) GOTO(STARS)
FOR(SLASH) PUT(BSLHOP) GOTO(TBLANK)
FOR(AT) PUT(BATOP) GOTO(TBLANK)
FOR(POUND) PUT(BPNDOP) GOTO(TBLANK)
FOR(PERCENT) PUT(BPCTOP) GOTO(TBLANK)
FOR(EXCLAM) PUT(BXLMOP) GOTO(TBLANK)
FOR(BAR) PUT(BBAROP) GOTO(TBLANK)
FOR(AND) PUT(BANDOP) GOTO(TBLANK)
```

```
FOR(NOT) PUT(BNOTOP) GOTO(TBLANK)
FOR(QUES) PUT(BQUEOP) GOTO(TBLANK)
ELSE ERROR
END BINOPS

BEGIN CRDTYP
  FOR(COMMENT) PUT(CMTCOD) STOPSH
  FOR(CONTROL) PUT(CTLCOD) STOPSH
  FOR(CONTINUE) PUT(CTNCOD) STOPSH
  ELSE PUT(STMCOD) STOPSH
END CRDTYP

BEGIN DQLIT
  FOR(DQUOTE) STOP
  ELSE CONTIN
STOP DQLIT

BEGIN ELEMNT
  FOR(DIGIT) PUT(INTCOD) GOTO(INTGER)
  FOR(LETTER) PUT(IDCOD) GOTO(IDENTF)
  FOR(SQUOTE) PUT(LITCOD) GOTO(SQLIT)
  FOR(DQUOTE) PUT(LITCOD) GOTO(DQLIT)
  FOR(LP) PUT(PRNCOD) STOP
  ELSE ERROR
END ELEMNT

BEGIN FGOTO
  FOR(LP) PUT(FGOCOD) STOP
  FOR(LB) PUT(FDRCOD) STOP
  ELSE ERROR
END FGOTO

BEGIN FORWRD
  FOR(BLANK) CONTIN
  FOR(EQUAL) PUT(EQCOD) STOP
  FOR(RP) PUT(RPCOD) STOP
  FOR(RB) PUT(RBCOD) STOP
  FOR(COMMA) PUT(CMACOD) STOP
  FOR(COLON) PUT(CLNCOD) STOP
  FOR(SEMI) PUT(EOSCOD) STOP
  ELSE PUT(NBKCOD) STOPSH
END FORWRD
```

```
BEGIN GOTO
    FOR(S) GOTO(SGOTO)
    FOR(F) GOTO(FGOTO)
    FOR(LP) PUT(UGOCOD) STOP
    FOR(LB) PUT(UDRCOD) STOP
    ELSE ERROR
END GOTO

BEGIN IDENTF
    FOR(ALPHANUMERIC) CONTIN
    FOR(BREAK) CONTIN
    FOR(TERM) PUT(IDCOD) STOPSH
    FOR(LP) PUT(FNCCOD) STOP
    FOR(LB) PUT(REFCOD) STOP
    ELSE ERROR
END IDENTF

BEGIN INITBL
    FOR(BLANK) GOTO FORWRD
    FOR(SEMI) PUT(EOSCOD) STOP
    ELSE ERROR
END INITBL

BEGIN INTGER
    FOR(DIGIT) CONTIN
    FOR(TERM) STOPSH
    FOR(DOT) PUT(RLCOD) GOTO(REAL)
    ELSE ERROR
END INTGER

BEGIN LABEL
    FOR(ALPHANUMERIC) GOTO(LABELR)
    FOR(BLANK) STOPSH
    FOR(SEMI) STOPSH
    ELSE ERROR
END LABEL

BEGIN LABELR
    FOR(BLANK) STOPSH
    FOR(SEMI) STOPSH
    ELSE CONTIN
END LABELR
```

```
BEGIN NBLANK
    FOR(TERM) ERROR
    ELSE STOPSH
END NBLANK

BEGIN REAL
    FOR(DIGIT) CONTIN
    FOR(TERM) STOPSH
    ELSE ERROR
END REAL

BEGIN SGOTO
    FOR(LP) PUT(SGOCOD) STOP
    FOR(LB) PUT(SDRCOD) STOP
    ELSE ERROR
END SGOTO

BEGIN SQLIT
    FOR(SQUOTE) STOP
    ELSE CONTIN
END SQLIT

BEGIN STARS
    FOR(BLANK) STOP
    FOR(STAR) PUT(BXLMOP) GOTO(TBLANK)
    ELSE ERROR
END STARS

BEGIN STMEND
    FOR(SEMI) STOP
    ELSE CONTIN
END STMEND

BEGIN TBLANK
    FOR(BLANK) STOP
    ELSE ERROR
END TBLANK

BEGIN UNOPS
    FOR(PLUS) PUT(UPLSOP) GOTO(NBLANK)
    FOR(MINUS) PUT(UMNSOP) GOTO(NBLANK)
    FOR(DOT) PUT(UDOTOP) GOTO(NBLANK)
    FOR(DOLLAR) PUT(UDOLOP) GOTO(NBLANK)
    FOR(STAR) PUT(USTROP) GOTO(NBLANK)
```

```
FOR(SLASH) PUT(USLHOP) GOTO(NBLANK)
FOR(AT) PUT(UATOP) GOTO(NBLANK)
FOR(POUND) PUT(UPNDOP) GOTO(NBLANK)
FOR(PERCENT) PUT(UPCTOP) GOTO(NBLANK)
FOR(EXCLAM) PUT(UXLMP) GOTO(NBLANK)
FOR(BAR) PUT(UBAROP) GOTO(NBLANK)
FOR(AND) PUT(UANDOP) GOTO(NBLANK)
FOR(NOT) PUT(UNOTOP) GOTO(NBLANK)
FOR(QUES) PUT(UQUEOP) GOTO(NBLANK)
ELSE ERROR
END UNOPS
```

Figure B.1.1 illustrates the relationships among these syntax tables. The symbols ◦ and • indicate stop conditions (STOP and STOPSH) and ERROR, respectively.

B.2 Syntax Tables Used to Analyze Prototypes

Four syntax tables are used to analyze prototypes. ARRAY uses APROTO and APROTT. DATA, DEFINE, and LOAD use FPROTO and FPROTT. The tables are:

```
BEGIN APROTO
```

```
FOR(DIGIT) GOTO(APROTT)
FOR(PLUS) GOTO(APROTT)
FOR(MINUS) GOTO(APROTT)
FOR(COMMA) PUT(CMACOD) STOPSH
FOR(COLON) PUT(CLNCOD) STOPSH
ELSE ERROR
```

```
END APROTO
```

```
BEGIN APROTT
```

```
FOR(DIGIT) CONTIN
FOR(COMMA) PUT(CMACOD) STOPSH
FOR(COLON) PUT(CLNCOD) STOPSH
ELSE ERROR
```

```
END APROTT
```

```
BEGIN FPROTO
```

```
FOR(LETTER) GOTO(FPROTT)
FOR(COMMA) PUT(CMACOD) STOPSH
FOR(RP) PUT(RPCOD) STOPSH
ELSE ERROR
```

```
END FPROTO
```

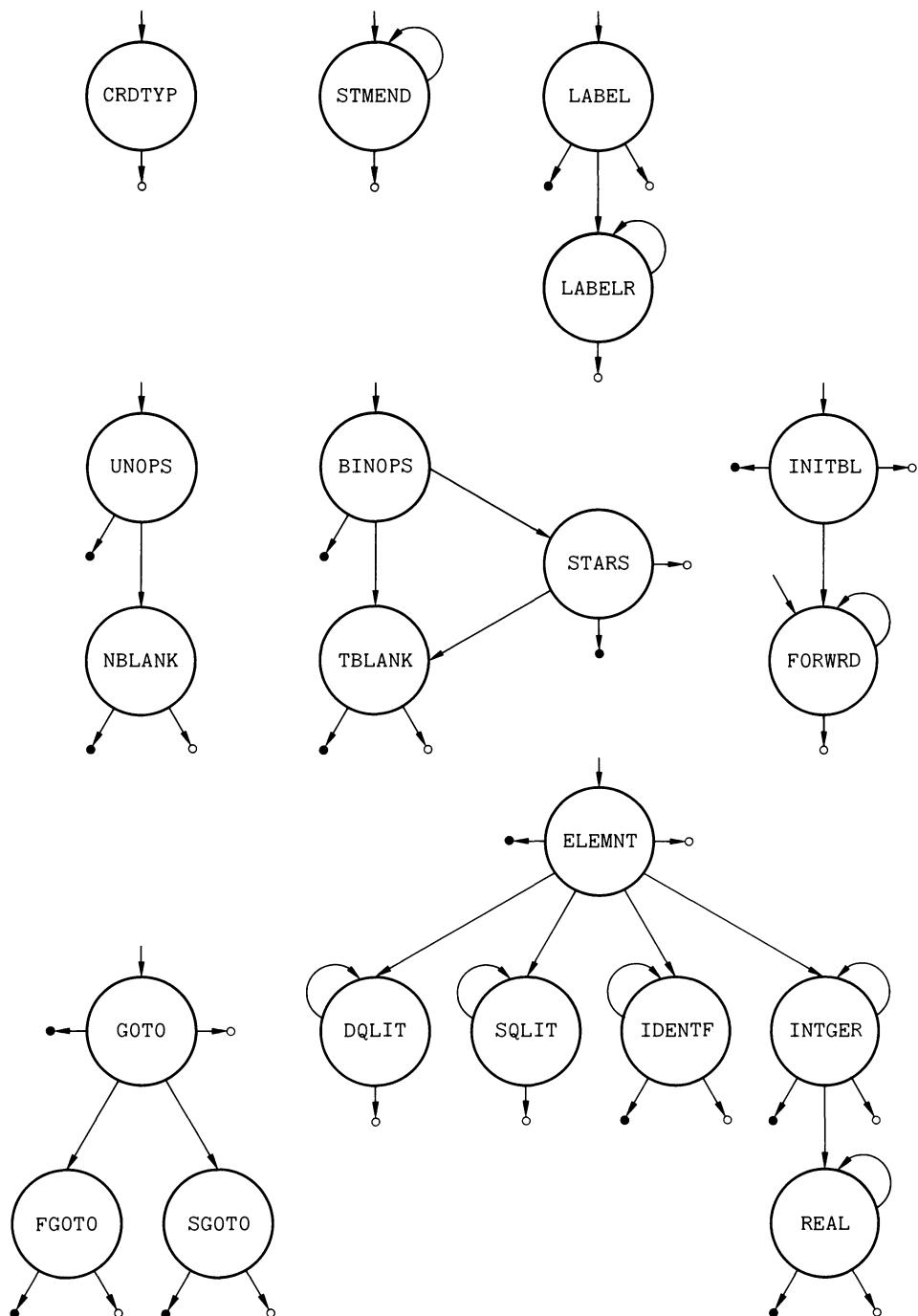


Figure B.1.1
Translator syntax table graphs.

```
BEGIN FPROTT
  FOR(ALPHANUMERIC) CONTIN
  FOR(BREAK) CONTIN
  FOR(LP) PUT(LPCOD) STOPSH
  FOR(COMMA) PUT(CMACOD) STOPSH
  FOR(RP) PUT(RPCOD) STOPSH
  ELSE ERROR
END FPROTT
```

Figure B.2.1 illustrates the relationships among these syntax tables.

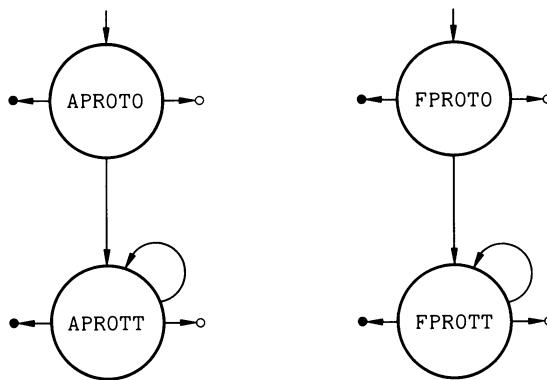


Figure B.2.1
Prototype syntax table graphs.

B.3 Syntax Table Used During Pattern Matching

The functions ANY, BREAK, NOTANY, and SPAN generate patterns that use STREAM and the syntax table MATCH. This table is modified during pattern matching by CLEART and SETUPT, as described in the text. The table has the following nominal definition initially:

```
BEGIN MATCH
  FOR(LETTER) CONTIN
  FOR(DIGIT) STOP
  FOR(COMMA) STOPSH
  ELSE ERROR
END MATCH
```

The reason for this apparently irrelevant definition is to provide information to the SNOBOL4 program that generates the IBM 360 syntax tables so that appro-

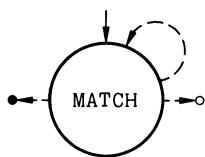


Figure B.3.1
Matching table syntax table graph.

priate exit code can be created. (See Section 11.1.5.) Figure B.3.1 illustrates this syntax table. The dashed lines indicate that the table is changed by matching procedures.

SIL Macros

The following sections list all SIL macros, arranged according to function. The mnemonic conventions given in Chapter 10 will be helpful in determining the meaning of macros not described in that chapter.

C.1 Data Assembly Macros

BUFFER	DESCR	QUAL
DBLOCK	FORMAT	STRING

C.2 Data Movement Macros

GETD	MOVQQ	PUTD
GETDC	MOVTC	PUTDC
GETIT	MOVTT	PUTIT
GETQC	MOVTV	PUTQC
GETVC	MOVVC	PUTTC
MOVDD	MOVVL	PUTVC
MOVLC	MOVVT	TRANDC
MOVLV	MOVVV	

C.3 Arithmetic Macros

ADDLV	DIVVV	NEGV
ADDRVV	EXPRVV	MPYVC
ADDVV	EXPVV	MPYRVV
DECLC	INCTC	MPYVV
DECVC	INCVC	SUBRVV
DIVRVV	NEGRV	SUBVV

C.4 Data Comparison Macros

CMPLL	CMPVV	EQLTT
CMPRRV	EQLDD	EQLVC
CMPSS	EQLLC	EQLVIC
CMPTIC	EQLTC	EQLVV
CMPVC		

C.5 Flag Macros

ADDF	CLRF	TESTF
ADDFI	CLRFI	TESTFI

C.6 Block Operation Macros

CLRBLK	LOCAD	LOCBD
CPYBLK		

C.7 Type Conversion Macros

CVTIR	CVTSR	CVTRS
CVTIS	CVTRI	CVTSI

C.8 String Manipulation and Syntax Table Macros

APDQQ	COMSQ	STREAM
CLEAR T	DELQC	SUBQQ
COMQNV	SETUPT	TRIMQQ

C.9 Procedure and Stack Macros

INITST	PROC	RCALL
POPD	PUSHD	RRETURN
POPQ	PUSHQ	STKPTR

C.10 Branching Macros

BRANCH

BRANIN

BRANLV

C.11 Pattern MacrosCMPPOS
COMBALCOMMML
CONALTCPYPAT
MAKPAT**C.12 Storage Management Macros**COMNVZ
COMSZ

LOCTTL

RELPTR

C.13 System Interface MacrosBKSPCE
DATE
ENFILE
INITEX
LINKLOAD
OUTPUT
PRINTQ
READQREWIND
TERMEX
TIME
UNLOAD**C.14 Tree-Building Macros**

APDSIB

APDSON

INSNOD

C.15 Assembly Control MacrosCOPY
ENDEQU
EQULOC

TITLE

C.16 Miscellaneous Macros

HASH

ORDNVT

RPLACE

Research Problems

The following problems are more difficult than the exercises in the body of the text. Depending on their interpretation, some constitute substantial research projects. Most call for language design in addition to implementation.

- (1) Design and implement language facilities that permit the changing of operator precedence and associativity during translation.
- (2) Design and implement language facilities for manipulating bit strings. Consider two cases: one in which the bit strings are of a fixed length, and another in which the bit strings are allowed to be of arbitrary length.
- (3) Design and implement language facilities for dealing with arbitrarily large integers.
- (4) Design and implement a new data type, STACK, that behaves like a push-down list.
- (5) In some applications, arrays are sparse, having only a few significant entries. In such cases the method of implementing arrays used in SNOBOL4 is wasteful of storage. Implement sparse arrays.

- (6) Extend operators to arrays. For example, if A and B are arrays,

$$C = A * B$$

creates an array in which each of the elements is the product of corresponding elements of A and B. Note that since an element of an array may itself be an array, operations on arrays are inherently recursive!

- (7) Design and implement “multidimensional” tables which permit references such as

$$T(A, B, C)$$

- (8) Implement a facility for printing the structure of a pattern.
- (9) Design and implement a facility for examining and modifying the structure of patterns.
- (10) Design and implement a format facility that does not use FORTRAN-like concepts.

Solutions to Selected Exercises

2.13.1 The index of LENGTH is out of bounds for words longer than ten characters, causing statement failure. The only effect is that words longer than ten characters are not counted.

2.13.2 Consider the following sentence:

ONE'S NON-ENGLISH BACKGROUND HAMPERS I/O.

WPAT produces the “words” ONE, S, NON, ENGLISH, BACKGROUND, HAMPERS, I, and 0. A good definition of a word is much more complicated than it may seem at first.

2.13.3 Replace statements 11 and 12 by statements such as

```
PLOOP  OUTPUT  =  GT(LENGTH(I),0) I ' : ' LENGTH(I)
I      =  LT(I,10) I + 1      :S(PLOOP)F(END)
```

2.13.4 An omitted second argument defaults to the null string, which is treated properly as a numeric zero when incremented. For zero counts, however, a null string rather than a zero is printed in the results (but see Exercise 2.13.3).

2.13.5 Add statements such as

```
&TRACE = 1000  
TRACE( .LENGTH(4), 'VALUE', 'FOUR-LETTER WORD' )
```

Note that while &TRACE may be set anywhere prior to processing of the input text, the trace association cannot be made until after the array LENGTH is created.

6.4.1 Any process may result from the evaluation of an argument. An argument may, for example, call a programmer-defined function, which, in turn, may perform any language operation, including input and output, pattern matching, definition of new functions, and even execution of the same procedure in which the argument is being evaluated. To prevent accidental destruction of temporary data, information which must be protected is saved on SYSSTK before argument evaluation and restored when evaluation is complete.

6.4.2 The prefix code for '10' and 10 is the same, except that in the first case the argument to lit₁ is a pointer to a natural variable and in the second it is an integer.

7.1.1 The ambiguity between a subject and pattern is a result of the fact that there is no explicit operator to indicate pattern matching. A notation such as

S ? P

could be used to indicate pattern matching. Note, however, that replacement is a function of three arguments.

7.1.2 An expression may be enclosed in parentheses but not in brackets. An array or table reference is preceded by an identifier. There is an entry for the left bracket in IDENTF.

7.1.3 A unary operator is the first thing to look for in an element.

7.1.4 The closing quotation mark must be of the same kind as the opening quotation mark. Otherwise 'ABS" would be a valid literal, and it would not be possible to enclose one kind of quotation mark within a literal bounded by the other. Two syntax tables are required to "remember" the kind of opening quotation mark.

7.3.1 These diagrams are left to the reader, but note that IDENT(X) results in a root node that has two sons. (See Section 2.2.2.)

7.3.3 The null expression returns a tree with the root lit₁, and a single son which is the null string.

7.3.4 For generality, the SNOBOL4 language permits a goto to be an expression. It might seem that there could be no valid expression except an element that returns a natural variable by name, as required by a goto. Consider, however, OPSYN('#', 'F', 2), where F is a programmer-defined function that returns a string by transferring to NRETURN.

7.3.5 A null rule generates no prefix code. A rule consisting only of a subject simply generates prefix code for the subject.

7.3.7 An omitted object corresponds to the null expression. (See Exercise 7.3.3.)

7.3.10 Code trees require a large amount of space compared with the space required for prefix code. One code descriptor in a tree node requires four other descriptors (counting the title). During interpretation, SYSSTK provides the transient space required to “remember” the structure of prefix code.

8.1.1 Most computers perform binary arithmetic. A string of n bits taken from a computation produces values in the range from 0 to $2^n - 1$, and hence no further computation is required if the number of chains is a power of two. The greater the number of chains, the shorter the length of the chains that have to be searched, on the average, providing that the quality of h_i is good. Each chain requires a header; thus, more storage is required for a larger number of chains.

8.1.3 The null string requires special handling in many cases (see Section 6.4.2). The form chosen for the null string is convenient since a zero field is easy and fast to check. If the null string were permitted to be a natural variable, a structure would have to be created for it like any other string. This structure would be created during program initialization. Some special mechanism would be required in the implementation of h_1 and h_2 since the null string has no characters. Finally, special zero tests for the null string would have to be replaced by comparisons with a descriptor known to point to the null natural variable.

8.1.4 For each string that is created, hash computation and a search of the table of natural variables must be performed. If the string does not exist, space must be allocated, including four descriptors in addition to the string itself. Furthermore, this scheme prevents shared use of substrings since each new string must be a separate natural variable. Usually only a relatively few strings are used as variables. For the rest, their representation as natural variables is inefficient. On the positive side, representing every string as a natural variable has the advantage of simplicity and uniformity. There is only one representation for strings throughout SNOBOL4. Every string has a unique representation as a pointer, which permits efficient searching of pair blocks. Operations that use strings as variables, such as indirect referencing, are simple and efficient.

8.1.5 Alternative methods of handling strings have many ramifications. One method is to have two representations for strings: one for strings as data objects and another for strings used as variables. In the former, strings can be constructed without consideration of the table of natural variables, and shared substrings are possible. When a string is used as a variable, a natural variable for that string is then created. An indirect reference applied to a data string requires creation of such a natural variable. (See Chapter 14.)

8.1.6 The algorithms for hash computation are highly machine-dependent. Consequently, the chains and the order of natural variables on chains vary consid-

erably from machine to machine. The resident data region, on the other hand, is mostly machine-independent in its structure and contents, making it a practical impossibility to prearrange the table of natural variables.

8.1.7 Only natural variables have unique identifying strings associated with them. Such strings are traditionally called “print names.” On the other hand, a variable such as $A(2,3)$ has no print name. Because of the syntactic significance of strings in the source program, only natural variables can be labels, names of functions, and so on. Labels are restricted to natural variables by the way SNOBOL4 is implemented, with the label descriptor in the natural variable. Any kind of variable could be a function since only a location, kept in FNCPBL, is needed to identify the variable.

8.2.1 Each defined function has a separate link descriptor. The number of arguments is in the T field of the link descriptor.

8.2.2 The DEFINE block for $F()$ consists of a single descriptor for the entry point, F. No special problems are encountered in handling a function which has no arguments. A call of $F()$ is a call with one (null) argument. This argument is evaluated by DEFFNC and discarded.

8.2.3 The prototype string may be arbitrarily long, depending on storage available. The total possible number of arguments and locals for a defined function may be limited in two ways. There may not be enough storage for the natural variables or for the DEFINE block, or there may not be enough room on SYSSTK to push all pointers to the arguments and natural variables. Both limits depend on the environment in which SNOBOL4 is run and on the program in which the definition occurs. There is, therefore, no explicit limit. Typically, the limitations are not of practical concern.

8.2.4 When a defined function is called, values are exchanged in order for $F, A_1, \dots, A_n, L_1, \dots, L_m$. Therefore, if F appears as an argument, the value of F on entry to function execution will be the value of the corresponding argument since this value is assigned last. Similarly, if F appears as a local, it will have a null value on entry. Any name that appears more than once will have a value corresponding to its last occurrence. A definition such as $F(F)$ is useful if the function F returns a modified version of its argument.

8.2.5 Values could be restored in any order except for the possibility of duplicate names, as discussed in the preceding exercise. Consider a function corresponding to the prototype $F(F)$. Suppose the value of F is 1 and a call $F(2)$ occurs. First, the value 1 is exchanged for the null string, the initial value for the function name. Then, this null is exchanged for 2, the value given in the call. If restoration were done in the same order, first the value 1 would be restored, and then the null string, resulting in different values of F before and after the call, which would be an error.

8.4.1 PROTOTYPE is easily implemented since the prototype pointer in the array can be returned without computation. The amount of space required to point to the prototype is only one descriptor. In most programs, the extra space devoted to prototype pointers is less than the space that would be occupied by a procedure for reconstructing prototypes.

8.4.2 The most important limit on the number of dimensions possible for an array is the amount of space high-dimensional arrays occupy. There are other limits, as indicated in Exercise 8.2.3. One curious case arises for a prototype of the form $1, 1, 1, 1, 1, \dots$. The array for such a prototype only requires $n + 4$ descriptors where n is the number of repetitions of 1. Presumably, such a prototype would not occur in a serious program.

8.4.3 According to the rules for treating omitted trailing arguments, a null string is supplied for the second index. The null string is equivalent to zero. If the second dimension has a positive lower bound, as is usually the case, the zero index is out of bounds and the reference fails, which may be mysterious to the erring programmer.

8.5.1 An array reference with one subscript, such as $X\langle N \rangle$, cannot be distinguished syntactically from a table reference. It could be either and *might* be one at one time and the other at another time. The data type of X determines the type of reference at the time the reference is made.

8.5.2 If X is an array, $X\langle 1 \rangle$ and $X\langle '1' \rangle$ reference the same element since the string '1' is automatically converted to the integer 1. If X is a table, $X\langle 1 \rangle$ and $X\langle '1' \rangle$ reference different elements since any data object can be used to reference a table and no data-type conversions are performed. This aspect of table referencing may cause subtle errors in source programs.

8.5.3 Entries in a table appear physically in the same order in which they are first referenced chronologically.

8.5.4 Entries in tables are variables, and references to them may occur throughout the resident and allocated data regions. An example is

$X = .T\langle K \rangle$

where T is a table. Thus, copying a pair block into a new area would require relocating all such references, a complicated and time-consuming task.

8.5.5 Use of integers in V fields to indicate special situations implies that these integers are distinguishable from addresses in the data regions. It is reasonable to assume that this is true of zero and one.

8.5.6 The creation of a table entry as a result of a table reference is a (possibly unfortunate) by-product of the generality of the SNOBOL4 system. Since a table reference is a variable, the referencing procedure must return a location, signifying return by name, assuming an assignment may be made. In fact, the value may

simply be tested. An example is `IDENT(T⟨K⟩)`, which may be designed to see if `K` is in the table `T`. However, for each distinct `K`, a variable `T⟨K⟩` is created and remains part of the table. On the other hand, as mentioned in the solution to Exercise 8.5.4, `.T⟨K⟩` is a reference to a variable that requires retention of the variable `T⟨K⟩`. Avoiding this problem is difficult. Table entries with null values can be deleted (“made empty”) only if no references to them exist. An automatic facility for doing this would be complicated, and circumstances under which it should be done are not easily determined.

8.5.7 A built-in function could be used to mark a table as being frozen. When a reference to a frozen table is made and the referencing index is not found in the frozen table, the reference could fail, rather than creating a new entry.

8.5.8 The natural alternative to a linear structure is a hashed structure similar to the table of natural variables. Note that all kinds of indexing values must be handled, not just strings.

8.7.1 The argument of `ANY` is expected to be a string. The order of the characters is unimportant since each character results in a syntax table entry, and the order of the entries is irrelevant. An integer argument is converted to a string. Thus, `1234567890` is equivalent to `'1234567890'`, and `0123456789` is equivalent to `'123456789'`. The zero is lost during conversion in the last case.

8.7.2 If tables were constructed during pattern construction when `ANY`, `BREAK`, `NOTANY`, and `SPAN` are executed, it would not be necessary to constantly set up tables during pattern matching, with a resulting saving in execution time. However, the space required for a table for each occurrence of `ANY`, `BREAK`, `NOTANY`, and `SPAN` might be significant.

8.7.3 If the argument of `@_1` is an unevaluated expression and evaluation of that expression by `m@_3` signals failure, then `m@_3` signals failure.

8.7.4 `LEN(*N)` produces a pattern component for `mlen3`, whose third argument is the unevaluated expression `*N`. `*LEN(N)` produces a pattern component for `m*3`, whose third argument is the unevaluated expression `*LEN(N)`. During pattern matching, `mlen3` evaluates `N`, but `m*3` constructs a pattern component for `LEN(N)`.

8.7.5 The value of `*¬ARB` is a pattern component for `m*3`. When `m*3` is called during pattern matching, `¬ARB` is evaluated, signaling failure. Therefore, `m*3` signals failure.

8.7.6 When `m*3` evaluates `.A⟨I⟩` during pattern matching, the result has the data type `NAME`, which is an error.

8.7.7 The argument of `@_1` may be an unevaluated expression, which is a value until it is evaluated by `m@3`, at which time a name is expected. An example is `@*A⟨I⟩`.

8.7.8 If P is used in constructing another pattern, the entire structure for P must be copied into Q. However, *P is simply a single component that points to P. Therefore, *P uses less space than P.

8.7.9 Consider the statement

```
'(A+CD)' ARB BAL LEN(3)
```

Initially, ARB matches the null string, BAL matches (A+CD), and LEN(3) fails. Subsequently, ARB matches (, BAL matches A, and LEN(3) matches +CD. Because BAL may match a shorter substring at a larger cursor position, BAL does not transmit length failure.

8.7.10 Although intuitively P and Q describe the same class of strings, the order of the alternatives causes P and Q to behave differently. For the subject string BAAA, P matches BAAA as described in the text, but Q simply matches B.

8.7.11 Consider a statement of the form

```
P ARB ARB 'Y'
```

where P is a string of n Xs. This statement inevitably fails. Using the heuristics, the first ARB matches the null string, and the second ARB matches successively longer strings as Y fails to match. Finally, length failure is signaled, transmitted backward by the ARBs, and the pattern match fails. The number of calls to matching procedures is on the order of n. Without the heuristics, the first signal of length failure does not terminate pattern matching, the first ARB advances the cursor position, and the process starts over on a string of n – 1 characters, and so on. The number of calls to matching procedures is on the order of n!. Therefore, the ratio is on the order of n!. Hence, for arbitrarily large values of n, the heuristics make an arbitrarily large improvement in speed.

8.7.12 Most programs do not contain patterns with the kind of combinatorial problems demonstrated in the solution above. The time spent testing for heuristics is often greater than the time saved by using them.

8.7.13 One solution to the combinatorial problems of pattern matching is to limit the number of calls to matching procedures during a given pattern match. If this number is exceeded, the pattern match could simply fail. An unprotected keyword, such as &MATCHLIMIT, would be a natural way to give the programmer control over combinatorial problems.

8.9.1 Use of file names instead of unit numbers involves evaluating the second argument of INPUT and OUTPUT as a string rather than as an integer and having the first descriptor in input and output blocks point to natural variables instead of being integers. Handling of the file name when I/O is actually done depends on the system interface.

8.9.2 FORTRAN formats provide a minimal amount of formatting, such as carriage control and literals, within a well-known context. Existing format conver-

sion routines can be used, avoiding the need for additional coding. FORTRAN formats are not natural to SNOBOL4, however, and introduce an alien and archaic feature.

8.9.3 Space could be reserved in natural variables for flags to be set if a natural variable is given I/O associations. When checking for possible I/O, the flags could be checked directly, rather than having to search a list. This procedure was used in SNOBOL3.

8.9.4 Space would have to be provided for I/O flags for each variable in the structure, and the appropriate flags would have to be tested for each variable reference, regardless of how the reference is made. It is impractical to put the flags in the *value* of a variable since such flags would have to be cleared whenever the value is used elsewhere and copied whenever the variable is assigned a new value.

8.10.1 Only two arguments are provided in the code block. If a function descriptor with an argument count greater than two is inserted, execution of the function would cause an attempt to fetch an argument beyond the block, with unpredictable results. This problem is overcome by simply setting the argument count to two when the code block is constructed.

8.11.1 In many circumstances programmers prefer to retain control, even briefly, however severe the error is. In some cases, such as exhausting the allocated data region, the programmer could deliberately eliminate data objects to make more space available. The problem is, of course, that returning control to the program in such cases may cause subsequent program malfunction. Looping, for example, must be avoided. Overflowing SYSSTK is a particularly dangerous situation because all SNOBOL4 procedures use SYSSTK. As a result, execution of an operation that is not itself recursive may cause overflow again.

8.11.2 Tracing &ERRTYPE may invoke a programmer-defined trace procedure which may execute any language operation. The code base pointer and code offset must be saved. If pattern matching is in progress, the cursor position, the location in PATSTK, and so forth must be saved also.

8.11.3 Continuing execution after an attempt to transfer to an undefined label is easy to do, but it is likely to cause more problems than it solves. Simply going on to the next statement is possible, but likely to cause serious malfunction of the source program.

8.11.4 One way of treating transfer to an undefined label as nonsevere is to transfer instead to some standard location in the source program. This location could be a fixed, “distinguished” label such as LBLERR or could be set by the program, as, for example, the value of a keyword.

8.12.1 Because of automatic conversion of data types, \$2 and '\$2' are equivalent and amount to using the string 2 as a variable. '\$'2.7' similarly uses the string 2.7 as a variable.

8.12.2 $\$\$.X$ is the same as X . $\$\X uses the value of the value of X as a variable, which is acceptable, provided the value of X and the value of the value of X are variables. The expression $\dots X$ is erroneous since the argument of \dots must be a variable, but the second (inner) \dots operator returns by value.

8.12.3 The expression $\$*X$ is erroneous since $*$, returns a value which has the data type EXPRESSION. $\*X simply fails since $*X$ always succeeds. $\neg\$*X$ is erroneous because $\$*X$ is erroneous.

8.12.4 Both $*$, and \neg , have one argument to be skipped. However, that argument may be a function which itself has several arguments, and so on. An example is $\neg\text{IDENT}(F(X), G(X, Y))$. Thus, CODSKP is much like INTERP, except that function procedures are not executed and the number of arguments must be counted by CODSKP itself.

8.12.5 When failure occurs, interpretation is suspended and remaining parts of the argument remain unevaluated. The failure signal is converted to return by value, and evaluation must resume beyond the unevaluated parts of the argument. If evaluation succeeds, however, the entire argument has been evaluated.

9.1.1 There should be no correlation between the logical and physical order of natural variables on a chain.

9.1.2 The correspondence between the physical and chronological order of allocated objects is quite useful in debugging the SNOBOL4 system. Objects often can be located in dumps by knowing the relative time of their creation.

9.2.1 “Garbage” refers to unneeded objects. The marking process never references garbage items, and, in fact, it is the nongarbage items that are collected. The more garbage there is, the faster the regeneration process is.

9.2.2 Interior pointers point below the title of a block and may result from array references, tables references, and field references. Such interior pointers exist transiently in some basic blocks and for arbitrary periods of time in SYSSTK, which is a basic block. The unary name operator applied to such a reference results in an interior pointer which is a source-language data object. Such data objects may occur in basic blocks and in the allocated data region as values of variables.

9.2.3 Every descriptor in a block must be examined by the marking process. Only the value and label descriptors of a natural variable are examined. The N flag in the title of a natural variable is used to distinguish between the two cases.

9.2.4 Strings are stored in descriptor-sized units within natural variables in the allocated data region. Any pattern of bits may occur in a string, and in particular a string may appear to be a descriptor with a T flag. Thus, a pointer into a string might result in an erroneous indication of a title and, hence, serious program malfunction.

9.2.6 New natural variables are created with null values and zero label descriptors. If such a variable is destroyed during storage regeneration, a future reference to the string simply results in the recreation of the natural variable in the same condition.

9.2.7 The V fields of descriptors are used for all kinds of data, including integers and real numbers. The A flag provides a simple and efficient way of separating pointers from other types of V fields that are not pointers but might have values within the address bounds.

9.2.8 A recursive process such as marking requires space on SYSSTK. Stack overflow can occur during storage regeneration, although that does not happen frequently.

9.2.9 Use counts may involve much unnecessary incrementing and decrementing. Marking performs the equivalent of use counting, except that it is done only when needed and only for objects that must be kept. Use counts are awkward in SNOBOL4 because of structured variables. A structure must be kept if it is referenced or any variable in it is referenced. Thus, if a pointer to a variable in a structure is created, the use count of the structure must be incremented, and similarly decremented when the pointer is destroyed. This involves locating the use count for the structure, which may be time-consuming and complicated.

9.2.11 The correspondence between physical order and chronological order is preserved.

9.2.12 It is not necessary to perform pointer adjustment on pointers that point to blocks at the top of the allocated data region since these blocks will not be moved.

9.2.14 Blocks must be moved starting at the top (beginning) of the allocated data region, working toward the end. Otherwise, data may be overwritten.

9.2.15 Blocks that are not unmarked will not be processed during the next regeneration, and needed objects may be destroyed.

9.2.17 Moving blocks only as necessary to fill in holes left by unmarked blocks offers the possibility of moving fewer blocks and performing less relocation. Since blocks are of various sizes, algorithms for filling holes become complicated, however. In general, there will be spaces which cannot be completely filled, and relocation is necessary to avoid fragmentation. Relocating all blocks upward has the merits of simplicity and generality.

9.2.18 The larger the allocated data region is, the longer the chains of natural variables may become before storage regeneration is required. If these chains contain many “garbage” entries, the time spent searching the table of natural variables may become excessive. After storage regeneration, the chains are shortened and the search is faster. Thus, more frequent storage regeneration, forced by a smaller allocated data region, may increase the running speed of the SNOBOL4 system.

10.3.1 A statement consisting only of the subject INPUT results in prefix code consisting of a single operand descriptor. Since there is no function descriptor, the subject is ignored. In particular, no input is performed since no procedure requires the value of INPUT.

10.3.2 If too few arguments are provided in the call of a built-in function, the translator supplies null strings as necessary. If too many arguments are provided, the translator permits the extra arguments to appear in the prefix code on the assumptions that the function will be redefined before execution and that the number of arguments given will be correct for the new definition. Function procedures (such as TRIM) evaluate their own arguments but do not check to see that the proper number of arguments are provided. This check is centralized in INVOKE as indicated. If this check were not made, an incorrect number of arguments would cause malfunction of the interpreter. Too few arguments (which can only result from the use of OPSYN) would result in a function procedure consuming prefix code beyond its arguments. Too many arguments would result in a function procedure leaving the extra arguments for other procedures to evaluate (erroneously). For example, in

```
IDENT(TRIM(X,Y),Z)
```

IDENT would compare TRIM(X) with Y since TRIM would leave the argument Y unevaluated.

10.3.3 In STRVAL, the integer data type of the argument 2 causes transfer to STRI, which calls an allocation procedure to obtain a natural variable for the string 2, which is returned to TRIM. Since this string has no trailing blanks, it is unmodified by TRIMQQ and the string 2 is returned as value.

11.1.1 The definition of LOCBD is the same as that for LOCAD, except that the comparison instruction is

```
C&SYSNDX CLC      16(8,15),&D3
```

11.1.2

	MACRO	
&LOC	MOVTV	&D1,&D2
&LOC	MVC	&D1+5(3),&D2+1
	MEND	

11.1.3

	MACRO	
&LOC	PUTD	&D1,&D2,&D3
&LOC	L	1,&D2
	L	2,&D1
	LD	0,&D3
	STD	0,0(1,2)
	MEND	

11.1.4

L	1 , D
SLA	1 , 2
B	*+0 , (1)
B	L1
B	L2
B	A0001
B	L3
A0001	EQU *

11.1.5 The four registers R11 through R14 correspond to two descriptors. R11 and R13 fall into V field positions while R12 and R14 span F and T field positions. Since the high-order bytes of R12 and R14 are zero, the F fields of the corresponding descriptor positions on SYSSTK are also zero. If this were not the case, stray bits in the F fields might be misinterpreted during storage regeneration.

11.2.1

LOCBD	MACRO	D1 , D2 , D3 , F , S
	LOCAUBD	D1 , D2 , D3 , F , S , (B1+B1)
	ENDM	

11.2.2

MOVTV	MACRO	D1 , D2
	SA1	D1
	SA2	D2
	AX3	B6 , X2
	MX4	42
	BX5	X4*X1
	BX6	X3+X5
	SA6	A1
	ENDM	

11.2.3

PUTD	MACRO	D1 , D2 , D3
	SA1	D1
	SA2	D2
	SA3	D3
	AX4	B6 , X1
	BX6	X3
	AX5	B6 , X2
	IX3	X4+X5
	SA6	X3
	ENDM	

11.2.4

SA1	D
AX2	B6,X1
SA3	X2+↓000001-1
SB2	X3
JP	B2
↑↓000001	BSS 0
+	VFD 60/L1
+	VFD 60/L2
+	VFD 60/↑000002
+	VFD 60/L3
+	VFD 60/↑↓000002
↑↓000002	BSS 0

Glossary

Implementation Terms and Symbols

A descriptor:	the first descriptor in an A-B pair.
A-B pair:	a pair of adjacent descriptors in a pair block used to associate two pieces of information.
allocated data:	data allocated by storage management procedures.
allocated data region:	a large region of contiguous descriptors from which blocks and natural variables are allocated.
B descriptor:	the second descriptor in an A-B pair.
basic block:	a block in the resident data region that may contain pointers to the allocated data region.
block:	a group of contiguous descriptors headed by a title descriptor.
chain:	a linked list of natural variables.
chain descriptor:	a descriptor in a natural variable, used to point to the next descriptor on the chain and to hold the order number.

chain offset:	a hash number used to select a chain of natural variables.
code base pointer:	the pointer to a position in prefix code corresponding to the last labeled statement.
code block:	a block of descriptors consisting of prefix code.
code offset:	the number added to the code base pointer to get the address of the next descriptor of prefix code to be processed.
code tree:	a structure used to represent an expression during translation.
CODSKP:	an interpreter procedure that skips over prefix code without executing it.
connector descriptor:	an operand descriptor in the prefix code of a pattern component which contains the offsets of the alternate and subsequent of the component.
current stack pointer:	the stack pointer (q.v.).
DATA block:	a block that contains the description of a defined data type.
data-type pair block:	a pair block that associates data-type codes with the corresponding data-type names.
DEFDAT:	the interpreter procedure that creates programmer-defined data objects.
DEFFNC:	the interpreter procedure that handles calls of programmer-defined functions.
DEFINE block:	a block that contains the description of a defined function.
definition descriptor:	a descriptor following a link descriptor, used to point to defining information for certain types of functions.
descriptor:	a fundamental unit of data used to represent all source-language data objects and most internal data.
DTPBL:	the data-type pair block.

DWDTH:	the address width of a descriptor.
ELEM:	the translator procedure that analyzes elements.
EXPR:	the translator procedure that analyzes expressions.
F field:	a field in the descriptor used to contain the flags that relate to the nature of the descriptor.
FIELD:	the interpreter procedure that handles references to fields of programmer-defined data objects.
FIELD block:	a block that associates field offsets with data types for a field defined on programmer-defined data objects.
FNCPBL:	the function pair block.
FORTXT:	the translator procedure that handles the input text stream.
free pointer:	the pointer into the allocated data region that identifies the beginning of free space for allocation.
function descriptor:	a descriptor in prefix code that points to a function procedure.
function pair block:	a pair block that associates function names with their link descriptors.
function procedure:	a procedure that implements a built-in function.
hash function:	a function that produces a pseudo-random number from a character string.
heuristic descriptor:	an operand descriptor in the prefix code of a pattern component which contains information used to increase the speed of pattern matching.
INPBL:	the input pair block.
input pair block:	a block that describes an input association, containing a unit number and a record length.
input text stream:	the source-program input to the translator.
interior pointer:	a pointer into the body of a block, below the title.

INTERP:	the basic interpretive procedure.
interpreter:	a collection of procedures that executes the program as it is represented by prefix code.
INVOKE:	a procedure, called by INTERP and function procedures, which accesses function procedures.
keyword pair block:	a pair block that associates keyword names with keyword values. There are two keyword pair blocks, one for protected keywords and one for unprotected keywords.
L field:	a field in the qualifier which contains the length of the given string.
label descriptor:	a descriptor in a natural variable which points to a location in prefix code if that variable is used as a label.
link descriptor:	a descriptor that forms a link between object code and a function procedure.
link descriptor pair:	a link descriptor followed by a definition descriptor (q.v.).
MARK:	the storage management procedure that marks structures in the allocated data region which must be saved during a storage regeneration.
matching procedure:	a procedure that performs pattern matching.
minimum match length:	the fewest number of characters a pattern component can match.
name:	a variable.
natural variable:	a structure which contains a nonnull string.
NULSTR:	the location of a descriptor in the resident data region which contains a representation of the null string.
O field:	a field in the qualifier which contains the offset of the beginning of a string from the location pointed to by the V field of the qualifier.

operand descriptor:	a descriptor in prefix code that contains an operand (argument).
order number:	a hash number used to order natural variables on a chain.
OUTPBL:	the output pair block.
output pair block:	a block that describes an output association, containing a unit number and a pointer to a natural variable for the output format.
pair block:	a block of descriptors logically arranged as a series of A-B descriptor pairs. Pair blocks are used for associating pieces of information. <i>See also</i> specific pair blocks.
PATSTK:	a stack used by pattern-matching procedures.
pattern component:	a section of pattern structure which consists of a call to a matching procedure.
PKYPBL:	the protected keyword pair block.
prefix code:	a representation of a SNOBOL4 program as a block of function and argument descriptors in which the function descriptors precede their arguments. Prefix code is created by the translator and executed by the interpreter.
procedure:	a logical unit of program.
procedure linkage:	linkage between prefix code and the procedure that is executed. <i>See</i> link descriptor.
qualifier:	a fundamental unit of data used to represent strings.
QWDTH:	the address width of a qualifier.
resident data region:	a region of data that is part of the SNOBOL4 system and is created by assembling SNOBOL4 as written in SIL.
residual:	the fewest number of characters that the pattern subsequent to a component can match.
return signal:	a signal made by a procedure when returning from a call, indicating the type of return.

SCNR:	the interpreter procedure that directs the pattern-matching process.
SIL:	SNOBOL4 Implementation Language. The language in which the machine-independent version of SNOBOL4 is written.
source program:	program written in SNOBOL4 which is input to the translator.
stack:	a push-down list used for temporary storage, generally on a last-in, first-out (lifo) basis.
stack pointer:	a pointer to the next available location on SYSSTK.
storage manager:	a collection of procedures that allocate and re-organize the allocated data region.
STREAM:	a process which analyzes text using syntax tables.
syntax table:	a table used to represent the token structure of the SNOBOL4 syntax.
SYSSTK:	a push-down list used for recursive calls and for temporarily saving the values of descriptors and qualifiers.
T field:	a field in the descriptor used to represent data-type codes, the sizes of structures, and so on.
TEXTQL:	a qualifier which points to the text being operated on by the translator.
title descriptor:	the first descriptor in every allocated structure.
trace-type pair block:	a pair block that associates types of traces with the data structures used by those types of traces.
translator:	a collection of procedures that convert the source-language program into prefix code.
TTPBL:	the trace-type pair block.
UKYPBL:	the unprotected keyword pair block.
V field:	a field of the descriptor used to represent integers, real numbers, pointers, and so on.

value descriptor:	the second descriptor in a natural variable, used to contain the value of the natural variable.
value trace pair block:	a pair block used to associate the names of traced variables with prefix code blocks used to perform the trace.
VTRPBL:	the value trace pair block.

References

1. Griswold, R. E. "A Guide to the Macro Implementation of SNOBOL4," SNOBOL4 Project Document S4D8d. Bell Telephone Laboratories, Incorporated. Holmdel, N.J. July 16, 1971.
2. Griswold, R. E. "SNOBOL4 Macro Source for Version 3.0," SNOBOL4 Project Document S4D17. Bell Telephone Laboratories, Incorporated. Holmdel, N.J. July 15, 1970.
3. Griswold, R. E. "IBM 360 Subroutines for Version 3.0 of SNOBOL4," SNOBOL4 Project Document S4D19. Bell Telephone Laboratories, Incorporated. Holmdel, N.J. July 16, 1970.
4. Griswold, R. E. "IBM 360 Macro Definitions for Version 3.0 of SNOBOL4," SNOBOL4 Project Document S4D20. Bell Telephone Laboratories, Incorporated. Holmdel, N.J. July 17, 1970.
5. Griswold, R. E. "SNOBOL4 Source and Cross-Reference Listings for Version 3.7," SNOBOL4 Project Document S4D26a. Bell Telephone Laboratories, Incorporated. Holmdel, N.J. July 15, 1971.
6. Griswold, R. E. "New Notation and Terminology for SNOBOL4 Source Material," SNOBOL4 Project Document S4D27. Bell Telephone Laboratories, Incorporated. Holmdel, N.J. August 25, 1971.
7. Griswold, R. E. "SNOBOL4 Written in SIL," SNOBOL4 Project Document S4D28. Bell Telephone Laboratories, Incorporated. Holmdel, N.J. August 25, 1971.
8. Griswold, R. E., J. F. Poage, and I. P. Polonsky. *The SNOBOL4 Programming Language*, 2nd ed. Prentice-Hall, Inc., Englewood Cliffs, N.J. 1971.
9. Farber, D. J., R. E. Griswold, and I. P. Polonsky. "SNOBOL, A String Manipulation Language," *Journal of the Association for Computing Machinery*, Vol. 11, No. 1 (January, 1964), pp. 21-30.
10. McIlroy, M. D. "A String Manipulation Program for FAP Programs." Bell Telephone Laboratories, Incorporated. Murray Hill, N.J. Unpublished, 1962.

11. Farber, D. J., R. E. Griswold, and I. P. Polonsky. "The SNOBOL3 Programming Language." *Bell System Technical Journal*, Vol. XLV, No. 6 (July-August, 1966), pp. 895-944.
12. Farber, D. J., et al. "Programming Machine-Language Functions for SNOBOL3." Bell Telephone Laboratories, Incorporated. Holmdel, N.J. Unpublished, May 13, 1965.
13. Griswold, R. E., and I. P. Polonsky. "Tree Functions for SNOBOL3." Bell Telephone Laboratories, Incorporated. Holmdel, N.J. Unpublished, February 1, 1965.
14. Griswold, R. E. "Linked-List Functions for SNOBOL3." Bell Telephone Laboratories, Incorporated. Holmdel, N.J. Unpublished, June 1, 1965.
15. Strauss, H. J. "External Functions for SNOBOL4," SNOBOL4 Project Document S4D10c. Bell Telephone Laboratories, Incorporated. Holmdel, N.J. July 31, 1970.
16. Gill, Arthur. *Introduction to the Theory of Finite-State Machines*. McGraw-Hill Book Company, Inc., New York. 1962.
17. Peterson, W. W. "Addressing for Random-Access Storage." *IBM Journal of Research and Development*, Vol. 1, No. 2 (1957), pp. 130-146.
18. Morris, R. "Scatter Storage Techniques." *Communications of the Association for Computing Machinery*, Vol. 11, No. 1 (1968), pp. 38-44.
19. Brooks, F. P., and K. E. Iverson. *Automatic Data Processing, System 360 Edition*. John Wiley, New York. 1969. Chapter 7.
20. Gimpel, J. F. "The Theory and Implementation of Pattern Matching in SNOBOL4 and Other Programming Languages," SNOBOL4 Project Document S4D24. Bell Telephone Laboratories, Incorporated. Holmdel, N.J. February 19, 1971.
21. "7090 Bell Telephone Laboratories Programmer's Manual." Bell Telephone Laboratories, Incorporated. Murray Hill, N.J. Unpublished, 1963.
22. IBM Systems Reference Library. "IBM System/360 Operating System Assembler Language," 6th ed. International Business Machines Corporation. June, 1969.
23. Control Data Corporation. "6400/6500/6600 COMPASS Reference Manual." *Applications Development Bulletin*. Control Data Corporation Documentation Department. March, 1967.
24. IBM Systems Reference Library. "IBM System/360 Principles of Operation," 7th ed. International Business Machines Corporation. January 13, 1967.
25. IBM Systems Reference Library. "IBM System/360 Operating System Concepts and Facilities," 6th ed. International Business Machines Corporation. July, 1969.
26. Thornton, J. E. *Design of a Computer, The Control Data 6600*. Scott, Foresman and Company, Glenview, Ill. 1970.
27. Control Data Corporation. "Control Data 6400/6600 Computer Systems SCOPE Reference Manual." Control Data Corporation Documentation Department. September, 1966.
28. IBM Systems Reference Library. "IBM System/360 Model 65 Functional Characteristics." International Business Machines Corporation. January 10, 1966.
29. Corbató, F. J., and V. A. Vyssotsky. "Introduction and Overview of the MULTICS System." *AFIPS Conference Proceedings*, Fall Joint Computer Conference, Vol. 27, Part 1. Spartan Books, Washington, D.C. 1965, pp. 185-196.

30. Gimpel, J. F. "A User Manual for BLOCKS Version 1.4," SNOBOL4 Project Document S4D11c. Bell Telephone Laboratories, Incorporated. Holmdel, N.J. August 15, 1970.
31. Gimpel, J. F. "A Guide to the Implementation of SNOBOL4B," SNOBOL4 Project Document S4D13. Bell Telephone Laboratories, Incorporated. Holmdel, N.J. August 11, 1969.
32. Dewar, Robert B. K. "SPITBOL Version 2.0," SNOBOL4 Project Document S4D23. Illinois Institute of Technology, Chicago, Ill. February 12, 1971.
33. Shantz, P. W., et al. "WATFOR—The University of Waterloo FORTRAN IV Compiler." *Communications of the Association for Computing Machinery*, Vol. 10, No. 1 (January, 1967), pp. 41–44.
34. Chai, David T., and Jack DiGuiseppi. "A Study of System Design Considerations in Computers for Symbol Manipulation." University of Michigan, Department of Electrical Engineering. Final Report for NSF Grant GP 786, Report 05635-1-F. May, 1965.
35. Shapiro, M. D. "System Architecture Considerations of a SNOBOL Machine," Purdue University, Department of Computer Science. Doctoral Dissertation, 1972.
36. Wilkes, M. V. "The Growth of Interest in Microprogramming—A Literature Survey." *Computing Surveys*, Vol. 1, No. 3 (September, 1969), pp. 139–145.
37. Rosin, R. F. "Contemporary Concepts in Microprogramming and Emulation." *Computing Surveys*, Vol. 1, No. 4 (December, 1969), pp. 197–212.
38. Husson, S. S. *Microprogramming Principles and Practices*. Prentice-Hall, Inc., Englewood Cliffs, N.J. 1970.
39. Barsamian, H., and A. DeCegama. "Evaluation of Hardware-Firmware-Software Tradeoffs with Mathematical Modeling," *AFIPS Conference Proceedings*, Vol. 38, 1971 Spring Joint Computer Conference, pp. 151–161.
40. Zaks, Rodney, David Steingart, and Jeffrey Moore. "A Firmware APL Time-Sharing System." *AFIPS Conference Proceedings*, Vol. 38, 1971 Spring Joint Computer Conference, pp. 179–190.
41. Lawson, H. W., Jr., and B. K. Smith. "Functional Characteristics of a Multi-lingual Processor." *IEEE Transactions on Computers*, Vol. C-20, No. 7 (July, 1971), p. 732.
42. "MLP-900 Multi-Lingual Processor, Principles of Operation." Standard Computer Corporation, Santa Ana, Calif. Form 809001-5. 1970.
43. Syrett, T. "The SNOBOL Machine, the First Virtual Machine Language for the SLAC MLP-900," Draft Report of SLAC Computation Group. Stanford, Calif. 1971.
44. Digital Scientific Corporation. *Digital Scientific META 4 (TM) Series 16 Computer System Reference Manual*. Publication No. 7032MO. Digital Scientific Corporation, San Diego, Calif. 1970.
45. Backus, J. "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference," *Proceedings of the International Conference on Information Processing*. UNESCO. (June, 1959), pp. 125–132.

Index

- A descriptor, 61, 67, 109, 113, 115, 133, 166
A flag, 57, 143, 146
A-B pair, 62, 67, 112, 115, 133, 166
ABORT, 32, 121
ADDF, 165
ADDFI, 166
ADDRVV, 164
ADDVV, 164
Aggregates, programmer-defined. *See also* Arrays; Data types, defined; Tables
Allocated data, 55, 59, 63, 67, 85, 96, 142–154, 167, 174, 184, 198–199, 225, 230, 238, 247–248
Allocation, 46, 53, 142–145, 247, 254
&ALPHABET, 23, 178
Alternate, 25, 118, 120, 121, 124, 127, 173
Alternation, 25–26, 117, 120, 127, 234
ANALYZ, 52, 55, 81, 85, 93, 94, 131, 238
&ANCHOR, 27, 116
Anchored mode, 27, 122
ANY, 33, 168, 234
APDQQ, 167
APDSIB, 176
APDSON, 176
ARB, 31, 36, 122, 123–124, 234
&ARB, 32
ARBNO, 36, 234
Argument count, 89, 103, 108, 179
Argument descriptor, 93, 118–119, 173
Argument evaluation, 69–71. *See also* Procedures, argument evaluation
Arguments, omitted. *See* Trailing arguments, omitted
ARGVAL, 55
Arithmetic, 9, 12–13, 164, 183, 201. *See also* Integers; Real numbers
ARRAY:
 data-type, built-in, 17, 235–236
 function, built-in, 20–21, 110–111, 168
Array dimension, 110–111
Array prototype, 110–111
Array reference, 72–74, 86, 90, 111–112, 139, 179, 235, 238
Arrays, 20, 44, 58, 61, 110, 234, 235
Assembly language, 5, 159, 160, 176, 178, 182, 237–238
Assignment, 9, 60, 72–73, 80, 93
Assignment operator. *See* Conditional value assignment; Immediate value assignment
Associativity, 13, 90–92
B descriptor, 61–62, 67, 88, 109, 113, 115, 133, 136, 166
Back referencing, 234–235
BAL, 31, 122, 174, 234
&BAL, 32
Basic blocks, 146–149, 151–153
Basing, 184, 187, 189, 190, 198, 200
Basing procedure, 78, 92
BEFAP, 4, 237

- BLOCK, 55
- Blocks, 61–63, 85–86, 105–110, 112–113, 119–120, 131–134, 136–137, 139, 142–153, 166, 173, 174, 183, 202, 237, 238
- BRANCH, 173
- BRANIN, 173, 179, 187, 208, 229
- BRANLV, 173, 187, 209, 211
- BREAK, 33, 169, 234
- BUFFER, 162
- Carriage control, 37
- CDC 6000, 182, 201–230, 240
- Chain descriptor, 59, 98, 143, 154, 192
- Chain of natural variables, 98–99, 143–145, 150, 154, 178, 184
- Chain offset, 98, 176, 191, 214
- Character set, 250
- CLEAR, 169
- CLRBLK, 166
- CLRF, 165
- CMPPOS, 174
- CMPSS, 165
- CMPVC, 165
- CNVRT, 55
- CODE:
 - data-type, built-in, 17, 37, 46, 54, 75, 84, 131–133, 235–236, 238, 253
 - function, built-in, 36–37, 131
- Code base pointer, 68, 69, 75–76, 78, 85, 104, 131–132, 136, 140, 141
- Code block, 132–133, 136
- Code descriptor, 65, 85–86, 88, 92, 93–94
- Code improvement, 246
- Code offset, 68, 69, 75–76, 78, 85, 94, 104, 140, 141, 179
- Code trees, 85–89, 91–94, 131, 176
- CODSKP, 140
- COL, 55
- COMBAL, 174
- Comment lines, 11, 52, 86
- COMMML, 174
- COMNVZ, 174
- Compiler, 51, 252–253
- COMQNV, 168, 180
- COMSQ, 168
- COMSZ, 174
- CONALT, 173
- Concatenation, 12, 25, 71, 117, 120, 127, 167
 - in patterns, 25, 234
- Conditional transfer, 10, 75–78
- Conditional value assignment, 124, 234
- Connector descriptor, 117–122, 173
- CONTIN, 82, 169, 194, 216, 221
- Continuation lines, 11, 52, 85–86
- Control lines, 11, 52, 86
- CONVERT, 17, 54, 166
- COPY, 176–177
- CPYBLK, 166
- CPYPAT, 173, 242
- Current stack pointer. *See* Stack pointer, current
- Cursor, 27, 243
- Cursor position, 31, 121–128, 174
- Cursor position operator, 31, 125, 243
- CVTIR, 162
- CVTIS, 166–167
- CVTSI, 166
- DATA, 18–19, 62, 66, 105–107, 168
- DATA block, 106
- Data representation, 56–63
- Data structures. *See* Blocks; Natural variables
- DATATYPE, 17, 19
- Data-type checking, 45, 70, 139, 253
- Data-type codes, 57, 61–63, 106–108, 113, 115, 125, 140, 177
- Data-type conversion, 38, 45, 70–71, 115, 119, 126, 139, 166, 181, 227, 236, 257
- Data-type pair block. *See* DTPBL
- Data types, 6, 17, 45, 234, 235, 250
 - built-in, 17, 57, 235–236
 - ARRAY, 17, 235–236
 - CODE, 17, 37, 46, 54, 75, 84, 131–133, 235–236, 238, 253
 - EXPRESSION, 17, 140, 235–236
 - INTEGER, 17, 71, 125, 139, 162, 166, 235–236
 - NAME, 17, 116, 235–236
 - PATTERN, 17, 235–236
 - REAL, 17, 71, 162, 166, 235–236
 - STRING, 17, 61, 71, 139, 162, 166, 235–236

- TABLE, 17, 236
defined, 18–19, 22, 44, 57, 105–
109, 234–235
external, 20, 57
DATE, 175
DBLOCK, 162
Declarations, 44–46, 235
DECVC, 164
DEFDAT, 106–108
Deferred pattern definition, 234–235,
238
DEFFNC, 102–105, 179
DEFINE, 14, 66, 100, 168
DEFINE block, 102, 104
Defined data objects. *See* Data types,
defined
Definition descriptor, 102–103, 107
DELQC, 168
DESCR, 162
Descriptors, 56–61, 162, 170–171,
174, 177, 179, 183–184, 200,
201–202, 205, 226–227, 237,
243, 254
Direct goto, 37, 132, 238
DTPBL, 61–63, 106, 146, 178
Dump messages, 178, 226
DUPL, 11, 229
DWDTH, 177, 183, 184, 186, 191, 201

ELEM, 55, 82, 85, 86, 88–89, 90, 93,
94, 238
Elements, 79–83, 85–88, 90
ELEMNT, 82, 88, 194–196, 217–219,
221
END, 176
End of file, 37, 76, 175
EQLDD, 164, 186, 206, 229
EQLTT, 165
EQLVC, 165
EQU, 176
EQULOC, 176
&ERRLIMIT, 41, 138
ERROR, 82–83, 94, 123, 168–169, 216
&ERRTYPE, 41, 138
Error handling, 40, 46, 73, 89, 94,
138, 140, 177, 179, 181, 187,
199, 200, 226, 235, 254
EVAL, 36, 131
EXPR, 55, 82, 85, 88–89, 90, 92, 93,
94, 131, 238
EXPRESSION, 17, 140, 235–236

Expressions, 79–83, 85, 90, 94
EXTERNAL, 20
External functions. *See* Functions,
external

F field, 56–61, 143, 146, 162, 165–
166, 177, 183, 192, 202, 204,
227, 242
F flag, 65, 179
FAIL, 32
Failure, 10, 12, 45, 72–73, 75–76, 173
Failure offset, 75–76, 93
Failure signal, 72–73, 94, 105, 111,
122, 126, 141, 179–181
Father, 86, 94
FENCE, 32
&FENCE, 32
FIELD, 107–109
FIELD block, 106–109
Field functions. *See* Functions, field
Field references. *See* Functions, field
File number. *See* Unit number
Flags, 145, 165–166, 177, 183, 202.
See also A, F, M, N, T, V flag;
F field
FNCPBL, 67, 69, 88–89, 92, 100–102,
108, 178
FOR, 82
Formal arguments, 14, 101, 103–105
FORMAT, 162
Formats, 38, 133–134, 162, 175, 178,
199, 226
FORTRAN, 38–39, 52, 133–134,
159, 162, 175, 177, 198–200,
225–227, 250
FORTXT, 52, 85, 168
FORWRD, 168
Free pointer, 143–145, 153
FRETURN, 14, 105, 179
&FTRACE, 40
&FULLSCAN, 36, 130
Function calls, 74, 75, 86, 88–90,
103, 126
Function descriptor, 65–66, 68–70,
117–119, 173
Function entry name, 14, 101, 104
Function name, 14, 101–105
Function pair block. *See* FNCPBL
Function procedures. *See* Procedures,
function

Function prototype, 14, 100–101
 Function tracing, 39, 40
Functions:
 built-in, 11–13, 53, 65, 89, 103,
 116, 122, 165, 166, 176–181,
 199, 226, 229
 ANY, 33, 168, 234
 ARBNO, 36, 234
 ARRAY, 20–21, 110–111, 168
 BREAK, 33, 169, 234
 CODE, 36–37, 131
 CONVERT, 17, 54, 166
 DATA, 18–19, 62, 66, 105–107,
 168
 DATATYPE, 17, 19
 DEFINE, 14, 66, 100, 168
 DUPL, 11, 229
 EVAL, 36, 131
 IDENT, 12
 INPUT, 38, 133
 ITEM, 90
 LEN, 32, 234
 LGT, 12, 165
 LOAD, 15, 66, 168, 199, 227
 LT, 12
 NOTANY, 33, 168, 234
 OUTPUT, 38, 133, 199
 OPSYN, 16, 66, 67
 POS, 33
 PROTOTYPE, 111
 REPLACE, 176
 RPOS, 33
 RTAB, 32, 234
 SIZE, 11
 SPAN, 33, 169, 234
 TAB, 32, 234
 TABLE, 22, 122
 TRACE, 39–40, 136, 254–255
 TRIM, 11, 180
 UNLOAD, 15, 66, 68, 175, 199,
 227
 defined, 14, 45–46, 66, 89, 100–
 105, 136, 178–179, 235
 returns
 FRETURN, 14, 105, 179
 NRETURN, 15, 105, 179
 RETURN, 14, 105, 179
 external, 15–16, 175, 177, 198–200,
 226, 227, 234, 236
 field, 18–19, 22, 105–109, 139
 object-creation, 18, 106–108
 pattern-valued, 32–33, 35, 117–120
 GETD, 163–164, 185, 205, 229
 GETDC, 163–164
 GOTO, 82, 216
 Gotos, 9, 37, 75–77, 80, 92–93, 132,
 238
 HASH, 176, 191–193, 214–216, 242
 Hash addressing, 97, 247, 254
 Hash functions, 97–100, 176, 191–
 193, 214–216, 247
 Heuristic descriptor, 118–119, 127,
 174
 Heuristics, 35–36, 118–119, 122,
 126–130, 173, 238
 IBM 360, 182–201, 228–230, 239–
 241, 246, 253
 IBM 7090/7094, 4–5, 237, 239–241
 IDENT, 12
 Identifiers, 23–24, 81, 83, 86, 88
 Immediate value assignment, 31, 35,
 124, 235
 Implicit alternatives, 123–124
 INCVC, 164, 186, 205
 Indirect referencing, 23–24, 139, 150,
 235, 247
 Infix form, 65
 INITEX, 174, 198–199, 225
 Initialization, 174, 198, 225
 INITST, 172
 INPBL, 133–134, 178
 INPUT:
 function, built-in, 38, 133
 I/O-associated variable, 37–38, 133
 &INPUT, 39, 45
 Input association, 38, 253
 Input block, 133, 175
 Input and output, 37, 133, 175, 177,
 181, 198–200, 225–226, 235,
 242, 250, 253. *See also I/O-*
 associated variables
 Input pair block. *See* INPBL
 Input text stream, 84–85, 94, 122
 INSNOD, 176
 INTEGER, 17, 71, 125, 139, 162, 166,
 235–236
 Integers, 12, 17, 86, 139, 162, 164,
 166, 183, 201, 227, 235–236,
 250
 Interior pointer, 154, 174
 INTERP, 68–70, 72, 75, 105, 132,
 178–179

- Interpretation, 64, 68, 75–76, 245
 Interpreter, 51–52, 54–55, 72, 177,
 237–238, 252–253
 Interrupt handling, 52, 198–200, 225
 INTGER, 194–196, 219–221
 INTERVAL, 71, 75
 INVOKE, 55, 69, 70, 71, 126, 131,
 140, 179–181
 I/O-associated variables, 37–38, 45,
 133–134, 255
 INPUT, 37–38, 133
 OUTPUT, 37–38
 PUNCH, 37–38, 133–134
 ITEM, 90
- Keyword pair blocks. *See* PKYPBL;
 UKYPBL
 Keywords, 22–24, 32, 115–116, 139,
 178, 235, 243
 protected, 22–23, 32, 115–116
 &ALPHABET, 23, 178
 &ARB, 32
 &BAL, 32
 &ERRTYPE, 41, 138
 &FENCE, 32
 &STFCOUNT, 22–23
 &STNO, 22–23, 254
 unprotected, 23–24, 115, 139
 &ANCHOR, 27
 &ERRLIMIT, 41, 138
 &FTRACE, 40
 &FULLSCAN, 36, 130
 &INPUT, 39, 45
 &OUTPUT, 39, 45
 &STLIMIT, 23, 138, 254
 &TRACE, 40
- L field, 58, 183, 202
 Label descriptor, 59, 75, 93, 94, 99,
 132, 143, 150
 Labels, 9, 75, 76, 78, 79, 80–81, 92–
 93, 94, 132
 Left recursion, 129
 Left son, 86, 93
 LEN, 32, 234
 Length failure, 128–129
 LGT, 12, 165
 LINK, 175
 Link descriptor, 66–70, 88–89, 92,
 102, 106–107, 118, 135, 173
- Link descriptor pair, 66–69, 89, 91,
 101, 108
 Literal procedure, 73–74, 86, 88
 Literals, 9, 81, 83, 86, 88
 LOAD:
 function, built-in, 15, 66, 168, 199,
 227
 SIL macro, 175
 LOCAD, 166–167, 186, 207–208
 Local variables, 14, 101, 103–105
 LOCBD, 166, 186, 207–208
 LOCTTL, 174
 LT, 12
- M* flag, 146, 153
 Machine independence, 5, 46–47, 51,
 159–160, 233, 244, 249–251,
 255
 Macros, 5, 160, 237–242, 245–247.
See also SIL macros
 MAKPAT, 173
 MARK, 55, 146, 148–150, 192
 Marking, 145–150
 MATCH, 122, 168
 Match abort, 121–122, 138
 Match failure, 121–123, 126, 128, 138
 Match success, 121–126
 Matching procedures. *See* Procedures,
 matching
 Minimum match length, 127, 129, 174
 MOVDD, 162, 185, 205, 229
 MOVQQ, 162
 MOVTC, 162
 MOVTI, 163
 MOVTI, 163
 MOVTT, 163
 MOVTI, 163
- N* flag, 59, 61, 174
 NAME, 17, 116, 235–236
 Name operator, 24–25, 74, 139–140
 Names, 24, 45, 72–73
 Natural variables, 20, 24, 59–61, 75,
 79, 88, 93, 96–100, 109, 132,
 134, 139, 142–145, 150, 154,
 168, 174, 178, 180–181, 191,
 237, 247, 251. *See also* Table
 of natural variables
 Negation operator, 12, 46, 141, 235
 Nodes, 85–86
 NOTANY, 33, 168, 234
 NRETURN, 15, 105, 179

- Null string, 10, 13, 61, 71, 89, 99, 103, 113, 123, 141, 143, 162, 184
 NULSTR, 162
- O field, 58, 183, 202
 Object, 9–10, 80, 93
 Object-creation function. *See* Functions, object-creation
 Old stack pointer. *See* Stack pointer, old
 Operand descriptor, 65. *See also* Argument descriptor
 Operating systems. *See* OS/360; SCOPE
 Operator block, 88
 Operators, 12–13, 45, 70–71
 binary, 12–30, 31, 65, 81, 83, 86, 88, 90–92, 124, 234
 unary, 12, 23–25, 31, 34, 46, 74, 81, 86–88, 125, 139–141, 150, 235, 243, 247
 OPERBL, 88, 91–92, 178
 OPSYN, 16, 66, 67
 Order number, 98–100, 143, 176, 191, 214
 ORDNVT, 176
 OS/360, 182, 198–201
 OUTPBL, 133–134, 178
 OUTPUT:
 function, built-in, 38, 133, 199
 I/O-associated variable, 37–38
 SIL macro, 175, 226
 &OUTPUT, 39, 45
 Output. *See* Input and output
 Output association, 38, 253
 Output block, 133–134, 175
 Output pair block. *See* OUTPBL
- Pair blocks, 61–63, 112–113, 115, 133–134, 135–137, 166, 178, 186, 207, 237, 243, 248, 251
 DTPBL, 61–63, 106, 146, 178
 FNCPBL, 67, 69, 88–89, 92, 100–102, 108, 178
 INPBL, 133–134, 178
 OUTPBL, 133–134, 178
 PKYPBL, 115–116, 146, 178, 254
 TTPBL, 135–136, 178
 UKYPBL, 115, 146, 178, 254
 VTRPBL, 135–136, 178
- Parenthesized expressions, 13, 81, 86, 88
 PATSTK, 121–122, 138, 178
 PATTERN, 17, 235–236
 Pattern base descriptor, 121
 Pattern components, 25–27, 30, 116–128, 173–174
 Pattern construction, 116–120, 125–128, 173, 177, 234–235, 238
 Pattern matching, 6, 10, 25, 34, 45, 80, 93, 116–117, 119, 121–130, 174, 177, 233–235, 237–238
 Pattern-matching procedure, 168
 Pattern offset descriptor, 121
 Pattern pointer, 27
 Pattern stack. *See* PATSTK
 Patterns, 6, 10, 25, 61, 71, 80, 82, 93, 116
 built-in, 31, 116, 122, 235
 ABORT, 32, 121
 ARB, 31, 36, 122, 123–124, 234
 BAL, 31, 122, 174, 234
 FAIL, 32
 FENCE, 32
 PKYPBL, 115–116, 146, 178, 254
 Pointer adjustment, 150, 152–154, 174
 Polymorphous operators, 12, 45, 70–71
 POPD, 170, 185, 187–188, 204, 209–210
 POPQ, 170, 185, 204
 Portability, 47, 51, 159, 233, 244, 248–249, 255
 POS, 33
 Precedence, 13, 25, 85, 90–92
 Precedence descriptor, 91–92
 Predicates, 12–13, 33, 72
 Prefix code, 51–52, 64–70, 73–78, 79, 85, 93, 103, 117, 119, 126, 131–133, 139–140, 173, 179–180, 238–239
 Prefix form, 65–66, 72, 75, 119
 PRINTQ, 175
 PROC, 171, 173, 184
 Procedure linkage, 66–68, 79, 117–118
 Procedures, 52–55, 171, 184, 238
 ANALYZ, 52, 55, 81, 85, 93, 94, 131, 238
 BLOCK, 55

- CNVRT, 55
- CODSKP, 140
- COL, 55
- ELEM, 55, 82, 85, 86, 88–89, 90, 93, 94, 238
- EXPR, 55, 82, 85, 88–89, 90, 92, 93, 94, 131, 238
- FORTXT, 52, 85, 168
- FORWRD, 168
- INTERP, 68–70, 72, 75, 105, 132, 178–179
- INVOKE, 55, 69, 70, 71, 126, 131, 140, 179–181
- MARK, 55, 146, 148–150, 192
- SCNR, 121–122, 126
- argument evaluation, 75
 - ARGVAL, 55
 - INTVAL, 71, 75
 - STRVAL, 180
- function, 65–70, 72–75, 136, 173, 179–180, 187, 238
 - DEFDAT, 106–108
 - DEFFNC, 102–105, 179
 - FIELD, 107–109
- matching, 117–126, 173, 177, 187
- Programmer-defined data types. *See*
 - Data types, defined
- Programmer-defined functions. *See*
 - Functions, defined
- Programmer-defined trace procedures.
 - See* Trace procedures, defined
- PROTOTYPE, 111
- Prototypes, 14, 15, 18, 20, 100–101, 106, 110–111, 168
- PUNCH, 37–38, 133–134
- PUSHD, 170, 185, 187–188, 204, 209–210
- PUSHQ, 170, 185, 204
- PUT, 82, 168, 194, 196, 216, 221
- PUTD, 163–164
- PUTDC, 164
- QUAL, 162
- Qualifiers, 58, 60–61, 162, 167–168, 170–171, 174, 175, 177, 180, 183–184, 196, 202, 237
- Quotation marks, 9–10, 25, 83, 86, 178
- QWDTH, 177, 183, 201
- RCALL, 171–172, 173, 184–185, 189–190, 204, 210–212, 242
- READQ, 175
- REAL, 17, 71, 162, 166, 235–236
- Real numbers, 12, 17, 71, 86, 162, 164, 166, 183, 202, 227, 235, 250
- Record length, 38, 133–134, 175
- Recursion, 14, 44, 55, 70, 72, 94, 129, 146, 170–172, 178–179, 235, 242
- Recursive patterns, 35, 234, 238
- Registers, 174, 184, 203–206, 226, 228, 239, 246, 251
- Relocation, 145, 150–153
- RELPTR, 174
- REPLACE, 176
- Replacement, 10, 80, 93
- Resident data, 55, 63, 67, 116, 145–149, 170, 176–178, 184, 198–200, 230, 243, 245, 247–248
- Residual, 127–129, 174
- RETURN, 14, 105, 179
- Return by name, 73, 105, 109, 115, 140, 141, 179, 238
- Return signal, 73, 177, 238
- Return by value, 73, 105, 108, 116, 140, 141, 179, 238
- Right sibling, 86, 93
- Root, 86, 88, 93
- RPLACE, 176
- RPOS, 33
- RRETURN, 171–172, 173, 184–185, 189–190, 204, 212–213, 242
- RTAB, 32, 234
- Rule, 9, 80–81, 92–93
- SCNR, 121–122, 126
- SCOPE, 201, 225
- Self pointer, 59, 151
- SETUPT, 169
- SIL, 160–181, 226, 237–242, 245–251
- SIL macros:
 - ADDFI, 166
 - ADDF, 165
 - ADDRVV, 164
 - ADDVV, 164
 - APDQQ, 167
 - APDSIB, 176
 - APDSON, 176
 - BRANCH, 173
 - BRANIN, 173, 179, 187, 208, 229
 - BRANLV, 173, 187, 209, 211

BUFFER, 162
 CLEART, 169
 CLRBLK, 166
 CLRF, 165
 CMPPOS, 174
 CMPSS, 165
 CMPVC, 165
 COMBAL, 174
 COMMML, 174
 COMNVZ, 174
 COMQNV, 168, 180
 COMSQ, 168
 COMSZ, 174
 CONALT, 173
 COPY, 176–177
 CPYBLK, 166
 CPYPAT, 173, 242
 CVTIR, 162
 CVTIS, 166–167
 CVTSI, 166
 DATE, 175
 DBLOCK, 162
 DECVC, 164
 DELQC, 168
 DESCR, 162
 END, 176
 EQLDD, 164, 186, 206, 229
 EQLTT, 165
 EQLVC, 165
 EQU, 176
 EQULOC, 176
 FORMAT, 162
 GETD, 163–164, 185, 205, 229
 GETDC, 163–164
 HASH, 176, 191–193, 214–216, 242
 INCVC, 164, 186, 205
 INITEX, 174, 198–199, 225
 INITST, 172
 INSNOD, 176
 LINK, 175
 LOAD, 175
 LOCAD, 166–167, 186, 207–208
 LOCBD, 166, 186, 207–208
 LOCTTL, 174
 MAKPAT, 173
 MOVDD, 162, 185, 205, 229
 MOVQQ, 162
 MOVTC, 162
 MOVT, 163
 MOVTV, 163
 MOVVT, 163
 ORDNVT, 176
 OUTPUT, 175, 226
 POPD, 170, 185, 187–188, 204,
 209–210
 POPQ, 170, 185, 204
 PRINTQ, 175
 PROC, 171, 173, 184
 PUSHD, 170, 185, 187–188, 204,
 209–210
 PUSHQ, 170, 185, 204
 PUTD, 163–164
 PUTDC, 164
 QUAL, 162
 RCALL, 171–172, 173, 184–185,
 189–190, 204, 210–212, 24:
 READQ, 175
 RELPTR, 174
 REPLACE, 176
 RRETURN, 171–172, 173, 184–185,
 189–190, 204, 212–213, 24:
 SETUPT, 169
 STKPTR, 172
 STREAM, 168–169, 194, 196–198,
 216, 221–225, 237
 STRING, 162
 TERMEX, 175, 226
 TESTF, 165
 TIME, 175
 TITLE, 176
 TRANDC, 164
 TRIMQQ, 167, 180
See also Appendix C
 SIZE, 11
 Size, 6, 47, 244–251
 SNOBOL, 3–7, 80
 SNOBOL2, 5
 SNOBOL3, 5–7, 46–47, 233–239
 SNOBOL4B, 250
 Source program, 51–52, 64, 66, 79,
 84–85
 SPAN, 33, 169, 234
 Speed, 6, 47, 244–251
 SPITBOL, 253–254
 SLIT, 83
 Stack. *See* PATSTK; SYSSTK
 Stack overflow, 187, 189
 Stack pointer:
 current, 170–172, 184–185, 187–
 190, 204, 210, 212
 old, 171–172, 189–190, 204, 210,
 212
 Stack position. *See* Stack pointer
 Statement, 9, 80, 92

- Statement initialization, 75–76, 92
Statements. *See* Assignment; Pattern matching; Replacement
&STFCOUNT, 22–23
STKPTR, 172
&STLIMIT, 23, 138, 254
&STNO, 22–23, 254
STOP, 82–83, 123, 168–169, 216
STOPSH, 82–83, 123, 168–169, 196, 216, 221
Storage management, 46, 52–53, 55, 59, 142–154, 167, 174, 177, 200, 237–238
Storage regeneration, 59, 142, 145–154, 174, 238, 242, 247
STREAM, 82–83, 85, 87–88, 90–94, 123, 168–169, 194, 196–198, 216, 221–225, 237
STRING:
 data-type, built-in, 17, 61, 71, 139, 162, 235–236
 SIL macro, 162
String base pointer, 58
String length, 58
String offset, 58
Strings, 23, 58–59, 71, 88, 96–100, 142, 162, 166–168, 174–176, 178, 183, 200, 202–203, 214, 226–227, 236, 247, 250–251
STRVAL, 180
STYPE, 82, 168–169, 196, 221
Subject, 9–10, 80–82, 93
Subscript. *See* Array reference; Table reference
Subsequent, 25, 118, 120, 121, 127
Substrings, 30, 58, 168, 254
Success, 10, 12, 173
Synonym. *See* OPSYN
Syntactic errors, 82, 94
Syntax, 79–94. *See also* Appendix A
Syntax table description:
 CONTIN, 82, 169, 194, 216, 221
 ERROR, 82–83, 94, 123, 168–169, 216
 FOR, 82
 GOTO, 82, 216
 PUT, 82, 168, 194, 196, 216, 221
 STOP, 82–83, 123, 168–169, 216
 STOPSH, 82–83, 123, 168–169, 196, 216, 221
Syntax tables, 82–85, 87–88, 91, 94, 122, 168–169, 192, 194–196, 216–221, 230, 248
ELEMNT, 82, 88, 194–196, 217–219, 221
INTGER, 194–196, 219–221
MATCH, 122, 168
SQLIT, 83
UNOPS, 87–88
 See also Appendix B
Syntax type codes, 88, 177
SYSSTK, 55, 72, 101–105, 106, 108, 111, 138, 146–147, 170–172, 177–178, 184, 188–190, 204, 210–212
System interface, 52, 174–175, 198–201, 225–227

T field, 56–59, 61, 65, 66, 86, 89, 91–92, 93, 98, 102, 106–107, 109, 115, 118, 120, 125, 140, 162–163, 165–166, 174–176, 183, 202, 204, 210
T flag, 59, 61, 146, 174
TAB, 32, 234
TABLE:
 data-type, built-in, 17, 236
 function, built-in, 22, 122
Table extents, 113
Table of natural variables, 97–100, 142, 176, 178, 191
Table reference, 86, 90, 113–114, 139, 247
Tables, 20, 22, 44, 61, 112–113, 235, 247
TERMEX, 175, 226
Termination, 138, 175, 177, 198–199, 225–226
TESTF, 165
TEXTQL, 85, 168
TIME, 175
TITLE, 176
Title adjustment, 150, 154
Title descriptor, 59–61, 139, 143, 145–146, 150–153, 173, 174, 185, 242
Token, 79, 82–83
Total length, 127
TRACE, 39–40, 136, 254–255
 &TRACE, 40
Trace association, 39–40, 135–136
Trace messages, 135–136
Trace procedures, 135–136, 138
 defined, 39–40, 135–136

- Trace tag, 39, 136
 Trace type, 39, 135
 Trace-type pair block. *See* TTPBL
 Tracing, 39–40, 45, 135–137, 177, 178–179, 253–255
 Trailing arguments, omitted, 13, 89, 103, 108
 TRANDC, 164
 Translation, 40, 79, 85, 92, 131–133
 Translation during execution, 36, 131–133. *See also* CODE
 Translator, 51–52, 54–55, 66, 79, 84, 168, 177, 192, 237–238, 242
 Trees. *See* Code trees
 TRIM, 11, 180
 TRIMQQ, 167, 180
 TTPBL, 135–136, 178
- UKYPBL, 115, 146, 178, 254
 Unanchored mode, 27, 122
 Unary name operator, 139, 235
 Undefined functions, 89
 Unevaluated expression operator, 34, 125–127, 129–130, 140
- Unevaluated expressions, 33–35, 45, 125–127, 129–130, 140, 235, 238
 Unit number 38, 133–134, 175, 177
 UNLOAD, 15, 66, 68, 175, 199, 227
 UNOPS, 87–88
- V field, 56–59, 61, 65, 82, 92, 93, 98, 107, 109, 113, 120, 125, 140, 150–151, 153, 162–166, 168, 172, 173, 175, 176, 183, 184, 185, 192, 196, 201–202, 204, 205, 210, 221
V flag, 179
 Value descriptor, 59–60, 99, 150
 Value trace pair block. *See* VTRPBL
 Value tracing, 39, 135–136
 Values, 45, 72–73, 139
 Variables, 20–25, 72–73, 96–97, 109
 Variables, local. *See* Local variables
 Variables, natural. *See* Natural variables
 Variables, subscripted. *See* Array reference; Table reference
 VTRPBL, 135–136, 178