INFORMATION TO USERS

This material was produced from a microfilm copy of the original document. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the original submitted.

The following explanation of techniques is provided to help you understand markings or patterns which may appear on this reproduction.

1. The sign or “target” for pages apparently lacking from the document photographed is “Missing Page(s)”. If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting thru an image and duplicating adjacent pages to insure you complete continuity.

2. When an image on the film is obliterated with a large round black mark, it is an indication that the photographer suspected that the copy may have moved during exposure and thus cause a blurred image. You will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., was part of the material being photographed the photographer followed a definite method in “sectioning” the material. It is customary to begin photoing at the upper left hand corner of a large sheet and to continue photoing from left to right in equal sections with a small overlap. If necessary, sectioning is continued again — beginning below the first row and continuing on until complete.

4. The majority of users indicate that the textual content is of greatest value, however, a somewhat higher quality reproduction could be made from “photographs” if essential to the understanding of the dissertation. Silver prints of “photographs” may be ordered at additional charge by writing the Order Department, giving the catalog number, title, author and specific pages you wish reproduced.

5. PLEASE NOTE: Some pages may have indistinct print. Filmed as received.

Xerox University Microfilms
300 North Zeeb Road
Ann Arbor, Michigan 48106
HALLYBURTON, John Camp, Jr., 1947-
ADVANCED DATA STRUCTURE MANIPULATION
FACILITIES FOR THE SNOBOL4 PROGRAMMING
LANGUAGE.

The University of Arizona, Ph.D., 1974
Computer Science

University Microfilms, A XEROX Company, Ann Arbor, Michigan
ADVANCED DATA STRUCTURE MANIPULATION FACILITIES
FOR THE SNOBOL4 PROGRAMMING LANGUAGE

by

John Camp Hallyburton, Jr.

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

1974
I hereby recommend that this dissertation prepared under my direction by John Camp Hallyburton, Jr. entitled Advanced Data Structure Manipulation Facilities for the SNOBOL4 Programming Language be accepted as fulfilling the dissertation requirement of the degree of Doctor of Philosophy.

Dissertation Director Date

After inspection of the final copy of the dissertation, the following members of the Final Examination Committee concur in its approval and recommend its acceptance:

This approval and acceptance is contingent on the candidate's adequate performance and defense of this dissertation at the final oral examination. The inclusion of this sheet bound into the library copy of the dissertation is evidence of satisfactory performance at the final examination.
STATEMENT BY AUTHOR

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

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

SIGNED: John [signature]
ACKNOWLEDGMENTS

The author wishes to express his deep gratitude to Frederick C. Druseikis who originally suggested the problem and offered many valuable criticisms during the writing of this paper. Additionally, I would like to thank John Doyle, John Lucas, and Bill Sears for contributing various pieces of advice during the course of this work.

A special debt of gratitude is owed to my committee, Dr. John D. Brillhart, Dr. David L. Clark, Dr. Ralph E. Griswold, Dr. David O. Lomen, and Dr. Murray Sargent III. They have provided enthusiasm and support, as well as technical and administrative advice that has saved me many headaches.

This work was supported by the National Science Foundation under grant GJ-36272. The programming and the preparation of this document were done at The University of Arizona Computer Center. The cooperation and assistance of the staff and management of the Center has been unfailing. This has resulted in a substantial savings of time and effort for which I am deeply indebted. I would also like to thank the individuals in the University administration for their support of machine-prepared theses and dissertations.
Most important, special thanks go to my advisor, Ralph E. Griswold, who has contributed vast amounts of his time and energy to this project. Without his constant encouragement the end result would have been significantly lower in quality. His comments and suggestions during the writing of this dissertation have also been invaluable.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF ILLUSTRATIONS</th>
<th>viii</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>ix</td>
</tr>
<tr>
<td>1. BACKGROUND</td>
<td>1</td>
</tr>
<tr>
<td>Purpose of this Study</td>
<td>1</td>
</tr>
<tr>
<td>Motivation</td>
<td>1</td>
</tr>
<tr>
<td>The Approach</td>
<td>2</td>
</tr>
<tr>
<td>Context for this Work</td>
<td>2</td>
</tr>
<tr>
<td>Survey of Suitable Language Contexts</td>
<td>3</td>
</tr>
<tr>
<td>COBOL</td>
<td>4</td>
</tr>
<tr>
<td>PL/I and ALGOL 68</td>
<td>4</td>
</tr>
<tr>
<td>SNOBOL3-SNOBOL4</td>
<td>5</td>
</tr>
<tr>
<td>Suitability of SNOBOL4</td>
<td>5</td>
</tr>
<tr>
<td>The Notion of a Data Structure</td>
<td>6</td>
</tr>
<tr>
<td>Aggregates</td>
<td>7</td>
</tr>
<tr>
<td>Uses of Data Structures</td>
<td>8</td>
</tr>
<tr>
<td>The State of the Art</td>
<td>9</td>
</tr>
<tr>
<td>Pattern Matching on Data Structures</td>
<td>10</td>
</tr>
<tr>
<td>Set-oriented Languages</td>
<td>11</td>
</tr>
<tr>
<td>Organization of this Paper</td>
<td>11</td>
</tr>
<tr>
<td>2. DATA STRUCTURES IN THE MACRO IMPLEMENTATION</td>
<td>13</td>
</tr>
<tr>
<td>Cells</td>
<td>13</td>
</tr>
<tr>
<td>Aggregates</td>
<td>14</td>
</tr>
<tr>
<td>Arrays</td>
<td>14</td>
</tr>
<tr>
<td>Programmer-defined Objects</td>
<td>15</td>
</tr>
<tr>
<td>Tables</td>
<td>17</td>
</tr>
<tr>
<td>Structure-valued Objects</td>
<td>19</td>
</tr>
<tr>
<td>Identifying Datatypes</td>
<td>20</td>
</tr>
<tr>
<td>A BNF Grammar</td>
<td>23</td>
</tr>
<tr>
<td>3. NEW DATA STRUCTURE FACILITIES FOR SNOBOL4</td>
<td>26</td>
</tr>
<tr>
<td>Extensions to Existing Facilities</td>
<td>26</td>
</tr>
<tr>
<td>Lists</td>
<td>26</td>
</tr>
<tr>
<td>Field Referencing</td>
<td>29</td>
</tr>
<tr>
<td>Associative Deques</td>
<td>30</td>
</tr>
<tr>
<td>Shortcomings in SNOBOL4 List Processing</td>
<td>35</td>
</tr>
<tr>
<td>Possibilities for Improvement</td>
<td>37</td>
</tr>
<tr>
<td>Diagrams and Notation</td>
<td>38</td>
</tr>
</tbody>
</table>
TABLE OF CONTENTS—Continued

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Operations</td>
<td>41</td>
</tr>
<tr>
<td>Sequeniation</td>
<td>41</td>
</tr>
<tr>
<td>Alternation</td>
<td>42</td>
</tr>
<tr>
<td>Forward Alternation</td>
<td>42</td>
</tr>
<tr>
<td>Special Directives</td>
<td>43</td>
</tr>
<tr>
<td>Null String</td>
<td>43</td>
</tr>
<tr>
<td>Lists</td>
<td>44</td>
</tr>
<tr>
<td>Unevaluated Expressions</td>
<td>47</td>
</tr>
<tr>
<td>Cursor Examination</td>
<td>47</td>
</tr>
<tr>
<td>The Driver Algorithm</td>
<td>49</td>
</tr>
<tr>
<td>Directive Interpretation of SNOBOL4 Patterns</td>
<td>54</td>
</tr>
<tr>
<td>Control Directives</td>
<td>55</td>
</tr>
<tr>
<td>RPT</td>
<td>55</td>
</tr>
<tr>
<td>The AND Operator</td>
<td>57</td>
</tr>
<tr>
<td>The FOR Operator</td>
<td>75</td>
</tr>
<tr>
<td>Conditionals</td>
<td>77</td>
</tr>
<tr>
<td>The Function IF</td>
<td>77</td>
</tr>
<tr>
<td>Null String Tests</td>
<td>78</td>
</tr>
<tr>
<td>Conditional Indexing</td>
<td>79</td>
</tr>
<tr>
<td>NOT</td>
<td>81</td>
</tr>
<tr>
<td>Structures Containing Loops</td>
<td>81</td>
</tr>
<tr>
<td>MARK</td>
<td>81</td>
</tr>
<tr>
<td>Unknown Structures</td>
<td>82</td>
</tr>
<tr>
<td>POINTER and SIMPLE</td>
<td>82</td>
</tr>
<tr>
<td>ENTER, NEXT, and EXIT</td>
<td>83</td>
</tr>
<tr>
<td>Other Directives</td>
<td>85</td>
</tr>
<tr>
<td>Replacement</td>
<td>85</td>
</tr>
<tr>
<td>SET</td>
<td>87</td>
</tr>
<tr>
<td>Cursor Positioning</td>
<td>88</td>
</tr>
<tr>
<td>Tagging the Cursor Position</td>
<td>89</td>
</tr>
<tr>
<td>4. EXAMPLES</td>
<td>91</td>
</tr>
<tr>
<td>AGDUMP</td>
<td>91</td>
</tr>
<tr>
<td>Sample Aggregate Dump</td>
<td>92</td>
</tr>
<tr>
<td>Constructing a Binary Tree</td>
<td>94</td>
</tr>
<tr>
<td>Tree Analysis</td>
<td>98</td>
</tr>
<tr>
<td>Left-listing a Binary Tree</td>
<td>100</td>
</tr>
<tr>
<td>The Directive COUNT</td>
<td>102</td>
</tr>
<tr>
<td>Traversing an Arbitrary Structure</td>
<td>103</td>
</tr>
<tr>
<td>Duplicating a Structure</td>
<td>106</td>
</tr>
<tr>
<td>Correctly Duplicating a Structure</td>
<td>110</td>
</tr>
<tr>
<td>Analyzing a BNF Grammar</td>
<td>112</td>
</tr>
<tr>
<td>Dumping Unprotected Keywords</td>
<td>117</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS—Continued

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5. EVALUATION 120</td>
</tr>
<tr>
<td>The Data-driven Approach 120</td>
</tr>
<tr>
<td>Control Directives 121</td>
</tr>
<tr>
<td>Other Benefits 122</td>
</tr>
<tr>
<td>Lessons from the Implementation 123</td>
</tr>
<tr>
<td>Preponderance of Operators 123</td>
</tr>
<tr>
<td>Use of Alternation 123</td>
</tr>
<tr>
<td>Problems with Directives 124</td>
</tr>
<tr>
<td>Primitives 124</td>
</tr>
<tr>
<td>Evaluation Time 125</td>
</tr>
<tr>
<td>Compatibility of Language Features 127</td>
</tr>
<tr>
<td>Possibilities for Further Research 130</td>
</tr>
<tr>
<td>6. THE IMPLEMENTATION 133</td>
</tr>
<tr>
<td>High-level Simulation 133</td>
</tr>
<tr>
<td>Modifications to SNOBOL4 134</td>
</tr>
<tr>
<td>Functions Required by NEXT 134</td>
</tr>
<tr>
<td>Basic Elements in the Simulation 135</td>
</tr>
<tr>
<td>Representation of Directives 135</td>
</tr>
<tr>
<td>The Main Algorithm 135</td>
</tr>
<tr>
<td>Recursive Calls to the Driver 137</td>
</tr>
<tr>
<td>Possibilities for a SIL Implementation 138</td>
</tr>
<tr>
<td>A Brief Look at SIL 139</td>
</tr>
<tr>
<td>Suitability of SIL 139</td>
</tr>
<tr>
<td>Notes on the COC 6400 Implementation 140</td>
</tr>
<tr>
<td>APPENDIX A: AGDUMP 142</td>
</tr>
<tr>
<td>APPENDIX B: DRIVER 149</td>
</tr>
<tr>
<td>REFERENCES 163</td>
</tr>
</tbody>
</table>
# LIST OF ILLUSTRATIONS

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>A cell</td>
<td>13</td>
</tr>
<tr>
<td>2.</td>
<td>A SNOBOL4 array</td>
<td>15</td>
</tr>
<tr>
<td>3.</td>
<td>A TIME(4362,21,11,4.96)</td>
<td>16</td>
</tr>
<tr>
<td>4.</td>
<td>A table</td>
<td>18</td>
</tr>
<tr>
<td>5.</td>
<td>A HAND of CARDS</td>
<td>20</td>
</tr>
<tr>
<td>6.</td>
<td>A SNOBOL4 representation of a BNF grammar</td>
<td>25</td>
</tr>
<tr>
<td>7.</td>
<td>Pertinent items in the traversal process</td>
<td>40</td>
</tr>
</tbody>
</table>
Contemporary programming languages vary greatly in their support of data structure processing. The first high-level programming language, FORTRAN, was designed with, and still possesses, only one data structure: the array. The designers of COBOL found one particular data structure, the tree, to be highly suited to their needs and a restricted class of tree structures is a basic element of the language.

The LISP programming language was the first major language to provide genuine list processing facilities. Although LISP suffers from a syntax that makes programming difficult, its concepts of list processing have influenced the design of data structure facilities in the major languages. SNOBOL4, PL/I, and ALGOL 68 offer facilities only slightly more high-level than LISP: The programmer can define data objects to consist of some fixed sequence of named fields, create data objects, and reference fields of them.

Such a set of facilities may be adequate to solve many problems, but their use can be cumbersome. A complicated data structure needs more than just these facilities to process it. Referencing a field of a data object is only a minor help if a data structure possessing complex sub-fields nesting down several levels is to be analyzed.
is a deficiency in language design that becomes evident when one attempts to process such a complex structure.

The SNOBOL4 programming language is chosen as a vehicle for experimentation, in which a new and different approach to data structure processing is designed and implemented. The new facilities include new data structures and access functions that provide a uniform way of visualizing SNOBOL4 data structures. From this point of view, a new set of data structure processing facilities are developed and implemented in a slightly modified version of SNOBOL4.

The pattern matching concept is extended to the notion of data structure traversal. The programmer defines a data object, called a directive, that in usage resembles a pattern. The process of traversing a data structure consists of applying a directive to a subject structure. Built-in operations exist for indexing into structures, saving the cursor position, testing various conditions, iterating, making replacements, and handling structures with loops. In general, operations that tend to arise frequently are provided in the form of built-in directives.

The result of the design and implementation, as demonstrated by examples, is that data structure operations are easy to write and read. The level of formulation is much higher since many of the clerical details are suppressed or handled automatically. Additionally, many
operations that are not otherwise possible in SNOBOL4 can be implemented using the new facilities. This generality, combined with the convenience of high-level facilities, greatly enhances the capabilities of the SNOBOL4 language.

The implementation is described along with some notes on the possibility of including the entire system as part of SNOBOL4.
CHAPTER 1

BACKGROUND

Purpose of this Study

Major contemporary programming languages possess a number of deficiencies in data structure handling facilities. The ability to define and reference fields of data objects hardly makes a complete repertoire for data structure manipulation, yet these are the only facilities provided. (Some languages, notably FORTRAN, do not even provide this minimal set of facilities.) This implies that each individual must write his own structure-processing procedures, often a simple task but tedious and time-consuming nonetheless. Thus, the data structure processing facilities of major programming languages deserve the term "low-level".

Motivation

The implications of low-level structure processing are obvious. The programmer has more work to do, increasing the time required to solve a given problem. The number of coding errors increases. Optimization becomes more difficult, since many statements are required to implement a single concept.
Another important consideration concerns psychology. When a programmer is forced to perform essentially clerical work, he may become demotivated, thus impairing his capabilities. All of this implies that problem-solving with low-level structure processing facilities is less efficient than it could be.

The Approach

The purpose of this study, then, is to design and implement a set of high-level data structure facilities. The design is based upon careful examination of what people really do, or would like to be able to do, when processing data structures. By studying the basic data structures and algorithms for processing them, a set of "building blocks" -- common processing algorithms -- can be determined. By providing reasonably simple means to access and combine these processing algorithms, a more high-level, human-oriented set of facilities can be constructed.

Context for this Work

When new language features are designed in computer science, a common technique for testing and evaluating them involves creating a new programming language, specifically designed to highlight the features. This approach allows some flexibility in the syntax of such features but also has some serious drawbacks. New concepts by themselves are rarely enough to comprise a programming language so other,
required facilities must be added. Such new programming languages are often machine-dependent and typically not well-used except by the designers. This may be the result of implementation bugs or inadequacies that force the designers to delay release of the new language. Additionally however, people tend to avoid learning a new language if it only is designed for experimentation.

For these reasons, this study does not include the construction of a new language. Rather, the new facilities designed herein are imbedded in the SN0B0L4 language. Thus, all the standard facilities of a high-level programming language are available in addition to the new features. Individuals desiring to use the new features need only learn them and not other irrelevant material.

Survey of Suitable Language Contexts

Although FORTRAN and ALGOL provide arrays, genuine list-processing facilities (above the assembly-language level) were not introduced until LISP (McCarthy et al., 1962).

LISP was the first major language to permit the use of pointers and thus provide list-processing capabilities. LISP and its dialects have been heavily used in the field of artificial intelligence. Unfortunately, LISP is quite primitive, if conceptually elegant. Part of the motivation
of this work is providing a convenient set of structure-processing facilities. LISP, however, is not a very easy language to program in. For this same reason extensions to LISP, such as GRASPE (Friedman et al., 1969) are also unsuitable.

COBOL

COBOL has the capability to describe data records in a tree-like structure. Any collection of data items within a record, by appropriate declarations, can be given a name and be referenced by that name. Various facilities, more or less specific to business data processing, are provided for moving trees, possibly editing some of the items in the process. However, the commercial orientation of COBOL precludes more general data structure facilities, since they are not normally required. COBOL does not have mechanisms for manipulating pointers, so general list processing cannot be performed directly.

PL/I and ALGOL 68

The PL/I language contains explicit list-processing facilities (Lawson, 1967). Its capabilities are representative of contemporary list-processing facilities in high-level languages. The programmer can declare an object to contain several fields, some of which contain values and others that point to other objects. Individual fields of a given object may be referenced and assigned values. Roughly
the same facilities exist in ALGOL 68 (Lindsey, 1968). Neither PL/I nor ALGOL 68 are suitable for this study because of unavailability of implementations.

SNOBOL3-SNOBOL4

One of the best arguments in support of programmer-defined data objects and list-processing in general is their wide use where available. The authors of SNOBOL3 at one time used external functions to simulate some basic structures: trees (Griswold and Polonsky, 1965) and linked lists (Griswold, 1965). In both cases the new facilities were well-received and some difficult operations were greatly simplified as a result. This encouraged the authors to include programmer-defined data objects in their development of SNOBOL4.

The facilities in SNOBOL4 parallel somewhat the features of PL/I and ALGOL 68: one can define data objects and reference their fields. However, SNOBOL4 possesses a more dynamic version of the facilities in that data structures are not fixed at compile time. For various reasons, given below, the facilities developed herein are incorporated in the SNOBOL4 programming language.

Suitability of SNOBOL4

SNOBOL4 is a high-level language containing all the standard list-processing facilities. The macro implementation is exceptionally well-suited for programming language
research and experimentation. This is in part due to the fact that one of the design goals of the authors was to provide a language that could be used for these purposes. The source code for SNOBOL4 is easily obtained (Griswold, 1971a) and is heavily commented, so modifications to it can be easily made. Since the macro implementation is highly machine-independent and portable, and exists on most large computers, new facilities can be widely distributed with little effort.

Additionally, SNOBOL4 contains all the data structure facilities common to contemporary high-level languages. It provides, therefore, a reasonable base to which extensions can be made.

The Notion of a Data Structure

The field of computer science contains from a number of terms that do not have precise definitions. Among these are "machine independent", "very high level", and "data structure". It seems evident that a formal definition of such terms would be difficult to formulate and any attempt to do so would result in a definition too intricate to be understandable. Since, in this work, communication of ideas is more important than precise definitions, most of the terms used are not formally defined. Instead, terms are introduced contextually to give the reader a reasonable idea
of their meaning. Confusion and ambiguity are minimized by the fact that SNOBOL4 serves as a framework for discussion, thereby limiting borderline cases (e.g., is a FORTRAN complex number a data structure?) that would otherwise appear.

Aggregates

In SNOBOL4 there are several ways that the programmer can group together values and reference the entire collection by a single variable. One way is to use an array, where a series of integer subscripts is used to locate a desired element. Another possibility is to use a table, which may be indexed by any SNOBOL4 object. Alternatively, the programmer can define his own datatypes that contain a fixed number of objects, and reference individual items by function names of his own choosing. These are all examples of aggregates: objects that represent collections of values. The terms "element", "item", "field", and "member" are used interchangeably to denote any one of such a collection.

Suppose now that A is the name of a one-dimensional array whose subscripts range from 1 to 10. Then A[3] is a particular element of the array. The SNOBOL4 language places no constraints on the value of A[3] simply because it is an array element. That is, A[3] can itself have any value and could, for example, point to another array. In fact, A[3] can have the array A as value. In general, any
member of an aggregate may have any object as its value, including another aggregate. The term data structure refers collectively to all the fields of an aggregate and, for every such field that itself points to an aggregate, all of its fields, and so on. Thus, a data structure embodies not just a collection of elements but also all the interrelationships between individual elements.

Uses of Data Structures

Many applications of data structures arise in the field of graph theory or areas related to it. Various networking problems, such as PERT/CPM, neural networks, and system simulation are greatly simplified when data structures natural to the problem can be used. The field of artificial intelligence depends heavily upon data structuring in representing search paths, scene recognition, and modeling of human behavior. A general rule is: whenever the data (in the most general sense of the word) to be analyzed involves structural relationships between various elements, appropriate data structuring provides maximum facility in dealing with the basic problems involved. Unnatural encodings of various interrelationships only complicate matters by forcing one to take extra steps along the way.

Sometimes, particularly in information management areas, performance can be drastically improved by introducing additional structure not necessarily inherent in
the Information. The Library of Congress cataloging system is a concrete example: by dividing books into various levels of categories, the amount of time and effort required to find a given book is reduced significantly. Data base management systems often employ this same type of search technique. There are many variations on this idea that are used in computer applications to cut down search times or simplify processing methods.

In many instances, certain specific types of structures (strings, trees, linked lists) are used in lieu of more appropriate but more complicated data structures. This may result if the language used supports only certain types of structures (as in LISP and SNOBOL3) or if the programmer feels more comfortable with a familiar data structure. Generally, then, the need for encoding techniques to simulate data structures indicates a lack of completeness in the language.

The State of the Art

The importance of this area of research is indicated by the large amount of current work. Major programming languages lack high-level facilities mainly because of their static nature, but a number of minor programming languages have been developed which feature new and interesting ways of processing data structures. What follows is not intended to be a complete review of current research, but rather an
indication of the kinds of facilities being considered, and how they differ from those described in this work.

Pattern Matching on Data Structures

Various systems have been proposed for performing pattern matching on data structures. These all involve searching a data structure for a specific item or class of items. Wolfberg (1972) is the only one discussed herein who claims to have a running implementation. The system he describes is called AMBIT/L (Christensen, 1971), and it involves two-dimensional diagrammatic input that represents the kinds of patterns to be matched and the transformations that are to occur. While the system is elegant and interesting, its two-dimensional nature involves time-consuming program preparation — something high-level languages should minimize.

Earley (1974) describes a system that is very large, powerful, and complex. Perhaps this explains why it has not been implemented. His goal is to design higher and higher level languages, to the point where at most minor changes are required to make mathematical notation into a programming language. This is an important area of research but of no use to the individual who wants to write a program today.

Standish (1967) uses pattern matching to construct access paths to a given data class. Access paths can be concatenated. Unfortunately, this is all that is mentioned
in the literature and an evaluation of the system is difficult.

Set-oriented Languages

The L* data language (Gray and Tomlinson, 1973) is designed to perform various operations on sets and sequences (ordered sets). The standard operations (union, merging, intersection, reduction, projection) and others are defined in the language, which has been implemented. The language appears to be useful in the area for which it has been designed, which is data base manipulation. It is not well-suited for general data structure operation as it does not (evidently) allow pointer manipulation. Furthermore, it does not handle structures with loops, so graph-theoretic problems cannot be conveniently solved.

In SETL (Schwartz, 1974) the basic data object is also the set. Again the concept of pointers is absent from the language so encoding techniques are used to process graphs. As with L*, the language appears to be useful in the area for which it is intended but does not possess the generality required to handle arbitrary structures conveniently.

Organization of this Paper

Since this work depends on the macro implementation of SNOBOL4, the appropriate framework must be provided.
Chapter 2 presents some descriptions and examples of data structures in the macro implementation.

The beginning of Chapter 3 discusses a few data structure additions to SNOBOL4 and the inadequacies of SNOBOL4 data structure processing. The remainder of the chapter describes a new approach to data structure manipulation and the various functions and operations associated with it. Included are some diagrams describing the workings of the system.

The fourth chapter presents ten examples of the new facilities, and explains how they work.

Chapter 5 is an evaluation of the facilities. It discusses why the facilities are so useful, what was learned from the implementation, possible improvements, and what problems exist with the facilities. The incompatibilities with standard SNOBOL4 are described and one potential solution is suggested. Finally, some additional areas for further research are proposed.

The last chapter describes the implementation, which is a simulation in SNOBOL4. It also describes the modifications to SNOBOL4 that were required, and discusses the possibility of implementing the entire system as part of SNOBOL4.
CHAPTER 2

DATA STRUCTURES IN THE MACRO IMPLEMENTATION

Cells

In the macro implementation of SNOBOL4, all objects are composed of one or more cells. A cell is a machine-dependent entity; an integral number of machine address units typically totalling 60-72 bits (Figure 1).

Figure 1. A cell

The V field of a cell contains either the value or a pointer. If the value is numeric or the null string, there is enough room in the V field to hold the value. Otherwise the V field is a pointer to a block of cells representing the value. In particular,

\[
\begin{array}{c|c}
T & V \\
\hline
I & 2 \\
\end{array}
\]

represents the integer 2 (the "I" standing for the code designating an integer datatype). Also,
represents the null string, and

represents the string HAT. Strings are represented by structures that contain more information than just the characters; braces are used here to indicate the entire structure.

Aggregates (arrays, tables, and programmer-defined datatypes) are represented by blocks of cells, in the format shown below. Usually, only one cell within an aggregate is of interest at any given moment. Each type of aggregate has a procedure associated with it to locate a given, desired element. These procedures are known as access functions and are an intrinsic part of the datatype. Each datatype has its own unique internal (to SNOBOL4) access function, so there is only one way to locate a particular cell.

Aggregates

Arrays

Execution of ARRAY(P) produces a block of cells that represents the array. Figure 2 below is a diagram of an array.
Certain information, such as dimensionality and subscript limits, is contained within the array as shown above. The array elements are stored in the cells following the descriptive information.

Accessing a particular array element requires computation and range testing. This is performed by the SNOBOL4 Internal procedure ITEM. Using the descriptive information, ITEM either fails (if an index is out of range), signals an error (if too many subscripts were supplied or the object is not an array), or locates the desired element and returns a pointer to it.

Programmer-defined Objects

New types of data objects may be added to the SNOBOL4 system by giving the prototype of the object as the argument to the DATA function:

\[ \text{DATA('TIME(JDATE,HOUR,MIN,SEC)')} \]
defines a new datatype TIME consisting of four fields. Each field requires one cell within the object, resulting in the following diagram (Figure 3) of a TIME.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>4362</td>
<td>JDATE field</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>21</td>
<td>HOUR field</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>11</td>
<td>MIN field</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>4.96</td>
<td>SEC field</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3. A TIME(4362, 21, 11, 4.96)

As demonstrated above, field positions correspond to the order of appearance in the prototype.

Accessing fields within a programmer-defined object is accomplished internally by use of certain blocks of cells associated with field function names. Since field functions may be defined for many different data objects, each field function has associated with it a block of cells that contains datatype codes and corresponding positions. Thus when

MIN(SOLSTICE)

is executed (where SOLSTICE is a TIME), an internal procedure FIELD examines the block for MIN and finds that for TIMES, the MIN field is the third field of the object. FIELD then constructs a pointer to the correct cell and returns it.
Note that this access method is similar in nature to the access method for a one-dimensional array. Instead of providing the relative position in the structure, the user supplies a mnemonic from which FIELD obtains a position. Thus

\[ \text{MIN(SOLSTICE)} \]

is not really a call to a MIN function, but rather a call to the FIELD procedure. A more precise form of referencing the MIN field of SOLSTICE is

\[ \text{SOLSTICE\<MIN\>} \]

since it reflects more precisely the operations taking place. Such a form also resembles a table reference both externally and internally.

There is a notable difference between a defined data object and an array. Prototype information is stored within each array but not within a programmer-defined data object. The reason for this is that a prototype of an array is an attribute of the array, but the prototype of a data object is an attribute of the datatype and is therefore shared by all objects having that datatype. Consequently, a savings in memory is achieved by placing the prototype information in a separate area.

Tables

Tables are basically just blocks of cells with their own datatype and access function. Figure 4 below displays a table, omitting for simplicity any extents.
Since tables can have any object as an index, it is necessary to retain, in some known location, the indices as well as the corresponding values. Thus, the macro implementation keeps table entries in pairs, the first member of which is the value and the second the index. The access function for a table involves a linear search to find the index; from this the corresponding value can be located. If the index is not found, a new table entry is created with the null string as value. As a consequence of this, table entries occur in chronological order.

Some special versions of the macro implementation (Griswold, 1972a) allow one to "close" tables to further entries: a reference to a closed table fails if the entry does not already exist.
Unfortunately there is only one access function for tables, although other kinds of accessing are conceivable. In particular, it seems reasonable to want to find the index or indices corresponding to a given value. It might also be desirable to want to change a table index without altering the corresponding value, or to remove the entry altogether. It might at times be useful to be able to find the index of an array having a particular value, or the field of a programmer-defined datatype having a specific value. The variety of possible alternate access methods suggests the need for user-defined access functions.

**Structure-valued Objects**

A cell is large enough to hold any SNOBOL4 value. This means that, for example, array elements may themselves represent arrays, tables, or anything else, for that matter. This is achieved by using a pointer when the object itself is too large to fit into a cell. The pointer points to a block of cells representing the value. Consequently, an array element may itself be an array, table, string, or any object since a cell can always contain a pointer. One example of how this is done follows.

Assume the following definitions have been executed:

```plaintext
DATA('HAND(LEFT,RIGHT)')
DATA('CARD(RANK,SUIT)')
```

Then after executing
DEALER = HAND()
LEFT(DEALER) = CARD('ACE', 'CLUBS')
RIGHT(DEALER) = CARD(9, 'HEARTS')

the resulting data structure is shown in Figure 5:

![Diagram](image)

Figure 5. A HAND of CARDS

From this it should be evident that any object may be represented by a single cell.

Identifying Datatypes

It is difficult to tell the difference between the datatypes HAND and CARD above. They are both represented as blocks of two cells and cannot be distinguished except by the datatype in the T field of a pointer to them. This is a general problem. Since objects are not self-identifying, an additional device is needed for clarity. A simple notational convention that allows easy distinction between datatypes is giving each datatype its own unique shape. For example, the shape
can be used to represent a HAND, while

\[ \text{LEFT} \]
\[ \text{RIGHT} \]

\[ \text{RANK} \]
\[ \text{SUIT} \]

could be used to represent a CARD. Each shape is divided into its component fields as shown above. Confusion is minimized by the convention that all figures are drawn with the same orientation. Thus, Figure 5 above can be redrawn as

\[ \text{DEALER} \] → \{ ACE \} → \{ CLUBS \} → \text{9} → \{ HEARTS \}

Now T fields are extraneous since datatypes are implied by the shape of the object pointed to. A convenient convention is that an empty field represents the null string. Following the above idea, arrays are diagrammed as
Lists are represented by

Tables are represented by a diagram suggestive of their associative nature:
A BNF Grammar

One final example demonstrates a complicated data structure and shows how it is represented in SNOBOL4. It is a representation of a BNF grammar. The grammar is:

\[
\begin{align*}
<E> &::= <T> \mid <E> + <T> \\
<T> &::= <I> \mid ( <E> ) \\
<I> &::= A \mid B \mid <I> .
\end{align*}
\]

This grammar represents arithmetic expressions consisting only of additions of identifiers; identifiers consist of the letter A or B, or one of the two followed by any number of periods. The grammar chosen is admittedly simple, but only because of space limitations.

The internal representation of this grammar involves several levels. At the first level is a table whose indices are the nonterminal symbols. The corresponding values are data structures representing the right-hand side of the nonterminal. For any grammar, each right-hand side consists of one or more alternates of the nonterminal. Each alternate possesses one or more subsequents. A subsequent is either a terminal symbol or a nonterminal. There is a visual distinction between terminals and nonterminals, but a data structure that relies upon this would require a pattern match to determine whether a symbol is a terminal or nonterminal. Processing time can be reduced if another technique is used to distinguish between the two. This can be done easily by defining a new datatype, ELEMENT, that has
a FLAG field. The FLAG field is 1 if the symbol is a
terminal symbol, and null otherwise. An ELEMENT also con­
tains a VALUE field that holds the symbol itself. Thus, a
prototype of an ELEMENT is

\[ \text{ELEMENT(FLAG, VALUE)} \]

This means that a given subsequent, terminal or nonterminal,
is represented by an ELEMENT. These, in turn, are built up
into alternates: an alternate is an array of ELEMENTs.
Thus the alternate \(<E> + <T>\) of \(<E>\) is a array of three
ELEMENTs. Alternates are themselves built up into right­
hand sides by the same technique: a right-hand side is an
array of alternates. Finally, as indicated above, each
right-hand side is the value in a table where the index is
the corresponding nonterminal. The entire grammar is
diagrammed in Figure 6.
Figure 6. A SNOBOL4 representation of a BNF grammar.
CHAPTER 3

NEW DATA STRUCTURE FACILITIES FOR SNOBOL4

The facilities described in this chapter fall into two major categories. The first category, described in the next section, consists of a number of minor, useful extensions to the SNOBOL4 language. These have been implemented in an extended version of SNOBOL4 (Hallyburton and Griswold, 1973). The remainder of the chapter describes a major set of data structure facilities that are simulated in SNOBOL4 itself. The implementation of the features described in this chapter is discussed in Chapter 6.

Extensions to Existing Facilities

Lists

Frequently the SNOBOL4 programmer wishes to combine two or more objects and refer to them as a unit. An example is an array of fixed size whose elements are constants. The typical approach in these cases is for the programmer to create an array and initialize its elements or, alternatively, to define a new datatype and work with it. The particular method chosen depends somewhat on taste and the means by which the individual elements are to be accessed.

Although these methods achieve the desired result, their use is often cumbersome. Consider using the array
approach to forming an aggregate of the three strings RETURN, NRETURN, and FRETURN:

\[ A = \text{ARRAY}(3) \]
\[ A[1] = \text{"RETURN"} \]
\[ A[2] = \text{"FRETURN"} \]
\[ A[3] = \text{"NRETURN"} \]

or, using programmer-defined datatypes:

\[ \text{DATA}(\text{\textsc{TRIPLE} (FIRST,SECOND,THIRD)}) \]
\[ \cdots \]
\[ A = \text{TRIPLE (\text{"RETURN"}, \text{"NRETURN"}, \text{"FRETURN"})} \]

The main shortcoming of the array approach is the inconvenience of having an explicit initialization statement for every element.

The use of programmer-defined objects provides a convenient, readable means of creating aggregates but often forces the programmer to impose needless structure on the object. This can result in confusing code and a number of extra datatypes (PAIR, TRIPLE, QUADRUPLE, and the like) that serve no real purpose other than allowing the programmer to group objects together. A high-level language like SNOBOL4 should not force unnecessary structure upon data objects.

What is needed, then, is a means of grouping objects that is natural, easy to use, and does not require any explicit structure. A new datatype, LIST, which meets all these requirements, has been added to SNOBOL4.
Lists are created by executing expressions of the form

\[ [e_1, e_2, \ldots, e_N] \]

A list represents an ordered collection of values with no other structure imposed.

These lists should not be confused with "linked lists" which are a different kind of structure. Conceptually, a list resembles a one-dimensional array but is more convenient to use than an array in many cases.

If \( L \) is a list, say

\[ L = ["X", 1, &BAL] \]

then the individual elements may be accessed by subscripting. This is similar to referencing elements of linear arrays. For example, the value of \( L[3] \) is the built-in pattern \&BAL and \( L[4] \) fails.

Associated with every list is an additional attribute called its type. This allows the programmer to include some additional information in a list for identification purposes. The built-in function TYPE is used to reference the type field of a list. An example list:

\[ L = ["X", 1, &BAL] \]

\[ \text{TYPE}(L) = "TRIO" \]

The type of a list may be any object, and is initially null.
Field Referencing

One-dimensional arrays, lists, and programmer-defined datatypes all have a common structure both intuitively and in their implementation: they are blocks of values. The method of accessing these values depends upon the particular datatype of the object. Arrays require subscripts and programmer-defined objects require field functions. A uniform access method for both is useful; positional addressing is convenient and unambiguous.

The ITEM function has been modified to operate on lists (as described above) and to operate on programmer-defined data objects. These modifications allow a string that is the name of a field function to be used as a subscript of a variable that has a programmer-defined object as value. An integer may also be used to specify the field referenced by position.

If these statements are executed

```
DATA("SYS(CPU,DATC, MEM, PER)"
DATA("PP(CIO, MEM)"
DEC = SYS(1,4,256,127)
PPCLS = PP("4LCIOP","4LMEMP")
```

then the following expressions all reference the same value:

```
DATC(DEC)
DEC["DATC"]
DEC[2]
```

but the following expressions fail:
Also, \texttt{PPCLSI["MEM"]} is equivalent to \texttt{PPCLS[2]} but \texttt{DEC["MEM"]} is equivalent to \texttt{DEC[3]}.

Note that a field reference fails if the subscript is a character string that is not the name of a field function of the object being referenced. An example is \texttt{PPCLSI["CPU"]}. This is inconsistent with the error that would occur if the reference were given as \texttt{CPU(PPCLS)}. This decision was made largely on the basis of taste. One thinks of \texttt{CPU(PPCLS)} as a call to the function \texttt{CPU} with argument \texttt{PPCLS}. The philosophy of SNOBOL4 is that arguments of functions must satisfy datatype restrictions; failure to do so is an error. On the other hand, \texttt{PPCLSI["CPU"]} is thought of as referencing the CPU field of \texttt{PPCLS}. The most natural approach is to relate nonexistent field references to out-of-range field and array references, which fail under such circumstances.

\textbf{Associative Deques}

SNOBOL4 contains a number of features that enable the programmer to simulate stacks. These are discussed by Griswold (1974) where the conclusion is drawn that a fully dynamic allocation scheme is in general best in terms of ease of use and memory utilization. A fully dynamic allocation scheme is one in which each addition and deletion
causes a separate allocation and deallocation of the exact amount of memory required. Implementation of such a scheme is best accomplished using programmer-defined data objects and functions that perform the necessary actions.

Experience has shown that stacks are required in programs frequently enough to warrant their addition to the language. This suggests the definition of built-in functions STACK, PUSH, and POP to create stacks and manipulate them. While such an implementation has the advantage of consistency of language features (Weinberg, 1971), it also carries an aura of staleness. This is undesirable because SNOBOL4 is, among other things, a tool for programming language research, where new ideas can be tested in the environment of a running language. For this reason a somewhat different approach is taken.

A stack is a special case of a more general data structure, the deque (Knuth, 1968). A deque is a linear structure having two ends, called the head and the tail. Items may be added at either end, and the items may be removed from either end. There are no facilities for accessing entries that are not at the ends, nor may entries be inserted anywhere except at an end. A stack mechanism may be provided by consistently accessing only one of the two ends.

As indicated above, deques have been implemented in an unusual manner. A deque is not a separate datatype.
Rather, deques are associated with objects in much the same manner that table entries are associated with their indices. The functions PUSH and PUT may be used to add objects to deques. A call of the form

\[ \text{PUSH}(X, Y) \]

pushes the value of \( X \) onto the deque associated with the value of \( Y \). Similarly,

\[ \text{PUT}(X, Y) \]

appends the value of \( X \) to the deque associated with the value of \( Y \). The difference between PUSH and PUT is that they add items to different ends of the deque. An arbitrary choice is to say that PUSH adds items to the tail of the deque and PUT adds items to the head.

If PUSH or PUT is invoked but no deque is associated with the second argument, a deque is created having a single entry; namely the first argument in the call.

The function POP(Y) returns by value the item currently at the tail of the deque associated with the value of \( Y \), removing it from the deque in the process. If there is only one item in the deque when POP is called, it is removed and the deque vanishes. If POP is called but there is no deque associated with its argument, POP fails.

Analogous to POP, there is a function GET(Y) which acts on the head of the deque.

The following diagram illustrates typical usage of the deque associated with the value of the variable GROUP1.
Typically, something like

\[ \text{GROUP1} = 1 \]

will have been executed. The term "deque X" is often used to refer to the deque associated with the value of X.

Initial configuration:

(there is no deque)

After \text{PUSH('Z',GROUP1)}:

\[ \begin{array}{c}
\circ \\hat{Z} \\
\end{array} \]

After \text{PUSH('W',GROUP1)}:

\[ \begin{array}{c}
\circ \\hat{Z} \quad \circ \\hat{W} \\
\end{array} \]

After \text{PUSH('T',GROUP1)}:

\[ \begin{array}{c}
\circ \\hat{Z} \quad \circ \\hat{W} \quad \circ \\hat{T} \\
\end{array} \]

After \text{GET(GROUP1)}:

\[ \begin{array}{c}
\circ \\hat{W} \quad \circ \\hat{T} \\
\end{array} \]
After PUT(-6, GROUP1):

\[
\begin{array}{c}
\circ \quad -6 \quad \circ \quad \circ \\
\end{array}
\]

After POP(GROUP1):

\[
\begin{array}{c}
\circ \quad -6 \quad \circ \quad \circ \\
\end{array}
\]

After GET(GROUP1):

\[
\begin{array}{c}
\circ \quad \circ \quad \circ \\
\end{array}
\]

After POP(GROUP1):

(there is no deque)

Subsequent GET(GROUP1) fails.

Note that PUSH and POP used in conjunction provide a stack mechanism, as do PUT and GET. Also, PUSH and GET used in conjunction provide a queue (first-in, first-out) discipline, as do PUT and POP.

PUSH and PUT return the null string if they succeed. They fail if the deque is 'full', which by default occurs if there are over 1000 entries in the associated deque. For the deque associated with the value of \( Y \), \( \text{DEQLIM}(Y, I) \) sets the maximum number of entries to the value of the integer \( I \).
Shortcomings in SNOBOL4 List Processing

The macro implementation of SNOBOL4 resembles other high-level languages in that list-processing facilities are primitive and low-level. Consider a typical problem: that of searching a linked list for a specific item. For definiteness, assume the statement

\[ \text{DATA('NODE(VALUE, LINK)*')} \]

has been executed. A linked list, composed of such NODEs, might look like

\[
\begin{array}{c}
\text{LL} \\
\text{Y} \\
\text{ZETA} \\
\text{F}
\end{array}
\]

where the symbol

is the unique shape for a node; the top half being the VALUE field and the bottom half the LINK field. An empty field represents the null string. Braces indicate string structures. Thus the rightmost node above has F as the value of the VALUE field and the entire list is the value of the variable LL.
A typical problem is finding the node with a specific value field VAL. A SNOBOL4 procedure for locating such a value is:

```
DEFINE( 'PTR(NODE,VAL)' )
  
  PTR IDENT NODE)
  IDENT(VALUE(NODE),VAL) IF(FOUND)
  NODE = LINK(NODE) IF(PTR)
  FOUND PTR = NCDE IF(RETURN)
```

To locate the string ZETA, for example, the statement

```
X = PTR(NODE,'ZETA') IF(OK) F(NO)
```

would be used.

Note first that the procedure is short and the statements are basically simple; they are at a low, assembly-language-like level. This is characteristic of SNOBOL4 data structure manipulations, and is undesirable because it forces the programmer to spend a great deal of time and effort on essentially clerical details. A high level of formulation is not possible in SNOBOL4. Programs that manipulate data structures extensively tend to abound in small modules, producing large, unwieldy programs that are boring to write and keyboard, and difficult to debug and maintain. There is also the possibility that a new approach may inspire other new ideas that were previously unfeasible.
Turning again to the example above, note the use of the variable PTR. It is an example of a very common occurrence: a variable is used to point to a "current" location within a data structure where most of the processing takes place. In this example PTR is tested for a null value, indicating the end of the structure (another common occurrence). If its value is nonnull, PTR is assumed to be a NODE and its VALUE field is tested. If this is not the required node, then the LINK field is used to obtain the next node in the list. The previous node is forgotten, and the process is repeated with PTR now pointing to a different object.

This use of PTR bears a strong resemblance to the notion of a cursor as it is used in SNOBOL4 pattern matching (Gimpel, 1973). In fact, there are a number of similarities between structure-manipulation and pattern matching. The simple statement

\[ A[3] = "XYZ" \]

means "set the third element of A to the string XYZ". This can be reformulated as:

\[ A \[3\] = "XYZ" \]

which means "In A look for the third element and replace it by the string XYZ". This conveys the same meaning as the previous formulation, but sheds new light on the operations taking place. The notions of success and failure fall very
nicely into place:

\[ A[3] = 'XYZ' \]


This then is the basis for design of high-level structure-manipulation facilities: the notion of pattern matching on a string is extended to the notion of traversal of a data structure. Just as pattern matching has its scanner, traversing has its driver. Built-in patterns and pattern-valued functions are mapped into directives and directive-valued functions. Concatenation of patterns ("match this then match that") is mapped into sequentiation of directives ("apply this then apply that"). Alternation followed by failure may cause the driver to backtrack.

The following table summarizes the extension of terms from pattern matching to traversing.

<table>
<thead>
<tr>
<th>Pattern Matching</th>
<th>Traversing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pattern</td>
<td>Directive</td>
</tr>
<tr>
<td>Scanner</td>
<td>Driver</td>
</tr>
<tr>
<td>To match a pattern</td>
<td>To apply a directive</td>
</tr>
<tr>
<td>Success</td>
<td>Success</td>
</tr>
<tr>
<td>Failure</td>
<td>Failure</td>
</tr>
<tr>
<td>Backup</td>
<td>Backtrack</td>
</tr>
<tr>
<td>Cursor</td>
<td>Cursor</td>
</tr>
<tr>
<td>Needle</td>
<td>Needle</td>
</tr>
<tr>
<td>Concatenation (blank)</td>
<td>Sequentiation ((\rightarrow))</td>
</tr>
<tr>
<td>Alternation ((\mid))</td>
<td>Alternation ((\mid))</td>
</tr>
<tr>
<td>To match a (subject) string</td>
<td>To accept a directive</td>
</tr>
<tr>
<td>Matching procedure</td>
<td>Applicator</td>
</tr>
</tbody>
</table>

**Diagrams and Notation**

Some notation is required in order to explain the traversal algorithm and display the behavior of the built-in
directives. Various pointers, drawn as arrows, play prominent roles in this process. The graphical conventions are described below.

Suppose \( X = ['A', 'B'] \)

and \( Y = .X[2] \)

This is diagrammed as:

\[
\begin{align*}
(X) & \rightarrow [A] \\
(Y) & \rightarrow [B]
\end{align*}
\]

That is, \( X \) points to the structure; the entire structure is the value of the variable \( X \). On the other hand, \( Y \) points into the structure; only a certain field (the second field in this case) of the structure is the value of \( Y \). Thus, the arrow emanating from \( Y \) terminates with the arrowhead inside the structure. This is how \( X \) is distinguished from \( X[1] \) in the diagrams that follow.

During the traversal process, the cursor and needle are often dealt with as a pair and it is convenient to have some concise notation to refer to them. The term "cursor-needle pair," abbreviated CNP, refers to the pair of positions.

Most of the examples involve diagramming the actions taken by the driver. The cursor position in these diagrams is indicated by the symbol \( \uparrow \rightarrow \) and the needle position is indicated by \( \downarrow \rightarrow \). When a CNP is saved (as may be required
by the directive) the pair is nevertheless diagrammed as it makes the process more understandable. Such a CNP is referred to as an inactive CNP. To distinguish inactive and active CNPs, level numbers of inactive CNPs are written next to the symbol, as in \( \rightarrow \). The most recently saved CNP has the highest level number and is restored prior to all other CNPs with lower level numbers.

When a CNP is saved, the needle that is saved is the position in the directive that the needle will have when that CNP is restored, as opposed to the position in the directive pointed to by the needle when the CNP is saved. The two are usually different; an example is alternation: the active needle points to the first alternate, but the needle saved in the CNP points to the second alternate.

Figure 7 is a sample diagram of a step in the traversal process. The directive being applied is written above the data structure being traversed. The binary operators in the directive are explained later; they are not essential to an understanding of the diagram.
The active cursor points into a field of the data structure and the active needle points to the element \([2]\). By convention, the needle points to the next directive to be applied. There are two inactive CNPs.

Certain useful information, local to the traversal, may also be displayed. In this example, \(N = 2\) is such an item.

**Basic Operations**

**Sequentiation**

The binary operator \(\rightarrow\) sequentiates two directives. An expression of the form

\[
A \rightarrow B
\]

constructs a directive that the driver interprets as "apply
A then apply B*. If both directives succeed, then the sequentiation succeeds. Whether a given directive succeeds or fails depends upon the directive and usually upon the nature of the structure being traversed.

Alternation

The binary operator $\mid$ forms the alternation of two directives. The behavior of an alternation of directives is very similar to the behavior of the alternation of patterns. If

$$D = A \mid B$$

then the driver accepts either A or B. Implicit in this process is the fact that if A fails, its alternate B is applied. If A succeeds, there is still the possibility that B may be applied. This occurs if D's subsequent fails. This requires saving the CNP, an action taken by the driver whenever it detects an alternate to a directive component. The process of restoring the CNP and applying an alternate is called backtracking.

Forward Alternation

The binary operator $\mid$ creates the "forward" alternation of its arguments. Thus

$$D = A \mid B$$

constructs a directive that the driver interprets as follows: If A fails, then backtrack and apply B, as in alternation. If A succeeds, then do not save the CNP;
proceed with the subsequent to D. In this case, subsequent backtracking to B is not possible.

Forward alternation is a restricted form of regular alternation. It is intended to prevent attempting alternates in a situation where such attempts would be erroneous or unnecessary.

The following table shows the relative precedence of the operators used for constructing directives. The FOR and AND operators are discussed later.

<table>
<thead>
<tr>
<th>operation</th>
<th>precedence</th>
<th>associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>FOR</td>
<td>5</td>
<td>left</td>
</tr>
<tr>
<td>forward alternation</td>
<td>4</td>
<td>left</td>
</tr>
<tr>
<td>sequentiation</td>
<td>3</td>
<td>left</td>
</tr>
<tr>
<td>alternation</td>
<td>2</td>
<td>left</td>
</tr>
<tr>
<td>AND</td>
<td>1</td>
<td>left</td>
</tr>
</tbody>
</table>

**Special Directives**

**NULL String**

The NULL string is accepted as a valid directive (the "null directive") that performs no action and succeeds. This allows directives of the form

\[ A \rightarrow *F(X) \rightarrow B \]

to operate in the expected manner if F returns the NULL string. The driver applies A, then evaluates F (which presumably produces some side effect) then moves on and applies B.
Lists

A list is interpreted as a valid directive. Its effect is that of moving the cursor by indexing: if the cursor points to an aggregate, the aggregate indexed by the contents of the list becomes the new location pointed to by the cursor. Thus, evaluating the expression A[3] and the traversal

A [3]

reference the same location, but in slightly different contexts. Evaluation of A[3] returns the name of the third element of A, and the traversal

A [3]

makes the cursor the name of the third element of A.

An indexing directive (i.e., a list) may fail for a variety of reasons:

1. out-of-range indices and closed tables
2. indexing a subject that is not an aggregate
3. ill-formed indices (e.g., indexing an array with a pattern)

The last two cases are a departure from conventional SNOBOL4 concepts and apply only during traversal. This is the most natural manner of operation as it avoids datatype checks or special end-of-pointer-chain tests in directives. Some examples of this usage appear in later sections.

Another example involves positioning the cursor to a specific spot in a data structure:
A = ARRAY(4)

This traversal may be diagrammed as follows:

Initial configuration

after applying [4]. Note cursor position.
after applying [1]. The traversal succeeds.

Note the positions of the cursor in this example. The cursor is always a NAME. Its values in this example are

- A
- A[4][1]

in that order. The use of A[4][1] is notational only; it represents ITEM(A[4],1). The reason the cursor is a name is that it is possible to change the values of the elements within the subject structure; to do this requires the name of the element. It is possible to alter the value of the entire structure (i.e., the value of A in the above example) if the subject is a name.

A slight abuse of terminology is convenient: both "the current object" and "the value of the cursor" are intended to mean "the object pointed to by the cursor".
Unevaluated Expressions

Unevaluated expressions may be used in directives in much the same manner that they are used in pattern matching. When the driver encounters an unevaluated expression, the expression is evaluated and the resulting value is used.

As with pattern matching, there are times when unevaluated expressions must be used in order to defer evaluation of a component until the directive is applied. An example is:

\[ \ldots \rightarrow *\text{IDENT}(X,Y) \rightarrow \ldots \]
as opposed to a directive of the form:

\[ \ldots \rightarrow \text{IDENT}(X,Y) \rightarrow \ldots \]
The former defers the test until the directive is used and the needle points to the expression. The latter uses the test to determine if the directive should be built.

Cursor Examination

Frequently it becomes necessary to choose one of several paths in a directive depending upon the value of some field of the current object. In general it is difficult to address the current object without some knowledge of the driver's implementation. There are two ways the object pointed to by the cursor may be examined:

The natural variable \( \text{CO} \) points to the current object. The driver does not use \( \text{CO} \) or reference it. Rather, the driver assigns \( \text{CO} \) the value of the cursor at each step.
in the traversal process. A typical application is:

\[ \ldots \rightarrow *\text{IDENT(DATATYPE(CO),'ARRAY')} \ldots \]

Another common need involves referencing fields of the current object. It is quite natural to want to test a certain field of the current object for a specific datatype or value, and succeed or fail depending on the result. Although

\[ \text{CO(FIELD)} \]

makes the correct reference, this method is somewhat cumbersome and not very pleasant to use. Once again, frequency of use indicates the need for a special operator.

The unary / operator returns the value of the current object indexed by the value of its argument. Thus,

\[ /\text{FIELD} \]

is equivalent to CO(FIELD).

It should be stressed that the variable CO and unary / are not directive-valued. Consequently, CO and / usually appear in unevaluated forms.

It is possible, using the / operator, to have self-processing data structures. Such a structure contains within itself directives necessary to process itself. This can be accomplished by including a PROC field in each element of the structure. The PROC field contains a directive to be invoked by the driver when the element is to be processed. Invoking the PROC field is quite simple,

\[ */'\text{PROC}' \]
as a directive, accesses the PROC field and uses the contents as the next directive.

**The Driver Algorithm**

The driver determines the next directive to be applied and, in case of failure, performs backtracking. It must also process termination of the traversal.

The directive components themselves are applied by applicators -- procedures written specifically to handle each type of component. The applicators may or may not advance the cursor, but perform some specific actions and signal either success or failure via transfer to one of two specific locations within the driver (referred to as S and F).

The term "current node" refers to the node pointed to by the active needle; a stack is used to save inactive CNPs for later use. The basic algorithm (Gimpel, 1973) is as follows:

1. Push null onto the stack. (This is not shown in diagrams.)

2. Set the needle to the first component of the pattern and the cursor to the name of the subject, creating one if none exists.

3. If an alternate to the current node exists, push the CNP onto the stack. (The needle position pushed is that of the alternate.)
4. Transfer control to the applicator associated with the current node. Control returns to either S or F.

(S) 5. If a subsequent to the current node exists, point the needle at it and return to 3.

6. Otherwise the traversal terminates successfully.

(F) 7. Pop the highest level CNP off the stack. If the needle is nonnull, return to 3.

8. Otherwise the traversal has terminated in failure.

An example of the basic algorithm is the traversal shown below where the directive PLURAL succeeds if and only if the (binary tree) structure contains more than one TNODE.

```
DATA('TNODE(VALUE,LEFT,RIGHT)')
PLURAL = ['LEFT'] -> ['LEFT'] | ['RIGHT'] -> ['RIGHT']
   ... 
TREE  PLURAL  S(MANY)F(ONE)
```

```
['LEFT'] -> ['LEFT'] | ['RIGHT'] -> ['RIGHT']

{TREE} -> {SMALL}
```

After step 2, not showing the null stack entry
After step 3; the stack is represented by the inactive CNP.

After step 4, the cursor is moved, control returns to S.
After step 5, the needle advances to the next directive.

No change after step 3 as there is no alternate.

After step 7, the CNP has been restored.

No change after step 3 as there is no alternate.
After step 4, the cursor is moved, control returns to S.

After step 5, the needle points to the next component.

No change after step 3 as there is no alternate.
After step 4, step 7. No alternates exist.

Step 8 is taken and the traversal fails.

Directive Interpretation of SNOBOL4 Patterns

The built-in patterns ABORT, FAIL, FENCE, and SUCCEED may be used as directives. They perform the same operations that they perform in pattern matching. The built-in pattern MATCH, which exists in some versions of SNOBOL4 (Hallyburton and Griswold, 1973), causes immediate successful termination of the pattern match. It, too, may be used as a directive and causes immediate successful termination of the traversal. DONE may be used in place of MATCH; the two are identical.

Other SNOBOL4 patterns and pattern-valued functions, excluding ARBNO(), are not directly applicable to data structures.
In many cases the subject structure has an iterative organization or some part of it is to be processed iteratively. Linked lists are one such group of data structures. Usually the number of elements in such a list is unknown and may even vary during traversal. The control structures in SNOBOL4 pattern matching are not well suited to iterative processing and force one to use recursive or failure-motivated patterns, iterative statement execution, or some combination of these.

The function RPT permits a straightforward approach to these problems. The directive returned by RPT(D), where D is a directive, repeatedly places the driver under control of the directive D as long as D succeeds. When D fails, RPT(D) succeeds, leaving the cursor pointing wherever it is after the last successful repetition of D.

If D contains alternates, their scope is local to each iteration. After a successful iteration, untried alternates (CNPs) are removed from the stack.

An example of the use of RPT is given by the problem of locating the end of a linked list of NODEs.
DATA("NODE(VALUE, LINK)")

LIST RPT([2])

For representative data, the diagram is

![Initial state diagram]

An initial state of the list with nodes containing 3, 3.1, and 3.14, with the list structure illustrated.
The AND Operator

Sequentiation can be thought of as the binary "then" operator in the sense of $A \rightarrow B$ meaning "apply $A$ then apply $B$". Similarly, alternation is the "or" operation and
forward alternation is the "exclusive or" operation. These colloquial terms are used to suggest the nature of the operations involved, and are convenient in that capacity.

The binary AND operator, &, has an interpretation, following the above idea, of "apply the first directive and then the second directive at the same place." This means that if

\[ D = A \& B \]

then the driver, upon encountering D, notes the CNP and applies A. When A terminates, either in success or failure, the CNP is restored and B is applied. The cursor is left wherever it is when B terminates.

As with RPT, directives of the form A & B always succeed and after each iteration (in this case there are two, one for A and one for B) alternatives established in that iteration are discarded.

The AND operator facilitates recursive traversal of structures, and of binary trees in particular. An example of the use of binary & is counting the number of nodes in a binary tree. (This and future examples presuppose the existence of a built-in function INCR which adds one to its argument. INCR exists in a slightly more general form in some implementations of SNOBOL4 (Hallyburton and Griswold, 1973). INCR returns the null string.)
DATA('TNODE(VALUE,LEFT,RIGHT)')
COUNT = *INCR(N) \rightarrow (\{2\} \rightarrow *COUNT \& \{3\} \rightarrow *COUNT)

\ldots

N = 0
TREE COUNT

The diagram for this traversal is:

*INCR(N) \rightarrow (\{2\} \rightarrow *COUNT \& \{3\} \rightarrow *COUNT)

\begin{center}
\begin{tikzpicture}
\node (tree) {\{TREE\}};
\node (two) [right of=tree] {2};
\node (one) [below of=two] {1};
\node (three) [right of=two] {3};
\node (n) [below of=three, yshift=-2cm] {N = 0};

\path [->] (tree) -- (two);
\path [->] (two) -- (one);
\path [->] (two) -- (three);
\end{tikzpicture}
\end{center}

1. Initial positions of the cursor and needle.
2. *INCR(N) is applied. This sets N = 1 and returns the null directive, which succeeds. Advancing the needle into a \&-component causes the driver to note the cursor position and the second half of the component. This causes a level-1 CNP to be pushed.

3. [2] causes the driver to move the cursor to the second field of whatever it was pointing to previously. [*LEFT*] would have the same effect here, as LEFT is the name of the second field.
4. This is a recursive invocation of the directive. The implementation actually notes the CNP but this is not displayed in order to avoid unnecessary cluttering of the diagrams.

5. Same as frame 2. Another CNP is pushed; these are at level 2 and are used before the level-1 pair.
6. Same as frame 3.

7. Same as frame 4.
8. As in frames 2 and 5, the \&-directive saves a new CNP.

9. The directive [2] signals failure as the cursor points to the null string. There are no alternates; this corresponds to failure (termination) of the A half of an A & B directive. The B half is invoked by restoring the highest level CNP, which is the level-3 pair in this case.
10. The directive [3] is invoked and, as did the directive constructed by [2], signals failure as the new (formerly level 3) cursor points to the null string. Failure is transmitted back and causes failure of COUNT as invoked from frame 6. As in frame 9, the highest level CNP is restored and becomes the active CNP.

11. The directive [3] is successfully invoked. The cursor and needle are updated.
12. COUNT is again invoked.

13. N is again incremented. Note that this is the second (and last) incrementing of N for this node. The needle again enters an &-directive so the driver saves the CNP.
14. [2] fails, the CNP at level 2 is restored.

15. [3] fails, the level-1 CNP is restored.
16. [3] moves the cursor to point to the third field of TREE.

17. COUNT is again invoked.
18. \( N \) is again incremented.

\[
\text{INCR}(N) \rightarrow ([2] \rightarrow \text{COUNT} \& \ [3] \rightarrow \text{COUNT})
\]

\[
\text{TREE} \rightarrow 2 \rightarrow \text{COUNT} \& \ [3] \rightarrow \text{COUNT}
\]

\[
N = 5
\]

19. \([2]\) is successful, the CNP is updated.
20. The driver performs a recursive invocation of COUNT.

21. N is incremented.
22. [2] fails, the CNP from frame 21 is restored.


25. Another recursive invocation of COUNT.
26. \( N \) is incremented. Again a new CNP is saved since an &-component is encountered.

27. \([2]\) fails. The CNP is restored.
\*INCR(N) -> (2) -> *COUNT & (3) -> *COUNT

28. \[3\] fails. There is no CNP to restore, and so the traversal fails.

---

N is incremented twice at each node, once after the \[2\] and once after \[3\]. Additionally, N is incremented at the start of the operation. Thus, the formula \((N - 1) / 2\) gives the total number of nodes -- in this case three.

This example illustrates many typical situations. Generally, failure is not regarded as undesirable, a connotation sometimes found in pattern matching. Rather, failure is used as a signal indicating the logical end of a process -- an exit. This fact, along with the conventional use of the null string to terminate a chain of pointers, is the basis for a number of useful directives.

The directive COUNT depends heavily upon recursion. This is acceptable, since the subject has a recursive structure. The objections to recursive directives occur
when recursion is used to simulate iteration, a situation that can usually be avoided by using RPT.

The directive COUNT can be used as a prototype for traversal of binary trees in a top-down, left-to-right fashion. The general form is:

\[ TBT = \text{PROC} \rightarrow \{[2] \rightarrow \ast TBT \& [3] \rightarrow \ast TBT\} \]

where PROC is a directive written by the user to process each node.

A slightly modified form of TBT can handle binary trees, "ternary" trees, or any mixture of the two:

\[ TRI = \text{PROC} \rightarrow \{[2] \rightarrow \ast TRI \& [3] \rightarrow \ast TRI \& [4] \rightarrow \ast TRI\} \]

There are obvious generalizations. The directive above works on binary trees because the directive [4] consistently fails.

In the previous examples it is assumed that the information for each node is contained in the first field of each node. Trivial modifications can be made if this is not the case.

There are, of course, many applications where a top-down, left-right traversal is unsuitable. There are simple directives for strict left-to-right and right-to-left processing. A typical example would be left-lsting a binary tree. Such a directive is presented in Chapter 4.
The FOR Operator

A generalization of the & operator is the binary FOR operator, \( \circ \). Unlike the other operators, where a call of the form

\[ A \circ B \]

produces a directive with a self-contained meaning, the effects generated by

\[ D = A \circ B \]

depend on the subsequents to \( D \). A typical directive involving the FOR operator has the form

\[ ... A \circ B \rightarrow D_1 \rightarrow ... \rightarrow D_n \rightarrow \text{ENDFOR} ... \]

(The precedence of \( \circ \) is highest of all binary operators.) Stated loosely, the FOR operator "distributes" its arguments \( A \) and \( B \) over the subsequents \( D_1, D_2, ..., D_n \). It is equivalent to the form

\[(A \rightarrow D_1 \rightarrow ... \rightarrow D_n) \& (B \rightarrow D_1 \rightarrow ... \rightarrow D_n)\]

The built-in directive ENDFOR is required to separate the FOR-construct from its possible subsequents. If the sequentiation \( D_1 \rightarrow ... \rightarrow D_n \) consistently fails the ENDFOR is not required.

When the driver encounters a directive of the form \( A \circ B \), the following occurs:

1. The CNP is saved, the needle half pointing to \( B \).
2. The driver applies \( A \). If \( A \) fails, control passes to step 5. Alternatives created by \( A \) are discarded.
3. The driver applies \( D_1, D_2, ..., D_n \). If failure
backtracks past D1, control passes to step 5, as if A had failed.

4. Otherwise either the traversal terminates or ENDFOR is encountered. If ENDFOR is encountered, control passes to step 5.

5. Alternatives created by A, D1, D2, ..., Dn are discarded.

6. The CNP from step 1 is restored.

7. The driver applies B. If B fails, control passes to step 10. Alternatives created by B are discarded.

8. The driver applies D1,D2, ..., Dn. Failure backtracking passes control to step 10, as if B had failed.

9. Otherwise ENDFOR is encountered.

10. Alternatives created by B,D1,D2, ..., Dn are discarded.

11. The cursor is left pointing wherever B,D1,D2,...,Dn indicate.

12. The traversal continues with the subsequent to ENDFOR.

FOR-constructs do not fail. In this way FOR resembles AND and RPT, which also do not fail. There is no intrinsic reason why this must be the case, but to provide the failure capability would require additional built-in directives. For simplicity, this has not been done.
The main use of the FOR-construct is to shorten forms such as:

\[(A \rightarrow D_1 \rightarrow \ldots \rightarrow D_n) \& (B \rightarrow D_1 \rightarrow \ldots \rightarrow D_n)\]

into forms of the type

\[A \& B \rightarrow D_1 \rightarrow \ldots \rightarrow D_n \rightarrow END\]

which amounts to "factoring out" the \(D_1, \ldots, D_n\). For example, the directive COUNT on page 58 can be written as

\[\text{FORCOUNT} = \ast\text{INCR}(N) \rightarrow [2] \& [3] \rightarrow \ast\text{FORCOUNT}\]

Note that END is not required since FORCOUNT cannot succeed.

The directive TRI on page 74 may be rewritten as


**Conditionals**

Most complicated directives require choosing from among various paths depending upon dynamic conditions. One possible way of making such a choice involves the use of failure and alternation to select the required path. This, however, is a rather poor method because it is awkward and leads to directives that are difficult to understand. The solution to this problem involves defining new directives that are specifically designed to handle the common cases.

The Function IF

The most basic and general conditional is that created by the function IF. Calls to IF take the form

\[\text{IF}(*\text{TEST}, \text{THEN}, \text{ELSE})\]
When such a directive is encountered during traversal, the argument TEST is evaluated. If this evaluation succeeds, the driver next applies the value of THEN and continues. Should evaluation of TEST fail, ELSE is applied next. Only one of THEN or ELSE is applied whenever the IF-construct is encountered.

A typical call is:

\[ \text{IF}(*\text{IDENT}(\text{DATATYPE}(\text{CO}),'\text{NODE}'), \text{ISNODE}, \text{ISNTNODE}) \]

where ISNODE and ISNTNODE represent the two alternate paths, only one of which is to be taken. As with forward alternation, subsequent failure and backtracking do not cause ISNTNODE to be invoked if ISNODE was selected first. Unlike forward alternation, failure of ISNODE causes failure of the IF-construct instead of invoking ISNTNODE.

The first argument to IF is almost always an unevaluated expression, since the test must be deferred until traversal.

Null String Tests

One of the most frequent kinds of IF-constructs is one where the current object, or one of its fields, is tested for a null value. Because of this frequency, special-purpose directives are provided. One of these is the function IFNULL which takes one of two paths, in the style of IF, depending upon the status (null or nonnull) of a specified field of the current object. For example,

\[ \text{IFNULL}(3, \text{ISNUL}, \text{NOTNUL}) \]
tests the third field of the current object and if that reference succeeds and the third field is null, then the directive ISNUL is invoked. If the reference fails or the third field is nonnull, the directive NOTNUL is invoked. In general,

     IFNULL(FIELD,NULLDIR,NOTNULLDIR)

is equivalent to

     IF(*IDENT(/FIELD),NULLDIR,NOTNULLDIR)

It is sometimes desirable to determine if the current object is the null string, rather than to examine some field of the current object. There are two built-in directives that perform this test. The directive IFNULL succeeds if the current object is null and fails otherwise. Thus, the directive

     IFNULL

is equivalent to

     IF(*IDENT(CO),FAIL)

Additionally, there is a built-in directive IFNONNULL that succeeds if and only if the current object is nonnull. Thus

     IFNONNULL

is equivalent to

     IF(*IDENT(CO),FAIL)

Conditional Indexing

Very often one encounters a directive of the form

     IFNULL(FIELD,FAIL,[FIELD])
that advances the cursor ([FIELD]) if that field is nonnull. This situation may occur when FIELD points to further structures to be analyzed. If there are no such structures, FIELD is null. (This, at least, is conventional.) A directive-valued function, CNDX, can be used in these circumstances: CNDX(F) fails if the F field of the current object is null, otherwise it advances the cursor to point to the F field of the current object.

An example of the use of CNDX is given by the problem of searching a binary tree for a node with null VALUE field. This particular example also uses some of the other facilities previously described.

\[
\text{SRCH} = \text{CNDX}(2) \to \ast \text{SBTREE} \& \text{CNDX}(3) \to \ast \text{SBTREE} \\
\text{SBTREE} = \text{IFNULL('VALUE', DONE, SRCH)}
\]

\[
\text{TREE \quad SBTree}
\]

If the VALUE field of TREE is null, IFNULL invokes the directive DONE and the traversal terminates successfully with the cursor pointing to the node with null VALUE field. Otherwise the directive SRCH is invoked, which checks the second and third fields, calling SBTREE recursively in the process. The traversal proceeds in a top-down, left-to-right manner until a null VALUE field is encountered. If no null VALUE field is encountered, the traversal fails.
Otherwise it succeeds. The use of CNDX here involves a test at each node (unless the traversal is terminated by finding an appropriate node) for null second and then null third field. If the tested field is null then it is not examined further.

**NOT**

NOT(D) applies the directive D and succeeds if D fails, failing if D succeeds. The directive NOT(NOT(D)) is not equivalent to D, since NOT(NOT(D)) does not move the cursor if D succeeds.

**Structures Containing Loops**

**MARK**

Since many data structures contain loops, traversing such structures can cause great difficulties if there is no way of avoiding looping during traversal. The built-in function MARK may be used to avoid these problems.

MARK(N)

constructs a directive that, during a given invocation of the driver, fails if the cursor points to the same location more than N times. That is, whenever a MARK() component is encountered, the driver checks the cursor position against an internal list of positions and sees if that position has been encountered more than N times. If this is the case, MARK(N) signals failure, otherwise success.
The directive MARK is equivalent to MARK(1). Thus, it can be used to prevent processing an object more than once in a given traversal.

Occasionally it may be of use to have the driver "forget" the list of objects encountered by a MARK directive. Such a case might occur if a directive consists of more than one logical section, where the second section performs processing unrelated to the first section. The natural variable CLEARMARK instructs the driver to discard the entire history of objects encountered.

Unknown Structures

Occasionally the need arises to handle a structure of unknown contents or form. One example involves program debugging: it would be very convenient to have a function that would take an arbitrary data structure and print its contents, or search it for a specific object. This type of problem warrants a general type of directive that can traverse an entire data structure without any a priori knowledge of the form of the structure. Several directives exist that make this kind of traversing possible.

POINTER and SIMPLE

The built-in directive POINTER succeeds if the current object is an aggregate; otherwise it fails. POINTER does not move the cursor. The built-in directive SIMPLE succeeds in situations in which POINTER would fail, that is,
when the cursor points to an object that is not an aggregate.

ENTER, NEXT, and EXIT

It is possible to traverse an arbitrary structure by positioning the cursor to the first field of the structure then advancing, step by step, through the remaining fields. Three directives make this possible: ENTER, NEXT, and EXIT.

The built-in directive ENTER fails if the current object is not an aggregate. Otherwise it moves the cursor and points it to the "first" field of the current object, regardless of the datatype of the current object. The following table summarizes what is meant by the "first" field of a given aggregate:

<table>
<thead>
<tr>
<th>Datatype</th>
<th>&quot;First&quot; Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td>[1] element</td>
</tr>
<tr>
<td>&lt;defined&gt;</td>
<td>[1] element</td>
</tr>
<tr>
<td>Array</td>
<td>the element with least subscripts</td>
</tr>
<tr>
<td>Table</td>
<td>the first value in chronological sequence</td>
</tr>
</tbody>
</table>

ENTER can thus be used to begin a traversal of an unknown structure. Once ENTER has been applied and the field pointed to has been processed, it may be desirable to process the next field of the aggregate ENTERed. This may be accomplished by using the directive NEXT, which moves the cursor to the next field of the object being examined.
Again, the directive does not need to know the datatype of the object being examined; NEXT just moves the cursor to the next field. In the case of lists and defined objects, the next field is defined to be the next field in sequence, as expected. In the case of tables, the next field is treated specially. Instead of moving the cursor to the next value in chronological sequence, NEXT positions the cursor to the index of the first element. After this, the cursor points to the second value, then its index, and so on. For arrays, the next field varies the leftmost subscript most rapidly, as the following example illustrates:

\[
A = \text{ARRAY}(-1:1, 2)
\]

The first field of \(A\) is \(A<-1,1>\), followed by

\[
A<0,1> \\
A<1,1> \\
A<-1,2> \\
A<0,2> \\
A<1,2>
\]

in that order.

A NEXT directive fails if there is no next field in the object being examined.

The built-in directive EXIT serves to terminate the examining process by restoring the cursor to its position prior to the most recent ENTER.
Replacement

A number of directives that can analyze data structures have been described. The reason there are so many directives is simply that there are so many different ways structures can be analyzed. Constructing a data structure, piece-by-piece, involves inserting elements at various locations.

The following two steps are required to add an element to a structure:

1. Depending upon the structure and the element to be added, locate the correct spot in the data structure where the element is to be placed.

2. Insert the element at the located spot.

The relative complexity of (1) compared to (2) indicates that one would expect the analysis facilities to be more complex than the synthesis facilities. This is indeed the case.

The basic replacement directive is constructed by the unary equal sign operator, =. When the driver encounters a directive of the form

= X

the current object is replaced by the value of X. The cursor is not moved -- only what it points to is changed.
Typically, the variable \( X \) is an unevaluated expression, since directives are normally constructed out of line, and the value to be inserted is defined some time after the directive is constructed. This means that the driver must perform an evaluation if the argument is an expression. If such an evaluation is required (i.e., the argument is an unevaluated expression), it is performed only once and the resulting value is used.

An example of the use of replacement is adding a term to a singly-linked list:

\[
\begin{align*}
\text{Initial state. NEWVALUE has been computed to be NODE}(3.14) .
\end{align*}
\]

\[
\begin{align*}
\text{After the first iteration of RPT(2)}.
\end{align*}
\]
The second iteration locates the end of the list.

After failure of $[2]$, RPT([2]) succeeds.

The replacement occurs. Note that further traversing is possible.

**SET**

SET(A,B) is a directive-valued function that performs assignment. When encountered by the driver, it evaluates A (which must return by name) and evaluates B,
then assigns B to A. It is equivalent to executing \( A = B \) during traversal.

Neither A nor B is evaluated until the driver encounters them. Until such time, they are kept as unevaluated expressions. It should be noted that this uses a nonstandard method of handling arguments and is possible only in a specially-modified version of SNOBOL4. (Hallyburton and Griswold, 1973).

One use of SET involves changing the value of a field of the current object without moving the cursor:

\[
\text{SET}(/2, /2 + 1)
\]

constructs a directive that adds one to the second field of the current object. It is equivalent to *\text{INCR}(/2).

Cursor Positioning

SNOBOL4 pattern matching does not allow the subject to be changed during the course of a pattern match. The main reason that this is not allowed is that certain concepts are not well-defined, particularly the results of pattern components that deal with the cursor position. It is, however, very reasonable to permit traversal of several different structures at once. One case where this might arise is when the contents of one structure depend on the contents of another.

The function CURSOR(X) changes the cursor to the value of X. The only restriction is that the value of X
must have datatype NAME or be a nonnull string. In particular, X need not be physically associated with the current subject structure. The following example shows how CURSOR can be used to alter the subject structure dynamically.

Problem: Obtain the VALUE field of the last element in a linked list of NODEs. This field is presumed to contain an index into a table T. Test to see if the table entry corresponding to the index is null, and succeed or fail accordingly.

This may be accomplished by the directive:

\[ \text{RPT(CMDX(2)) \rightarrow CURSOR(*.T[1]) \rightarrow IDENT(CO)} \]

Here RPT(CMDX(2)) locates the last element in the list. The cursor still points to this last element while the argument to CURSOR is evaluated. Evaluation results in a call to unary / which returns the first (VALUE) field which is then used as the subscript to T. Unary \( \cdot \) returns a NAME that the CURSOR function stores into the traversal cursor. This makes CO the value of T[1], which is the element to be tested.

Tagging the Cursor Position

It is occasionally necessary to save the cursor position for later reference. One possible way to do this is

\[ \text{SET(TEMP, CURSOR)} \]

but this does not permit saving a series of cursor positions that might, for example, result from an RPT loop. This same
problem precludes the use of a unary operator \$ that, in the style of pattern matching, assigns the cursor to its operand. Consequently some means must be provided for collecting an unknown number of cursor positions. Once this has been done, some means must also be provided for returning the collected values, one at a time. There are advantages to returning these values in the order they were saved, and there are times when reverse order is preferable. This suggests a deque mechanism, and the deque facilities, described in this chapter, are a logical choice. The associative facility also lets the programmer save the cursor positions on a particular, named, deque. This is useful if more than one series of cursor positions must be kept.

The unary \$ operator \$USHes the cursor position on the deque associated with the value of its operand. An example is saving the cursor position on the deque associated with the string HAT:

... \$ 'HAT' \$ ...

The argument to \$ may have datatype EXPRESSION, in which case it is EVALed when encountered by the driver.

After unary \$ has created the deque, the functions POP and GET may be used to remove the cursor positions. Care must be taken, however, since the value returned is a NAME. Unary $, as in $POP(X)$, must be used if the value is desired; this would correspond to pushing CO.
CHAPTER 4

EXAMPLES

The examples in this chapter demonstrate the new facilities roughly in increasing order of complexity. Various types of data structures are used, including some with loops. When a given example does not involve a structure with loops, a binary tree is usually the structure chosen. In these cases, usually only minor modifications to the directives are necessary to apply them to another, different structure without loops.

AGDUMP

Some of the following examples construct data structures and others analyze them. In order to verify the results, there must be some way to print the data structures. A function AGDUMP has been written to do this. AGDUMP is given two arguments: the first is a pointer to the structure to be dumped, and the second is a maximum depth if only the first levels are to be dumped. If the second argument is null, as it is in all of the examples, the entire structure is dumped. If an object is pointed to by more than one cell, the second and subsequent cells indicate this by naming the first cell, rather than dumping the same substructure again. This technique, demonstrated below, permits
structures with loops to be dumped. The SNOBOL4 source code
for AGDUMP is given in Appendix A.

Sample Aggregate Dump

The first example merely demonstrates AGDUMP, con­
structing some simple structures and then dumping them.

* * A SIMPLE STRUCTURE * *
X = AGDEX(100,200,300)
AGDUMP(X)
* * MAKE THE SECOND FIELD OF X A STRUCTURE * *
BFLD(X) = AGDEX("210","FIELD3 OF BFLD")
AGDUMP(X)
* * NOW FOR A STRUCTURE WITH TWO POINTERS * *
* TO THE SAME OBJECT.*
* AGDUMP([X,BFLD(X)])
END

The dump follows.
AGGREGATE DUMP:

OBJECT = AGDEX

1. AFLD = 100
2. BFLD = 200
3. FIELD3 = 300

AGGREGATE DUMP:

OBJECT = AGDEX

1. AFLD = 100
2. BFLD = AGDEX

2.1. AFLD = '210'
2.2. BFLD = **
2.3. FIELD3 = 'FIELD3 OF BFLD'

3. FIELD3 = 300

AGGREGATE DUMP:

OBJECT = LIST(2)

[1] = AGDEX

[1]1. AFLD = 100
[1]2. BFLD = AGDEX

[1]2.1. AFLD = '210'
[1]2.2. BFLD = **
[1]2.3. FIELD3 = 'FIELD3 OF BFLD'

[1]3. FIELD3 = 300

Constructing a Binary Tree

The second example shows a directive that constructs a lexically-ordered binary tree. This example, and later examples, use a programmer-defined function WORD() that returns the next word from the input file and fails when there are no more. The coding of WORD is straightforward (Griswold and Griswold, 1973) and has been omitted here.

TREE-CONSTRUCTING DIRECTIVES.

CHOOSE = (IF(*LGT(/1,WORD),['LEFT'],['RIGHT']) ->
+ IFNONN) ! (=*LEAF(WORD) -> DONE)

LOOK = RPT(IF(*IDENT('/VALUE',WORD),DONE,CHOOSE))

INITIALIZE

HEAD = LEAF(WORD()) IF(ERROR)

LOOP TO CONSTRUCT THE TREE

READ WORD = WORD() IF(TREPU8)

HEAD LOOK *(READ)

DUMP THE TREE

TREPU8 AGDUMP(HEAD)

END

SEARCH FOR THE SWORD THAT WAS BROKEN. IN ILMADRIS IT DWELLS

Understanding how this directive was written may help in understanding how it works. The tree is initialized so it has at least one node. Doing it by hand, one would start at the top and see if the current node is the correct one. If this is the case, the algorithm terminates as the word is already in the tree. If not, then one must test the
new word against the word in the current node, and move down to the left or right accordingly. Consequently, if a null field is encountered, the end of the tree has been reached and no match was found. In this case, a node containing the word is inserted in place of the null field.

Corresponding to this are the directives LOOK and CHOOSE. The first action taken is testing the new word against the word in the current node. The conditional IF(*IDENT(/'VALUE*',WORD),DONE,CHOOSE) does this. It is interpreted as follows: if the VALUE field of the current node is equal to the WORD to be inserted, then DONE is invoked and the traversal terminates. If not, then a choice of paths must be made. In order to keep the directive LOOK reasonably neat in appearance, the choice is made in another directive, CHOOSE, that is constructed elsewhere. It should be noted that this construction should be performed prior to building LOOK. Otherwise, when LOOK is constructed the value of CHOOSE (the directive that chooses which path to take down the tree) will not be the desired value (it would most likely be null). This of course could be circumvented by making the call to CHOOSE an unevaluated expression. The method used was selected on the basis of taste.

Therefore, the directive CHOOSE will be invoked if the word in the current node does not match the new word. CHOOSE must perform a lexical comparison between the two. This is done by calling IF. If the first field of the
current node is lexically greater than the word to be inserted then the cursor moves down to the left, otherwise to the right. Thus, the indexing directives ['LEFT'] and ['RIGHT'] are the "true" and "false" sections of the call to IF. After this, a test is made to see if the end of the tree has reached. If not, then the entire process is to be repeated at the new node. This implies that the LOOK directive has an RPT organization. If the end of the tree has been reached, IFNONN fails and the (forward) alternate is taken. This corresponds to not finding the word in the tree. Thus, a node containing the word is created and inserted, replacing the null string. Then the directive DONE is applied to terminate the traversal.

The result of dumping the tree produced by the traversal follows:
AGGREGATE DUMP:

OBJECT = LEAF
1. VALUE = 'SEARCH'
2. LEFT = LEAF

2.1. VALUE = 'FOR'
2.2. LEFT = LEAF

2.2.1. VALUE = 'BROKEN'
2.2.2. LEFT = ''
2.2.3. RIGHT = LEAF

2.2.3.1. VALUE = 'DWELLS'
2.2.3.2. LEFT = ''
2.2.3.3. RIGHT = ''

2.3. RIGHT = LEAF

2.3.1. VALUE = 'IN'
2.3.2. LEFT = LEAF

2.3.2.1. VALUE = 'IL MADRIS'
2.3.2.2. LEFT = ''
2.3.2.3. RIGHT = ''

2.3.3. RIGHT = LEAF

2.3.3.1. VALUE = 'IT'
2.3.3.2. LEFT = ''
2.3.3.3. RIGHT = ''

3. RIGHT = LEAF

3.1. VALUE = 'THE'
3.2. LEFT = LEAF

3.2.1. VALUE = 'SWORD'
3.2.2. LEFT = ''
3.2.3. RIGHT = LEAF

3.2.3.1. VALUE = 'THAT'
3.2.3.2. LEFT = ''
3.2.3.3. RIGHT = ''

3.3. RIGHT = LEAF

3.3.1. VALUE = 'WAS'
3.3.2. LEFT = ''
3.3.3. RIGHT = ''
Circumstances may dictate that, if the word is already in the tree, further processing is to be done. For example, if this technique were used to build a concordance, one might add an additional field, LINES, to the prototype of a LEAF. Then the LINES field of a given leaf could point to a linked list of the line numbers in which the word appears. In such a case, a directive to perform the additional processing would be written and replace the occurrence of DONE in LOOK.

**Tree Analysis**

The next example analyzes the same binary tree. The directive SCHK (Size Check) creates two deques from the tree. The first contains all the nodes where the word in the VALUE field is longer than four characters. The second deque is for nodes containing the remaining words, those that are four or less characters long. The deque for the longer words is associated with 5; the other is associated with 4. (Small integers are handy when tagging the cursor position.)
TREE-CONSTRUCTING DIRECTIVES.

CHOOSE = (IF(*LGT(/1,WORD),['LEFT'],['RIGHT']) -> IFNONN) ! (=*LEAF(WORD) -> DONE)

LOOK = RPT(IF(*IDENT('/VALUE',WORD),DONE,CHOOSE))

ADDED DIRECTIVES

SCHK = IF(*GT(SIZE(/i),@5,@4) -> *MORE

INITIALIZE

HEAD = LEAF(WORD()) :F(ERROR)

LOOP TO CONSTRUCT THE TREE

READ WORD = WORD() :F(TREHUNT)
HEAD LOOK t(READ)

ANALYZE THE TREE

TREHUNT HEAD SCHK

OUTPUT = 'LONGER THAN 4 CHARACTERS!'
OUTPUT =
G4LOOP OUTPUT = VALUE($GET(5)) :S(G4LOOP)
OUTPUT =
OUTPUT = '4 OR LESS CHARACTERS!'
OUTPUT =
L4LOOP OUTPUT = VALUE($GET(4)) :S(L4LOOP)
END
SEARCH FOR THE SWORD THAT WAS BROKEN. IN ILMADRIS IT DWELLS

The SCHK directive decides which deque to tag the current object with and calls MORE. MORE uses the FOR operator to move down the tree, first to the left and then to the right. Part of the FOR-construct is the directive

IFNONN -> SCHK

which calls SCHK recursively as long as the current object
is nonnull. When the current object is the null string, IFNONN fails and the process backs up and moves on to other modes.

LONGER THAN 4 CHARACTERS:
SEARCH
BROKEN
DWELLS
ILMADRI
SWORD

4 OR LESS CHARACTERS:
FOR
IN
IT
THE
THAT
WAS

**Left-listing a Binary Tree**

The fourth example left-lists a binary tree. If the tree is constructed with a lexical ordering, as is the case here, then a left-listing produces the words in alphabetical order. The simplicity of the directive is surprising.

This example is also essentially a solution to a problem posed by Richard A. Stone at the SHARE XXXVIII Conference. Stone proposed this to be a standard problem used to compare all list-processing languages (Stone, 1972).
DATA('LEAF(VALUE,LEFT,RIGHT)')

TREE-CONSTRUCTING DIRECTIVES.

CHOOSE = (IF(*LGT(/I,WORD),['LEFT*],['RIGHT*]) ->
+ IFNONN) ! (=*LEAF(WORD) -> DONE)

LOOK = RPT(IF(*IDENT(/'VALUE*",WORD),DONE,CHOOSE))

ADDED DIRECTIVE

LL = (CNDX(2) -> LL) & @'TREE' & (CNDX(3) -> *LL)

INITIALIZE

HEAD = LEAF(WORD()) $F(ERROR)

LOOP TO CONSTRUCT THE TREE

READ WORD = WORD() $F(TREHUNT)
HEAD LOOK $(READ)

LEFT-LIST THE TREE AND DELETE IT AFTERWARDS

LLIST HEAD LL
OUTPUT =
OUTPUT =
OUTPUT = 'LEFT LISTED:'
OUTPUT =
LLOOP OUTPUT = VALUE($GET(*TREE*)) $S(LLOOP)
HEAD =
END

SEARCH FOR THE SWORD THAT WAS BROKEN. IN ILMADRIS IT DWELLS

For every node in the tree, starting at the root, the directive LL left-lists the LEFT field of the node, then the node itself, then the RIGHT field of the node. Here the term "left-list" means "pushes nodes onto the deque associated with the string TREE in a left-to-right manner."

This example also shows the use of the binary & operator to initiate the two different subtraversals from the same cursor position.
The function CNOX is also used here. Its effect is to either halt or continue the downward plunge of LL, depending on whether the end of the branch has been reached.

**LEFT LISTED:**

BROKEN
DWELLS
FOR
*ILMADRIS*
IN
IT
SEARCH
SWORD
THAT
THE
WAS

**The Directive COUNT**

In Chapter 3 a directive COUNT was presented for counting the number of nodes in a binary tree. The next example verifies the result. In the light of the previous examples, the directive should require no explanation.
DATA('LEAF(VALUE,LEFT,RIGHT)')

* TREE-CONSTRUCTING DIRECTIVES.
* CHOOSE = (IF(*LGT(/I,WORD),['LEFT'],['RIGHT'])) -> IFNONN) ! (=*LEAF(WORD) -> DONE)

LOOK = RPT(IF(*IDENT('/VALUE',WORD),DONE,CHOOSE))

* ADDED DIRECTIVE
* COUNT = *INCR(N) -> ([2] -> *COUNT) & ([3] -> *COUNT)

* INITIALIZE
* HEAD = LEAF(WORD()) IF( ERROR)

* LOOP TO CONSTRUCT THE TREE
* READ WORD = WORD() IF(COUNT)
  HEAD LOOK :(READ)

* COUNT NODES
* COUNT N = 0
  HEAD COUNT

* OUTPUT THE RESULT
* OUTPUT = (N - 1) / 2 * WORDS IN THE TREE.*

END

SEARCH FOR THE SWORD THAT WAS BROKEN. IN ILMADRIS IT DWELLS

The output is:

11 WORDS IN THE TREE.

Traversing an Arbitrary Structure

The next example is a directive that traverses an arbitrary structure, visiting each cell exactly once. In a given application, what is to be done at each cell depends on circumstances. In this example, each cell is pushed on a
deque. This amounts to creating a "linearization" of the structure.

* CREATE A DATA OBJECT

 DATA('LEAF(VALUE,LEFT,RIGHT)')
 LTOP = LEAF('VAL')
 LEFT(LTOP) = ARRAY(3,'HERE')
 RIGHT(LTOP) = LEAF('RLV',LEFT(LTOP),LTOP)

* PRINT IT OUT

 AGDUMP(LTOP)

* CREATE A DIRECTIVE TO TRAVERSE ANY STRUCTURE

 ALOOP = RPT((TRBT -> FAIL) ! NEXT)
 TRBT = MARK -> &6 -> ENTER -> ALOOP -> EXIT

* TRAVERSE

 LTOP TRBT

* PRINT THE LINEARIZATION

 OUTPUT = 'THE TRAVERSAL IS'
 TL OUTPUT = $GET(6) $S(TL)
 OUTPUT = '...DONE...' END

The best way to understand how the directive works is to examine TRBT first. MARK is used to prevent possible program looping due to loops in the structure. Next the cursor position is pushed. The remainder of TRBT is significant only if the cursor points to an aggregate. If this is the case, the aggregate is ENTERed and ALOOP is called, recursively applying TRBT to each cell in the aggregate.
TRBT may either succeed or fail. Failure occurs if MARK sees the same object twice, or if an object that is not an aggregate (this is checked by ENTER) is found. In each case the failure is of no consequence as the object has already been pushed. On the other hand, TRBT may instead succeed by EXITing from an aggregate. In other words, the result of a call to TRBT in ALOOP can be success or failure. Which one occurs is immaterial from the user's standpoint, however, the traversal algorithm requires both cases to be treated. In this case the method used is forcing failure (FAIL) if TRBT succeeds.

AGGREGATE DUMP:

OBJECT = LEAF

1.VALUE = 'VAL'
2.LEFT = ARRAY("3")
2.[1]. = 'HERE'
2.[2]. = 'HERE'
2.[3]. = 'HERE'

3.RIGHT = LEAF
3.1.VALUE = 'RLV'
3.2.LEFT = OBJECT 2.
3.3.RIGHT = OBJECT

THE TRAVERSAL IS
LEAF
VAL
ARRAY("3")
HERE
HERE
HERE
LEAF
RLV
ARRAY("3")
LEAF

...DONE...
Duplicating a Structure

One use of the linearizing directive above is as part of a directive to duplicate a data structure. The directives BLOOP and NMCOPY strongly resemble ALOOP and TRBT; the only real difference is the action taken when each cell is found. Instead of pushing the cursor position, NMCOPY makes a copy of the object and replaces the old value with the copy. Modifications to the SNOBOL4 COPY procedure permit any object to be copied. Objects that cannot be copied (names, expressions, and so forth) are returned unaltered so that the copy is the same as the original.
CREATE A DATA OBJECT

DATA('LEAF(VALUE, LEFT, RIGHT)')
LTOP = LEAF('VAL')
LEFT(LTOP) = ARRAY(3, 'HERE')
RIGHT(LTOP) = LEAF('RLV', LEAF(1, 2), LEAF('R'))

CREATE A DIRECTIVE TO TRAVERSE ANY STRUCTURE

ALOOP = RPT((TRBT -> FAIL) ! NEXT)
TRBT = MARK -> 36 -> ENTER -> ALOOP -> EXIT

THE COPYING DIRECTIVE

BLOOP = RPT((NMCOPY -> FAIL) ! NEXT)
NMCOPY = MARK -> =*COPY($GET(6)) -> ENTER ->
+ BLOOP -> EXIT
COPYNL = TRBT -> CURSOR(.MTOP) -> NMCOPY

TRAVERSE

LTOP   COPYNL

DUMP. THE FORM OF THE CALL TO AGDUMP INSURES THAT
INCORRECT COPYING WILL RESULT IN A POINTER INTO LTOP
BEING EASILY RECOGNIZED.

AGDUMP([LTOP, MTOP])

This particular implementation forms a linearization
of the subject structure LTOP, repositions the cursor to the
name of the variable MTOP, and performs the duplicating pro-
cess. The value of MTOP before the traversal is unimport-
ant; after the traversal it is a duplicate of LTOP. The
resulting output follows.
AGGREGATE DUMP:

OBJECT = LIST(2)

[1] = LEAF

[1]1.VALUE = 'VAL'
[1]2.LEFT = ARRAY("3")

[1]2.[1]. = 'HERE'
[1]2.[2]. = 'HERE'
[1]2.[3]. = 'HERE'

[1]3.RIGHT = LEAF

[1]3.1.VALUE = 'RLV'
[1]3.2.LEFT = LEAF

[1]3.2.1.VALUE = 1
[1]3.2.2.LEFT = 2
[1]3.2.3.RIGHT = 

[1]3.3.RIGHT = LEAF

[1]3.3.1.VALUE = 'R'
[1]3.3.2.LEFT = 
[1]3.3.3.RIGHT = 

[2] = LEAF

[2]1.VALUE = 'VAL'
[2]2.LEFT = ARRAY("3")

[2]2.[1]. = 'HERE'
[2]2.[2]. = 'HERE'
[2]2.[3]. = 'HERE'

[2]3.RIGHT = LEAF

[2]3.1.VALUE = 'RLV'
[2]3.2.LEFT = LEAF

[2]3.2.1.VALUE = 1
[2]3.2.2.LEFT = 2
[2]3.2.3.RIGHT = 

[2]3.3.RIGHT = LEAF

[2]3.3.1.VALUE = 'R'
[2]3.3.2.LEFT = 
[2]3.3.3.RIGHT = 

The traversal also could have been written

\texttt{LTOP TRBT}

followed by

\texttt{MTOP NMCOPY}

breaking the process into two parts. This method differs slightly from the former method in that this one succeeds when the subject is not an aggregate, the former fails.

The COPYNL directive is reasonably simple and not difficult to write if the programmer has some familiarity with the facilities. Unfortunately, it is not very general. If the subject structure contains two pointers to the same object, two different copies of that object are made. Thus

\begin{center}
\begin{tikzpicture}
  \node (s) at (0,0) [draw, circle] {1};
  \node (a) at (-1,-1) [draw, circle] {2};
  \node (b) at (1,-1) [draw, circle] {2};
  \draw (s) -- (b);
\end{tikzpicture}
\end{center}

is "copied" into

\begin{center}
\begin{tikzpicture}
  \node (a) at (0,0) [draw, circle] {1};
  \node (b) at (-1,-1) [draw, circle] {2};
  \node (c) at (1,-1) [draw, circle] {2};
  \draw (a) -- (b);
\end{tikzpicture}
\end{center}

This amounts to a copy-and-unfold directive that may be useful in some instances, but a general copying directive is more worthwhile. It is possible to add a new directive
that would handle the situation correctly, but such a solution is artificial as it would have extremely limited use. When situations of this nature arise the programmer must supply a function to perform the desired processing. The next example shows how this is done.

**Correctly Duplicating a Structure**

The function DUP uses a table to keep track of the objects that have been copied. Any object that has been copied has an index into DUPTABLE with the corresponding value being the copy. Each time DUP is called, it looks in the table to see if its argument is an index. If not, it is added and the copy is made and stored as the value. If its argument is already in the table, the copy is returned directly. Since the table must be global to the traversal, it is necessary to initialize it prior to a particular traversal.
CREATE A DATA OBJECT

DATA('LEAF(VALUE,LEFT,RIGHT)')
LTOP = LEAF('VAL')
LEFT(LTOP) = ARRAY(3,'HERE')
RIGHT(LTOP) = [LTOP,LEFT(LTOP),RIGHT(LTOP)]

CREATE A DIRECTIVE TO TRAVERSE ANY STRUCTURE

ALOOP = RPT((!*TRBT -> FAIL) ! NEXT)
TRBT = MARK -> @6 -> ENTER -> ALOOP -> EXIT

THE NEW COPYING DIRECTIVE

CLOOP = RPT((!*XXCOPY -> FAIL) ! NEXT)
XXCOPY = MARK -> =$DIDP($GET(6)) -> ENTER ->
+ CLOOP -> EXIT

COPYANY = TRBT -> CURSOR(.NEW) -> XXCOPY

THE SERVICE FUNCTION TO DETECT PREVIOUSLY COPIED OBJECTS.

DEFINE('DUP(OBJ) X') *(DUP.END)
DUP = DIFFER($X) $X =$S(RETURN)
$X = COPY(OBJ)
DUP = $X = = (RETURN)

DUP.END

TRAVERSE

DUPTABLE = TABLE()
LTOP COPYANY

DUMP. AGAIN THE FORM OF THE CALL WILL DETECT INCORRECT COPYING.

AGDUMP([LTOP,NEW])

Now, instead of calling COPY, the directive calls DUP which supplies the correct copy of the object. The output for this traversal follows.
AGGREGATE DUMP:

OBJECT = LIST(2)

[1] = LEAF

[1]1.VALUE = "VAL"
[1]2.LEFT = ARRAY("3")

[1]2.[1] = 'HERE'
[1]2.[2] = 'HERE'
[1]2.[3] = 'HERE'

[1]3.RIGHT = LIST(3)

[1]3.[3] = ""

[2] = LEAF

[2]1.VALUE = "VAL"
[2]2.LEFT = ARRAY("3")

[2]2.[1] = 'HERE'

[2]3.RIGHT = LIST(3)


Analyzing a BNF Grammar

Another example involves analyzing a BNF grammar and producing a list of left-derived symbols from a given non-terminal. The data structure presented in Chapter 2 is analyzed for all terminal symbols that are left-derived from the nonterminal A. This directive does not handle empty right-hand sides.
Note that a rather complex data structure does not, at least in this case, require a very complicated directive. This problem would require much more work if conventional techniques had to be used. In this example, MARK prevents looping due to left recursion. ENTER then initiates a search among all the alternates wherein the list [1] picks up the first (leftmost) subsequent. The nature of the RPT loop shows that this procedure is followed for the first subsequent to each alternate for the nonterminal. These are precisely the candidates for being left-derived symbols. With each of the leftmost subsequents, GETLDS tests field 1 (the FLAG field). A nonnull FLAG field denotes a terminal
symbol that gets pushed onto deque 5 and causes GETLDS to return to inside the RPT loop, to restore the cursor position (pointing somewhere inside the array of alternates) and to apply the null directive. Then NEXT moves on to the next alternate, terminating the traversal if there is none.

If the test inside GETLDS indicates that the symbol is nonterminal, the cursor is repositioned to point to the grammar entry for the nonterminal and ALTHUNT is called recursively with a new subject. Note that MARK causes failure if the nonterminal has previously been examined in this traversal. This is correct, since a previous examination will have already pushed the left-derived symbols for the nonterminal.

The next pages present an aggregate dump of the grammar followed by the traversal output.
AGGREGATE DUMP:

OBJECT = TABLE(10,10)

[*E*] = ARRAY("2")

[*E*][1]. = ARRAY("1")

[*E*][1].[1]. = ELEMENT

[*E*][1].[1].1.VALUE = *

[*E*][1].[1].2.FLAG = *T

[*E*][2]. = ARRAY("3")

[*E*][2].[1]. = ELEMENT

[*E*][2].[1].1.VALUE = *

[*E*][2].[1].2.FLAG = *E

[*E*][2].[2]. = ELEMENT

[*E*][2].[2].1.VALUE = 1

[*E*][2].[2].2.FLAG = *+

[*E*][2].[3]. = ELEMENT

[*E*][2].[3].1.VALUE = *

[*E*][2].[3].2.FLAG = *T

[*T*] = ARRAY("2")

[*T*][1]. = ARRAY("1")

[*T*][1].[1]. = ELEMENT

[*T*][1].[1].1.VALUE = *

[*T*][1].[1].2.FLAG = *I

[*T*][2]. = ARRAY("3")

[*T*][2].[1]. = ELEMENT

[*T*][2].[1].1.VALUE = 1

[*T*][2].[1].2.FLAG = *("}

[*T*][2].[2]. = ELEMENT
LEFT-DERIVED SYMBOLS FOLLOW

A
B
C
D

"T"[2].[2].1.VALUE = "E"
"T"[2].[2].2.FLAG = 'E'
"T"[2].[3]. = ELEMENT
"T"[2].[3].1.VALUE = 1
"T"[2].[3].2.FLAG = '

"I" = ARRAY("3")
"I"[1]. = ARRAY("1")
"I"[1].1. = ELEMENT
"I"[1].1.1.VALUE = 1
"I"[1].1.2.FLAG = 'A'

"I"[2]. = ARRAY("1")
"I"[2].1. = ELEMENT
"I"[2].1.1.VALUE = 1
"I"[2].1.2.FLAG = 'B'

"I"[3]. = ARRAY("2")
"I"[3].1. = ELEMENT
"I"[3].1.1.VALUE = "I"
"I"[3].1.2.FLAG = 'I'

"I"[3].2. = ARRAY("1")
"I"[3].2.1. = "."
This example uses the technique of

\[
((1) \rightarrow \text{GETLDS} \& \text{NULL})
\]

to assure that the cursor position is saved and restored so that no matter where GETLDS positions the cursor, and regardless of the depth of recursion that follows, the cursor will be properly restored. Thus, NEXT always causes the cursor to point to the next alternate. The right operand, NULL, of binary & exists simply because & requires two operands. The directive NEXT would not work in place of NULL as eventual failure of NEXT is required to break the RPT loop, and & does not transmit failure.

**Dumping Protected Keywords**

The final example is more lighthearted. The function KEYDUMP() produces a dump of all the unprotected keywords by traversing through the SNOBOL4 unprotected keyword block. Some knowledge of the internals of SNOBOL4 (Griswold, 1972b) is helpful. Unprotected keywords are stored in a block that looks like a programmer-defined object to the driver. Each keyword name is stored in a cell, preceded by a cell containing its value. These pairs of cells are arranged internally in reverse alphabetical order. Since &TRIM is last in order, it is first in the block. This is why &TRIM is the subject of the traversal. The directive and its usage is shown below.
* THE DIRECTIVE
* KEYCUMP.DIR = RPI(@1 -> NEXT)
* THE FUNCTION
* DEFINE('KEYDUMP()') *(KEYDUMP.END)
KEYDUMP OUTPUT = 'UNPROTECTED KEYWORDS'
OUTPUT =
&TRIM KEYDUMP.DIR
KEYDUMP. OUTPUT = '&' $POP(1) * = ' $POP(1)
+ 'S(KEYDMP.)F(RETURN)
KEYDUMP.END
* THE CALL
* KEYDUMP()
END

UNPROTECTED KEYWORDS
&ABEND = 0
&ADUMP = 0
&ANCHOR = 0
&ATRACE = 0
&BTRACE = 0
&CODE = 0
&DUMP = 2
&ERRLIMIT = 0
&FDUMP = 0
&FTRACE = 0
&FULLSCAN = 0
&INPUT = 1
&MATCHLIMIT = 5000
&MAXLENGTH = 5000
&MTRACE = 0
&OUTPUT = 1
&STLIMIT = 50000
&TRACE = 0
&TRIM = 0

The directive pushes the cursor position onto deque 1 and moves to the next element in the block. Note that next is applied without a preceding ENTER. This is
perfectly legal and well-defined, since the cursor started out as an interior pointer.

Similarly, if \( T \) is a table,

\[
T["X"] \quad \text{NEXT} \rightarrow ="Y"
\]

is well-defined. The cursor initially points to the value of \( T["X"] \). Then \( \text{NEXT} \) moves the cursor to point to the "next" field, which is the index of \( T["X"] \), namely the string \( X \). This is replaced by the string \( Y \). The effect is therefore one of altering the table index rather than the value. This operation could only be done with great difficulty in standard SNOBOL4, even though it is perfectly legitimate and well-defined.
CHAPTER 5

EVALUATION

Experience has shown that directives are easy to write and debug, as well as easy to understand. The examples in the previous chapter demonstrate that complicated data structures can be handled by fairly simple directives. Thus, the evidence indicates that directives provide a powerful tool for eliminating much of the overhead that complicates data structure manipulations. There are two reasons for this.

The Data-driven Approach

Directives are data objects. As such, they may be constructed out of line and combined with other directives. For example, a directive to traverse an arbitrary structure may be used as part of a duplicating directive and also as part of a directive to print a data structure. Creating directives is particularly simple; all one does is call the appropriate functions and operators. An actual traversal involves very little code, which improves the overall readability of programs. Contrast this with having to define a number of small functions, and then having to call them one after (or inside) another. This leads to many problems: having to execute a DEFINE of each function but
placing the code for it out of line, explicitly RETURNing from each function, and other details one would rather avoid.

Another advantage to a data-driven approach is that once used, a directive can be deleted (by assigning the variable another value) and its space reclaimed. It is generally not possible to reclaim the space that code occupies.

**Control Directives**

One surprising result of the driver's implementation is the large number of control directives. While this is not good in itself (fewer primitives make learning easier), at least the directives are reasonably familiar in nature and are uniform in their usage. For example, the different IF-constructs all have the second argument as the "success" directive and the third argument as the "fail" directive.

Use of control functions does make directives easy to write and read. A statement of the form

```plaintext
LOOK  X = DIFFER(X) LINK(X)  IS(LOOK)
```

is more complex than the equivalent form

```plaintext
RPT(['LINK'])
```

One reason directives are so terse and control directives so simple is that the cursor position is an implicit parameter in most operations. This in turn demonstrates what is normally a characteristic of data
structure operations -- structures are analyzed in a regular, point-to-point manner. Human beings tend to design data structures so that directives can be used. In other words, processing a data structure is usually done on a localized basis: it is the conditions prevailing at some "current" spot within the data structure that determine the action to be taken. A good example of this is the problem of inserting a word into a binary tree, depending upon a lexical comparison at a given node. The result of this comparison in turn determines the location of a new node where the process is repeated. This continues until some specified conditions are met at the current node. Only rarely do problems arise where two or more different locations determine the course of action.

Other Benefits

Directives provide a number of additional benefits. Certain operations, such as traversing an arbitrary structure, use directives that only could be simulated with great difficulty in standard SNOBOL4. The concept of indexing into any object, regardless of its datatype, allows one to view aggregates in a very uniform manner. Similarly, such directives as ENTER, NEXT, EXIT, and MARK perform the same action regardless of the type of aggregate on which they operate. This unified approach simplifies thinking and permits a straightforward solution to many problems.
Lessons from the Implementation

Preponderance of Operators

Most of the traversal facilities are represented by operators. In retrospect, a functional notation for some, such as AND and FOR, probably would have been better. One of the most frequent causes of bugs in directives is operator precedence errors. By reducing the number of binary directive-valued operators, the source of these errors would be removed. There are other advantages to be gained -- there is no reason why AND and FOR have to take exactly two arguments.

Use of Alternation

Curiously, none of the examples in the previous chapter involve (regular) alternation. Yet the driver algorithm is in part motivated by considerations of alternation and backtracking. Evidently the type of processing performed by directives reduces the need for backtracking. This echoes the philosophy that conditions prevailing at one spot in the data structure are the major indicators for further processing. Alternation provides the user with the ability to process a structure in a different way depending upon occurrences that take place (perhaps considerably) later. This is simply not necessary. The driver algorithm perhaps could be rewritten and simplified, after further experimentation with the facilities.
Problems with Directives

The directive functions and operations essentially constitute a programming language of their own, with all the implied advantages and shortcomings. The advantages have been described, but there are some shortcomings that should be discussed.

Primitives

There are a fair number of primitive functions and operations that have to be learned before they can be used. Although each one is simple in itself, there are many of them. There are times when even more built-in directives would be useful. An example is a directive that would cause an RPT loop to terminate signaling failure. In general, there could be directives to abort execution of the current control structure and signal either success or failure. Perhaps a cursor assignment operator, in the style of pattern matching, could be added. Similarly, immediate and conditional value assignment operations are candidates for inclusion. The decision was made at one point during the implementation to provide all these facilities. After working some examples and problems, the least useful operations were dropped as overcomplicating the system. (The SNOBOL4 patterns that have a directive interpretation were retained even though some of them, such as SUCCEED, are not of much use. In this case, consistency was judged more worthwhile than simplicity.)
Evaluation Time

The examples in Chapter 4 make careful use of unevaluated expressions. This is a problem for the inexperienced programmer; all "forward references" to directives that have not been constructed must be kept unevaluated. This is required because statements of the form

\[ D = A \rightarrow B \]

use the current values of A and B to construct D. On the other hand,

\[ D = \ast A \rightarrow \ast B \]

uses whatever values that A and B have when the needle points to them during traversal. Also, when

\[ D = A \rightarrow \ast B \]

is executed, the current value of A is used in constructing D, but the value of B is obtained during traversal.

The directive D constructed by

\[ D = \ast A \rightarrow \ast B \]

is slightly different from the expression constructed by

\[ D = \ast(A \rightarrow B) \]

Knowing the difference between the two formulations indicates a fairly good understanding of evaluation times. The first operation sequentiates two expressions and forms a directive. When encountered by the driver, A is evaluated and applied. If A succeeds, then B is evaluated and applied. The sequentiation occurs once, when D is built. On
the other hand, the second statement causes very little to be done when it is executed. When encountered by the driver the expression is automatically evaluated, resulting in the evaluation of both A and B prior to applying A. Then the two are sequentiated, and the resulting directive is applied. The sequentiation must take place each time the expression is encountered during traversal, as opposed to only once prior to traversal. This is probably a waste of time (a small waste, however) unless A does something unusual such as redefining the value of B. The two forms also may produce different results if evaluation of B produces side effects. In the first case, B is not evaluated until and unless A succeeds, but in the second case, B is evaluated prior to applying A. Without such side effects, the two are equivalent.

Another time unevaluated expressions are needed is when conditional directives are being used. For example

```plaintext
IF(*LGT(*VALUE',WORD))
```

requires the argument to be unevaluated since the test must be deferred until the driver encounters the IF-construct.

Unevaluated expressions are also usually required when the variable CO or unary / are used, since the value at traversal time is typically wanted. Otherwise the value that the operand has when the directive is built will be used.
Similarly, the replacement operator = sometimes needs to have an unevaluated expression as its argument, if the value depends upon dynamic conditions. Example 2 in Chapter 4, and subsequent examples, follow this pattern.

In all the examples, unevaluated expressions are used only if required. This may help in understanding how they are used.

Compatibility of Language Features

In its present form, the traversal system is incompatible with the syntax of the standard version of SNOBOL4. Here a statement of the form

```
SUB OIR
```

represents a traversal, but in standard SNOBOL4 this statement represents a pattern match. A natural solution is to have a binary operator ? to represent one of the operations, as in

```
SUB ? DIR
```

However, this is not possible in standard SNOBOL4. Such a statement is syntactically incorrect, since the syntax of SNOBOL4 does not allow such an expression to be the subject. This restriction is due to the positional nature of the subject. In a statement such as

```
S P
```

the blank between S and P does not represent concatenation but rather pattern matching. The replacement statement is also positional. The SNOBOL4 compiler therefore does not
allow any binary operations in the subject unless parentheses or brackets enclose them. Thus

\[(S \quad P)\]

is necessary to represent concatenation.

It is possible to modify the compiler so that

\[S \quad ? \quad D\]

becomes an acceptable statement, but then the replacement operation would have to be abandoned, since the two statement forms are incompatible. Griswold (1973) discusses the merits of this approach.

Another solution is to formulate the traversal system in functional syntax. Thus

\[\text{TRAVERSE(SUB,DIR)}\]

could represent a traversal. This method has the advantage of fitting comfortably into SNOBOL4 syntax, but there are two disadvantages. First, it is not a natural way to write traversals. Traversals can be complex and the functional notation masks the important role of a traversal. The main reason the pattern-matching statement form was redefined in this work to indicate traversal is that in such a statement the subject serves as a focus of attention (Galler and Perlis, 1970). Also, the statement form itself draws the reader's attention. When reading the two statements

\[P = \text{ARBNO}(\text{LEN}(1) \text{ BAL}) \quad \& \quad \text{OUTPUT FAIL}\]

and

\[S \quad P\]
the SNOBOL4 programmer skims over the first statement and concentrates his attention on the second. The same seems to be true for traversal; the statement form draws attention to itself and, at the same time, the subject serves as a focus of attention for the operation. This attraction of attention is highly desirable: it helps the reader skim over details and obtain an outline of program flow.

Another disadvantage of the functional notation is that, in a degenerate case at least, it is possible to modify the value of the subject. Thus

\[
S = 77
\]

is a valid traversal that sets \( S \) equal to 77. This means that the name of the subject must be available. There is no standard SNOBOL4 function that requires an argument by name, although there are operators that do. Consequently,

\[
\text{TRAVERSE}(S, = 77)
\]

does not fit comfortably into the context of SNOBOL4, and could be misleading. In any event, it is unnatural.

Another solution is to let the form

\[
X Y
\]

represent either pattern matching or traversal, depending upon the datatypes of \( X \) and \( Y \). The similarity between pattern matching and traversal is quite strong and indicates that perhaps there is a more general mechanism that could subsume the two. If future development leads to such a
unification of string and list processing, then the statement form might be a very natural representation. In fact, such a notation could encourage development in this area.

Even without a unified view of pattern matching and traversal, there is no reason why the two operations cannot share the same statement form. SNOBOL4 contains many polymorphic operators and functions. In some instances (such as concatenation), operations have a wide range of effects and many different internal procedures can be called, depending upon the datatypes of the arguments. In this case, some modifications to the pattern matching routine are required, but they are not very difficult: the routine would have to check to see if the datatypes of the operands indicate a traversal, and, if so, call the driver.

There are other advantages that could result from this approach. Pattern matching can use some of the primitive operations defined for the driver. In particular, the function RPT has a very natural interpretation in pattern matching. Interpretations could also be defined for many other functions, such as AND and FOR. Thus, it is possible to enrich the pattern matching system without introducing new primitives.

Possibilities for Further Research

The traversal system opens up some interesting areas for further research. A few ideas are presented below.
The concept of multiple cursors may be quite useful. Additionally, there may be some use for "vector mode" processing where the number of cursors varies dynamically, depending upon the number of pointers that emanate from any given node. (This kind of processing has application in simulation of neural networks.)

Now that high-level processing of structures is possible, it may be feasible to look into new kinds of aggregates since they can be easily simulated. Sets, for example, can be studied and a number of operations on them evaluated for suitability. A more complicated type of set is a hash structure. A hash structure is a set whose elements are automatically partitioned into a number of different subsets by an internal procedure. This technique shortens search time when an element is to be located, but makes certain other operations, such as determining the (chronologically) first element that was inserted, more difficult. The table of natural variables in the macro implementation is an example.

There is still the possibility that new primitives can be added to the system. A good idea of what would be useful can be derived from working a large number of problems and looking at the results. Particular attention could be directed to processing structures with loops since there are not very many built-in directives for doing this. In particular, processing of graphs could be simplified if
there were a built-in aggregate, along the lines of a set, that could freely grow and contract. This in turn would inspire new directives for operating on such a datatype, or perhaps generalizations of existing directives.
CHAPTER 6

THE IMPLEMENTATION

High-level Simulation

The traversal system has been fully implemented and tested on an experimental basis. Most of the functions are written in SNOBOL4 and thus are simulated rather than being incorporated as part of the language (see Appendix B). This approach has several advantages. Changes can be made easily and quickly, since system reassemblies are not required. New facilities are easy to add; this encourages experimentation and testing in a working environment. Additionally, facilities that turn out to be inadequate or just bad ideas can be discarded without any inhibitions that might have resulted from having put a lot of effort into them.

Experience with the implementation indicates that high-level simulation of proposed facilities is a reasonable approach to software design. In particular, SNOBOL4 excels in this area for several reasons. There are extensive diagnostic facilities for simple, high-level tracing and debugging. If required, language features can be added by redefining standard functions and operators. If necessary, programs can be processed and converted to code dynamically.
Modifications to SNOBOL4

In implementing the facilities described here, simulation provided the maximum flexibility. However, some modifications to SNOBOL4 itself were necessary, in order to implement features that cannot be handled by simulation. The list, field referencing and deque facilities were implemented by modifying the SNOBOL4 system. Similarly, pattern matching was made available as a function so that it could be redefined using OPSYN. This facility was included so that traversal could be represented by a statement of the form

```
S    D
```

A number of other modifications have been made locally to the macro implementation. These include the list and deque facilities, as well as the field referencing additions. Additionally, some special modifications were required to permit implementation of one of the driver's facilities, NEXT. The rest of the system is implemented in SNOBOL4X (Hallyburton and Griswold, 1973).

Functions Required by NEXT

When a NEXT directive is encountered, the driver must determine the datatype of an aggregate, given a pointer (the cursor) to the interior of the aggregate. This is necessary because NEXT must take different actions for each datatype. The function DCODE(N) takes a name as its
argument and analyzes the structure into which the name points. Arrays and tables can be identified because they contain, in computable locations, cells that identify them. The only other aggregates are lists and programmer-defined objects. Lists are distinguished by a special flag that is placed in a known location within the list (an F flag in the title). Any object that is not one of the three discernable datatypes must, by elimination, be a programmer-defined object. DCODE returns an integer representing either the datatype of the object or the number zero if the aggregate is programmer-defined. Determining the exact datatype of a programmer-defined object is difficult and (fortunately) not needed by the driver.

Once DCODE has determined the datatype of the aggregate, the function DNEXT is called to create a pointer to the "next" element in the structure. DNEXT is passed two arguments: the cursor and the result computed by DCODE. DNEXT fails if there is no "next" field in the aggregate.

**Basic Elements in the Simulation**

The SNOBOL4 source for the driver (Appendix B) comprises about 400 statements that precede the user's program. The major parts are outlined below.

**Representation of Directives**

Since directives are data objects, they have their own prototype. In the simulation, directives are composed
of DNODEs, where the prototype of a DNODE is

\[ \text{DATA('DNODE(PROC, ALT, SUB, ARGA, ARGB)')} \]

Here the PROC field contains the label of the applicator for this directive. The ALT and SUB fields point to the alternate and subsequent of this directive, and are filled in by the alternation and sequentiation operators. The ARGA and ARGB fields contain argument information specific to the directive. Each directive-valued function constructs a DNODE and returns it. Natural variables, such as ENTER and MARK, are also DNODEs that are constructed when execution begins.

The SNOBOL4 patterns that have a directive interpretation are still patterns. Similarly, lists are unchanged nor are unevaluated expressions processed in any special manner. When given as an argument to a directive-construction function (or as an argument to the driver itself), such an object is used to construct a DNODE with a special applicator. It is this applicator that, when invoked by the driver, determines the datatype of the object and takes appropriate action.

The Main Algorithm

The traversal algorithm has already been described in Chapter 3. The implementation is straightforward and follows that description. However it was necessary to make some additions in case the directive is a pattern, list, or expression, and to process the "nonlinear" directives ABORT,
FENCE, MATCH, and DONE (Gimpel, 1973). The first problem is easily solved; the second was solved by having the various applicators set a variable to one of two values, 1 or 2, depending on whether or not the traversal is to terminate prematurely in success or failure. This variable is then tested each time an applicator returns success or failure, and if it is set to 1 or 2, the signal is propagated. This technique permits premature termination signals to be effective even though they may have been started several levels down in recursive calls to the driver. These are described in the next section.

Recursive Calls to the Driver

Several built-in directive functions, such as NOT, IF, and FOR, do not fit directly into the algorithm. Un-evaluated expressions share the same problem in both pattern matching and traversal. Stated briefly, there are functions that require an argument to be dynamically sequentiated into the directive at traversal time. The situation is complicated further by the fact that such a dynamic sequentiation may create alternates that are not needed until much later in the traversal.

When such a situation occurs, the appropriate applicator must take several steps. What follows is a general outline of the process and various details, specific to a given applicator, are omitted. The term "new directive" refers to the argument to be dynamically sequentiated in.
First, the new directive must be determined. This may be quite simple, such as evaluating an unevaluated expression, or it could be a complex process eventually leading to the selection of an IF argument. The only real problem here is that selection may cause side effects that involve recursive calls to the driver. The recursive nature of SNOBOL4 greatly facilitates handling of the problem.

Next, the applicator must set the needle to the root of the new directive and invoke the driver, recursively, entering it at a location that bypasses (harmful) initialization code.

Finally, upon return from the recursive call, the applicator must restore the needle to point to the DNODE that originally produced the new directive, since that is the point from which the alternate or subsequent is to be taken.

These problems are resolved by defining additional functions that save various elements prior to re-entering the driver. These functions do not actually call the driver; rather they enter it at various locations. This is another example of the advantages of simulation in SNOBOL4; the code required to do this is rather simple.

Possibilities for a SIL Implementation

If the traversal system is ever to be really useful and generally available, it must be implemented as part of
the SNOBOL4 system. This section discusses the problems involved in such an implementation.

A Brief Look at SIL

The macro implementation of SNOBOL4 is written in a machine-independent language called SIL. (SNOBOL4 Implementation Language). The source code for SNOBOL4 is a series of calls to the macros in the SIL language.

The SNOBOL4 system is implemented by writing definitions of SIL macros for a particular machine, and then assembling the definitions and SIL code using a macro assembler for that machine. Most of the macros are reasonably simple to define and all of them are documented in an implementation guide (Griswold, 1971b). The success of this method can be measured by the variety of computers that offer the macro implementation; almost every large system has a running version.

Suitability of SIL

There should be no problem implementing the system in SIL. The code is very straightforward and needs little in the way of high-level facilities. What little is needed is already present in SIL: recursion is inherent in the language, conditionals and datatype tests exist, and all the list processing primitives are present and fairly easy to use.
As an example of the relationship between statements in the SNOBOL4 simulation and SIL, consider the function \( Y \) defined in the driver. \( Y \) returns its argument unless it is of datatype EXPRESSION, in which case the argument is evaluated and the result returned. (Here the function DTC succeeds if and only if the datatype of the first argument is the second argument.)

\[
\begin{align*}
\text{DEFINE('Y(Y))} & : (Y\text{-END}) \\
Y & \text{DTC(Y,'EXPRESSION')} : (\text{F(FRETURN)}) \\
Y & = \text{EVAL(Y)} : (\text{S(RETURN)}F(\text{FRETURN})) \\
Y\text{-END}
\end{align*}
\]

This function, written in SIL, would look like:

Y\text{SIL} \quad \text{PROC} , \\
\text{RCALL} \quad \text{XPTR, ARGVAL, , FAIL} \\
\text{EQLTC} \quad \text{XPTR, E, RTXPTR} \\
\text{RCALL} \quad \text{XPTR, EXPVAL, (FAIL, RTXPTR)}

This example illustrates the simplicity of translating the SNOBOL4 source code into SIL.

\textbf{Notes on the CDC 6400 Implementation}

The implementation of the facilities described in this dissertation was carried out on a CDC 6400. There are a few differences between the implementation as described in Chapter 3 and the actual implementation. For various reasons, binary composite operators, readily available in SNOBOL4\text{x}, were used to represent directive-constructing operations. Similarly, some unary operations had to be
redefined. The following table summarizes the differences between the "publication" graphics (used herein) and the actual graphics of the implementation. Operations that are not mentioned below are unchanged. Precedences are also unchanged.

<table>
<thead>
<tr>
<th>publication</th>
<th>actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>A -&gt; B</td>
<td>A /- B</td>
</tr>
<tr>
<td>A ! B</td>
<td>A // B</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>A &amp; B</td>
<td>A /&amp; B</td>
</tr>
<tr>
<td>A  B</td>
<td>A  B</td>
</tr>
<tr>
<td>@A</td>
<td>U.EQ(A)</td>
</tr>
<tr>
<td>=A</td>
<td></td>
</tr>
</tbody>
</table>

The need for an alternative notation for unary equal sign arises because of limitations in the compiler.
APPENDIX A

AGDUMP
TRANSFER VECTOR. INDEX WITH THE DATACODE OF THE OBJECT TO FIND THE CORRECT LABEL INSIDE THE FUNCTION STRUCTURE FOR DUMPING THE OBJECT.

\[
\text{STRUCTURE.TV} = \text{ARRAY}(13, "\text{STRUCTURE.X}")
\]
\[
\text{STRUCTURE.TV}_1 = "\text{STRUCTURE.S}")
\]
\[
\text{STRUCTURE.TV}_6 = "\text{STRUCTURE.IR}"
\]
\[
\text{STRUCTURE.TV}_7 = "\text{STRUCTURE.IR}"
\]

DETERMINE THE APPEARANCE OF A GIVEN OBJECT.

\[
\text{DEFINE("STRUCTURE(OBJECT,KEY) X") ;(STRUCTURE.END)}
\]
\[
\text{STRUCTURE X} = \text{DATACODE(OBJECT)}
\]
\[
K = 1
\]
\[
\text{LE}(X,17)
\]
\[
\text{S}($\text{STRUCTURE.TV}_X$)
\]

OBJECT IS NOT REAL, INTEGER, OR STRING.

SEE IF IT HAS BEEN DUMPED BEFORE.

\[
\text{STRUCTURE.X}
\]
\[
X = \text{.HISTORY(OBJECT)}
\]
\[
\text{IDENT(OBJECT,OBJEC.)}
\]
\[
\text{S}($\text{STRUCTURE.OBJEC.}$)
\]
\[
\text{STRUCTURE.Y}
\]
\[
\text{STRUCTURE} = \text{DIFFER}(X) "OBJECT " X
\]
\[
\text{S}(\text{return})
\]
\[
\text{X} = \text{IDENT(KEY) PREFIX}
\]
\[
\text{STRUCTURE.Z}
\]
\[
K = -1
\]
\[
\text{STRUCTURE.IR}
\]
\[
\text{STRUCTURE} = \text{CONVERT(OBJECT,"STRING") ;(RETURN)}
\]
\[
\text{STRUCTURE.S}
\]
\[
\text{STRUCTURE} = """ OBJECT ""
\]
\[
\text{S}(\text{return})
\]
\[
\text{STRUCTURE.OBJEC.}
\]
\[
X = \text{IDENT(X) ' '}
\]
\[
\text{S}($\text{STRUCTURE.Z}$)
\]
\[
\text{F}($\text{STRUCTURE.Y}$)
* CONSTANTS

AGDUMP.AP = (BREAK("","", T LEN(1) ! REM . T)
AGDUMP.FAP = BREAK("","", *L<1> LEN(1) REM . *U<1>
AGDUMP.FAP = AGDUMP.FAP ! LEN(1) *L<1> FAIL
AGDUMP.FAP = AGDUMP.FAP ! RPOS(0) ABORT
AGDUMP.FAP = AGDUMP.FAP ! REM . *U<1>
AGDUMP.TV = ARRAY(13, "AGDUMP.X")
AGDUMP.COUNT = FENCE BREAK(""",") LEN(1) *INCR(N) *AGDUMP.COUNT
AGDUMP.TV<2> = "END"
AGDUMP.TV<4> = "AGDUMP.A"
AGDUMP.TV<5> = "AGDUMP.T"
AGDUMP.TV<9> = "AGDUMP.N"
AGDUMP.TV<10> = "AGDUMP.N"
AGDUMP.TV<12> = "END"
AGDUMP.TV<13> = "AGDUMP.L"
AGDUMP.STRIP = LEN(10) $ LEFT RTAB(10) REM $ RIGHT

DEFINE("AGDUMP.(PREFIX,OBJECT,LIMIT,TAG)F,L,U,C,I,D,T,K"

OPSYN OUR OWN COPY OF BASIC SNOBOL4 FUNCTIONS SO THAT THIS
WILL NOT INTERFERE WITH ANY FUTURE USES OF THE FUNCTIONS.
IN PARTICULAR, THEY MAY BE REDEFINED BY THE PROGRAM.

OPSYN("AGDUMP.CAT","",2)
OPSYN("AGDUMP.PAT","$4SCAN",2)
OPSYN("AGDUMP.REP","$4REPL")
OPSYN("AGDUMP.ADD","+",2)
OPSYN("AGDUMP.SUB","-",2)

AGDUMP.HISTORY = AGDUMP.CAT("AGGREGATE DUMP CALLED FROM LINE ",
&LASTNO)

OUTPUT = HISTORY
SAVE AND TURN OFF VARIOUS TRACE KEYWORDS SO THAT THEY DO NOT INTERFERE.

AGDUMP.ATRACE = &ATRACE ; AGDUMP.BTRACE = &BTRACE
&ATRACE = 0 ; &BTRACE = 0
FTRACE = &FTRACE
FULLSCAN = &FULLSCAN
&FULLSCAN = 1
&FTRACE = AGDUMP.FTRACE
HISTORY = TABLE(50,50)
OUTPUT = AGDUMP.(PREFIX,OBJEC.,LIMIT,*OBJECT*)
&FTRACE = FTRACE
&ATRACE = AGDUMP.ATRACE ; &BTRACE = AGDUMP.BTRACE
&FULLSCAN = FULLSCAN

AGDUMP. (LIMIT,DIFFER(LIMIT))
GT(SIZE(PREFIX),20) «S(AGDUMP.OK)
AGDUMP.REP(PREFIX,AGDUMP.STRIP,AGDUMP.CAT(AGDUMP.CAT(LEFT,
"<ETC.>"),RIGHT))
AGDUMP.OK TAG = AGDUMP.CAT(TAG,"") = "")
O = DATACODE(OBJECT)
LE(O,17) «S($AGDUMP.TV<D>)

DUMP A PROGRAMMER - DEFINED DATATYPE

OUTPUT = AGDUMP.CAT(AGDUMP.CAT(PREFIX,TAG),STRUCTURE(OBJECT))
EQ(K,1) ;S(RETURN)
EQ(LIMIT,1) ;S(RETURN)
I = 1
OUTPUT = LIMIT = AGDUMP.CAT(DIFFER(LIMIT),AGDUMP.SUB(LIMIT,1))
AGDUMP.(AGDUMP.CAT(AGDUMP.CAT(PREFIX,"")","")",OBJECT<I>,LIMIT
+ FIELD(DATATYPE(OBJECT),I) ) ;F(AGDUMP.R)
I = AGDUMP.ADD(I,1) ;F(AGDUMP.D)
* * * STRING, INTEGER, REAL, PATTERN, EXPRESSION * *

AGDUMP.X OUTPUT = AGDUMP.CAT(AGDUMP.CAT(PREFIX, TAG), STRUCTURE(OBJECT))
+ 1(RETURN)
* *

TABLE
* *

AGDUMP.T OUTPUT = AGDUMP.CAT(AGDUMP.CAT(PREFIX, TAG), STRUCTURE(OBJECT))
EQ(K,1) 1S(RETURN)
EQ(LIMIT,1) 1S(RETURN)
T = ARRAY(OBJECT) 1S(AGDUMP.TA)
OUTPUT = AGDUMP.CAT(OUTPUT," HAS NO NONNULL ENTRIES")
+ 1(RETURN)

AGDUMP.TA EQ(LIMIT,1) 1S(RETURN)
OUTPUT =
LIMIT = AGDUMP.CAT(DIFFER(LIMIT), AGDUMP.SUB(LIMIT,1))
I = 1
AGDUMP.TB F = AGDUMP.CAT(AGDUMP.CAT(PREFIX,"[") , AGDUMP.CAT(
+ STRUCTURE(T[I,1,1","])) 1F(RETURN)
AGDUMP.(F,T<I,2>,LIMIT)
I = AGDUMP.ADD(I,1) 1(AGDUMP.TB)
* *

ARRAYS
* *

AGDUMP.A N = 1
D = PROTOTYPE(OBJECT)
OUTPUT = AGDUMP.CAT(AGDUMP.CAT(PREFIX, TAG), STRUCTURE(OBJECT))
EQ(K,1) 1S(RETURN)
EQ(LIMIT,1) 1S(RETURN)
OUTPUT =
AGDUMP.PAT(D, AGDUMP.COUNT)
L = ARRAY(N)
U = COPY(L)
I = 0
AGDUMP.AA AGDUMP.REP(D, AGDUMP.AP,)

146
I = AGDUMP.ADD(I, 1)
AGDUMP.PAT(T, AGDUMP.FAP)  *S(AGDUMP.AA)
AGDUMP.AB C = COPY(L)
*
* WE WILL NOW CAREFULLY CONSTRUCT A STRING THAT
* EVALS TO THE CORRECT ARRAY ELEMENT.
*
AGDUMP.AC D = AGDUMP.CAT("\[", C[I])
I = 2
AGDUMP.AD D = AGDUMP.CAT(AGDUMP.CAT(D,"\],"),AGDUMP.CAT(C[I], INCR(I)))
+  *S(AGDUMP.AD)
D = AGDUMP.CAT(D,"\]")
I = N
F = AGDUMP.CAT(AGDUMP.CAT(PREFIX, D), "\"")
AGDUMP.(F, EVAL(AGDUMP.CAT("OBJECT", D)), LIMIT)
+  *F(AGDUMP.O)
+  C[I] = AGDUMP.CAT(LT(C[I], U[I]), AGDUMP.ADD(C[I], 1))
+  *S(AGDUMP.AC)
AGDUMP.O K = I
AGDUMP.OA C[K] = L[K]  *F(AGDUMP.OB)
K = AGDUMP.ADD(K, 1)  *F(AGDUMP.OA)
AGDUMP.OB I = AGDUMP.CAT(GT(I, 1), AGDUMP.SUB(I, 1))  *F(AGDUMP.R)
C[I] = AGDUMP.CAT(LT(C[I], U[I]), AGDUMP.ADD(C[I], 1))
+  *S(AGDUMP.AC) F(AGDUMP.OB)
*
* NAMES
*
AGDUMP.N OUTPUT = AGDUMP.CAT(AGDUMP.CAT(PREFIX, TAG), "NAME")
EQ(LIMIT, 1)  *S(RETURN)
EQ(K, 1)  *S(RETURN)
LIMIT = AGDUMP.CAT(DIFFER(LIMIT), AGDUMP.SUB(LIMIT, 1))
OUTPUT = AGDUMP.(AGDUMP.CAT(PREFIX, "1."), $OBJECT, LIMIT)
+  *(RETURN)
AGDUMP.R OUTPUT =  *(RETURN)
* LISTS

AGDUMP.L

OUTPUT = AGDUMP.CAT(AGDUMP.CAT(PREFIX, TAG), AGDUMP.CAT("LIST(",
AGDUMP.CAT(AGDUMP.SUB(SPACE(OBJECT),2),"")))
LE(LIMIT, DIFFER(LIMIT)) = S(AGDUMP.R)
OUTPUT =
I = 0
LIMIT = AGDUMP.CAT(DIFFER(LIMIT), AGDUMP.SUB(LIMIT, 1))

AGDUMP.LA

I = I + 1
AGDUMP.(AGDUMP.CAT(PREFIX, AGDUMP.CAT(AGDUMP.CAT("[D", I, "]")))
+ OBJECT[I], LIMIT) = S(AGDUMP.LA) F(AGDUMP.R)

AGDUMP.END
APPENDIX B

DRIVER
OPSIN (. STRING. PATTERN. MATCH. . . SCAN) 

* * *

STRUCTURE-MANIPULATOR 4/1/74 

& DUMP = 2 
DATA ('DNODE (PROC, ALT, SUB, ARG A, ARGB)') 
OPSIN ('?', 'ITEM', 2) 
OPSIN ('DTC', '/ _', 2) 

* * *
CURSOR ADVANCEMENT * *

DEFINE ('ADVANCE.CURSOR (OBJ)') : (ADVANCE.CURSOR.END) 
ADVANCE.CURSOR 
IDENT (OBJ) : S (RETURN) 
DTC (OBJ, "LIST") : F (ERR (1)) 
ADVANCE.CURSOR 
AGGREGATE ($CURSOR) : F (RETURN) 
* 
CURSOR = .ITEM ($CURSOR, OBJ[1]) : S (RETURN) F (RETURN) 
ADVANCE.CURSOR.END 
* 
THE PREDICATE TO SEE IF X IS AN AGGREGATE * * 

BUILT.IN.AGGREGATES = ['ARRAY', 'TABLE', 'LIST'] 
DEFINE ('AGGREGATE (X)') : (AGGREGATE.END) 
AGGREGATE GE (DATACODE (X), 100) : S (RETURN) 
IS. IT. IN (X, BUILT.IN. AGGREGATES) : S (RETURN) F (RETURN) 
AGGREGATE.END * 
* 
FORM THE ALTERNATION OF TWO DIRECTIVES * *

DEFINE ('ALTERNATE (OBJ, WITH)') : (ALTERNATE.END) 
ALTERNATE ALTERNATE = OBJ 
ALTLOOP OBJ < 2 > = IDENT (OBJ < 2 >) WITH 
OBJ = OBJ < 2 > : S (RETURN) 
ALTERNATE.END
COPY A DNODE TO NONNULL FIELDS

DEFINE("COPYNOD(P)"") (COPYNOD.END)

COPYNOD DTC(P,"DNODE") s(COPYNOD.) (RETURN)
COPYNOD = P (RETURN)
COPYNOD = DNODE(PROC(P),COPYNOD(ALT(P)),COPYNOD(SUB(P)));
+ ARGA(P),ARGB(P)) s(COPYNOD.END)

CURSOR(X)

DEFINE("CURSOR(X)"") (CURSOR.END)

CURSOR CURSOR = DNODE("M.CURSOR",,X) (RETURN)
M.CURSOR TEMP = ARGA(NEEDLE)
DTC(TEMP,"EXPRESSION") f(M.CURSOR.)
CURSOR = EVAL(TEMP) s(TRAV.S) f(TRAV.F)
M.CURSOR CURSOR = DTC(TEMP,"NAME") TEMP s(TRAV.S)
CURSOR = DTC(TEMP,"STRING") DIFFER(TEMP) TEMP s(TRAV.S) f(ERR(4))
CURSOR.END

DOBOTH(D1,D2) = D1 /* D2

DEFINE("DOBOTH(D1,D2)"") (DOBOTH.END)
OPSYN("*/","DOBOTH",2) (DOBOTH.END)
DOBOTH DOBOTH = D1 /& D2 /= ENDFOR (RETURN)
DOBOTH.END

EITHER(D1,D2) = D1 // D2 (FORWARD ALTERNATION)
DEFINE("EITHER(D1,D2)")
OPSyn("//","EITHER",2) : (EITHER.END)
EITHER = DNode("M.//",Makenode(D1),Makenode(D2)) + (RETURN)
M.//
   TEMP = X(ARGA(NEEDLE))
   XSubtrav(TEMP) : F(M.1F)
   S(Trav.S)
M.1F
   TEMP = X(ARGB(NEEDLE))
   XSubtrav(TEMP) : F(Trav.F)
   S(Trav.S) F(Trav.F)

EITHER.END
*

--------------------------------- FOR
*
*
FOR(D1,D2) = D1 & D2
*
*
DEFINE("FOR(D1,D2)")
OPSyn("//","FOR",2) : (FOR.END)
FOR = DNode("M.FOR",Makenode(D1),Makenode(D2)) + (RETURN)
M.FOR
   Push(,B)
   Push(Cursor,B)
   Push(Needle,B)
   Push(4,4)
   Push(Cursor,A)
   Push(Endfor,A)
   Tubtrav(ARGA(NEEDLE)) : S(Trav.S) F(Trav.F)
*
*
FOR-DIRECTIVE TERMINATION. POP UNUSED ALTERNATIVES
*
M.ENDFOR
   Ident(Pop(A),4) : F(M.ENDFOR)
   Temp = Pop(B)
   Ident(Temp)
   Needle = Temp
   Cursor = Pop(B)
   Push(4,4)
   Push(Cursor,A)
PUSH(ENDFOR,A)  \[S(\text{TRAV.S})F(\text{TRAV.F})\]
\[\text{FOR.END}\]

CNDX

\[\begin{align*}
\text{CNDX}(F) & \\
\text{DEFINE(''CNDX(F)'')} & \\
\text{CNDX} & = \text{DNODE(''M.CNDX'',F)} & \text{(CNDX.END)} \\
\text{M.CNDX} & = \text{Y(ARGA(NEEDLE))} & \text{(RETURN)} \\
\text{TEMP} & = \text{EVAL(ARGA(NEEDLE))} & \text{F(\text{TRAV.F})} \\
\text{IDENT(CO[TEMP])} & & \text{S(\text{TRAV.F})} \\
\text{ADVANCE.CURSOR([TEMP])} & & \text{(TRAV.S)} \\
\text{CNDX.END} & & \\
\end{align*}\]

IF

\[\begin{align*}
\text{IF(Test,Then,Else)} & \\
\text{DEFINE(''IF(Test,Then,Else)'')} & \text{(IF.END)} \\
\text{IF} & = \text{DNODE(''M.IF'',TEST,[THEN,ELSE])} & \text{(RETURN)} \\
\text{M.IF} & = \text{EVAL(ARGA(NEEDLE))} & \text{F(M.IF.F)} \\
\text{M.IF.S} & = \text{XUBTRAV(TEMP)} & \text{S(\text{TRAV.S})F(\text{TRAV.F})} \\
\text{M.IF.F} & = \text{XUBTRAV(TEMP)} & \text{F(\text{TRAV.F})S(M.IF.)} \\
\text{IF.END} & & \\
\end{align*}\]

IFnull

\[\begin{align*}
\text{IFnul(INDEX,YES,NO)} & \\
\text{DEFINE(''IFnul(INDEX,YES,NO)'')} & \text{(IFnul.END)} \\
\text{IFnul} & = \text{DNODE(''M.IFNULL'',INDEX,[T,F])} & \text{(RETURN)} \\
\text{M.IFNULL} & = \text{IDENT(CUSOR? Y(ARGA(NEEDLE))}) & \text{S(M.IF.S)F(M.IF.F)} \\
\text{IFnul.END} & & \\
\end{align*}\]
* PREDICATE: DOES X HAVE A DATATYPE FOUND IN LIST
* DEFINE('IS.IT.IN(X,LIST,I,T)') : (IS.IT.IN.END)
IS.IT.IN  X = DATATYPE(X)
IS.IT.IN  I = I + 1
IS.IT.IN-END
T = LIST<I>
IDENT(X,T)
IS.IT.IN-END
*
MAKE A DNODE
*
DEFINE('MAKENODE(X)') : (MAKENODE.END)
MAKENODE  DT(X,'DNODE')
MAKENODE = COPYNOD(X)
MAKENODE-END
MAKENODE-END
*
MARK
*
MARK(N)
*
DEFINE('MARK(N)') : (MARK.END)
MARK = DNODE("M.MARK",,N)
MARK-END
M.MARK
TEMP = MARKTABLE[CURSOR]
IDENT($TEMP)
$TEMP = +Y(ARGA(NEEDLE))
M.MARK-END
$TEMP = GT($TEMP,0) $TEMP - 1
*
NOT
*
NOT(P) - SUCCEEDS IF P FAILS AND FAILS IF P SUCCEEDS
*
DEFINE('NOT(P)') : (NOT.END)
NOT = DNODE("M.NOT",,P)
NOT-END
SERVICE FUNCTION FOR CONCATENATION

```
DEFINE("SVC.CAT(OBJ,PUT,FP,FSRCH)"
SVC.CAT OBJ[FP] = IDENT(OBJ[FP]) PUT
SVC.CAT OBJ[FP],PUT,FP,FSRCH)
SVC.CAT.S IDENT(OBJ[FSRCH])
SVC.CAT OBJ[FSRCH],PUT,FP,FSRCH)
SVC.CAT.END
```

---

```
RPT

DEFINE("RPT(P)"
RPT = DNODE("M.RPT",,MAKENODE(P))
M.RPT. M.RPT.PAT = X(ARGA(NEEDLE))
M.RPT. XUBTRAV(M.RPT.PAT)
IDENT(TRAVER)
S(M.RPT.)F(TRAVER)
RPT.END
```

---

THE TRAVERSAL DRIVER

```
OPSIN("PM","S4SCAN")
```

```
DEFINE("SUBTRAV(P) NEEDLE","ENTER.HERE.FROM.SUBTRAV")
DEFINE("XUBTRAV(P) NEEDLE")
DEFINE("TRAV(S,P) CURSOR, NEEDLE, A, TEMP, MARKTABLE, M.RPT.PAT,"
"M.LIM, F.LEV, ENTRY, P.T., SUB,, SUB,, B,, I")
DEFINE("TUBTRAV(P)","XUBTRAV")
```

```
OPSIN("S4SCAN","TRAVER")
```

---

```
155
```
XUBTRAV SUBTRAV(P)
XUBTRAV DIFFER(POP(A))

* * *

A AND B ARE UNIQUE VARIABLES FOR PUSH/POP PAIRS

* * *

TRAV

A = LEN(1)
B = LEN(1)

* VARIOUS INITIALIZATIONS
PUSHTABLE = TABLE()
P.T. = PUSHTABLE
MARKTABLE = TABLE()
F.LEV = &FNCLEVEL
M.LIM = &MATCHLIMIT
TEMP = DOT(EVAL(S))
DTC(TEMP,"STRING")
DTC(TEMP,"NAME")
SUB. = TEMP
CURSOR = .SUB.

* SUBJECT IS VALUE NOT NAME...

* SUB. = SUB. !(ENTER.HERE.FROM.SUBTRAV)

* SUBJECT BY NAME..O.K. TO CHANGE IT.

TRAV.N CURSOR = TEMP

ENTER.HERE.FROM.SUBTRAV

* PUSH(NULL,A)
P = "DTC(P,"DNODE") MAKENODE(P)
NEEDLE = P

TRAV.GOTO IDENT(ALT(NEEDLE))
PUSH(CURSOR,A)
PUSH(ALT(NEEDLE),A) !F(ERR(16))
TRAV.GOTO. M.LIM = GT(M.LIM,0) M.LIM - 1 $F(ERR(29))
IDENT(TRAV)
 CO = $CURSOR
*
* BRANCH TO THE MATCHING PROCEDURE. CONTROL
* RETURNS TO TRAV.S OR TRAV.F
*
TRAV.S IDENT(TRAV)
trav.. EQ(TRAV,2)
trav.S. NEEDLE = SUB(NEEDLE)
IDENT(NEEDLE)
*
* POP UNUSED ALTERNATES OFF THE STACK
*
TRAV.R IDENT(F.LEV,&FNCELEVEL)
trav.R. DIFFER(POP(A))
*
*
FAILURE...POP AN ALTERNATE
*
TRAV.F IDENT(TRAV)
needle = POP(A)
IDENT(NEEDLE)
CURSOR = POP(A)
$(TRAV.GOTO)

TRAV.END
*

----------------------------- SET
*
* SET(A,B)
*
DEFINE("SET(A,B)"",1)
SET SET = DNODE("M.SET",A,B)
M.SET EVAL(ARGA(NEEDLE)) = EVAL(ARGB(NEEDLE)) 

SET.END


* SLASH
* SLASH(X) = /X
*
DEFINE("SLASH(X)"")
OPSYN("/", "SLASH", 1)
SLASH SLASH = "CO(Y(X))"
SLASH END
*
DEFINE("SUBSEQUENT(OBJ,WITH)"")
SUBSEQUENT SUBSEQUENT = OBJ
SVC.CAT(OBJ,WITH,3,2)
SUBSEQUENT END
*
* TAG
* UNARY TAG \ OPERATOR
*
DEFINE("TAG(OBJ)"")
OPSYN("\", "TAG", 1)
TAG TAG = DNODE("M.TAG", OBJ)
M.TAG PUSH(CURSOR,Y(ARGA(NEEDLE))))
TAG END
*
* U.EQ
* UNARY EQUAL SIGN
*
DEFINE("U.EQ(X)"")
U.EQ U.EQ = DNODE("M. =", X)
M. = $CURSOR = Y(ARGA(NEEDLE))
*
**CHECK FOR CHANGED SUBJECT**

```plaintext
IDENT(SUB., SUB..) U.EQ.END OPSYN("=","U.EQ",1)
```

**WORD**

```plaintext
LET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
WORD.PAT = BREAK(LET) SPAN(LET). WORD
DEFINE("WORD()")
```

```plaintext
WORD WORDS WORD.PAT = WORDS = INPUT
WORD.END
```

**X**

**THE CURIOUS FUNCTION** **X**. **EVALUATES ITS**
**ARGUMENT REPEATEDLY UNTIL IT IS NOT AN**
**EXPRESSION.**

```plaintext
DEFINE("X(X)"
```

```plaintext
X = DTC(X,"EXPRESSION") X = EVAL(X)
X.END
```

**ALTERNATION**

```plaintext
DEFINE("X.ALT(A,B)"
```

```plaintext
X.ALT X.ALT = MAKENODE(A)
```

```plaintext
ALTERNATE(X.ALT,MAKENODE(B))
```

```plaintext
X.ALT.END OPSYN("/\","X.ALT",2)
```

**CONCATENATION**
DEFINE("X.CAT(A,B)"")

X.CAT

X.CAT.DO

X.CAT = MAKENODE(A)

SUBSEQUENT(X.CAT,MAKENODE(B))

X.CAT.END

OPSYN("/"","X.CAT",2)

* * *

THE CURIOUSER FUNCTION Y. EVALUATES ITS ARGUMENT ONCE.

* *

DEFINE("Y(Y)"")

Y

OTC(Y,"EXPRESSION")

Y = EVAL(Y)

Y.END

* * *

BEGIN PRIMITIVES

*

CLEARMARK = DNODE("M.CLEAR")

DONE = MATCH

ENDFOR = DNODE("M.ENDFOR")

IFNULL = DNODE("M.IFNUL")

IFNONN = DNODE("M.IFNONN")

MARK = MARK(1)

NEXT = DNODE("M.NEXT")

POINTER = DNODE("M.POINTER")

XBALQQ = COPY(&BAL)

ENTER = POINTER /- \XBALQQ /- DNODE("M.ENTER")

EXIT = CURSOR(*$POP(XBALQQ)) // *ERR(18)

SIMPLE = DNODE("M.SIMPLE")

*

TRANSFER VECTOR FOR PRIMITIVE PATTERNS

*

TRAV.TV = TABLE(5,70000)

TRAV.TV[ABORT] = "M.ABORT"

TRAV.TV[FAIL] = "TRAV.F"

TRAV.TV[MATCH] = "M.MATCH"

TRAV.TV[FENCE] = "M.FENCE"
TRAV.TV[SUCCEED] = "M.SUCCEED"
CLOSE(TRAV.TV)
SUCCEED. = MAKE NODE (SUCCEED) *(M.PRIM.END)

* MATCHING FOR PRIMITIVES

M.ABORT TRAV = 1 *(TRAV.R)
M.POINTER AGGREGATE($CURSOR) *(S(TRAV.S)F(TRAV.F)
M.CLEAR MARK TABLE = TABLE () *(TRAV.S)

M.ENTER TEMP = DATATYPE($CURSOR)
IDENT(TEMP,"ARRAY") *(S(M.ENTER.ARRAY)
IDENT(TEMP,"TABLE") *(S(M.ENTER.TABLE)
ADVANCE.CURSOR[1]) *(TRAV.S)

M.ENTER.ARRAY
TEMP = DESCR(9,"A",LOCATION($CURSOR) + 1)
CURSOR = DESCR(9,"A",LOCATION(TEMP) ++ $TEMP ++ 1) *(TRAV.S)

M.ENTER.TABLE
CURSOR = DESCR(9,"A",LOCATION($CURSOR)) *(TRAV.S

M.NEXT DTC(CURSOR,.NAME) *(F(TRAV.F)
CURSOR = DNEXT(CURSOR,DCODE(CURSOR)) *(S(TRAV.S)F(TRAV.F)

M.FENCE PUSH(,A)
PUSH(DNODE("M.ABORT",1),A) *(S(TRAV.S)F(ERR(16))

M.IFNUL IDENT($CURSOR) *(S(TRAV.S)F(TRAV.F)
M.IFNONN DIFFER($CURSOR)

*          \( S(\text{TRAV}.S)F(\text{TRAV}.F) \)
M.SIMPLE AGGREGATE($CURSOR)

*          \( S(\text{TRAV}.F)F(\text{TRAV}.S) \)
M.SUCCEED POP(A) \( S(\text{M}.\text{SUCCEED}) \)
PUSH($CURSOR,A)
PUSH($\text{SUCCEED},A)

*          \( (\text{TRAV}.S) \)
M.MATCH TRAV = 2

*          \( (\text{TRAV}.R) \)

* MATCH AN OBJECT

M.OBJECTX

-------------------------------------------OBJECT

M.OBJECT TEMP = X(ARGA(NEEDLE))

*          \( S(\text{M}.\text{OBJECTS}) \)
M.OBJECTS PUSHTABLE = P.T.
PUSHTABLE = P.T.
 TEMP = TRAV.TV[TEMP]
 DTC(TEMP, "DNODE")
 SUBTRAV(TEMP)
M.OBJECT ADVANCE.CURSOR(TEMP)

*          \( S(\text{TRAV}.S)F(\text{TRAV}.F) \)
M.PRIM.END

OUTPUT(OUTPUT, 4, "(1X135A1)")

--------------------------------------- ENO PRIMITIVES ------------------------------------------
REFERENCES


