GRISWOLD

String and List Processing in SNOBOL4
Techniques and Applications

PRENTICE
HALL

covers techniques and
applications with
emphasis on string
and list processing

PRENTICE-HALL
SERIES IN
AUTOMATIC
COMPUTATION

RALPH E. GRISWOLD

String and List Processing in SNOBOL4

Techniques and Applications

The two aspects of this book, techniques and applications, are frequently integrated throughout the text. The technique material is designed to include portions of string and list processing that underlie applications and to emphasize programming methods in SNOBOL4.

The topics included in this book may display a particular aspect of string or list processing in context providing motivation that would otherwise be lacking or, may show what can be accomplished if string and list processing techniques are applied to an area in which manual techniques are conventional.

Programming examples are emphasized and some of the programs may be applied as they stand. Most of the programs are comprised of the bare essentials in order to be embellished, extended, and filled out with the amenities of good programs. Detecting and handling errors is experienced by the reader through explicit exercises or by implication.

". . . The exercises, provided throughout the book, are an important component of the material presented. The exercises come in all varieties from trivial drills to research projects. Many exercises suggest extensions necessary to complete a program introduced in the text. Other exercises

(continued on back flap)

STRING AND LIST PROCESSING IN SNOBOL4: Techniques and Applications

Prentice-Hall
Series in Automatic Computation

AHO, editor, *Currents in the Theory of Computing*
AHO AND ULLMAN, *The Theory of Parsing, Translation, and Compiling, Volume I: Parsing; Volume II: Compiling*
ANDREE, *Computer Programming: Techniques, Analysis, and Mathematics*
ANSELONE, *Collectively Compact Operator Approximation Theory and Applications to Integral Equations*
ARBIB, *Theories of Abstract Automata*
BATES AND DOUGLAS, *Programming Language/One*, 2nd ed.
BLUMENTHAL, *Management Information Systems*
BRENT, *Algorithms for Minimization without Derivatives*
BRINCH HANSEN, *Operating System Principles*
COFFMAN AND DENNING, *Operating Systems Theory*
CRESS, et al., *FORTRAN IV with WATFOR and WATFIV*
DAHLQUIST, BJÖRCK, & ANDERSON, *Numerical Methods*
DANIEL, *The Approximate Minimization of Functionals*
DEO, *Graph Theory with Applications to Engineering and Computer Science*
DESMONDE, *Computers and Their Uses*, 2nd ed.
DESMONDE, *Real-Time Data Processing Systems*
DRUMMOND, *Evaluation and Measurement Techniques for Digital Computer Systems*
EVANS, et al., *Simulation Using Digital Computers*
FIKE, *Computer Evaluation of Mathematical Functions*
FIKE, *PL/1 for Scientific Programmers*
FORSYTHE AND MOLER, *Computer Solution of Linear Algebraic Systems*
GAUTHIER AND PONTO, *Designing Systems Programs*
GEAR, *Numerical Initial Value Problems in Ordinary Differential Equations*
GOLDEN, *FORTRAN IV Programming and Computing*
GOLDEN AND LEICHUS, *IBM/360 Programming and Computing*
GORDON, *System Simulation*
GRISWOLD, *String and List Processing in SNOBOL4: Techniques and Applications*
HANSEN, *A Table of Series and Products*
HARTMANIS AND STEARNS, *Algebraic Structure Theory of Sequential Machines*
HULL, *Introduction to Computing*
JACOBY, et al., *Iterative Methods for Nonlinear Optimization Problems*
JOHNSON, *System Structure in Data, Programs, and Computers*
KANTER, *The Computer and the Executive*
KIVIAT, et al., *The SIMSCRIPT II Programming Language*
LAWSON AND HANSON, *Solving Least Squares Problems*
LORIN, *Parallelism in Hardware and Software: Real and Apparent Concurrency*
LOUDEN AND LEDIN, *Programming the IBM 1130*, 2nd ed.
MARTIN, *Design of Man—Computer Dialogues*
MARTIN, *Design of Real-Time Computer Systems*
MARTIN, *Future Developments in Telecommunications*
MARTIN, *Programming Real-Time Computing Systems*
MARTIN, *Security Accuracy and Privacy in Computer Systems*
MARTIN, *Systems Analysis for Data Transmission*

MARTIN, *Telecommunications and the Computer*
MARTIN, *Teleprocessing Network Organization*
MARTIN AND NORMAN, *The Computerized Society*
MATHISON AND WALKER, *Computers and Telecommunications: Issues in Public Policy*
MCKEEMAN, et al., *A Compiler Generator*
MEYERS, *Time-Sharing Computation in the Social Sciences*
MINSKY, *Computation: Finite and Infinite Machines*
NIEVERGELT, et al., *Computer Approaches to Mathematical Problems*
PLANE AND MCMILLAN, *Discrete Optimization: Integer Programming and Network Analysis for Management Decisions*
PRITSKER AND KIVIAT, *Simulation with GASP II: a FORTRAN-Based Simulation Language*
PYLYSHYN, editor, *Perspectives on the Computer Revolution*
RICH, *International Sorting Methods Illustrated with PL/1 Programs*
RUSTIN, editor, *Algorithm Specification*
RUSTIN, editor, *Computer Networks*
RUSTIN, editor, *Data Base Systems*
RUSTIN, editor, *Debugging Techniques in Large Systems*
RUSTIN, editor, *Design and Optimization of Compilers*
RUSTIN, editor, *Formal Semantics of Programming Languages*
SACKMAN AND CITRENBAUM, editors, *On-Line Planning: Towards Creative Problem-Solving*
SALTON, editor, *The SMART Retrieval System: Experiments in Automatic Document Processing*
SAMMET, *Programming Languages: History and Fundamentals*
SCHAEFER, *A Mathematical Theory of Global Program Optimization*
SCHULTZ, *Spline Analysis*
SCHWARZ, et al., *Numerical Analysis of Symmetric Matrices*
SHAW, *The Logical Design of Operating Systems*
SHERMAN, *Techniques in Computer Programming*
SIMON AND SIKLOSSY, editors, *Representation and Meaning: Experiments with Information Processing Systems*
STERBENZ, *Floating-Point Computation*
STERLING AND POLLACK, *Introduction to Statistical Data Processing*
STOUTEMYER, *PL/1 Programming for Engineering and Science*
STRANG AND FIX, *An Analysis of the Element Method*
STROUD, *Approximate Calculation of Multiple Integrals*
TAVISS, editor, *The Computer Impact*
TRAUB, *Iterative Methods for the Solution of Equations*
UHR, *Pattern Recognition, Learning, and Thought: Computer-Programmed Models of Higher Mental Processes*
VAN TASSEL, *Computer Security Management*
VARGA, *Matrix Iterative Analysis*
WAITE, *Implementing Software for Non-Numeric Application*
WILKINSON, *Rounding Errors in Algebraic Processes*
WIRTH, *Systematic Programming: An Introduction*

STRING AND LIST PROCESSING IN SNOBOL4: Techniques and Applications

RALPH E. GRISWOLD

Department of Computer Science
The University of Arizona

PRENTICE-HALL, INC.

ENGLEWOOD CLIFFS, NEW JERSEY

Library of Congress Cataloging in Publication Data

GRISWOLD, RALPH E.
String and list processing in SNOBOL4.

(Prentice-Hall series in automatic computation)
Includes bibliographical references.

1. SNOBOL (Computer program language) 2. List processing (Electronic computers) I. Title.
QA76.73.S6G77 001.6′424 74-9661
ISBN 0-13-853010-6

© 1975 by Prentice-Hall, Inc., Englewood Cliffs, N.J.

All rights reserved. No part of this book may be reproduced in any form or by any means without permission in writing from the publisher.

10 9 8 7 6 5 4 3 2 1

Printed in the United States of America

PRENTICE-HALL INTERNATIONAL, INC., *London*
PRENTICE-HALL OF AUSTRALIA, PTY. LTD., *Sydney*
PRENTICE-HALL OF CANADA, LTD., *Toronto*
PRENTICE-HALL OF INDIA PRIVATE LIMITED, *New Delhi*
PRENTICE-HALL OF JAPAN, INC., *Tokyo*

CONTENTS

PREFACE x

1 PATTERN MATCHING 1

1.1. A Few Examples 1
  1.1.1. Removing Items from a List 1
  1.1.2. Matching Words 2
  1.1.3. Matching Numbers 4
  1.1.4. Search and Replacement Specifications 5
1.2. Grammars and Patterns 7
1.3. Patterns as Procedures 12
1.4. Structuring Data to Utilize Pattern Matching 15

2 DEFINED FUNCTIONS 20

2.1. Defined Functions as a Mechanism for Language Extension 20
  2.1.1. Basic Extensions 21
  2.1.2. More Elaborate Extensions 26
2.2. Recursive Functions 30
2.3. Generators and Successors 34
  2.3.1. Random Number Generation 35
  2.3.2. Generation of Strings 36
2.4. Use of Functions in Program Design 40

3 STRUCTURES 47

3.1. Values and Objects in SNOBOL4 47
  3.1.1. Built-In Data Types 48
  3.1.2. Defined Data Types 55
3.2. Implementation of Some Specific Structures 60
  3.2.1. Stacks 60
  3.2.2. Queues 69
  3.2.3. Linked Lists 73
  3.2.4. Binary Trees 78
  3.2.5. Trees 86
3.3. Processing Structures that Contain Loops 91

4 APPLICATIONS IN MATHEMATICS 95

4.1. Representation and Manipulation of Mathematical Objects 95
  4.1.1. Rational Numbers 95
  4.1.2. Large Integers 98
  4.1.3. Polynomials 103
4.2. Operations on Expressions 107
  4.2.1. Expressions 108
  4.2.2. Differentiation 111
  4.2.3. Tree Representation of Expressions 115
  4.2.4. Alternative Representations 117

5 CRYPTOGRAPHY 122

5.1. Ciphers 124
  5.1.1. Types of Ciphers 124
  5.1.2. Alphabets 125
5.2. Monoliteral Substitution Ciphers 126
  5.2.1. Basic Substitution Methods 126
  5.2.2. Keyed Alphabets 128
  5.2.3. Polyalphabetic Substitutions 129
5.3. Transposition Ciphers 132
  5.3.1. Route Transposition 132
  5.3.2. Columnar Transposition 134
  5.3.3. Inscription and Transcription Paths 135
  5.3.4. Programming Methods for Transportation 136
  5.3.5. Grilles and the Use of Nulls in Enciphering 138
  5.3.6. Related Uses of REPLACE 142
5.4. More Complicated Substitutions 147
  5.4.1. Polyliteral Substitution 147
  5.4.2. Polygraphic Substitution 150
5.5. Cryptanalysis 152
  5.5.1. Statistical Aspects of Language 153
  5.5.2. Regularities Inherent in Enciphering Techniques 154
  5.5.3. Application of Enciphering Techniques to Decrypting 156

6 DOCUMENT PREPARATION 160

6.1. Representation of Text 161
6.1.1. Character Sets 161
6.1.2. Processing Input Text 163
6.1.3. Other Problems in Representing Text 167
6.2 Formatting 168
6.2.1. Vertical Formatting 169
6.2.2. Horizontal Formatting 174
6.2.3. Combining Vertical and Horizontal Formatting 178
6.3. Other Aspects of Document Preparation 184
6.3.1. Indexing 184
6.3.2. Abbreviations 185
6.3.3. Automatic Numbering 188
6.3.4. Modifying the Formatting Program During Execution 189

7 ADDITIONAL APPLICATIONS 192

7.1. A Random Sentence Generator 192
7.1.1. The Generation Process 193
7.1.2. Representation of the Grammar 194
7.1.3. The Generation Functions 197
7.2. Turing Machines 200
7.2.1. Simulation of Turing Machines 202
7.3. A Macro Processor 209
7.3.1. Description of MP 210
7.3.2. The Implementation of MP 213
7.3.3. A Conditional Facility in MP 217
7.4. A Context Editor 220
7.4.1. A Description of ED4 221
7.4.2. The Implementation of ED4 222

APPENDIX

A CHARACTER SETS 226

APPENDIX

B SOLUTIONS TO SELECTED EXERCISES 231

REFERENCES 275

INDEX OF DEFINED FUNCTIONS 279

SUBJECT INDEX 282

PREFACE

"String and list processing" in computer programming is a topic with which some mystery has been associated. Like many somewhat mysterious subjects, the quality of mystery is derived from two sources: a lack of understanding and the fact that the subject is not well defined.

In the most general sense, string and list processing is defined negatively as "nonnumerical computation"; in other words, anything not involved with numbers. This definition is too general and at the same time too restrictive. Nonnumerical computation includes much business and administrative data processing, information retrieval, and so forth. While these areas have components of string and list processing, they include many other aspects of computation. Conversely, string and list processing is used in certain kinds of numerical manipulations, some of which are illustrated in this book.

String and list processing is a catch-all for types of computation that are not conventional or "traditional". In a sense, string and list processing is an awkward marriage of terms. String processing pertains to character and text manipulation, while list processing is concerned with structures and the relationships among aggregates of objects. Despite the apparent dissimilarity of the two, many problem areas require both kinds of facilities.

If string and list processing is not a generally well-defined subject, it is even more true that string and list processing techniques in programming are not well understood. To a large extent, string and list processing involves the use of a hodge-podge of unsystematic, ad hoc devices. Many methods that are used are clumsy and contorted to a degree that masks their defects. In an area without much systematic literature, "reinvention of the wheel" is commonplace. Unfortunately, the flat tire is frequently reinvented also. It is not my ambition to rectify this situation; that is far too great a task.

This book does strive to display some techniques in string and list processing, and to explore some areas of application. From this, the reader will, hopefully, be able to pull some things together and see common threads that unite the subject.

A significant reason for grouping string and list processing together is the nature of the programming tools available. There are many programming languages [1][1]. Some are designed primarily for "scientific computation", i.e., numerical calculation. Others are oriented toward business and administrative data processing. Some languages are "special purpose", and are intended for specific, restricted applications. Other languages are "general purpose", combining many features. Few programming languages, however, have a strong emphasis on string and list processing facilities. SNOBOL4 [2, 3] is a notable exception among programming languages that have achieved wide acceptance.

SNOBOL4 is usually described as a string-processing language, partly because its predecessors (SNOBOL [4] and SNOBOL3 [5, 6]) were true string-processing languages, and partly for the valid reason that SNOBOL4 does have extensive and powerful facilities for manipulating strings of characters. SNOBOL4 is, however, a general-purpose language (in the sense indicated above) that stresses "nonnumerical" facilities. The list-processing facilities in SNOBOL4 are not as well known as the string-processing facilities. This is due in part to the historical development of the SNOBOL languages, which only recently included list-processing facilities, and in part to the fact that the list-processing facilities are less explicit than those for string processing. The particular context for this book is the SNOBOL4 programming language. This language is an integral part of the material presented. In a sense, this book is about programming in SNOBOL4. In all fairness, had another language been chosen, the book would have been quite different. It is, nonetheless, simply a fact that the nature of the tools, to a large extent, defines the work.

This book presumes a good working knowledge of SNOBOL4 and the willingness of the reader to review or to refer to the language description as necessary. There are, however, sections that discuss pertinent aspects of SNOBOL4 or provide alternative ways of viewing certain language features. The reader who is not already familiar with SNOBOL4 is strongly advised to first acquire the prerequisite background and enough programming experience so that simple matters follow naturally without conscious thought. Even for the experienced programmer, some of the material in this book will, perhaps, be novel and require thought and study.

The book has two parts: one on techniques and one on applications. The division between the two is not particularly clear cut. Some of both aspects appear everywhere in the book. The material on techniques is designed to cover portions of string and list processing that underlie applications and to give special attention to programming methods in SNOBOL4. The choice

[1] Numbers in brackets refer to references listed at the end of the book.

of application areas presented a number of problems. The most serious problem was the limitation on the amount of material that can reasonably be presented in a single book. The topics that were finally chosen for inclusion were selected for various reasons. In some cases, a topic displays a particular aspect of string or list processing in context and provides motivation that otherwise would be lacking. In other cases, a topic shows what can be accomplished if string and list processing techniques are applied to an area in which manual techniques are conventional. The coverage of application areas is by no means comprehensive. Some areas which use string and list processing heavily are simply omitted. These include artificial intelligence, music theory, compiler writing, and computer-assisted instruction. Other subjects, such as linguistic analysis, are not presented as unified topics, but, instead, are distributed throughout the book. Some readers may be disappointed that their particular area of interest is not included and that the book does not provide ready-made programs for their use. I can only apologize and encourage them to apply the concepts and techniques that are presented here to their work.

The emphasis in this book is on programming examples. Some of the programs may be useful to some readers as they stand. One of the major problems in accommodating the material to the confines of a book was providing significant programming examples in a relatively small amount of space. The approach to this problem has been to strip programs down to their bare bones. Most of the programs in this book are skeletons, albeit working skeletons. They are designed so that they can be embellished, extended, and filled out with the amenities that good programs should have. Comments in the programs themselves have been sacrificed for more compact (and hopefully more literary) description in the body of the text. Generally speaking, detection and handling of errors has been left to the reader, either in the form of explicit exercises or by implication. Quite frequently, only one of a number of similar components of a program is actually given in the text.

Programming inevitably involves questions of style. The programs in this book reflect my style, modified somewhat by pedagogical considerations and the constraints of publication format. Readers who wish to use the programs or elaborate on them will no doubt wish to modify the programs to suit their own styles. I make no claim that my style is the best or even suitable for use by others; it simply suits me.

All the programs in this book are real. They have been run and tested. Nevertheless, it is inevitable that they contain errors, logical, clerical, and otherwise. I will appreciate errors being called to my attention so that they can be corrected in subsequent printings. It is also inevitable that my programs will not always be the "best", even by my own standards. Suggestions for better programs are welcome.

The programs in this book all conform to Version 3 of SNOBOL4 [2]. There are a number of dialects of SNOBOL4 [7-9], some of which contain features not available in the standard version. Users of such systems may have to make minor modifications for their particular versions, and may find that program solutions can be improved by using some of the available language extensions. Most of the programs in this book are independent of any particular computer. Implementations of Version 3 of SNOBOL4 are sufficiently standardized so that computer differences usually do not cause problems. The few references to input and output are given in the form that is used for the IBM 360/370 operating under OS. See Reference 2. The programs were actually developed and tested on a CDC 6400. The standard SNOBOL4 publication graphics [2] are used in the programs, but the ASCII character set is used in data in some places. Character-set considerations are discussed in Section 6.1 and there are tables in Appendix A.

With a few exceptions, discussion of interactive computing is omitted. Many installations offer SNOBOL4 in a time-sharing environment on an interactive basis. The nature of such facilities varies considerably, however. Many of the programs given in this book can be modified to operate interactively. Users with time-sharing facilities may find it interesting to adapt the programs to their particular systems.

The exercises, provided throughout the book, are an important component of the material presented. The exercises come in all varieties from trivial drills to research projects. Many exercises suggest the embellishments and extensions necessary to complete a program introduced in the text. Other exercises suggest next steps to be taken toward more significant material, or indicate related areas not covered in the text. Solutions to many of the exercises are given in Appendix B. These solutions, and accompanying discussions, are intended to supplement the text. Some further exercises are suggested in the solutions.

Throughout this book there are citations by number (as illustrated in this preface) to references that are listed after the appendices. The list of references, although by no means comprehensive, includes material for supplementary reading in addition to specific citations.

I am grateful to a number of people who have contributed to this book. Jim Gimpel, who has made a number of important and original contributions to string and list processing, introduced me to the types of transformations that can be performed by character replacement. The material in Section 5.3 depends heavily on these ideas. The random sentence generator presented in Section 7.1 is patterned after a similar program written by Jim. The basic idea underlying the context editor developed in Section 7.4 is also due to Jim. John Hallyburton and Fred Druseikis provided critical readings of the manuscript, suggested a number of improvements, and provided some of the exercises. John Hallyburton also suggested the method used for diagramming defined data objects. The students in my classes on string and list processing served as guinea pigs, and were subjected to a variety of experiments in methods of presenting the material and techniques for formulating program solutions. Their many corrections and specific suggestions are distributed throughout the text. They have, in the process of course work, verified most of the exercises. I am indebted to the staff of the University of Arizona Computer Center for providing excellent service, often under difficult circumstances, which made it possible to develop and test the programs given in this book. My secretary, LeeAnn Underwood, was most helpful in assisting with the preparation of the manuscript. Finally, and most importantly, I am enormously indebted to my wife, Madge. She assisted in all aspects of the book and devoted many hours of her time over a period of years to the material that appears here. She provided advice on content, critical readings of the manuscript, copyediting, and keyboarding; most of all she provided constant encouragement, support, and understanding.

RALPH E. GRISWOLD

Tucson, Arizona

STRING AND LIST PROCESSING IN SNOBOL4: Techniques and Applications

1 PATTERN MATCHING

Pattern matching is the most powerful and most extensive major feature of the SNOBOL4 language. There are ten built-in pattern-valued functions, six operators that relate to patterns, seven built-in patterns, nine keywords relating to patterns and pattern matching, and two of the three statement types are devoted to pattern matching. Pattern matching presents more difficulties to the SNOBOL4 programmer than any other aspect of the language; it is poorly understood, often abused, and its potential is rarely realized. For some programmers, pattern matching is an obsession. Operations that could be performed in a straightforward way are formulated in an obtuse fashion using patterns. The purpose of this chapter is to point out some of the ways of using pattern matching and to clarify some of its more obscure aspects.

1.1. A FEW EXAMPLES

One of the problems with pattern matching is the large number of facilities that are available. It is often difficult to decide what approach to take. A few examples of typical pattern-matching problems and their solutions serve to illustrate some useful techniques.

1.1.1. Removing Items from a List

Consider a list of items separated by commas:

```
        LIST   =   'LEFT,HALT,LEFT,CANCEL,RIGHT,'
```

Removing an item is trivially simple:

```
        NEXTI  =   BREAK(',') . ITEM LEN(1)
                  .
                  .
                  .
GETI    LIST  NEXTI   =                         :F(NONE)
                  .
                  .
                  .
```

In the pattern above, LEN(1) could just as easily be replaced by a literal comma. Suppose, however, that the last item of the list is not followed by a comma. One alternative is to handle this as a special case in another statement. If the pattern NEXTI is modified appropriately, this is unnecessary. A first attempt might be

```
        NEXTI  =   BREAK(',') . ITEM LEN(1) | REM . ITEM
```

This pattern works as intended until the last item has been removed. Since REM can match the null string, this pattern never fails. The problem can be circumvented as follows:

```
      NEXTI  =  BREAK(',') . ITEM LEN(1) | (LEN(1) REM) . ITEM
```

This pattern works properly whether or not the last item is followed by a comma.

The purpose of the problem above is to illustrate that alternative situations which arise in pattern matching can usually be handled by including alternatives in a pattern, rather than by providing alternative statements. This problem also illustrates a situation in which no single built-in pattern provides the necessary facilities. Stated another way, there is no single pattern in SNOBOL4 that matches the remainder of a string provided it is nonnull. The use of LEN(1) in conjunction with REM amounts to constructing such a pattern out of simpler components.

1.1.2. Matching Words

The term "word", in the written sense, while generally understood, is not particularly well defined in a syntactic sense. Superficially, a word might be described as a contiguous sequence of letters. Of course there are many counter examples to this definition. Words exist in the context of written material, as components of sentences. Thus, words might be described as sequences of letters between blanks or punctuation marks. Consider, therefore, the problem of getting the next word from a string of text, which is analogous to getting the next item from a list. An example of such a string of text is

```
       TEXT   =   'EVEN IF HE SAW ME, I WILL DENY IT.'
```

This string is similar to the more highly structured list given earlier: a sequence of significant substrings with separating characters. A first approach might be to use a pattern such as:

```
       LETTERS   =   'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
       NEXTW     =   SPAN(LETTERS) . WORD
```

Then a statement such as

```
GETW   TEXT   NEXTW   =                    :F(NONE)
```

could be used to get the words, in order, from TEXT. This method works in the sense that it produces the desired result. However, examination of successive values of TEXT illustrates a disadvantage of this approach:

```
EVEN IF HE SAW ME, I WILL DENY IT.
 IF HE SAW ME, I WILL DENY IT.
  HE SAW ME, I WILL DENY IT.
   SAW ME, I WILL DENY IT.
    ME, I WILL DENY IT.
    , I WILL DENY IT.
    ,  WILL DENY IT.
    ,   DENY IT.
    ,    IT.
    ,    .
```

In the first place, the blanks and punctuation marks are not removed. Consequently, they must be rescanned with every match—an obvious inefficiency. In the second place, NEXTW only works properly in the unanchored mode, which may be in conflict with other parts of the program. To avoid these problems, a second attempt might be

```
      NEXTW  =  SPAN(LETTERS) . WORD BREAK(LETTERS)
```

This pattern eliminates the characters between words. It is erroneous, however, because it does not match the last word, since BREAK(LETTERS) fails in this case. This can be avoided by providing a second alternative for the last word, but there is a better solution:

```
      NEXTW  =  BREAK(LETTERS) SPAN(LETTERS) . WORD
```

Since BREAK can match the null string, this pattern handles the first word of a sentence as well as those preceded by blanks and punctuation marks. The pattern also works properly in either the anchored or unanchored mode.

The method of arriving at this pattern, rather than one of the two previous ones (or some other), might be the result of trial and error. More frequently, a pattern that does not quite work is "patched up", adding alternatives, until it is much more complicated than it need be. One method of evaluating such a pattern is to determine whether or not the way it operates during pattern matching corresponds in a procedural way to what is desired. For NEXTW, this operation can be described loosely as "Search for a letter, consuming any other characters. If there is no letter, fail. If a letter is found, match all subsequent letters, and assign them to WORD." This description illustrates that pattern matching starts at the beginning of the string, and that any irrelevant separating characters before a word are included in the match and hence are discarded as a result of replacement by the null string.

It is important to realize that the definition of word as used in this example is a naive one. There are compound words, possessive forms, abbreviations, acronyms, and all sorts of special notations that appear in written language. The specific problem dictates what constitutes an appropriate definition of a word [10].

1.1.3. Matching Numbers

In SNOBOL4, both integers and real numbers can be specified literally in a source program. Consider the problem of creating a pattern that matches such numbers, ignoring limitations on magnitudes, which vary from implementation to implementation. Integers consist of a sequence of consecutive digits. Real numbers contain a period, but that period cannot be the first character of a real number in SNOBOL4.

It is not difficult to devise a pattern to match a number, taking into account all the possibilities and writing alternatives for the various situations:

```
      DIGITS   =   SPAN('0123456789')
      NUMBER   =   DIGITS '.' (DIGITS | NULL) | DIGITS
```

The first alternative must appear before the second. Otherwise an integer substring of a real number would be matched instead of a real number. An alternative formulation is

```
      NUMBER   =   DIGITS ('.' (DIGITS | NULL) | NULL)
```

which essentially factors out the initial string of digits. These patterns are satisfactory for the specific definition of numbers given above. Suppose, however, that the definition is extended to include signs, exponent representations of real numbers, and so forth. The pattern may have to be rewritten in its entirety. If the definition is complicated, such patterns become unmanageably difficult to understand or modify.

In such cases, it is good practice to separate the definition into logical components and build the final pattern in a series of steps. (The pattern DIGITS above serves this purpose to some extent, but primarily saves repetition.) Basically, a number is an integer or a real number. This basic division can be reflected in the patterns:

```
     INTEGER  =  DIGITS
     REAL  =  DIGITS '.' (DIGITS | NULL)
     NUMBER  =  REAL | INTEGER
```

So far, these patterns offer little advantage over the previous ones except that NUMBER is somewhat more "self documenting" than before. Suppose that an optional sign is added to the definition. The patterns might be changed as follows:

```
     SIGN  =  ANY('+-') | NULL
     NUMBER  =  SIGN (REAL | INTEGER)
```

Similarly, if the exponent form of real numbers is allowed, as in FORTRAN, REAL might be defined as follows:

```
     DFORM  =  DIGITS '.' (DIGITS | NULL)
     EFORM  =  DFORM 'E' SIGN DIGITS
     REAL  =  EFORM | DFORM
```

1.1.4. Search and Replacement Specifications

File editors usually provide a means of locating a desired place in a file by context, i.e., in terms of the data contained in the file. One method is to allow the user of the editor to specify a string that initiates a search. Presumably, a file may contain any combination of characters. Consequently, the way in which a search string can be specified is a problem. One technique is to preempt a character to delimit the search string. (The SNOBOL4 quoted literal is of this nature.) This method precludes the use of the delimiting character in the search string itself. A way of overcoming this problem is to permit the delimiter to be specified when the search request is made. A common form for such a specification is to interpret the first character of the search request as the delimiter [11]. Thus, in the search request

```
/PROGRAMMING/
```

the delimiter is / and the string to be searched for is PROGRAMMING. On the other hand

```
$A*B/C$
```

uses $ as a delimiter to specify a search for A*B/C. The facility of this mechanism is based on the fact that in almost all cases there is some character not in the desired string that can therefore be selected as a delimiter. Note that any character, even a blank, can serve as a delimiter.

An extension of this technique allows for a search that is followed by a replacement. The replacement string is simply appended to the search specification and followed by another instance of the delimiter. An example is

```
/PROGRAMMING/CODING/
```

which specifies a search for PROGRAMMING and a replacement by CODING when it is found.

Consider the problem of analyzing specifications of this type. Since the delimiter is determined from the specification itself, one method is to pick off the first character, and having determined what it is, analyze the rest of the specification. Immediate value assignment, coupled with unevaluated expressions, permit this to be done in a single pattern match:

```
        SPECPAT   =   LEN(1) $ D BREAK(*D) . SEARCH LEN(1)
+                     BREAK(*D) . REPLACE
```

When applied to a specification, the first character is (immediately) assigned to D, which is then used as an argument for BREAK to isolate the two desired strings.

This is an example of context dependence in pattern matching. A value determined dynamically during pattern matching is used as a parameter of the pattern itself. More is said about context dependence, immediate value assignment, and unevaluated expressions later in this chapter. This example provides a simple instance of an extremely powerful facility.

EXERCISES

1.1 Modify NEXTW to include compound words.

1.2 A common approach to the problem of matching words is to define words by exclusion rather than by inclusion, that is, to look for any characters that may occur between separators. Discuss this approach and contrast it with the method used above.

1.3 In writing and printing, it is conventional to punctuate integers by placing commas between groups of three digits. An example is 2,173,406,315. Write a pattern to match integers in this form.

1.4 Write a pattern that matches even numbers. Include both signed and unsigned numbers. Consider a real number to be even if its integer part is even.

1.5 Write a pattern that matches FORTRAN Hollerith literal specifications.

1.6 Write a pattern to match strings of the form $A^n B^n C^n$ ($n \geqslant 0$).

1.2. GRAMMARS AND PATTERNS

Pattern matching is often concerned with determining whether a string is a member of a set of strings having some desired property. An example is determining whether a string is an integer, i.e., whether it is composed entirely of digits.

In formal language theory [12], a language is a set of strings. There are many ways of characterizing a language. Languages of interest generally contain an infinite number of strings (as in the example above). Therefore, a language cannot, in general, be characterized by simply listing all the strings it contains.

Even though languages are generally infinite, finite characterizations are possible for cases of practical interest. A *grammar* is the commonest characterization. A grammar describes the structure of strings ("sentences") in the language. In programming contexts, the best-known type of formal grammar is Backus-Naur Form [13], abbreviated BNF. (There are a number of different terminologies for describing BNF. The one used below is the author's preference.)

BNF characterizes a language in terms of *terminals*, which are characters from which strings of the language are composed, and *nonterminals*, which specify classes of structures or substructures of interest. Nonterminals are enclosed in angular brackets to distinguish them from terminals. Terminals are written just as they ordinarily appear. A grammar consists of definitions of nonterminals. One nonterminal, called the *goal*, is designated as characterizing the language. In the formal notation of BNF, the definition of a nonterminal is indicated by giving its name, followed by ::= (indicating definition), followed by the specific definition. A very simple example is

```
<comma>::=,
```

which defines a nonterminal, `<comma>`, consisting of a single terminal character. The name "comma" has no meaning of itself, but is chosen to be suggestive.

Nonterminals may have several alternatives in their definitions. This is indicated in BNF notation by using vertical bars to separate the alternatives. An example is

```
<digit>::=0|1|2|3|4|5|6|7|8|9
```

which defines `<digit>` to be any one of ten terminal strings.

An alternative may consist of several concatenated components, called *subsequents*. For example

```
<dpair>::=(<digit>,<digit>)
```

consists of five subsequents, three of which are terminal and two of which are nonterminal. An equivalent definition, using <comma> defined above, is

```
<dpair>::=(<digit><comma><digit>)
```

When writing a grammar, the choice of nonterminals depends on convenience, on essential aspects of the language, and on the attributes of the language that are to be emphasized. Thus, <comma> might be included in a grammar, in spite of its trivial definition, to emphasize its syntactic role.

A nonterminal may be used in its own definition. An example is

```
<integer>::=<digit>|>digit><integer>
```

Such a use is recursive and is implied as such in BNF notation. Here, an integer is either a single digit or a digit followed by another integer. Recursive references are necessary in BNF to describe indefinite repetition. This definition illustrates one of the most important aspects of grammars: that an infinite number of strings can be represented by a finite number of nonterminals.

In general, a nonterminal may appear anywhere in a grammar, even before its definition appears. A BNF grammar is a description of a language, not a series of executable statements. In fact, most grammars contain essential, circular, recursive references. Consider the following grammar:

```
<element>::=a|b|c|d
<member>::=<element>|<ntuple>
<list>::=<member>|<member>,<list>
<ntuple>::=(<list>)
```

In this grammar, <member>, <list>, and <ntuple> are defined in terms of each other. The nonterminal <element> provides an "escape" through which the other nonterminals have alternatives leading to terminals. If <ntuple> is the designated goal, the language consists of strings such as

```
(b)
(a,c)
(b,b,c,d)
((b))
(a,(b,c))
(b,c,(a,(d,d)))
```

and so on.

There is a close analogy between BNF definitions and SNOBOL4 patterns. For example, the statement

```
ELEMENT   =   'A' | 'B' | 'C' | 'D'
```

constructs a pattern that matches any `<element>`. The second and third BNF definitions cannot be directly mapped into SNOBOL4 patterns. Construction of a pattern uses existing values, and both these nonterminals reference themselves or nonterminals that follow. Use of unevaluated expressions to defer reference until pattern matching is performed solves this problem, since both patterns can be constructed before they are used in pattern matching. The three patterns

```
MEMBER   =    ELEMENT | *NTUPLE
LIST     =   *MEMBER | *MEMBER ',' *LIST
NTUPLE   =    '(' LIST ')'
```

complete the definition. Note that unevaluated expressions are needed only in cases where a pattern is referenced before it is constructed.

To write a recognizer for `<ntuple>`s, i.e., a program that determines if a string is an `<ntuple>`, it is important to realize that `NTUPLE` will succeed, in general, if it successfully matches a *substring* of a given string. To assure that an entire string is matched, the following pattern may be used:

```
GOAL   =   POS(0) NTUPLE RPOS(0)
```

This pattern provides "context" for `NTUPLE`. Such context is required for recognizers in general.

Since BNF is widely used to describe formal languages, the close correspondence between BNF and SNOBOL4 patterns provides a convenient approach to developing patterns that recognize strings of a language. There are some precautions to be taken, however, before simply translating a BNF grammar into SNOBOL4 patterns. In the first place, the implications of using unevaluated expressions are significant. As noted in the example above, unevaluated expressions are needed only in places where a pattern is referenced *before* it is constructed. However, unevaluated expressions may be used anywhere. The patterns above could be rewritten as follows:

```
MEMBER   =    *ELEMENT | *NTUPLE
LIST     =   *MEMBER | *MEMBER ',' *LIST
NTUPLE   =    '(' *LIST ')'
```

This leads to the question of what difference, if any, there is between the two sets of patterns. Superficially, the two are the same and recognize the same strings. There are, however, subtle differences.

Any time an unevaluated expression is used in a pattern, the significance of pattern-matching heuristics becomes important. These heuristics, which are applied in the "quickscan" mode, are designed to do two things:

(1) Avoid futile attempts to match in contexts where success is not possible.
(2) Prevent looping due to "left recursive" unevaluated expressions.

The details of these heuristics are described in Reference 2. For recognizers corresponding to BNF grammars, the second point is of greatest importance. Suppose the definition of <list> above is slightly rewritten:

```
<list>::=<list>,<member>|<member>
```

This definition is equivalent to the one above, at least in terms of describing what a <list> may be. For the purposes of description, the order of alternatives is not important, although one order may give a definition that is easier to understand than another. Operationally, however, using patterns corresponding to a grammar, there is a great deal of difference. Alternatives are attempted in order from left to right. The SNOBOL4 pattern corresponding to the revised definition is:

```
          LIST    =    *LIST ',' *MEMBER | *MEMBER
```

In an attempt to match LIST, the first alternative results in an attempt to match LIST, and so on. Unless something is done to "break the loop," there is no possibility of getting to the second alternative. Usually BNF grammars are written in a way that avoids this problem. The problem is, nevertheless, inherent in this type of grammar and corresponding patterns. The quickscan mode breaks the loop by taking length considerations into account and assuming that any unevaluated expression matches at least one character. Consequently, the first alternative above is assumed to match at least three characters. Similarly, when LIST causes another match for LIST, three more characters are required for the first alternative. The heuristics use this information to prevent looping, causing the first alternative to fail and the second to be attempted. As a result, most "recursive references" in patterns do not cause operational difficulties. It is important, however, that such pattern matching be done in the quickscan mode. In the "fullscan" mode, the heuristics are not used and recursive references cause "overflow" as a result of information that is stored each time a pattern is referenced recursively.

The heuristic used to prevent overflow usually causes pattern matching to operate correctly. However, the assumption that every unevaluated expression matches at least one character can cause failure to match when a pattern should, in fact, match successfully. This happens when an unevaluated expression may legitimately match a null string, but is prevented from doing so by the heuristics that erroneously bypass an alternative on the assumption that it requires too many characters.

These problems are partially inherent in the pattern-matching algorithm used in SNOBOL4 and partially inherent in the types of languages that

SNOBOL4 patterns attempt to describe. For the most part, patterns can be constructed without worrying about these problems. When a problem arises, a first attempt at a remedy is to change the pattern-matching mode—from quickscan to fullscan, or vice versa—in hope that the problem will go away. If this fails, the offending pattern may have to be examined in detail, or reformulated.

As noted above, unevaluated expressions are mandatory in some situations, but optional in others. Since some uses are optional, there are questions concerning the advantages and disadvantages of the use of unevaluated expressions.

Unnecessary unevaluated expressions should be avoided where they may lead to left-recursive references as discussed above. There is also a minor penalty in execution time for unevaluated expressions, but this consideration is usually insignificant compared with others.

There is one substantial advantage of the use of unevaluated expressions: saving storage space. In pattern construction, an unevaluated expression is simply a reference to a name (or expression) and occupies a negligible amount of space. On the other hand, when a previously constructed pattern is used in constructing a new pattern, a copy of the previous pattern, however large, is incorporated in the new pattern. The amount of storage required may be substantial, particularly in constructing patterns from formal grammars where there is a tendency to define progressively more complicated structures from simpler ones. The extra storage required also affects execution speed, since more time is spent in storage management. The judicious use of unevaluated expressions in optional contexts may make the difference between a practical and an impractical program.

A word of caution is in order about the utility of patterns corresponding to grammars. Such patterns are easy to construct and make it possible to write recognizers for complicated languages with little difficulty. Consequently, the power of pattern matching is often overestimated. Although recognizers are easy to write, the value of recognition alone is relatively minor. Recognition is not parsing. Pattern matching does not reveal the entire structure of a string, even though that structure is analyzed (internally) during pattern matching. Only the outermost level of a recursively-defined structure is easily accessed, using attached names. It is a common mistake to assume that because complicated recognizers are easy to create, parsers, translators, and compilers follow naturally. That is not the case, and, in fact, patterns derived from grammars are of little use in such problems.

It should also be noted that SNOBOL4 provides a number of pattern-matching facilities that are more efficient and concise than the equivalent BNF patterns. BNF requires recursion to describe constructions that are iterative in nature. SNOBOL4 often permits such constructions to be specified directly. In most contexts, BNF definitions

```
<letter>::=a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<identifier>::=<letter>|<identifier><letter>
```

can be described by the pattern

```
      IDENTIFIER   =   SPAN('ABCDEFGHIJKLMNOPQRSTUVWXYZ')
```

Note also the simplification obtained by specifying a string of characters rather than by the alternation of individual characters. The pattern

```
      ELEMENT   =   ANY('ABCD')
```

is more direct, simpler, and more efficient than the pattern given earlier in mimicking BNF. Similarly, LEN permits characterizations of strings that are awkward to describe in BNF. Finally, SNOBOL4 permits construction of patterns characterizing languages that cannot be described by BNF. The major distinction is context dependence, which BNF cannot describe. Refer to Section 1.1.4 for an example.

EXERCISES

1.7 Write a recognizer for ALGOL 60 arithmetic expressions.

1.8 Write a BNF grammar to describe all strings consisting of two characters.

1.9 Write a BNF grammar to describe all strings of even length.

1.10 Write a BNF grammar that describes quoted literals in SNOBOL4.

1.11 Write a BNF grammar that describes balanced strings as matched by the SNOBOL4 pattern BAL. Write a corresponding SNOBOL4 pattern, not using BAL itself.

1.12 Write a recognizer for SNOBOL4 expressions.

1.13 Write a program that reads a BNF grammar as input and creates a corresponding recognizer.

1.3. PATTERNS AS PROCEDURES

Unevaluated expressions are discussed several places in this chapter. They provide a way of specifying context dependence and a method for providing recursive pattern matching. Independent of these specific applications, unevaluated expressions, used in conjunction with other dynamic features of pattern matching, are sufficiently important to deserve discussion as a separate topic.

There are several ways that changes in a program environment can be effected dynamically during pattern matching. Values can be assigned to variables by immediate value assignment and by the cursor position operator. These types of assignments may produce output and may be traced. In turn, programmer-defined trace procedures may perform any program operation whatsoever. More directly, an unevaluated expression may perform any program operation. It is possible, although not common or necessarily desirable, for a built-in or programmer-defined function to be invoked in the course of pattern matching. In a sense, there is no limit to what it is possible to do during pattern matching, nor is there a limit to the side effects that may result from pattern matching.

The sequence of events that occurs in pattern matching depends on the nature of the string being matched. Furthermore, there is little flexibility in specifying, within a pattern, what sequence of events is to occur during pattern matching. Stated another way, the control structures of pattern matching are quite restricted. Nevertheless, it is possible to write patterns to perform quite sophisticated operations. In this sense, patterns resemble small programs written in a pattern language—a language quite different from the expressions and statements of the rest of SNOBOL4. Applying such a pattern in a pattern-matching statement is similar to invoking a function: the pattern procedure is "executed" by the program that performs pattern matching, using the subject string as data.

Consider the following problem: given a string, print it out, one character per line. Writing a program to do this is trivially simple. The same operation can be performed using a single pattern match. The basic operation is performed by the following pattern:

```
        ROUT    =    LEN(1) $ OUTPUT *ROUT
```

A single character is matched, the character is printed, the pattern calls itself recursively to process the next character, and so on. To assure that the pattern is anchored, the following pattern may be used:

```
        VERTICAL    =    POS(0) ROUT
```

The single pattern-matching statement

```
        STRING   VERTICAL
```

serves to print the characters in STRING regardless of how long STRING is. Note that this pattern requires the fullscan mode to operate properly.

In the sense described above, the pattern VERTICAL corresponds to a procedure that is invoked with STRING as the subject. The relatively awkward control structures of the "pattern-matching language" are displayed by the necessity of using recursion to accomplish an essentially iterative task.

A slightly more complicated situation is posed by the problem of locating the last blank of a string. Pattern matching is highly asymmetrical with respect to direction. Locating the first blank is trivial. Short of resorting to reversing the string, locating the last blank is substantially more complicated. An approach is suggested by the previous example: use a recursive reference to force repeated matching until failure occurs. The basis of the method is a pattern such as:

```
NEXTBL   =   BREAK(' ') @L SPAN(' ') *NEXTBL
```

To assure the match is anchored, the following pattern can be used:

```
LOCBL   =   POS(0) NEXTBL
```

During pattern matching, each time BREAK succeeds, the current cursor position is assigned to L. Eventually NEXTBL fails, and the value of L indicates the position of the last blank in the subject string.

This pattern fails to account for a subject string that contains no blanks. LOCBL always fails; there is no way it can succeed. Therefore it is only useful because it assigns a value of L. Even if L is set to zero before matching, there is no way to distinguish, as a result of matching, whether the subject string starts with a blank or simply contains no blanks at all. This problem may be resolved by using another cursor assignment to note the position after blanks:

```
NEXTBL   =   BREAK(' ') @L SPAN(' ') @M *NEXTBL
```

Furthermore, both L and M can be initialized to zero at the beginning of the pattern match, and the value of M tested at the end:

```
LOCBL   =   POS(0) @L @M NEXTBL | *GT(M,0)
```

After the first alternative fails (as it must), the value of M is checked. If it is greater than zero, at least one span of blanks must have been found, and the value of L is the position of the last span of blanks. If M is zero, there are no blanks in the subject string. LOCBL now succeeds if blanks are found and fails otherwise.

This kind of programming—writing pattern procedures—appeals to some programmers and repels others. It offers the advantages of compactness, efficiency in some situations, and intellectual challenge. On the other hand, such programming methods tend to be difficult, obscure, error-prone, inefficient in some situations, and particularly susceptible to idiosyncrasies in the pattern-matching algorithm.

EXERCISES

1.14 What happens if the pattern VERTICAL is used in the quickscan mode?

1.15 What happens if the pattern ROUT is used directly, without anchoring the pattern match?

1.16 Write a pattern that prints the location of all spans of blanks in a string. Generalize the pattern so that the characters in spans of interest are not limited to blanks, and may be specified at the time pattern matching is performed.

1.17 Write a pattern that divides a string at its last (rightmost) span of blanks, assigning the part before the span to one variable, and the part after the span to another.

1.18 Write a pattern to print the items of a list as described in Section 1.1.1.

1.19 Write a pattern to print the words in a string of text.

1.4. STRUCTURING DATA TO UTILIZE PATTERN MATCHING

One of the most difficult problems in programming is the selection of good representations for data. SNOBOL4 offers many possibilities, especially in data structures. A good part of Chapter 3 is devoted to this subject. Since SNOBOL4 has so many string-manipulation facilities, it is tempting to represent all kinds of data as strings. Succumbing to this temptation is perhaps the most frequent cause of poor SNOBOL4 programs. Nevertheless, there are some natural uses of strings for representing data. In addition to data that is inherently string-structured, such as written text, there are situations in which mathematical expressions are best handled as strings. This matter is discussed in Chapter 4.

Occasionally the possibility of a string representation of data occurs in unexpected contexts. Consider, for example, the question of patterns of occurrences of integers with certain properties [14]. Let the characters of a string represent the positive integers by position. Suppose that X is used to indicate an integer having a specified property, and 0 is used to indicate an integer that does not have the property. A "property string" corresponding to the odd integers is

```
XOXOXOXOXOXOXOXOXOXOX ...
```

Similarly, the $\overset{(2,3,5,7,\ldots)}{\text{primes are}}$ represented by

```
0
XXXOXOXOOOXOXOOOX...
```

Alternatively, discarding the lone even prime, the primes among the odd integers are represented by

```
0
XXXXOXXOXXOXOOXXOO ...
```

A little knowledge of number theory suggests that one would not expect to find regular patterns in such sequences. In any event, the strings of interest would be too long to manipulate in a program. Nonetheless, this approach provides an interesting, if somewhat unconventional, way of considering such data.

Another example of the use of string representation is afforded by the familiar game of tic-tac-toe. One of the more common demonstrations of interactive computing is a program that plays this game against a human opponent. The game itself is simple, and the program is not difficult to write. One of the first considerations is the representation of the board on which 0s and Xs representing the player's moves are to be placed. The board is usually pictured geometrically as shown by a game illustrated in Figure 1.1.

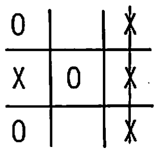


Figure 1.1 A Tic-Tac-Toe Board

A program representation, analogous to this illustration, is a 3-by-3 array. To obtain symmetry, the array might have its center at zero:

```
          BOARD   =   ARRAY('-1:1,-1:1')
```

There is no question that such a representation is natural. Examination of the operations involved in playing tic-tac-toe shows that such an array representation presents some clerical difficulties. The essence of the game involves adjacencies. Any three similar marks along a line constitute a win. Two marks along a line containing an empty space represent a potential win (or loss, depending on whose turn it is). While it is no technical problem to write a program to determine such configurations, there are many combinations to be tried, even for such a small board.

In SNOBOL4 it is natural to think of such configurations as patterns. Unfortunately, SNOBOL4 does not provide two-dimensional strings. The two-dimensional structure can be easily "unfolded", however, into a one-dimensional string of nine characters. Suppose the board positions are identified by numbers as shown in Figure 1.2.

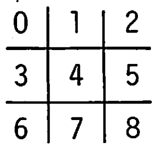


Figure 1.2 A Numbered Board

Then a string of nine characters, corresponding by position to the numbers above, represents the board. The reason for starting the numbering at zero, rather than one, will become apparent in the subsequent discussion. An empty board, corresponding to the beginning of the game, is created by a statement such as:

```
         BOARD   =   '.........'
```

Dots instead of blanks are chosen to represent empty positions in order to provide visibility to the board.

Consider some typical operations that occur during a game. One is placing a mark at a given position. A pattern that can be used is

```
         PLACE   =   TAB(*N) . H LEN(1)
```

and the statement to perform the operation is

```
         BOARD   PLACE   =   H MARK
```

where the value of N determines the position, as indicated in Figure 1.2, and the value of MARK determines which type of mark is made. Starting the numbering at zero avoids the necessity for constantly subtracting one when locating the position.

The center of a tic-tac-toe board is crucial in the game. A pattern to determine whether the center is empty is

```
         CENTER   =   TAB(4) '.'
```

More complicated board positions require more elaborate patterns. The rows are described by three consecutive characters beginning at positions 0, 3, and 6. If C1, C2, and C3 are three consecutive marks of interest, a pattern to determine if such a row exists is given by

```
         ROW   =   (TAB(0) | TAB(3) | TAB(6)) C1 C2 C3
```

Anticipating the need for such patterns in more general contexts, the following function might be useful:

```
         DEFINE('ROW(C1,C2,C3)')
         R   =   TAB(0) | TAB(3) | TAB(6)
                  .
                  .
                  .
ROW      ROW   =   R C1 C2 C3                 :(RETURN)
```

ROW returns a pattern that successfully matches a board that has a row consisting of C1, C2, and C3 (in that order). An example is

```
         ROWX   =   ROW('X','X','X')
```

which produces a pattern that matches a row of Xs.

Columns start at positions 0, 1, and 2. A function COL, similar to ROW, is given by

```
          DEFINE('COL(C1,C2,C3)')
          C    =    TAB(0) | TAB(1) | TAB(2)
                         .
                         .
                         .
COL       COL    =    C C1 LEN(2) C2 LEN(2) C3      :(RETURN)
```

A function to create a diagonal-matching pattern is:

```
          DEFINE('DIAG(C1,C2,C3)')
                         .
                         .
                         .
DIAG      DIAG     =     TAB(0) C1 TAB(4) C2 TAB(8) C3 |
+                        TAB(2) C1 TAB(4) C2 TAB(6) C3  :(RETURN)
```

A losing board position for the player with the mark X is described by

```
          LOSEX    =    ROW('O','O','O') | COL('O','O','O') |
+                      DIAG('O','O','O')
```

Note that the patterns created by these functions are naturally thought of as pattern procedures that go through a series of alternatives during pattern matching.

Another use of patterns is illustrated by the problem of printing the board. Printing involves "folding" the string that represents the board. A row of the two-dimensional representation of the board is printed by the pattern

```
          ROW    =    LEN(3) . OUTPUT
```

A pattern to print the entire board is

```
          PRINT    =    ROW ROW ROW
```

The printing statement is simply

```
          BOARD   PRINT
```

If blank lines are needed above and below the printed board to set it off from other output, PRINT can be rewritten as follows:

```
          SKIP     =    NULL . OUTPUT
          PRINT    =     SKIP ROW ROW ROW SKIP
```

where the value of NULL is the null string.

As mentioned before, the use of pattern matching to this extent is a matter of taste. This example is designed to illustrate the systematic application of pattern matching to procedural problems.

EXERCISES

1.20 Write a program to print
(a) The positions of interest in property strings.
(b) The distances between positions of interest in property strings.

1.21 A palindromic number is one that reads the same forward and backward. An example is 8135318. Generate property strings for
(a) The palindromic numbers among the first 500 positive integers.
(b) The odd palindromic numbers among the first 500 positive integers.
(c) The palindromic numbers among the first 500 squares.
Apply the results of Exercise 1.20 to these strings.

1.22 Write a pattern that locates a free corner on a tic-tac-toe board.

1.23 Write a complete program to play tic-tac-toe. Design the program to operate interactively to play against a human opponent. Use the string representation of the board and employ pattern-matching techniques wherever possible.

1.24 Describe how to minimize the size of the patterns needed in the solution to Exercise 1.23.

1.25 Perform Exercise 1.23 using an array representation of the board. Avoid the use of pattern matching as far as possible. Compare the two programs.

2 DEFINED FUNCTIONS

Defined functions serve many purposes in SNOBOL4 programs. They provide a mechanism for isolating commonly used sections of a program in one place, they serve as mnemonic aids in identifying certain operations with common notation, and they are a vehicle within which recursion may be performed.

In a sense, the use of functions is a programming discipline—a way of thinking and a framework for problem solving. No other technique is as important. The proper use of functions often makes feasible a program solution that otherwise would be intractable.

The virtues of the use of functions are generally well understood, although they receive more lip service than practice. We will assume that the virtues of functions are not arguable, and focus attention instead on different contexts in which functions are important, describe good programming practices with respect to functions, and give a number of examples of functions, some of which will be useful later in the book.

2.1. DEFINED FUNCTIONS AS A MECHANISM FOR LANGUAGE EXTENSION

Defined functions provide the major facility for extending the SNOBOL4 language. SNOBOL4 has a number of built-in functions (about 60—the number varies somewhat with the particular version). The syntax for all these functions is the same; the difference is in the name of the function and the number of arguments, but not in the form. Defined functions use the same syntax. There is no general way of telling, from a function call itself, whether the procedure being called is built into the SNOBOL4 system or is part of the particular SNOBOL4 program. A defined function is used in a program in the same way that a built-in function is used. Thus, the definition of a function can be thought of as an extension of the SNOBOL4 language. Defined functions, for the most part, could just as easily be implemented as built-in functions; the program would be written the same way and the results of execution would be the same. (Recursive functions, whose definitions involve calling the functions themselves, are not as naturally thought of in this way.)

2.1.1. Basic Extensions

When a language like SNOBOL4 is designed and implemented, certain operations are included in the form of built-in functions and operators, and other operations are left to be defined if they are needed. The considerations that apply are:

(a) Generality of the need for a particular operation. SIZE, for example, is an operation that has general use. If an operation is frequently needed, a built-in function provides convenience and efficiency.
(b) Relative efficiency of a built-in procedure as opposed to a defined procedure. SIZE, for example, can be implemented as a defined function using pattern matching. This defined function is awkward and extremely inefficient compared with the built-in function.
(c) The impossibility of defining the operation. REWIND, for example, cannot be defined in terms of other built-in operations.

Working against the inclusion of an operation in the built-in repertoire is the cost associated with each built-in function. Cost can be measured in terms of

(a) Implementation effort.
(b) Increased size of the resulting system.
(c) The documentation required.
(d) The burden imposed on the user of an increasingly large and complex language.

Working within the framework and philosophy of a particular language, the designer makes decisions based on the contending factors listed above. The result is inevitably a compromise, and experience usually shows that some of the decisions are incorrect. In any event, the repertoire of built-in functions suits some users in some situations, but not others. In most cases it is easy to extend the repertoire of built-in functions by adding defined functions. Consider the following example.

SNOBOL4 has few facilities for automatic formatting on output. When printing columns on a page, it is necessary to pad with blanks in order to place strings in their appropriate places. This is generally done using DUPL. For example, to print a string left justified ten spaces from the left margin, the following statement might be used:

```
          OUTPUT   =   DUPL(' ',10) STRING
```

Similarly, to print the string right justified at column 20, the following statement might be used:

```
          OUTPUT   =   DUPL(' ',20 - SIZE(STRING)) STRING
```

If several columns are to be printed side by side, this method becomes cumbersome. For such situations, two functions, LPAD and RPAD, are useful. Some implementations of SNOBOL4 have these functions built in. LPAD and RPAD, in their simplest forms, have two arguments: a string and an integer. The value returned is the string with blanks added on the left or right respectively so that the length of the string returned is the specified integer. Simple definitions of these functions follow.

```
          DEFINE('LPAD(S,N)')
          DEFINE('RPAD(S,N)')
                        .
                        .
                        .
LPAD      LPAD   =    DUPL(' ',N - SIZE(S)) S      :(RETURN)
RPAD      RPAD   =    S DUPL(' ',N - SIZE(S))      :(RETURN)
```

With these functions, the SNOBOL4 language is extended to provide padding with blanks. Such functions can be used to build up a library for use with other programs.

The procedures for LPAD and RPAD given above illustrate some general principles that should be followed in writing functions. These principles are especially important when functions are used with language extension in mind and are intended to form part of a library. Such functions should be carefully designed to produce meaningful results or error indications if an unexpected or erroneous situation is encountered. For example, suppose that N is less than SIZE(S). In the functions above, the call of DUPL fails and a null string is returned as value. Return of a null string without any indication of a problem is probably not what a user of these functions would want. One solution is to signal failure in such cases as illustrated by the following alternative procedure for LPAD:

```
LPAD   LPAD = DUPL(' ',N - SIZE(S)) S   :S(RETURN)F(FRETURN)
```

This solution has the intrinsic merit of causing the padding function to treat an impossible situation in the same way that the built-in function DUPL does. When thinking of defined functions as extensions of the language, it is usually desirable, for consistency, to have defined functions behave in the same general way that built-in functions do.

There is always the question of how far to carry error checking and what to do if an error is detected. Suppose, for example, N is erroneously given as a (nonnumeral) string. This results in an argument for subtraction that has an illegal data type. The procedure could test the data type of the argument before executing the padding statement. This is cumbersome and time consuming, however. There are two other approaches: (1) use &ERRLIMIT to have the error cause failure, or (2) let the error occur. Both methods have their advantages and disadvantages, and a particular situation or programmer preference may dictate the choice. The second alternative is simpler and has the virtue of treating such errors with essentially the same philosophy that SNOBOL4 itself uses.

Some individuals prefer to provide error exits for erroneous situations. This technique is particularly useful during program debugging. For example, the procedure above might be changed to

```
LPAD   LPAD = DUPL(' ',N - SIZE(S)) S      :S(RETURN)F(ERROR)
```

If the string cannot be padded, a branch to ERROR occurs. A statement at ERROR could print an error message and then return, successfully or signaling failure, depending on preference. Alternatively, in the debugging stage, it may be convenient not to provide a statement with the label ERROR. If a branch to ERROR occurs, program termination results. This pinpoints the location of the problem and has the additional advantage of leaving the values of formal arguments (and local variables, if any) of the function intact and available for inspection in the string dump. A statement with the label ERROR can be added later when the debugging phase is complete.

For functions that serve as language extensions and are likely to be placed in a library or distributed to other programmers for their use, error messages as described above may prove annoying. For example, error messages printed in an otherwise acceptable output are likely to be intrusive. This situation essentially amounts to a question of control. In general, the less a function does to take control away from the user, the better. Nothing essential is lost, for example, if a function simply fails without printing an error message. The failure of the function call can be sensed and an appropriate message printed, if desired, by the routine that made the call. Such a situation is illustrated in the following statement:

```
       OUTPUT  =  LPAD(S,20)                  :F(ERROR)
```

In some cases it may be more desirable not to treat the inability to pad a string as an error at all. The padding functions can simply return the string unmodified if it is too long. This behavior can be obtained without extra programming by using a simple device that is useful in many similar contexts. Rather than thinking of the padding functions as having two arguments from which a value is computed, think of the padding functions as modifying a string. Let the first argument of LPAD be LPAD itself:

```
          DEFINE('LPAD(LPAD,N)')
                      .
                      .
                      .
LPAD      LPAD   =   DUPL(' ',N - SIZE(LPAD)) LPAD   :(RETURN)
```

When LPAD is called, the value of the variable LPAD is the string to be modified. If DUPL fails, LPAD is unchanged, and the string is simply returned, unmodified, as value.

Another matter that should be kept in mind in writing functions is generality. Often, if a little care is taken in design, a function can be made general enough to use in a variety of situations, whereas if a function is written only for a specific situation, other similar and essentially repetitive functions must be written whenever a new situation arises. The padding functions given above illustrate this point. While it is generally true that padding is done with blanks, sometimes other characters are used for padding; periods and dashes are most typical. The functions as written above are useless for such situations, although a minor change would provide the desired generality. Using the form of LPAD where failure is signaled if the string cannot be padded, the result is:

```
          DEFINE('LPAD(S,N,C)')
                      .
                      .
                      .
LPAD      LPAD = DUPL(C,N - SIZE(S)) S       :S(RETURN)F(FRETURN)
```

Here the third argument provides the padding character. A further touch of elegance can be obtained by considering that most padding is done with blanks, and that having to specify this in every call is cumbersome. Therefore establish blank as the default value for C as follows:

```
LPAD      C   =   IDENT(C) ' '
          LPAD = DUPL(C,N - SIZE(S)) S       :S(RETURN)F(FRETURN)
```

Not only does this technique obviate the explicit writing of the third argument when blank padding is desired, but it is also an "upward compatible" generalization of the previous version of LPAD. No matter how much care is taken to design functions with generality, it is frequently the case that hindsight suggests better methods. Where new procedures can be added without requiring modification of existing function calls, that approach should be taken. It might appear more logical, for example, to have the padding character as the second argument rather than the third. This choice has two disadvantages, however: (1) if the padding functions with two arguments had previously been used in programs, every call would have to be rekeyboarded, and (2) for the most common case, padding with blanks, the second argument would have to be written with an explicit blank or would have to be explicitly omitted using a second comma to indicate a null argument. As a general rule, extra, optional arguments are best placed in trailing positions.

When writing functions to extend SNOBOL4, it is particularly important that the procedures be as independent as possible of conditions in the programs in which they may be used. Since SNOBOL4 provides no facility for local labels, it is obviously not possible to write functions so that they will work in any program. Other global aspects of the program environment are similarly beyond the control of a particular function. Since a function can only achieve an approximation of independence from the context in which it is used, there is always the question of how much effort should be expended in this regard. In practice, the problem is not as severe as it may seem. Some matters, such as specification of local variables, are simple. By selecting labels that are unlikely to be used in other programs, the probability of the collision of label names can be minimized. Such methods produce procedures that are difficult to read, and hence are not used in this book. The commonest cause of conflict is the anchored mode of pattern matching. Consider a function COMPRESS(S,C) that compresses spans of characters. A simple procedure is:

```
         DEFINE('COMPRESS(COMPRESS,C)')
         COMPAT    =    *C SPAN(*C)
                   .
                   .
                   .
COMPRESS COMPRESS   COMPAT    =    C      :S(COMPRESS)F(RETURN)
```

This procedure requires the unanchored mode. If used in a program that is operating in the anchored mode, COMPRESS does not work properly. One rather awkward solution to this problem is to save the current mode on entry to the function, establish the unanchored mode, and then restore the original mode on exit from the function. Generally speaking, it is better practice to write the procedure in a way that is independent of the anchored mode. An alternative procedure is:

```
         DEFINE('COMPRESS(S,C)H')
         COMPAT    =    BREAK(*C) . H SPAN(*C)
                   .
                   .
                   .
COMPRESS S   COMPAT   =                         :F(COMRET)
         COMPRESS   =   COMPRESS H C            :(COMPRESS)
COMRET   COMPRESS   =   COMPRESS S              :(RETURN)
```

EXERCISES

2.1 Write procedures for LPAD and RPAD allowing for a third argument that specifies the padding character and taking into account the problem that arises if a string more than one character long is given as the third argument.

2.2 Write a function for centering a string within a specified width.

2.3 Write a function whose value is the result of interleaving the characters of two strings. Take into account the possible situations that may arise if the strings are not of the same length.

2.4 Write a function to reverse a string.

2.5 Write a function to rotate a string by *n* characters.

2.6 Write a function to delete all occurrences of a specified character from a string.

2.7 Frequently there is a need to use arrays where the number of items to be referenced is not known. Write functions to truncate and extend arrays.

2.1.2. More Elaborate Extensions

The functions described in the preceding section are of a kind that could easily have been included in the basic repertoire of built-in functions. In reference to the criteria for determining whether or not a function should be built-in, all of the defined functions listed above are at least reasonable candidates for being built-in.

There are often situations where a given function or set of functions is of considerable utility in a specific program or set of programs, but not of sufficient general utility to warrant serious consideration for inclusion as part of the language. Nonetheless, such a function or set of functions may be a reasonably consistent extension within a limited context. Consider the following example.

SNOBOL4 has facilities for representing and performing computations on integers and real numbers. Some problems require the manipulation of complex numbers, which consist of two parts: a real part and an imaginary part (both parts are typically real numbers). The DATA function in SNOBOL4 provides a way for representing such objects. The statement

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

creates a defined data type, COMPLEX, and three functions. COMPLEX creates COMPLEX objects, and R and I are field functions referencing the real and imaginary parts, respectively. These three functions provide the capability for performing complex arithmetic. For example, the following statements create three complex numbers, the third of which is the (complex) sum of the first two.

```
        C1   =   COMPLEX(1.3,-2.5)
        C2   =   COMPLEX(-4.1,6.0)
        C3   =   COMPLEX(R(C1) + R(C2),I(C1) + I(C2))
```

The three functions COMPLEX, R, and I do not, in themselves, provide any complex arithmetic operations or functions. A natural extension for a program that manipulates complex numbers is a set of functions for performing complex arithmetic. Consider the addition of complex numbers:

```
        DEFINE('ADD(N1,N2)')
                   .
                   .
                   .
ADD     ADD   =   COMPLEX(R(N1) + R(N2),I(N1) + I(N2))   :(RETURN)
```

With this function, any two complex numbers can be added.

When thinking of functions as a language extension for a specific use, such as complex arithmetic, there is the question of what functions should be provided. For complex numbers, ADD, SUB, MUL, and DIV are obvious. Less obvious is the need for data-type conversion functions. There are a variety of automatic conversions in SNOBOL4 for built-in data types. For example, conversions among integers, real numbers, and numeral strings are performed automatically in most contexts. Since data read into or output from a program is necessarily in the form of strings, a natural extension is to provide for conversions between STRING and COMPLEX data types. It is necessary to decide how a complex number is to be represented as a string. If the conventional printed form is taken as a guide, the first two complex numbers given above might have the following string representations:

```
        S1   =   '1.3-2.5I'
        S2   =   '-4.1+6.0I'
```

Thus, the real part is given first with an initial sign only if it is negative, and the imaginary part follows, always with a sign and followed by an I. Other representations are possible; the choice is largely a matter of personal preference. It is desirable to select a representation that is natural, close to the conventions used in other contexts, easy to read, and reasonably easy to process.

Two functions, STRCPX and CPXSTR, for converting from STRING to COMPLEX and vice versa now follow naturally:

```
         DEFINE('STRCPX(S)I,R')
         DEFINE('CPXSTR(C)')
         CPLX    =    (LEN(1) BREAK('+-')) . R BREAK('I') . I
                          .
                          .
                          .
STRCPX S    CPLX                                    :F(FRETURN)
         STRCPX    =    COMPLEX(+R,+I)              :(RETURN)

CPXSTR CPXSTR    =    LT(I(C)) R(C) I(C) 'I'        :S(RETURN)
         CPXSTR    =    R(C) '+' I(C) 'I'           :(RETURN)
```

In the first function, pattern matching is used to locate the two parts. The unary + is used to convert these parts to numbers. If this were not done, the two parts would be strings. Although they would have the correct numerical values, and arithmetic would be performed correctly, constant conversion would be performed. More seriously, a positive imaginary part would have an initial plus sign which would complicate CPXSTR, because it would then have to deal with both numeral strings and numbers. It is better to produce COMPLEX numbers in "canonical form" from the beginning. In CPXSTR, note that the sign separating the real and imaginary parts must be given special consideration.

With the functions given above, complex numbers can be read in, complex calculations can be performed, and the results can be printed out. It is nonetheless awkward to have to use function calls when performing arithmetic. Consider the statement

```
         C5    =    SUB(MUL(C1,C2),MUL(C3,C4))
```

If integers or real numbers, rather than complex numbers, were being operated on, the statement could be written as

```
         C5    =    C1 * C2 - C3 * C4
```

Operator notation is, in fact, a convenient shorthand. There is no basic difference between functions and operators except syntax. Operations that are performed very frequently are represented by operators rather than by functions; an operator expression is generally easier to read and is briefer. Conventions about precedence and associativity of operators make most parentheses unnecessary. If much complex arithmetic is being performed, an operator notation would offer the convenience customarily associated with integer and real arithmetic. In addition, complex arithmetic operations paralleling integer and real operations could have the same form.

SNOBOL4, through the use of OPSYN, makes such extension of operators possible. For addition, the statement

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

changes the meaning of the binary + operator, making it equivalent to ADD so that when + is used, ADD is called to perform the operation. Now it is possible to write

```
        C1    =    C2 + C3
```

instead of

```
        C1    =    ADD(C2,C3)
```

If nothing except the OPSYN above is executed, however, an error results. The OPSYN makes + and ADD synonymous. When ADD is called, + is encountered in the procedure for ADD. Since + is synonymous with ADD, ADD is called again, this time with real numbers as arguments. Subsequent application of field functions to these real numbers is an error. The basic problem is that there are two kinds of addition: the defined one for complex numbers, and the built-in one for real numbers and integers. This problem may be solved by first "saving" the built-in operation for binary + by OPSYNing another operator, say &, to it. This operator can then be used instead of + in the ADD procedure. The resulting changes are

```
        OPSYN('&','+',2)
        OPSYN('+','ADD',2)
                  .
                  .
                  .
ADD     ADD   =    COMPLEX(R(N1) & R(N2),I(N1) & I(N2))     :(RETURN)
```

Of course, any part of the program that performs integer or real addition must be changed similarly. If complex arithmetic is to be considered an extension to SNOBOL4, requiring the use of & for + everywhere (and other operators similarly) is extremely unattractive.

A further extension solves the problem. Since the occurrence of + now calls ADD, the procedure for ADD can be extended to test the data types of its operands and perform the appropriate kind of arithmetic accordingly. For the purpose of example, we will assume that if either operand is COMPLEX, both are.

```
ADD     IDENT(DATATYPE(N1),'COMPLEX')           :S(ADD1)
        ADD   =    N1 & N2                      :(RETURN)
ADD1    ADD   =    COMPLEX(R(N1) & R(N2),I(N1) & I(N2))     :(RETURN)
```

Now + can be used anywhere in the main program and the correct operation will be performed. Of course, some efficiency is lost, since any occurrence of + results in a call to ADD, regardless of whether integer, real, or complex addition is being performed.

Complex arithmetic is just one example of the use of functions for language extension. There are numerous examples in other chapters.

EXERCISES

2.8 Write a full set of functions for complex arithmetic as follows:
   (a) First consider only complex operands.
   (b) Then consider extension to integers and real arithmetic.
   (c) Finally consider the case in which mixed forms of arithmetic may occur.

2.9 Write functions corresponding to the unary + and - operators. Consider all cases listed in Exercise 2.8.

2.10 Write a function that computes the absolute value of a complex number.

2.11 Add error checking to STRCPX.

2.12 Extend STRCPX and CPXSTR to allow for operands of various types.

2.13 Generalize the built-in function TRIM so that the trailing character to be trimmed can be specified.

2.2. RECURSIVE FUNCTIONS

The technique of recursive definition—defining a thing in terms of itself—is a powerful and concise method commonly used in mathematics. Well-known examples are the factorial:

$n! = 1$	for $n = 0$
$n! = n*(n-1)!$	for $n > 0$
$n!$ is undefined otherwise	

and the Fibonacci numbers:

$f(n) = 1$	for $n = 1, 2$
$f(n) = f(n-1) + f(n-2)$	for $n > 2$
$f(n)$ is undefined otherwise	

Such definitions suggest a similar approach to the computation of values for these functions. For example, the recursive definition of Fibonacci numbers has a direct counterpart in the following function:

```
          DEFINE('F(N)')
                    .
                    .
                    .
F         LT(N,1)                                        :S(FRETURN)
                    N,2
          F    =    LE(~~N2~~) 1                         :S(RETURN)
          F    =    F(N - 1) + F(N - 2)                  :(RETURN)
```

The close correspondence between the recursive definition and the function for computing values makes programming functions like these very easy indeed. Note that an argument for which the function is not defined causes failure. Validity checking could be extended to assure that the argument is of the correct data type.

Although recursion is often convenient, it may be inefficient, and is certainly not the best approach to use if many values are to be computed. There are several reasons for this. One is inherent in recursive function calls: values of formal arguments and local variables are saved when a call is made and are restored upon return. This process necessarily takes time and space. There is, of course, computational overhead in the call itself. Another reason for inefficiency results from the structure of some definitions themselves. To compute F(5), for example, the recursive method requires the computation of F(4) and F(3). But F(4) requires the computation of F(3) and F(2), and so on. Figure 2.1 shows all the values computed (and hence calls made) in the process of computing F(5).

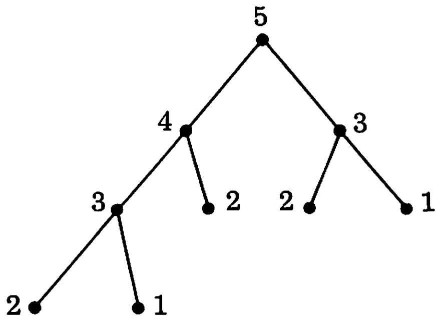


Figure 2.1 Calls in Computing F(5)

Thus, F(3) is called twice, F(2) three times, and F(1) twice, all in a single computation of F(5). For these reasons, recursive procedures are usually avoided in such computations. Iterative methods, without the inefficiencies mentioned above, are sought instead.

In spite of these problems, however, recursion has a very important place in programming. The underlying motivations for using recursion are its essential conciseness and the close relationship that often exists between the statement or structure of a problem and its recursive solution. In this sense, recursion is as much a problem-solving tool as it is a programming tool. There are many programming problems for which it is so difficult to formulate any solution at all that the simplicity of a recursive formulation may provide the only method.

Consider the following well-known recursive function due to Ackermann [16,17]:

$$
\begin{array}{ll}
a(m,n) = n + 1 & \text{for } m = 0, n \geqslant 0 \\
a(m,n) = a(m-1,1) & \text{for } m > 0, n = 0 \\
a(m,n) = a(m-1, a(m,n-1)) & \text{for } m > 0, n > 0 \\
a(m,n) \text{ is undefined otherwise} &
\end{array}
$$

While this function is of theoretical, not practical, interest it nevertheless provides an example for which a recursive solution is easily formulated by mimicking the definition. An iterative solution is another matter.

The structure of data in a problem may suggest a recursive solution. Consider the case of an arithmetic expression containing, for sake of example, only binary operators. Usually such expressions are written in infix notation with the operators between the operands. An example is

```
((7+X)-(Y*(N+3)))/2
```

An alternative form uses prefix notation in which the operators precede their operands. In this notation, the expression above is

```
/(-(+(7,X),*(Y,+(N,3))),2)
```

While prefix notation is more difficult for human beings to read, it is often easier to process in a program. Here we are concerned simply with the problem of converting from one form to another. Assume that a function PREFIX for doing this conversion is to be written. To further simplify the problem, we will assume that infix expressions are sufficiently well parenthesized so that implied associativity and precedence do not have to be considered. To convert from infix to prefix form, an expression can be considered to be a string in which an operator stands between two operands. Of course, the operands may contain expressions themselves, but the entire string consists of an operator separating two operands. Assuming the four most frequently encountered operators, for sake of example, a pattern to match an expression is:

```
INFIX  =   POS(0) BAL . L ANY('+-*/') . OP BAL . R RPOS(0)
```

The use of POS and RPOS assures that the entire string is matched, and BAL assures that the operands are correctly matched according to their parenthesization. INFIX matches correctly unless the entire expression is an operand. This may occur either because there is no operator at all, or because the entire expression is surrounded by parentheses. Examples of these cases are:

```
X
(Y+2)
```

The first case does not require conversion to prefix form, and the second case can be resolved by removing any parentheses that surround the entire expression. Assuming that these matters have been taken care of, the pattern INFIX can be used to identify the operator, and to assign operands to L and R. Both L and R are, necessarily, infix expressions. Since PREFIX is a function for converting from infix to prefix, it can be applied to L and R. The transformation is:

```
        EXP   INFIX  =  OP '(' PREFIX(L) ',' PREFIX(R) ')'
```

Of course this statement is itself a part of PREFIX. At this point, recursion enters the picture. A complete function, including the handling of expressions that are operands, follows:

```
        DEFINE('PREFIX(EXP)L,R,OP')
        STRIP  =  POS(0) '(' BAL . EXP ')' RPOS(0)
                 .
                 .
                 .
PREFIX  EXP  STRIP                             :S(PREFIX)
        EXP  INFIX  =  OP '(' PREFIX(L) ',' PREFIX(R) ')'
        PREFIX  =  EXP                         :(RETURN)
```

The first statement of PREFIX removes any parentheses that may surround the entire expression. Note that the result within parentheses is assigned to EXP, and hence transforms EXP. If INFIX fails to match, EXP is not transformed. In either case, the expression is assigned to PREFIX which is returned as value.

One problem that frequently arises is how to detect when a recursive solution should be employed. It is not possible to give a simple recipe to follow, but the presence of one factor may suggest the possibility of a recursive solution: if the data to be processed can be described recursively, recursive processing is usually feasible. The expressions processed above have this quality. That is, an expression may consist of an operator separating two operands, but an expression is also an acceptable operand for another operator.

One final word of caution about the use of recursion is in order. Most implementations of SNOBOL4 have a fixed amount of space reserved for saving the values of formal arguments and local variables. The maximum depth of function call therefore depends on the number of values that have to be saved and in some cases on other processes, such as storage regeneration, that may share the space used for saving values. A maximum depth of 30 to 50 is typical. This limitation may impose very real restrictions on the use of recursion in some instances.

This section merely touches on the use of recursion. Many of the programs given in subsequent chapters use recursion in one way or another. Some individuals find recursion natural and appealing while others have difficulty understanding it and using it. Since recursion is an essential part of much of the subsequent material in this book, the reader who is uncomfortable with recursion should spend some time studying the concept of recursion and attempting specific examples before proceeding. The best method for gaining practical insight is to actually run some simple programs containing recursive procedures and to observe the results. In particular, use &FTRACE to obtain a printed record, in sequence, of calls and values returned.

EXERCISES

2.14 Write both recursive and iterative definitions for the factorial function. Compute *n*! for several (reasonably small) values of *n*, using &FTRACE to explicate the processing performed by the two procedures.

2.15 Write an iterative procedure for computing Fibonacci numbers. Compare this procedure to the recursive one given in the text.

2.16 Write a (recursive) procedure for computing Ackermann's function. Compute $a(1,2)$, $a(2,3)$, and $a(3,2)$. Use &FTRACE to print the course of the computations.

2.17 Modify the solution to Exercise 2.16 to print a histogram of the depth of recursive call during the computation of Ackermann's function.

2.18 Use the suggestion given in Section 2.1.1 and make the formal argument of PREFIX the same as the function name.

2.19 What would be the effect if PREFIX were not defined with L, R, and OP as local variables?

2.20 Write a recursive procedure for reversing strings (see Exercise 2.4). Compare the iterative and recursive solutions.

2.21 Describe how the structure of an infix expression is related to the maximum depth of recursive call in PREFIX.

2.22 Write a function that converts expressions from prefix form to infix form.

2.23 Write a function that evaluates expressions written in prefix form. Assume that all operands are integers.

2.3. GENERATORS AND SUCCESSORS

In a number of situations, it is necessary to generate, on demand, successive values according to some set of rules. The most familiar example is the generation of pseudo-random numbers.

2.3.1. Random Number Generation

The generation of pseudo-random numbers has been the subject of a great deal of investigation, both theoretical and practical. Discussion of the properties of pseudo-random number generators is beyond the scope of this chapter. For excellent coverage of this subject, see Reference 18. The process can be conceptualized as the selection of successive integers R_0, R_1, ... R_j, ... from a set of integers as shown in Figure 2.2.

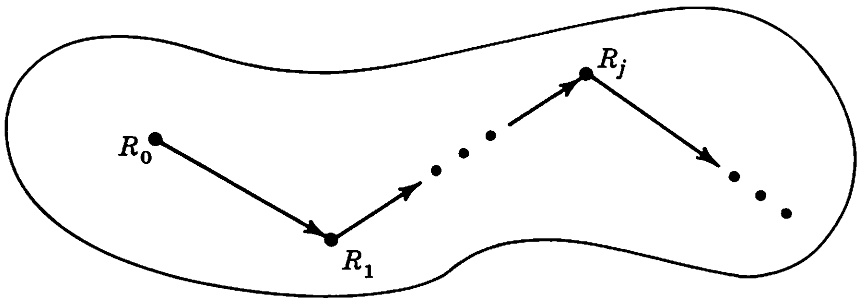


Figure 2.2 Successive Integers

The process of generating successive integers has two fundamental properties: (1) the jth number, R_j, is derived from the $(j-1)$st number, R_{j-1}, according to some fixed set of rules, and (2) the rules are designed to provide properties desired of the set of numbers. In the case of pseudo-random numbers, rules are chosen so that the set of numbers generated has the statistical properties that would be expected of truly randomly-chosen numbers. In most cases, the rules are the same regardless of the value of j and the time that the rules are applied. Such rules constitute a successor function that can be applied to any generated number to get the next number.

One of the best methods of generating pseudo-random numbers is the use of a linear congruence sequence which has the form:

$$R_{j+1} = p^*R_j + c \qquad \text{mod } m \text{ for } j \geqslant 0$$

Where

R_0	initial value	$R_0 \geqslant 0$
p	multiplier	$p \geqslant 0$
c	increment	$c \geqslant 0$
m	modulus	$m > R_j$, $m > p$, $m > c$

An example is provided by choosing $R_0 = p = c = 7$ and $m = 10$. The resulting sequence is 7,6,9,0,7,6,9,0,... which is periodic with period four. Obviously this sequence does not produce pseudo-random numbers of good quality. Reference 18 gives recommendations for selecting values to produce a pseudo-random number generator of high quality. For our purposes, we will use

$$\begin{aligned} R_0 &= 0 \\ p &= 12{,}621 \\ c &= 21{,}131 \\ m &= 100{,}000 \end{aligned}$$

Given a number R, the next number is computed by

```
        R    =    REMDR(R * 12621 + 21131,100000)
```

Usually, random numbers are desired within some range that is small compared with the value of *m* given above (in fact, *m* should be chosen to be large compared with the maximum number desired). Therefore, the numbers are scaled down. Suppose numbers are desired in the range of 0 to $n - 1$. For the sequence above, the best results are obtained by multiplying the number generated by *n* and then truncating the result. A complete random number generator is:

```
        DEFINE('RANDOM(N)')
                   .
                   .
                   .
RANDOM RANVAR  =   REMDR(RANVAR * 12621 + 21131,100000)
       RANDOM  =   RANVAR * N / 100000       :(RETURN)
```

This procedure illustrates an important aspect of generators: the modification of the value of a global variable (here RANVAR). The value returned is computed from the new value of the global variable. The value upon which the computation is to be made cannot be provided as an argument; the function can only change the value of a formal argument or local variable during the period in which it is invoked. Once a function returns, the former values of all such variables are restored. When a function such as RANDOM requires a global variable, care must be taken to assure that this variable is not modified inadvertently elsewhere in the program. Usually the name for a global variable can be chosen to minimize the likelihood of such an error. Note also that the previous value of RANDOM cannot be used to compute the next value. The mapping from RANVAR to RANDOM is many-to-one in general, and information is consequently lost.

2.3.2. Generation of Strings

Sometimes it is necessary to proceed systematically through a set of strings that have some property of interest. For example, one may wish to determine whether or not any one of a class of strings is contained within another string. In some situations, this kind of problem can be characterized in terms of a pattern that matches substrings with the specified property. In other situations, it may be necessary to generate all the strings of interest. SNOBOL4 is substantially more powerful in its analytic facilities than in its synthetic facilities. For example, only a few statements are required to construct a pattern that will match an algebraic expression. Writing a procedure to generate algebraic expressions in any generality is a much more difficult task. We will start here with some fairly simple cases and return to the problem of generation in more generality in Chapter 7.

An *n*gram is simply a string of *n* characters. Usually, the character set from which *n*grams are composed is restricted. Thus, 2grams (digrams) of letters might be of interest. Consider the problem of generating such strings systematically. A first attempt is given by a function GRAM(N,CSET) where N is the length of the *n*gram and CSET is a string of characters from which *n*grams (specifically, Ngrams) may be composed. A procedure is:

```
        DEFINE('GRAM(N,CSET,HEAD)C,TEMP')
        ONECH    =   LEN(1) . C
                  .
                  .
                  .
GRAM    OUTPUT    =   EQ(SIZE(HEAD),N) HEAD       :S(RETURN)
        TEMP    =   CSET
GRAM1   TEMP    ONECH    =                        :F(RETURN)
        GRAM(N,CSET,HEAD C)                       :(GRAM1)
```

A typical call is GRAM(5,'ABC') which provides

```
AAAAA
AAAAB
AAAAC
AAABA
AAABB
AAABC
AAACA
AAACB
AAACC
AABAA
  .
  .
  .
```

The key to this function is the third argument, HEAD, which does not appear in the specification of GRAM as given above. HEAD is null, by virtue of being omitted, when GRAM is called initially. GRAM builds up HEAD through successive recursive calls. If HEAD is not of the desired length (N), the loop at GRAM1 adds a character from CSET to HEAD and calls GRAM (recursively). Output occurs when HEAD is of the desired length.

The utility of a function like GRAM is quite limited. What is produced is an "*n*gram dump" without any possibility of intervention between successively generated *n*grams. While such "wallpaper" may be useful in gaining insight about the properties of some classes of strings, a more controlled generation, in the fashion of random number generation, is needed. Visualizing the *n*grams in the way used in Figure 2.2 for the generation of successive integers illustrates that the *n*grams can be considered to be strings in an ordered set. See Figure 2.3.

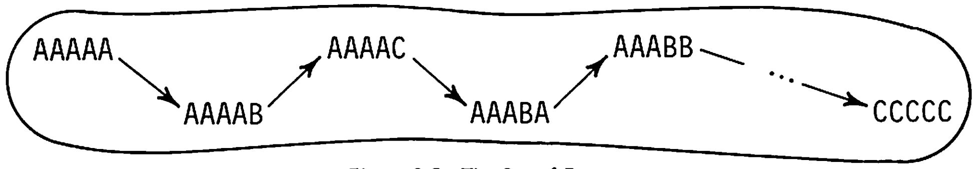


Figure 2.3 The Set of 5grams

Note that the first member of the set is AAAAA and the last is CCCCC. This choice is arbitrary, but natural, and uses the order of characters in CSET as a basis. Such a set of strings can be characterized by a successor function similar to that used for the generation of random numbers. A function NEXTG, for generating successive *n*grams, follows:

```
        DEFINE('NEXTG(GRAM)C')
        LAST   =   RTAB(1) . GRAM LEN(1) . C
        SCSR   =   BREAK(*C) LEN(1) LEN(1) . C
        CSET   LEN(1) . F
                 .
                 .
                 .
NEXTG   GRAM   LAST                          :F(FRETURN)
        CSET   SCSR                          :F(NEXTG1)
        NEXTG   =   GRAM C                   :(RETURN)
NEXTG1  NEXTG   =   NEXTG(GRAM) F            :S(RETURN)F(FRETURN)
```

Here ~~N,~~ F, and CSET are global variables, serving much the same function that RANVAR serves in random number generation. NEXTG, given an *n*gram as an argument, returns the next *n*gram as value, failing only if the last *n*gram is given as an argument. So long as the last character of the *n*gram is not the last character of CSET, the last character is replaced by its successor in CSET. When the last character is encountered, NEXTG is called recursively with the first N - 1 characters as an argument, and the first character of CSET, F, is appended to the result. This type of formulation is possible, where it was not possible in RANDOM, because all the information necessary to compute the next value is contained in the current value.

While the function above fails if given the last *n*gram as an argument, it is possible to make the generation circular, returning the first *n*gram if the last *n*gram is given as value.

EXERCISES

2.24 What limits the size of the modulus, increment, and multiplier in the function RANDOM?

2.25 In most uses, a random number generator is associated with some process. In some situations, there are several concurrent, but independent, processes, each of which requires an independent sequence of random numbers. Devise a method for implementing this without writing separate random number generating procedures.

2.26 Write a function that generates a sequence of randomly selected characters.

2.27 Write a function that generates random strings whose lengths are randomly distributed between 0 and *n*.

2.28 Introduce an element of randomness into the play of the tic-tac-toe program developed in the solution of Exercise 1.23.

2.29 Specify an appropriate default value for the second argument of GRAM.

2.30 Write a function to generate the set of all alphabetic strings.

2.31 Write a function that returns the successor of any string.

2.32 Write a function that generates palindromic strings.

2.33 Modify NEXTG to make it return the first *n*gram, given the last.

2.34 Write a function to shuffle a deck of playing cards. Choose a representation for the cards that is mnemonic.

2.35 One method of forming acronyms is to select the initial letter of each word of a phrase. A more general method is to select any characters from the phrase, provided that the letters occur in order in the phrase. An example is: StriNg Oriented symBOlic Language. Write a function to generate acroynms in this way.

2.4. USE OF FUNCTIONS IN PROGRAM DESIGN

One of the natural results of the systematic use of defined functions is program modularization. The advantages of modularization are well known, especially with respect to the ease of program modification and maintenance. Thinking of the components of a program as functions may also provide a substantial additional benefit in terms of problem solving and implementation of the program. Separating a program into functional components requires sorting out ideas, isolating and specifying operations, and generally getting a clear logical understanding of relationships.

As an example, consider the problem of generating a simple concordance that lists the line numbers of a text in which each word appears. With only this statement, the problem is poorly defined, as is typical of most problems. One of the first steps in the solution is to determine what operations have to be performed and how the data is to be organized.

We may presume that the text to be processed is on a sequential data file, since the problem mentions line numbers. There is, however, no indication of how these lines have been prepared. Assuming that the text is prepared in a manner closely resembling printed text, there is still the question of how the text is split between successive records. Perhaps a long stream of text has been split into records at regular intervals. On the other hand, the text may have been prepared with an explicit continuation convention that indicates when a word is continued on the next record. The easiest situation to handle is when the text is prepared so that words are not split between records at all. Each record then ends with blanks if there is not enough room for the next word.

The definition of a word is, in itself, a substantial problem. A simple approach is to define a word to be a span of letters. An alternative approach is to consider words to be whatever occurs between blanks and punctuation marks. Then questions arise about hyphenated compound words, abbreviations, various nonstandard constructions, and so on. In any event, a precise definition of a word is a hopeless task; the problem is to make a definition suitable for a particular situation and to write a program that will break up the text accordingly. The method used in Section 1.1.2 will be followed here.

Developing a concordance requires some type of symbol table mechanism. Since a word may appear several times on several different lines, it is necessary to be able to look a word up in a table in order to add a new line reference. Since the words in the text cannot be determined in advance, the mechanism must be general enough to accommodate any specific words and any number of different words that may occur. The `TABLE` data type is designed for just this kind of situation.

After all the text has been processed, the table will contain lists of line numbers associated with each word. Printing the results remains. Again the problem is vague. In what order should the results be printed? A natural order is alphabetical. Other possible orderings are by first occurrence, by length, by number of occurrences, and so on. It is reasonable to expect that ordering the table will present programming problems.

The discussion above is rather detailed; more detailed than similar discussions elsewhere in this book. Such a discussion is typical of the first thoughts about solving a programming problem. The reason for going to such length here is to study an approach to program development rather carefully. It is not uncommon for a programmer to start with a poorly defined problem and start programming at once. This is particularly likely if the problem is apparently as simple as the one given above. Some difficulties are resolved naturally in the course of programming, but others get swept under the rug. Still other problems may not be recognized until the program has been run on a variety of data. Unless substantial care and effort are taken to assure flexibility, the program may have limited utility or may require substantial rewriting if requirements are changed. Suppose, for example, that an alphabetic listing is programmed, but that it is later decided that the listing should be by number of occurrences.

Many such difficulties can be avoided if the problem is broken down into functional components, isolating operations logically and physically, and thus making debugging and revision easier. In the problem above, several functions can be specified, with operations as follows:

(1) GET: a function that returns the next word from the text each time it is called, and fails when the text is exhausted. Input and output of text and the assigning of line numbers are the responsibility of GET.
(2) CITE: a function that updates the symbol table, adding the association of a line number with a word.
(3) SORT: a function that orders the symbol table in the desired way.
(4) PRINT: a function that prints the symbol table in the desired format.

These four functions amount to specifying the *vertical* organization of the program. Depending on specific approaches, some of these functions may call other functions, providing *horizontal* organization. The concordance program, stripped of initialization and lacking procedures for the functions above, is quite simple:

```
NEXT   CITE(GET(),LINE.NO,T)                    :S(NEXT)
       PRINT(SORT(T))
END
```

Here T is presumed to be the symbol table containing the words and line-number references. This program may appear somewhat mysterious and could give the impression of $\overset{e}{\text{slight}}$ of hand, since none of the functions has yet been written. Nevertheless, a person with some experience in writing programs in SNOBOL4 will recognize that the four functions can be written without great difficulty, and that they are substantially independent of one another. We will go through the programming details to provide an illustration that the overall program organization can be addressed first and details can be left until later.

GET presents the greatest conceptual problems. This function must handle any problems concerning the representation of the text and must isolate the words. As a start, we will take the simplest of the alternatives for the form of the data and assume words are not broken over record boundaries. The more difficult cases are left as exercises. An important aspect of program organization is isolating the input of text within GET. GET can be thought of as a generator in the sense used in the preceding section. Each time GET is called, it returns a word, regardless of the form of data on the text file and, in fact, independently of the existence of a data file at all. Words might as well be coming from a word generator. When the source of words is exhausted, GET fails. In the program being developed here, GET will operate on a global variable which is the last line read. GET will, in general, be called several times to generate the words from a line of input. Only when a line is exhausted is another line read. The procedure for GET is:

```
       DEFINE('GET()')
       WORD   =   BREAK(LETTERS) SPAN(LETTERS) . GET
                     .
                     .
                     .
GET    LINE   WORD   =                            :S(RETURN)
       LINE.NO   =   LINE.NO + 1
       LINE   =   INPUT                           :F(FRETURN)
       OUTPUT   =   LINE LINE.NO                  :(GET)
```

LINE and LINE.NO are global variables. GET prints the text, numbering the lines, to provide the initial part of the listing that results from executing the program. Observe that GET is self-initializing. LINE is null when GET is called the first time, and LINE.NO is zero (or null). Therefore, WORD fails to match. As a result, a record is read.

CITE is a simple function, so simple that it is unnecessary except for organizational purposes.

```
       DEFINE('CITE(WORD,N,TABLE)')
                     .
                     .
                     .
CITE   TABLE<WORD>   =   TABLE<WORD> N ','    :(RETURN)
```

A list of line numbers is built up using commas as separators. Note that TABLE and N are formal arguments. This is not necessary, and is largely a matter of style. The table and line numbers could be global variables and could be transparent to the main program. The advantage of including them as formal arguments is that the functions developed in this program can be used in other contexts where these values might have other meanings. Ideally it would be desirable to have LINE.NO more visible, but our organization precludes that, since a function can return only a single value unless tricks are employed.

SORT is a trivial or a substantial problem, depending on what kind of sorting is to be done. In the spirit of starting with a simple solution to get the program running and later making revisions to handle more realistic situations, a sort by order of first occurrence will be used.

```
         DEFINE('SORT(TABLE)')
                        .
                        .
                        .
SORT     SORT    =   CONVERT(TABLE,'ARRAY')       :S(RETURN)F(FRETURN)
```

The purpose of converting the table to an array is to produce a structure that can be easily accessed in an orderly fashion. The array produced by CONVERT is ordered by first occurrence simply because of the way that SNOBOL4 works. Such an ordering is called "chronological".

This procedure brings a new problem to light: If there are no words in the table, the conversion fails. This might happen because there is no input text, because there are no words in the text, or because of an error in the program. In any event, this situation must be recognized and handled properly. The failure return from SORT provides the necessary information to the calling statement without making any presumption that an empty table is erroneous. A minor modification of the main program will provide for an appropriate error message.

Unfortunately, chronological order is usually not sufficient, and a more elaborate sorting procedure is called for. Methods of sorting, like random number generation, have been the subject of a great deal of study. Such considerations are beyond the scope of this book. A simple sorting procedure follows for reference and use in test programs. The method used is a version of the Shell sort [19]. The interested reader is referred to Reference 20 for an extensive discussion of sorting. As a general rule, the sorting of a large number of items should not be attempted in a SNOBOL4 program, regardless of the specific method used. The time required may be astronomically large. Highly efficient system utilities are available for production sorting.

```
         DEFINE('SORT(TABLE)I,J K,G,N,M,T1,T2')
         ALEN    =   BREAK(',') . N
                        .
                        .
                        .
SORT     SORT    =   CONVERT(TABLE,'ARRAY')       :F(FRETURN)
         PROTOTYPE(SORT)   ALEN
         G   =   N
SORTG    G   =   GT(G,1) G / 2                    :F(RETURN)
         M   =   N - G
SORTK    K   =   0
         I   =   1
SORTJ    J   =   I + G
         LGT(SORT<I,1>,SORT<J,1>)                 :F(SORTI)
         T1   =   SORT<I,1>
         T2   =   SORT<I,2>
         SORT<I,1>   =   SORT<J,1>
         SORT<I,2>   =   SORT<J,2>
         SORT<J,1>   =   T1
         SORT<J,2>   =   T2
         K   =   K + 1
SORTI    I   =   LT(I,M) I + 1                  :S(SORTJ)
         GT(K,0)                                :S(SORTK)F(SORTG)
```

The difficulty in writing the procedure for PRINT depends, again, on how elegant the solution is. A very simple procedure is sufficient for a first attempt.

```
         DEFINE('PRINT(A)I')
                        .
                        .
                        .
PRINT    I   =   I + 1
         OUTPUT   =   A<I,1> ' : ' A<I,2>    :S(PRINT)F(RETURN)
```

To make the listing more attractive, a few lines to print identifying information and spacing can be added to the main program. The complete main program, with initialization, follows:

```
         LINE.NO   =   0
         LETTERS   =   'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
         WORD   =   BREAK(LETTERS) SPAN(LETTERS) . GET
         T   =   TABLE()
         DEFINE('GET()')
         DEFINE('CITE(WORD,N,TABLE)')
         DEFINE('SORT(TABLE)I,J,K,G,N,M,T1,T2')
```

```
       DEFINE('PRINT(A)I')
       OUTPUT  =   'LISTING OF TEXT';   OUTPUT  =
NEXT   CITE(GET(),LINE.NO,T)                   :S(NEXT)
       OUTPUT  =;   OUTPUT   =   'CONCORDANCE';   OUTPUT   =
       PRINT(SORT(T))                          :S(END)
       OUTPUT  =   'NO CITATIONS'              :(END)
```

EXERCISES

2.36 Modify GET to handle input text in which
- (a) Words are broken arbitrarily at fixed intervals (i.e., words are split at the ends of records).
- (b) Words which are broken at the end of a line are indicated by hyphens (dashes) in the conventional manner used for typing.

2.37 Write a program to produce a concordance of *n*grams rather than words. Assume *n*grams consist only of letters, but provide a way of specifying the value of *n* at run time.

2.38 Write a program to produce a concordance of numbers that appear in a text.

2.39 Write a program to produce a concordance of punctuation marks.

2.40 Write a program to count the number of times each different word occurs.

2.41 For many purposes, words such as "a", "the", and so forth are of no interest in a concordance. Add a facility to the concordance program to suppress specified words.

2.42 Thinking of GET as a generator, use the concordance program as modified for Exercise 2.40 to tabulate randomly generated strings.

2.43 Modify CITE to avoid duplicate line numbers in cases where a word occurs more than once on a line.

2.44 Modify CITE so that only four-letter words are cited.

2.45 Write a program that tabulates word lengths.

2.46 Write a program to produce both a concordance of all words and also a list of words that begin with the letter A.

2.47 Write a program to produce concordances of assembly-language programs. Select a locally available assembly language.

2.48 Write a program to produce concordances of SNOBOL4 programs. Test the program on itself.

2.49 What are the reasons for putting the output statements for titling and spacing in the main program rather than in the functions GET and PRINT?

2.50 Generalize SORT so that the predicate used for sorting and the column on which the sort is performed can be specified when SORT is called.

2.51 Use the results of Exercises 2.40 and 2.50 to produce a listing of words in order of number of occurrences.

2.52 Expand PRINT to improve the appearance of the output listing as follows:
(a) Remove the terminal comma from the list of line numbers.
(b) Indent the line number listings so that the words occupy columns 1 through 20 and the line numbers start at column 21. In case a line number listing is too long to print on one line, indent subsequent lines to column 21 also.
(c) Format the listing as for (b), but assure that the listings of line numbers are broken at commas, not in the middle of numbers.
(d) Provide a way of specifying the columns for formatting and the width of the listing.

2.53 Write a function to sort a hand of playing cards. See Exercise 2.34.

3 STRUCTURES

Data to be manipulated by a program frequently is complex, and there may be a variety of relationships among parts of the data that must be represented. Many kinds of processing require storing and accessing data in a structured way according to specified rules.

SNOBOL4 provides a number of ways of structuring data. An array is one of the most commonly used structures. Elements of an array are referenced by integer subscripts and an array represents a rigidly structured aggregate of values. Other structures are more loosely organized or have different facilities for referencing their values. In addition to the built-in data types of SNOBOL4, new data types can be created. Even more important is the ability to link objects together to form structures that are more complicated.

This chapter begins by discussing the kinds of values and objects that are available in SNOBOL4 and progresses to a discussion of structures that can be built from the basic components.

3.1. VALUES AND OBJECTS IN SNOBOL4

For the most part, programming in SNOBOL4 can be done with only a vague idea of how the various features are implemented. In order to be facile in the manipulation of structures, however, it is necessary to have a good grasp of the relationships among objects and how they are created and modified. A method for visualizing relationships, with the aid of diagrams, is particularly helpful. The discussion that follows is somewhat of a digression, but it is included because misunderstandings are common. It is designed to provide a system for depicting such relationships which parallels an actual implementation of SNOBOL4 [21] fairly closely, but suppresses unnecessary detail. There are, of course, many alternative ways of implementing SNOBOL4. The system that follows is adequate and self-consistent, even if it does not correspond, in all cases, to every implementation.

3.1.1. Built-In Data Types

All structures that can be created in a program are necessarily constructed out of the objects that can be created by language operations. SNOBOL4 has a variety of built-in data types and the ability to define others during execution. Every kind of value that can occur in a program has a distinguishable data type. For the purposes of visualization, values can be thought of as occupying cells, and a cell can be thought of as having two parts: a data type designation and a representation of the value itself. There are two basic representations for values: numbers and pointers. The values of integers and real numbers are numbers. Figure 3.1 illustrates the integer 734 and the real number 2.0.

Figure 3.1 Integer and Real Number Values

The letters I and R indicate INTEGER and REAL data types respectively. A similar method of abbreviation is used for other data types.

The values of all other types of data in SNOBOL4 are pointers to objects. A pointer is an address that identifies the location of the object in the memory of the computer. The term pointer is used because of the connotation that suggests an arrow, which in turn provides a convenient way of diagramming the relationship. Figure 3.2 indicates how this may be visualized for an array of five elements.

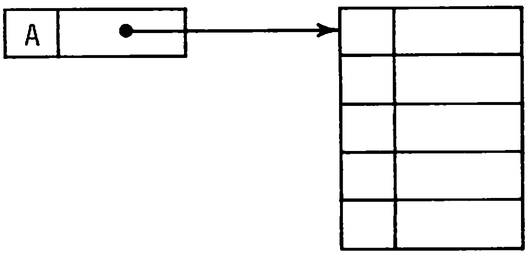


Figure 3.2 An Array Value

The cell at the left represents the array value. The arrow is a pointer to the location where the five array elements actually exist.

Arrays

As indicated in Figure 3.2, each element of the array is, in itself, a cell. Consider the following statement:

```
    X   =   ARRAY(5,0,0)
```

Figure 3.3 illustrates the data resulting from executing this statement.

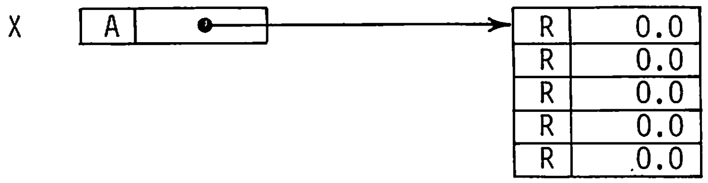


Figure 3.3 An Array

The symbol X indicates that the value of X is a cell that points to an object consisting of an array of five elements. In this situation, X is a variable, which may be thought of as the name of a location where the value is placed.

It is important to understand that the value assigned to X by the statement above is *not* a block of five cells, but rather a pointer to such a block. This concept of a cell containing a *pointer* makes it possible for all values to be represented uniformly by a single cell. If the object represented by the value requires more space than is available in a cell (as in the case of an array), the object is located elsewhere and is pointed to. Now consider the statement

```
    Y  =  X
```

The relationships established by this statement are shown in Figure 3.4.

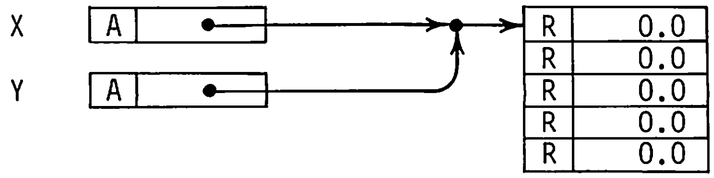


Figure 3.4 Two Variables with the Same Value

Assignment simply copies the cell that is the value of X and stores it at Y. Both X and Y point to the *same* object. Assignment does not copy the object that is pointed to. In fact the (cell) values of X and Y are *identical*, and the predicate IDENT(X,Y) succeeds. The fact that X and Y point to the same object leads to an important concept in manipulating structures. If an element of the array pointed to by X is changed, this change is shared by Y. For example,

```
X<2>   =   4.5
```

produces the situation shown in Figure 3.5.

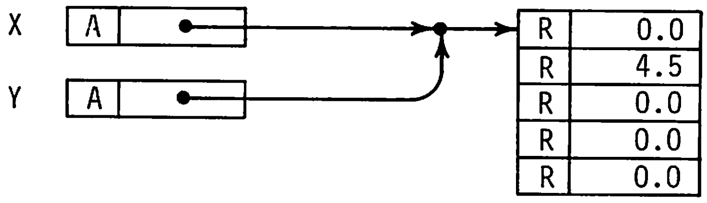


Figure 3.5 A Change in the Array

Although the value of Y itself is not changed (it still points to the same object), part of the object it points to has been changed. This kind of effect must be kept in mind when more than one variable points to the same object. Note the essential difference between the situation illustrated above and the result of executing the statements

```
X   =   ARRAY(5,0.0)
Y   =   ARRAY(5,0.0)
```

as shown in Figure 3.6.

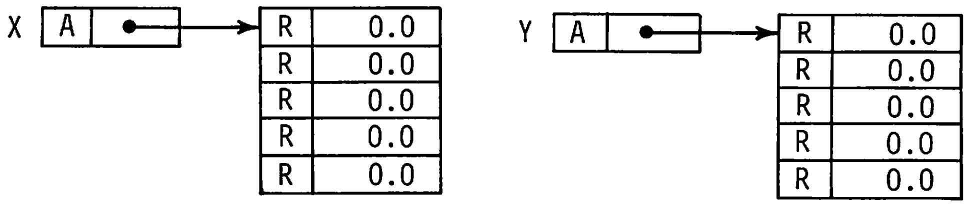


Figure 3.6 Two Separate Arrays

Although the two calls of ARRAY are identical, two separate and totally unrelated arrays are created. A change in an element of X has no effect on Y, and vice versa. Similarly, IDENT(X,Y) fails. ARRAY is a function that creates objects. Every time that ARRAY is called, it creates a new object, distinct from any other object.

The built-in function of COPY makes a copy of an object. The statements

```
X  =  ARRAY (5,0.0)
Y  =  COPY (X)
```

produce the same structures as shown in Figure 3.6.

Strings

String values are also represented by pointers to objects. This representation is partly due to the fact that strings can be very long and hence cannot in general be stored in a single cell. Another reason for the use of a pointer to an object is that strings also serve as variables in SNOBOL4 and can have values. Consider the statement

```
     COLORATION  =  'BEIGE'
```

The value can be pictured as shown in Figure 3.7.

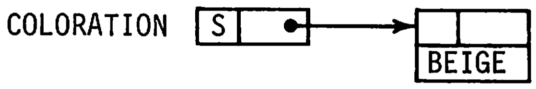


Figure 3.7 A String

The object pointed to has two parts: a cell (as yet unspecified) and the actual character string BEIGE. As pictured, BEIGE occupies one cell. The number of characters that fit into a cell varies from computer to computer. In the discussion here, we will assume that there are six characters per cell. In any event, it is convenient to think of data in terms of cells and to divide strings up into cells accordingly. COLORATION is also a string, but it has a value as well. The first cell in the object for a string contains the value that the string has if it is used as a variable. The statement above produces the situation shown in Figure 3.8.

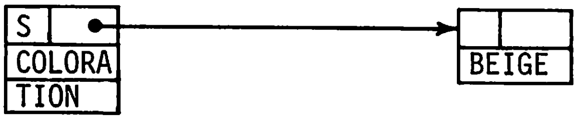


Figure 3.8 Strings as Variables and Values

In SNOBOL4 all nonnull strings have the capability of being used as variables, even if such use is not explicit in the program. The indirect referencing operator provides a dynamic means of using a string as a variable. Thus,

```
     $COLORATION  =  12
```

produces the result shown in Figure 3.9.

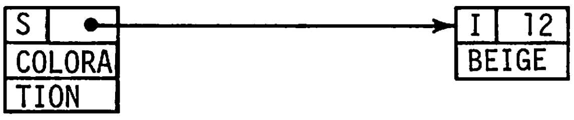


Figure 3.9 Strings as Variables

A string that is not given a value explicitly has the null string as value by default. The null string is conveniently represented in a special way as shown in Figure 3.10.

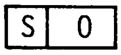


Figure 3.10 The Null String

The data type is indicated by S, but, instead of a pointer, the value is simply zero. This representation is more a matter of convenience than an essential feature of the null string. In this representation, the null string is the exception to the earlier statement that only integers and real numbers have values that are numbers.

Strings occur frequently in SNOBOL4, but for the most part their representation as objects with cells for values is unnecessarily cumbersome. A shorthand representation, retaining the essential concept of a pointer to an object, is useful. Figure 3.11 illustrates this convention, where braces indicate the object for the string.

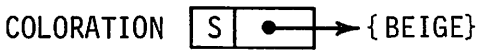


Figure 3.11 An Abbreviated Representation for a String

This abbreviated representation will be used except where it is necessary to emphasize variable-value relationships among strings.

Two other aspects of strings are important:

(1) Each distinct string is represented by a unique object.
(2) There are no operations in SNOBOL4 that change the internal structure of a string.

The uniqueness of strings is a consequence of the fact that any string can be used as a variable. To avoid inconsistencies and program malfunctions, each distinct string is represented by a unique object. Consider the statements

```
X   =   'A' 'LOOPZ'
Y   =   'ALOOP' 'Z'
```

The uniqueness of strings means that the results of these statements can be visualized as shown in Figure 3.12.

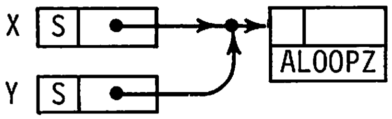


Figure 3.12 The Uniqueness of Strings

No matter how a string is created, it is always represented by the same object. (The reader who is interested in how this is accomplished should consult Reference 21.) Referring to the abbreviated notation, a string in braces represents a unique object.

Earlier in this section we illustrated a situation in which part of an object used to represent an array was changed. There is no operation in SNOBOL4 that can modify the object representing a string. A replacement statement gives the appearance of doing so, but this is *not* the case. A replacement statement may create a new string, but it cannot change an existing one. The statement

```
X  '00'   =   'U'
```

produces the situation shown in Figure 3.13.

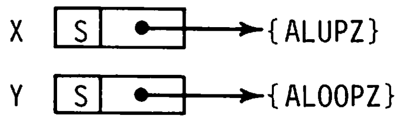


Figure 3.13 The Result of Changing the Value of X

The entire value of X is different as a result, and X and Y now point to two different strings.

Tables

The function TABLE creates objects in a fashion similar to ARRAY. Tables are not as easy to visualize as arrays, since tables are not fixed in size and the elements in tables are not ordered numerically as they are in arrays. One way to visualize a table is as a block of cell pairs. Suppose the following statements are executed.

```
INDEX   =   TABLE()
INDEX<'LANGUAGE'>   =   10
INDEX<'COMPOSITION'>   =   15
```

The table INDEX can be visualized as shown in Figure 3.14.

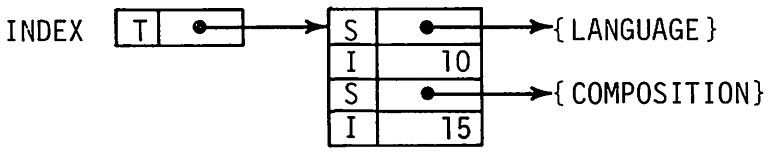


Figure 3.14 A Table

Note that the value of INDEX is a pointer to the table itself. As other items are referenced and values are specified, the table grows and is modified accordingly. Figure 3.15 shows the results of executing the additional statements

```
INDEX<'LANGUAGE'>  =  INDEX<'LANGUAGE'>  + 1
INDEX<'AUTHOR'>   =  8
```

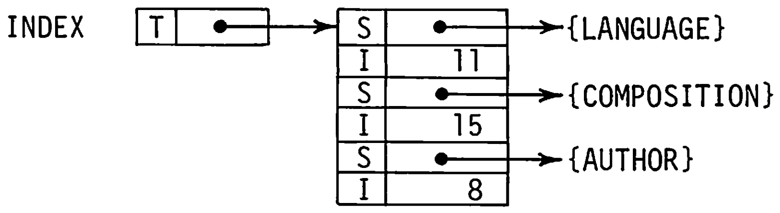


Figure 3.15 An Enlarged and Modified Table

Other Data Types

Other built-in data types have other forms. For the most part, these forms are not relevant, both because there is no need to deal with such data types as objects and because the internal structure of such objects is complex and highly implementation-dependent. It has become customary, for example, to draw patterns as clouds [22], suggesting a structure that is vague and poorly understood. Figure 3.16 illustrates how this representation might be used to represent the results of executing the statement

```
BC   =   BREAK(',')
```

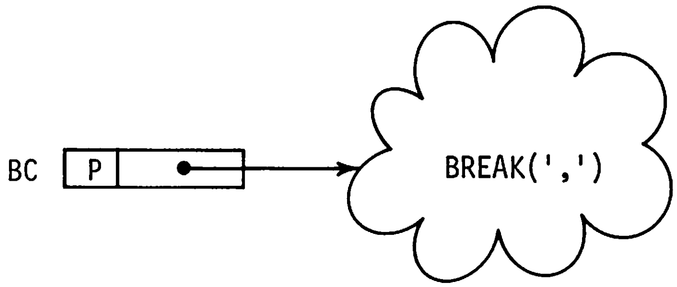


Figure 3.16 Representation of a Pattern

If a pattern is represented as a cloud, who knows how CODE might be visualized (but see Reference 21)!

Some remarks about one other data type may prove helpful to some readers. In all the data types discussed above that point to a block of cells, the pointer points to the top cell. This is somewhat a matter of convention, and emphasizes, if subtly, that cells in a block have an inherent order.

Mechanisms that process such blocks are always given pointers to the same relative position in the block (the top) by convention.

There is a data type that may point to a cell in the interior of a block: the NAME data type. Consider the following statement with respect to the value of X shown in Figure 3.3.

```
E   =   .X<4>
```

The result is shown in Figure 3.17.

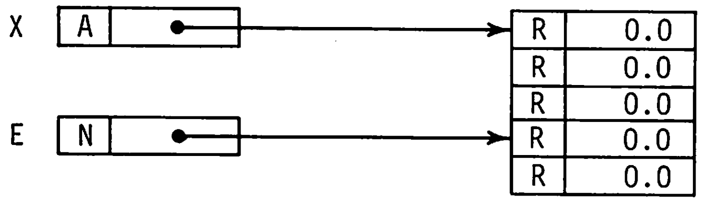


Figure 3.17 A NAME

The value of E designates a particular element of the array value of X and hence a particular cell in the object pointed to by X. A statement such as

```
$E   =   3.1
```

produces the effect shown in Figure 3.18.

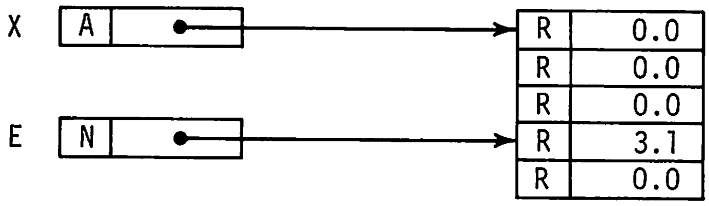


Figure 3.18 Modification of a Value in an Array

3.1.2. Defined Data Types

Defined data types are particularly useful in creating structures. The DATA function adds new "data types" to SNOBOL4, defines creation functions for these new types, and provides field functions for accessing them. The quotation marks above are used to indicate that the data types created by DATA are not as distinctly different as the built-in data types. In fact, the data types created by DATA form a class whose members have similar representations, differing only in their names, the names of their field functions, and the number of fields. Consider the statements

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

The first statement defines the object creation function NODE and the three field functions. The second statement creates a "NODE". The result may be visualized as shown in Figure 3.19.

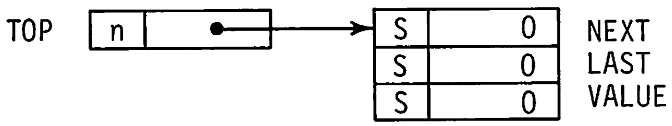


Figure 3.19 A Defined Object

The NODE consists of three cells, one for each field. The values of these fields are null strings, since no field values are given explicitly in the call to the function NODE. The result assigned to TOP is a cell pointing to a NODE. The abbreviation n is used to indicate that the data type of the object pointed to is NODE.

Values may be assigned to fields of defined objects when the object is created or by assignment to field references. An example is given by the statements

```
A   =   NODE(,,'MANDATE')
B   =   NODE(,A,'STYLE')
C   =   NODE(,B,'PROBLEM')
NEXT(A)   =   B;   NEXT(B)   =   C
```

which create the structure shown in Figure 3.20.

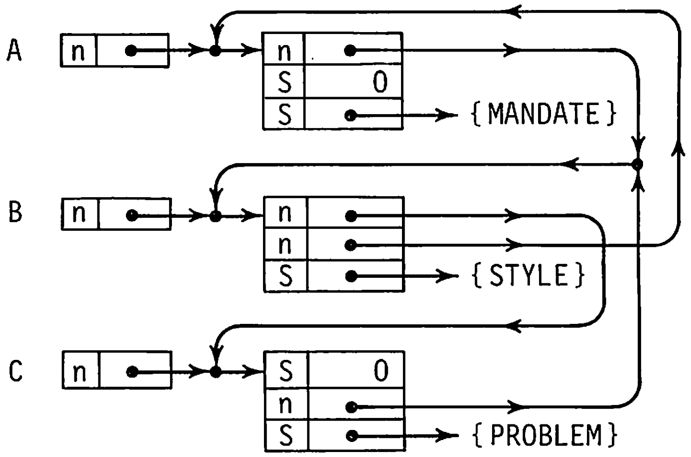


Figure 3.20 A Structure of NODEs

This figure illustrates several important facts. A statement such as

```
NEXT(A)   =   B
```

assigns to the NEXT field of A a cell that is identical to the cell at B. It is important to understand that such a statement does *not* copy the object pointed to by B. In fact IDENT(NEXT(A),B) succeeds because the two cells are identical—both have the same data type and the same pointer. A reference to the value of a variable such as B simply yields the cell at B, regardless of what the cell contains. There is often some confusion on this point when structures are created using defined data objects. The difference between a cell pointing to an object and the object itself must be clearly understood.

The example above also illustrates that NODE, like ARRAY and TABLE, is a function that creates objects. Every time NODE is called, a new object is created. The three calls of NODE in the example above create three separate NODEs. Similarly, NODEs are created *only* when NODE is called. The statement

```
        AGGREGATE   =   ARRAY(4,NODE())
```

creates the structure shown in Figure 3.21.

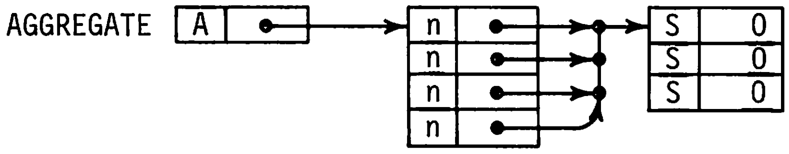


Figure 3.21 An Array with a NODE Value

Note that only one NODE is created because NODE is only called once—as the second argument of ARRAY. All the array elements point to this one NODE. Failure to understand this aspect of object creation is a common cause of error. An array of four distinct NODEs is created by the following statements:

```
        AGGREGATE   =   ARRAY(4)
        I   =   1
NLOOP   AGGREGATE<I>   =   NODE()          :F(DONE)
        I   =   I + 1                      :(NLOOP)
DONE
                 .
                 .
                 .
```

The result is shown in Figure 3.22.

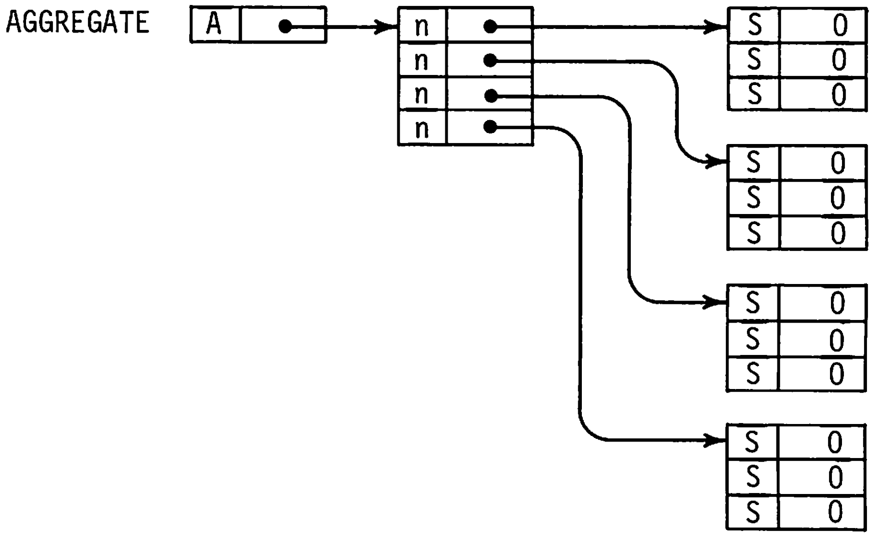


Figure 3.22 An Array of Four NODEs

The visualization of defined data objects as blocks of cells is consistent with the representation given for other objects. There are several drawbacks to this method. It is difficult to arrange the objects and draw the pointers in a way that makes the relationships clear (as evidenced by Figure 3.20) because the cells are necessarily adjacent and the orientation of the objects is fixed. In addition, there may be many types of defined data objects; the data-type abbreviations may be unambiguous, but it is, nonetheless, difficult to distinguish the data types easily by a glance at the drawing. More important, the data type of an object can only be determined from a pointer to the object, not from the object itself. For these reasons, a simplified and more diagrammatic method will be used for representing defined data types in most of the figures that follow. This method is based on assigning a shape to a defined data type and on partitioning this shape into fields in such a way that the fields can be unambiguously distinguished regardless of the orientation of the shape. For example, a NODE might be assigned the shape shown in Figure 3.23.

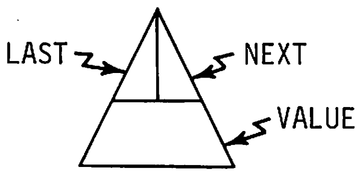


Figure 3.23 A Shape for a NODE

NODEs can be drawn straight up, lying on their sides, or standing on their heads, and the different fields can still be distinguished. Figure 3.20 can now be redrawn as shown in Figure 3.24.

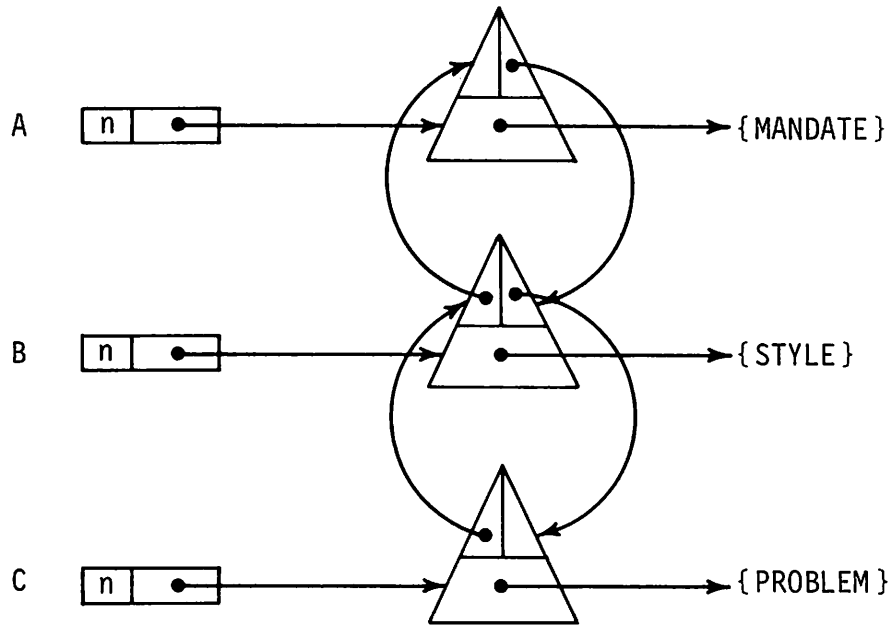


Figure 3.24 A Structure Composed of NODEs

In such diagrams, the field cells are distorted to fit the chosen shape, and the data type abbreviations are discarded. As long as the data type can be distinguished by shape (as in the case of NODEs) or by its form (braces for strings, for example), there is no ambiguity. An empty field is used to represent the null string. The purpose of such diagrams is to make the structure easier to understand, not to provide a complete and consistent system of notation. Consider the following statements, using the COMPLEX data type discussed in Section 2.1.2.

```
X   =   NODE(,,COMPLEX(2.0,3.7))
Y   =   NODE(,X,COMPLEX(-4.7,0.0))
NEXT(X)   =   Y
```

The resulting structure is shown in Figure 3.25. Rectangles are used for COMPLEXes, a minor violation of the statement above that shapes would be chosen so that fields could be determined independently of the orientation of the shape. We simply assure the reader that rectangles are not drawn upside down—which is quite reasonable for COMPLEXes.

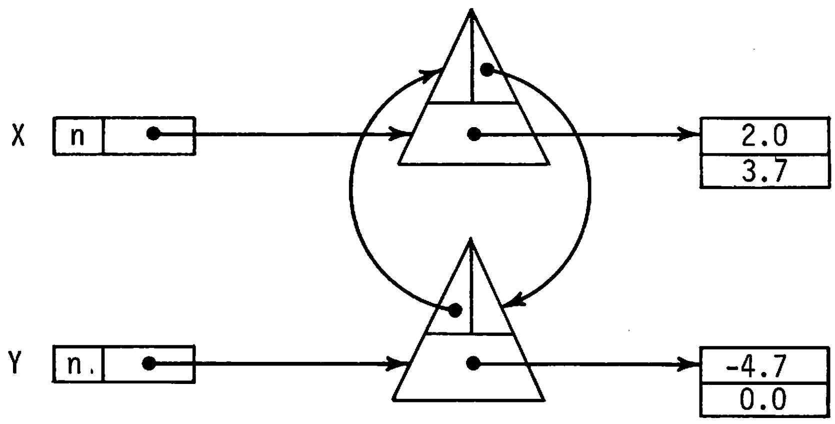


Figure 3.25 A Structure of NODEs and COMPLEXes

Data type indications have been dispensed with in the COMPLEXes, according to the rules outlined above, since the values are self-identifying.

3.2. IMPLEMENTATION OF SOME SPECIFIC STRUCTURES

3.2.1. Stacks

One of the most commonly used information structures is the stack. Stacks are also called pushdown lists and last in-first out (*lifo*) lists. The term stack is derived from the fact that objects are put on and taken off a stack from the "top". "Pushdown" is a descriptive term derived from devices used in restaurants where plates are stacked on a spring-loaded platform. Only the top plate is accessible. As plates are added, the stack settles into a recessed area and as plates are removed, new ones come to the surface as the weight on the spring diminishes. It is the way that operations are performed on stacks—by adding and removing from the top only—that characterizes this type of information structure. That is, it is not "proper" to slip an item into the middle of a stack, or to take an item off the bottom. The term lifo refers to the order in which items are serviced in a stack. The last object put onto a stack is necessarily the first object taken off.

Stacks are commonly used in programming contexts where recursive operations are involved. It is the nature of recursion that the most recently saved information is the first to be restored. For example, SNOBOL4 uses a stack internally for implementing recursive processes.

Programming languages provide no direct physical analogy to the spring-loaded restaurant cart or, in fact, to a stack of plates. Nevertheless, operations analogous to those described above can be simulated.

There are usually two operations that are performed on a stack: "pushing" an object, that is, placing it on the top of the stack, and "popping" an object—removing it from the top of the stack. These two operations can be characterized by functions:

```
PUSH(Y)
Z    =    POP()
```

The first operation places the value of Y on the top of the stack. The second operation removes the top item from the stack and assigns it to Z. POP fails if the stack is empty. An empty stack is not necessarily an error condition. It may simply indicate the lack of stored information. Generally speaking, PUSH should not fail.

There is no reason why there can only be one stack at any given time. Although one stack is sufficient for most situations, more generality is desirable. Different stacks can be associated with different identifiers such as S1, S2, and so forth. Stacks do not simply exist; they must be created, as illustrated by the following statements:

```
S1    =    STACK()
S2    =    STACK()
```

In this sense, STACK is a creation function similar in concept to ARRAY and TABLE. Of course, the function STACK is not built-in, but must be defined.

Since a stack is a structure, the operations of pushing and popping are applied to specific stacks. The previous functions might have an additional argument:

```
PUSH(Y,S1)
Z    =    POP(S2)
```

These operations may now be read "push the value of Y onto stack S1" and "pop the top value off stack S2 and assign it to Z". In a situation where there is only one stack whose name is known globally, the extra argument could be omitted and a suitable default assumed. In the sections that follow, the stacks operated on will be specified explicitly.

The three functions STACK, PUSH, and POP are sufficient to characterize stacks. If these three functions are written, stacks can be used in a program.

Rarely in a programming language is there only one way to do something. Frequently there are many ways, and the best choice is not clear. The material that follows describes several ways of implementing stacks, and discusses the relative advantages and disadvantages of each. As a result of this discussion, some general rules will appear that are applicable to a variety of other types of structures.

String Implementation

One way to implement a stack is to use a string for representing the data, selecting some character, such as the comma, as a marker to separate the items. If S is a stack, pushing a value V onto S can be done by

```
        S   =   V ',' S
```

Similarly, an item is popped off the stack by a statement such as

```
        S   BREAK(',') . V LEN (1)   =
```

If these statements are used in writing the functions PUSH and POP, a problem immediately becomes evident. A call of the form PUSH(V,S) does not change the value of S. A way around this problem would be to redesign the form of the pushing operation to return a value that is then reassigned to the variable which is the name of the stack. Pushing would have to be written as follows:

```
        S   =   PUSH(V,S)
```

If, however, two variables point to the same stack, changing one in this fashion would not change the other. A more attractive alternative, and one that does not require changing the original design, is to place an intermediate linkage between the variable and the value. This can be done conveniently and with elegance by using a defined data type as follows:

```
        DATA('STACK(VALUE)')
```

This defined data type provides a ready-made stack creation function and also gives a stack the data type STACK. The functions PUSH and POP now can be written:

```
        DEFINE('PUSH(V,S)');   DEFINE('POP(S)')
        NEXTI   =   BREAK(',') . POP LEN(1)
                    .
                    .
                    .
PUSH    VALUE(S)    =   V ',' VALUE(S)       :(RETURN)

POP     VALUE(S)   NEXTI   =                 :S(RETURN)F(FRETURN)
```

The advantages of this method of implementation are:

(1) There is no intrinsic limit to the size of a stack (although string-length limits apply).
(2) The stack functions are simple.

(3) There is no necessity to reserve space for the stack; space is provided as needed when items are pushed onto a stack.

The disadvantages are:

(1) Some character must be reserved as a marker to separate items and hence cannot appear within an item. (This problem can be overcome at the expense of additional complexity. See Exercise 3.2.)
(2) String processing is relatively slow compared with other operations in SNOBOL4. Long strings, in particular, are costly to manipulate.
(3) Items can only be strings or types of data that can be converted to strings without essential loss of information. For example, a pattern cannot be stored in this kind of stack.

Actually, the use of strings to implement stacks is not a good method. Some programmers tend to use strings to represent all types of structures, finding a method of encoding structural relationships in strings by using various syntactic devices. More sophisticated use of SNOBOL4 language features produces better results, as is illustrated by the following sections. The inability to handle items of various data types is a fatal flaw, and is alone sufficient reason for seeking other methods.

Array Implementation

A logical approach to allowing any type of data to be stored on a stack is to use an array. The elements of an array can have any data type, and incrementing and decrementing an index naturally corresponds to pushing and popping. Figure 3.26 illustrates how an array implementation of a stack might look.

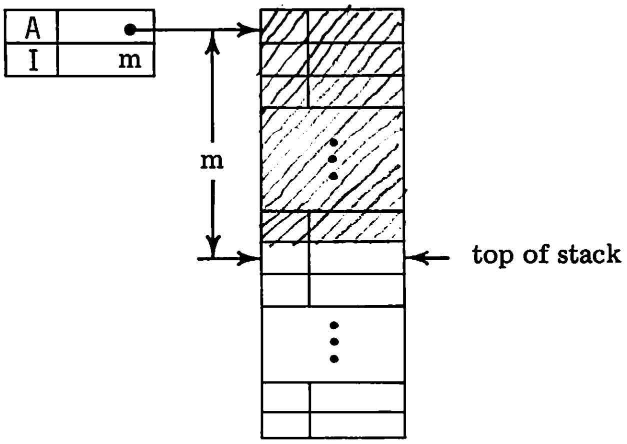


Figure 3.26 An Array Implementation of a Stack

Again there is a defined data object serving as an intermediate linkage, this time to an array. Since creating a stack now requires allocation of an array, the object creation function itself cannot be used to implement the stack creation function STACK. The complete set of functions follows.

```
        DATA('STK(LIST,INDEX)')
        DEFINE('STACK(N)')
                   .
                   .
                   .
STACK   STACK   =   STK(ARRAY(N),1)               :(RETURN)

PUSH    ITEM(LIST(S),INDEX(S))   =   V            :F(FRETURN)
        INDEX(S)   =   INDEX(S) + 1               :(RETURN)

POP     INDEX(S) = GT(INDEX(S),1) INDEX(S) - 1 :F(FRETURN)
        POP   =   ITEM(LIST(S),INDEX(S))          :(RETURN)
```

The advantages of this method of implementation are:

(1) Access to the stack is efficient and natural.
(2) Space is allocated only when a stack is created, not every time an item is pushed.
(3) Most important, any kind of object can be stored in the stack.

The disadvantages are:

(1) Space must be allocated for the stack even if it is never used.
(2) The amount of space allocated is fixed and does not increase automatically. A failure exit is provided accordingly. It is not a difficult matter, however, to provide for stack extension.

Table Implementation

The problem of the fixed size of the stack for the array implementation suggests the use of a table instead. The function STACK could simply allocate a table rather than an array. A table can be indexed by integers in the same way that an array is indexed, so the stack manipulation functions used for the array implementation apply to the table implementation substantially unchanged. The advantage of this method is that the stack increases in size automatically as the index is increased. Stack overflow does not occur.

To understand the disadvantages of this method, some knowledge of the way that tables are implemented is required. In summary:

(1) Access to the stack is less efficient.
(2) For a large stack, the table implementation requires more space—approximately twice as much as the array implementation.
(3) Although more space is allocated as needed, this space is not released, even if the stack becomes empty.

The advantage of automatic growth is outweighed by the disadvantages. In a sense, a table is not a natural way to implement a stack. Table references are associative, rather than numerical as in the case of arrays. The use of numerical subscripts is only a ruse to make it appear that the structure is being indexed numerically.

Defined Data Type Implementation

One way of conceptualizing a stack is based on the physical analogy given earlier. A stack can be thought of as a number of "plates" which are piled up by pushing and unpiled by popping. A defined data type PLATE is suggested:

```
DATA('PLATE(VALUE,LAST)')
```

Each PLATE has a value and a pointer to the previous plate on the stack. The arrangement of plates can be visualized as shown in Figure 3.27.

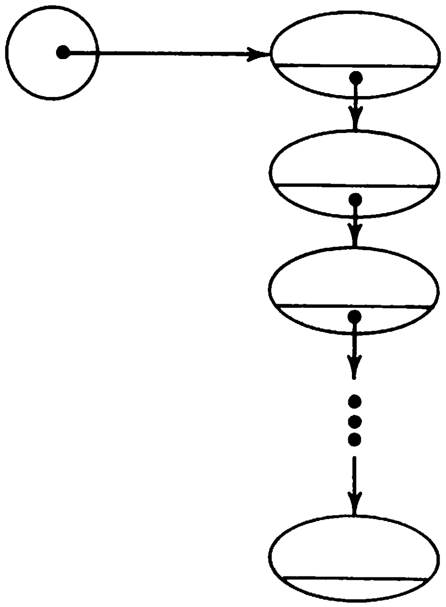


Figure 3.27 A Stack of PLATEs

As pushing and popping take place, PLATEs come and go. An empty stack contains no plates, so STACK can be implemented by the object-creation function. The complete set of stack functions follows:

```
        DATA('STACK(TOP)')
                  .
                  .
                  .
PUSH    TOP(S)    =    PLATE(V,TOP(S))              :(RETURN)
                                     T
POP     POP    =   DIFFER(TOP(S)) VALUE($OP(S))    :F(FRETURN)
        TOP(S)    =    LAST(TOP(S))                 :(RETURN)
```

The advantages of this method are:

(1) Stacks grow in size automatically.
(2) Space is allocated only when it is needed.
(3) The stack manipulation functions are simple and efficient.
(4) Space is released when an item is popped.
(5) Any kind of object can be stored in the stack.

The disadvantages of this method are:

(1) The amount of space required for each item on the stack is greater than for other methods.
(2) Each push requires allocation of storage; PLATEs are not reused.

Actually, for most purposes, the defined data type implementation is the best. The efficiency of the stack manipulation functions outweighs the time spent allocating space. The automatic deallocation of space has an advantage that is not obvious unless the internal workings of SNOBOL4 are well understood. In the array and table implementations, not only is space not deallocated when an item is popped, but the item also remains in the structure even though it cannot be accessed by the stack manipulation functions. As a result, the items that have been popped are still retained by the SNOBOL4 system, even if they are of no more use to the program. Only when they are overwritten by subsequent pushes are they released so that the SNOBOL4 storage management procedures can reclaim the space they occupy. While allocation of space in many programming languages is an expensive and cumbersome process, SNOBOL4 is designed to allocate and regenerate storage automatically and efficiently. Programming methods that require allocation of objects should not be of concern to the programmer as long as the objects are not excessively large and as long as they are discarded when they are no longer needed.

Other Stack Functions

STACK, PUSH, and POP are the basic stack functions. They include the necessary means for creating stacks and accessing them. There are a number of other, auxiliary, functions that may be useful in some situations.

A fairly common operation consists of popping an item off the stack and then immediately pushing another in its place. While this can be done with POP and PUSH, it is both awkward and inefficient. A function to perform this operation in one step follows. The defined data type implementation is assumed.

```
        DEFINE('SWITCH(V,S)')
                   .
                   .
                   .
SWITCH  SWITCH  =    DIFFER(TOP(S)) VALUE(TOP(S))    :F(FRETURN)
        VALUE(TOP(S))   =   V                        :(RETURN)
```

Procedures for the other kinds of stack implementations follow naturally.

For diagnostic purposes, it may be useful to print the items on a stack in a sequence of lines, giving a picture of the current state of the stack. Such a function is:

```
        DEFINE('PRTSTK(S)P')
                   .
                   .
                   .
PRTSTK  P   =   TOP(S)
PRTS1   OUTPUT   =   DIFFER(P) VALUE(P)           :F(RETURN)
        P   =   LAST(P)                           :(PRTS1)
```

Other lines of printing can be added, if desired, to set off and delimit the stack contents.

The procedure above prints the stack from the top down. Suppose that the printing is desired in the opposite order, from the bottom up. This might seem to be particularly inconvenient, since the pointers within the stack are from the top down. One approach to the problem is to start from the top, linking through the stack, saving (pushing) pointers to PLATEs along the way. When the bottom is reached, printing of values can be done as the pointers are popped. A separate stack could be used for this purpose, but the same effect can be achieved by using a recursive procedure.

```
       DEFINE('PRTSTK(S)')
       DEFINE('PLUNGE(P)')
                .
                .
                .
PRTSTK PLUNGE(TOP(S))                              :(RETURN)

PLUNGE (DIFFER(P) PLUNGE(LAST(P)))                 :F(RETURN)
       OUTPUT   =   VALUE(P)                       :(RETURN)
```

Since PLUNGE calls itself before printing the value of the current item, the last (bottom) value is printed first, followed in order by previous items as successive calls return. A second procedure is needed only because there is a heading object between S and the first item. Using recursion amounts to the use of the internal SNOBOL4 stack to save intermediate results. This technique is often applicable when objects are not linked together in the desired order of processing.

A variety of other stack functions can be developed. Some are suggested in the exercises.

EXERCISES

3.1 In many circumstances, only a single stack is needed, and the additional arguments of the stack manipulation functions are unnecessary. Specify a reasonable default in case the additional argument of PUSH and POP is omitted.

3.2 For the string implementation of stacks, devise a method for separating items that does not preclude any character from appearing in an item.

3.3 Modify the array implementation of stacks to provide for automatic extension when a stack becomes full.

3.4 For each method of stack implementation, write a function to count the number of items on a stack.

3.5 For each method of stack implementation, write a function to copy a stack.

3.6 For each method of stack implementation, write a function to reinitialize a stack, discarding any contents it may have and restoring it to its condition when first created.

3.7 Write a procedure to print the contents of a stack from the bottom up, using an auxiliary stack for storage, rather than recursion.

3.2.2. Queues

While a stack is a lifo list, a queue is a *fifo* list, served on a first-in first-out basis. Typical examples of queues are waiting lines for buses and grocery check-out lines. In these cases individuals are handled on a first-come first-served basis. A queue may be visualized as shown in Figure 3.28.

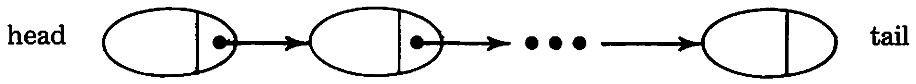


Figure 3.28 A Queue

Insertions are made at the tail and deletions are made from the head. Arrows indicate the order of service.

As for stacks, there are standard functions for manipulating queues:

```
Q   =   QUEUE()
INSERT(V,Q)
V   =   DELETE(Q)
```

QUEUE creates a queue. INSERT puts the value of V on the tail of Q and DELETE removes the head of Q and returns its value. DELETE fails if the queue is empty.

Implementation of queues is somewhat more complicated than the implementation of stacks. The principles discussed in the preceding section can be applied, although the less-desirable approaches will not be discussed again.

Since a queue is a linear structure as shown in Figure 3.28, it is natural to consider the possibility of an array implementation. Since items are added to the tail and taken from the head, a single index is not sufficient for maintaining a queue. Figure 3.29 illustrates a typical situation.

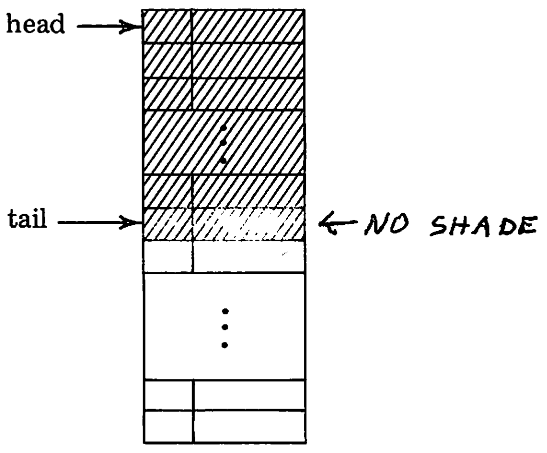


Figure 3.29 A Queue in an Array

The shaded area indicates the portion of the array that is in use. As items are added to the tail, that index increases. As items are deleted from the head, that index increases also. Thus, unlike stacks, both indices increase as items are added and removed from a queue. Obviously, both indices cannot increase indefinitely. The standard method of handling this situation, where space for a queue is preallocated in a contiguous block, is to "wrap around" to the beginning when the end of the array is reached. This can be done as long as enough items have been deleted from the queue to free space at the beginning. Figure 3.30 illustrates the positions of the two indices after wrap around.

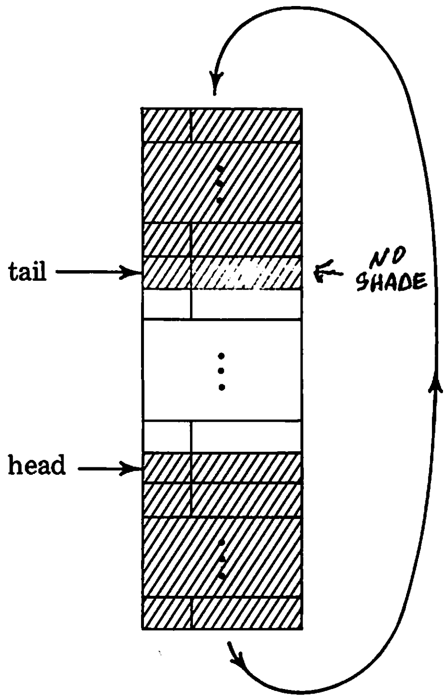


Figure 3.30 Wrap-Around in a Queue

If the two indices ever meet, the queue is full.

The problem with implementing queues in this way lies in the complexity of the programming required. Since wrap-around is not a process that occurs naturally, it has to be simulated when the end of the array is reached. There are numerous special cases that have to be handled, including making provisions for distinguishing an empty queue from a full one. The process can be more easily understood with respect to a structure that provides physical wrap-around: a ring. Such a structure is shown in Figure 3.31.

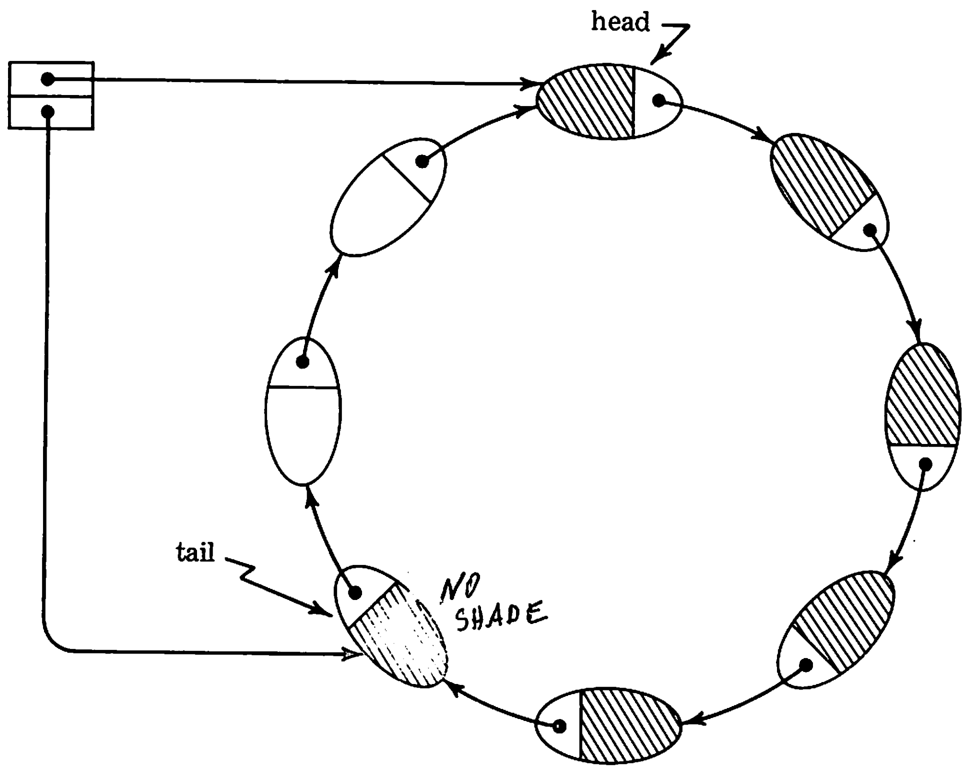


Figure 3.31 A Ring Implementation of a Queue

The shaded elements on the ring indicate those nodes that are currently "on the queue". Assuming that the ring has already been created as shown, the functions INSERT and DELETE are:

```
        DATA('ELEMENT(VALUE,NEXT)')
        DEFINE('INSERT(V,Q)')
        DEFINE('DELETE(Q)')
                  .
                  .
                  .
INSERT  IDENT(NEXT(TAIL(Q)),HEAD(Q))           :S(FRETURN)
        VALUE(TAIL(Q))    =   V
        TAIL(Q)    =   NEXT(TAIL(Q))           :(RETURN)
DELETE  IDENT(TAIL(Q),HEAD(Q))                 :S(FRETURN)
        DELETE    =   VALUE(HEAD(Q))
        HEAD(Q)    =   NEXT(HEAD(Q))           :(RETURN)
```

In this implementation, it is necessary to distinguish between an empty queue and a full queue. In the procedures above, this is accomplished by using one element as a separator which can never hold a value.

While the ring implementation is an easier way to simulate a circular structure than an array is, rings are still fixed in size and hence are subject to overflow. A more natural method is based on the implementation of stacks using defined data objects which are allocated as needed. Such a structure is shown in Figure 3.32.

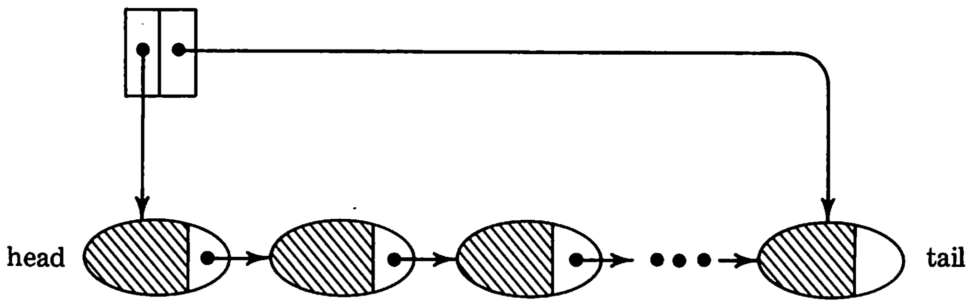


Figure 3.32 A Queue Composed of Defined Data Objects

Insertion requires creation of a new element, and deletion discards an element. The procedures are:

```
       DATA('ELEMENT(VALUE,NEXT)')
       DATA('QUEUE(HEAD,TAIL)')
                 .
                 .
                 .
INSERT E    =   TAIL(Q)
       TAIL(Q)  =   ELEMENT(V)
       HEAD(Q)  =   IDENT(E) TAIL(Q)        :S(RETURN)
       NEXT(E)  =   TAIL(Q)                 :(RETURN)
DELETE DELETE = DIFFER(HEAD(Q)) VALUE(HEAD(Q)) :F(FRETURN)
       HEAD(Q)  =   NEXT(HEAD(Q))
       TAIL(Q)  =   IDENT(HEAD(Q))          :(RETURN)
```

An empty queue is distinguished by null values for the HEAD and TAIL fields.

EXERCISES

3.8 Write a string implementation of queues.

3.9 Write an array implementation of queues.

3.10 Write a function to construct the ring for the ring implementation of a queue.

3.11 Modify the ring implementation of queues to allow for automatic enlargement when a queue becomes full.

3.12 Write a function to transfer the contents of a stack to a queue. Make the procedure independent of the methods of implementation for the stack and queue.

3.13 For each method of queue implementation, write a function to count the number of items on a queue.

3.14 For each method of queue implementation, write a function to copy a queue.

3.15 For each method of queue implementation, write a function to print the contents of a queue as a sequence of items on separate lines. Write procedures to print:
(a) From the head to the tail.
(b) From the tail to the head.

3.2.3. Linked Lists

The implementations of stacks and queues using defined data objects are just two examples of *singly-linked* lists. These structures are linear, consisting of objects linked together, one after another. Given a pointer to an object, the next object (if any) can be reached directly by use of a field function. Successive objects on the list can be accessed by successive use of field functions.

Linked lists are useful in a number of situations where neither stacks nor queues are necessary. For example, in some types of linguistic processing it is useful to represent a segment of text (such as a sentence) as a list of words. The queue or stack mechanisms might suffice in such a case, but the relatively rigid constraints imposed by the access functions for these structures might be more of a hindrance than a help.

When the use of linked lists is being considered, a number of questions arise:

(1) What operations can easily be performed on lists?
(2) To what extent are related pointers to objects necessary or convenient?
(3) What operations are difficult, inefficient, or impossible to perform on lists?

Some answers are obvious. For example, getting from one element to the next is straightforward and efficient. Getting from an element to the previous element is not possible unless there is some related pointer elsewhere that leads to the previous element. Related pointers, i.e., pointers that are not part of the list itself, determine many of the processes that can be performed on lists. In the case of stacks, only one related pointer, to the top, is required. For queues, two pointers, one to the head and another to the tail, are used. Of course, given a pointer to the tail of a queue, it is possible to get to the head, but the operation is time consuming and inefficient.

Copying a list illustrates many of the aspects of list processing. Suppose that E points to the first element on a list, and that NEXT is the field function used to link the elements together. The following function creates a (physically distinct) copy of the list:

```
       DEFINE('COPYL(E)')
                 .
                 .
                 .
COPYL  COPYL   =   COPY(E)
       E   =   COPYL
COPYLP NEXT(E) = DIFFER(NEXT(E)) COPY(NEXT(E)) :F(RETURN)
       E   =   NEXT(E)                         :(COPYLP)
```

This procedure may be a little obscure unless the built-in function COPY is clearly understood. Figure 3.33 illustrates the steps in the process for a typical list.

At step (a), E points to the first element on the list. The first statement of COPYL creates a copy of the first element as shown in (b). The second statement changes the value of E to point to the new element as shown in (c). COPYL continues to point to the first element on the new list being created. It is this value that is eventually returned by the function. The copying process continues as long as there are elements on the original list. In (f) the entire list has been copied and E points to the last element. E is then assigned the null string (NEXT(E)) and the statement labeled COPYLP fails. A pointer to the new list is returned.

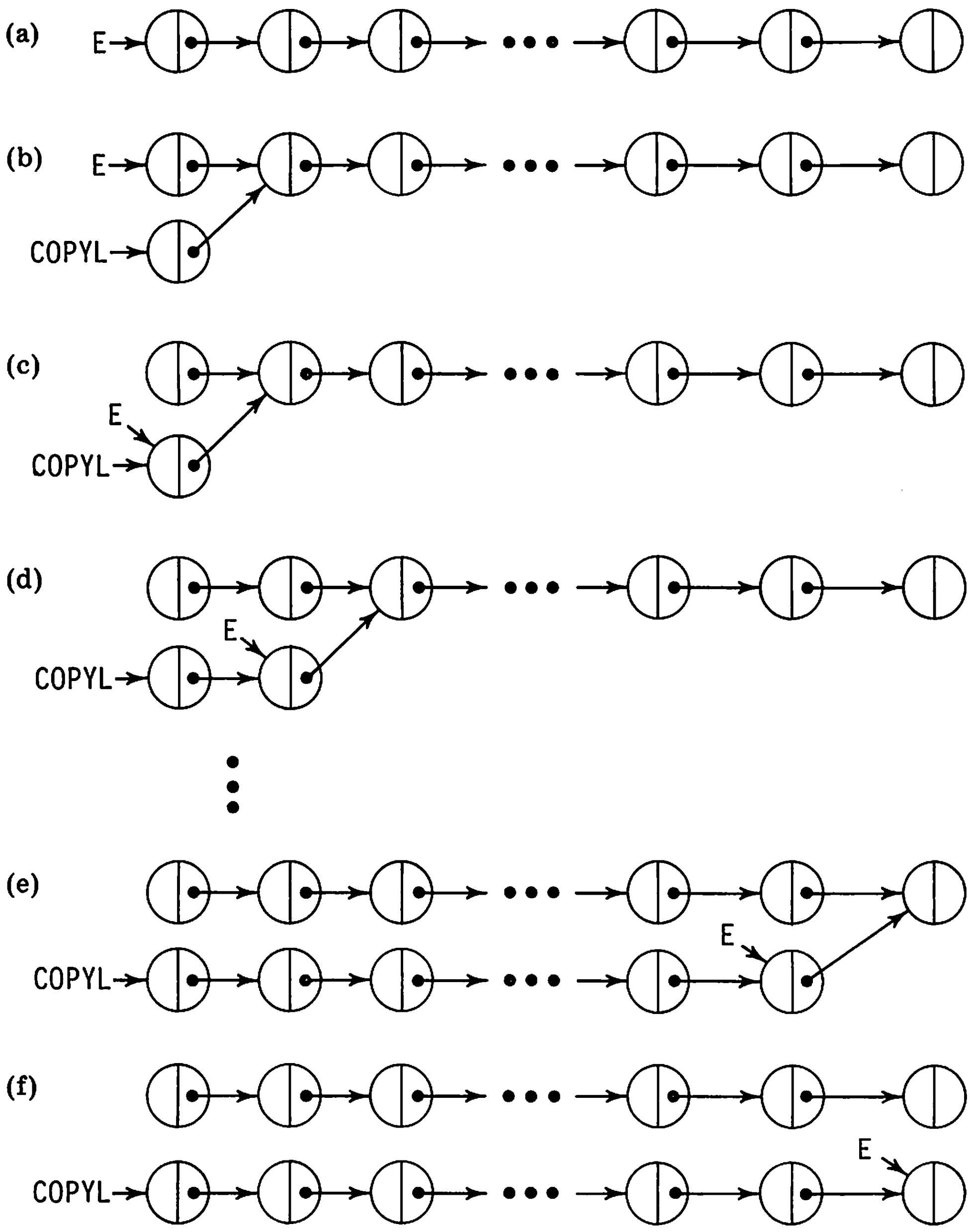


Figure 3.33 Copying a List

Although COPYL makes copies of the elements comprising the original list, it does not copy any objects pointed to by other fields of these elements. If an element on the original list points to an object, the corresponding element on the new list points to the *same* object. This situation is illustrated in Figure 3.34.

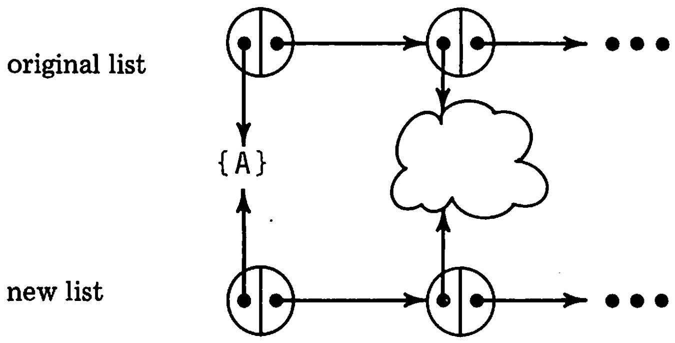


Figure 3.34 Lists Pointing to Common Objects

The two lists share objects. This is another manifestation of the problem mentioned in Section 3.1.1.

Modification of a singly-linked list can be awkward. Suppose, for example, that it is desired to delete an element. This can only be done, in general, if there is a pointer to an element before the element to be deleted. Similarly, an element can be inserted after an element, but not before it. This problem is often overcome by the use of related pointers which make it possible to get to any element on a list. If, however, a list is to be used in a situation where modification is frequently performed, a *doubly-linked* list may be called for. Figure 3.35 shows the structure of a doubly-linked list in which there are pointers in both directions between adjacent elements.

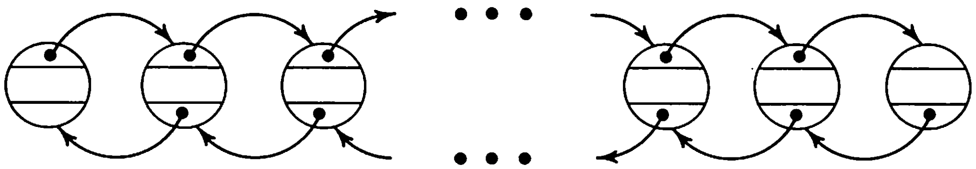


Figure 3.35 A Doubly-Linked List

Elements on such a list might be defined by

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

where LAST is used to link back to the previous element.

In a doubly-linked list, deletion or insertion can be performed anywhere in the list. A function to delete an element is:

```
       DEFINE('DELELE(E)')
                .
                .
                .
DELELE IDENT(NEXT(E))                                  :S(DELNN)
       IDENT(LAST(E))                                  :S(DELNL)
       NEXT(LAST(E))   =   NEXT(E)
DELNL  LAST(NEXT(E))   =   LAST(E)                     :(RETURN)
DELNN  IDENT(LAST(E))                                  :S(RETURN)
       NEXT(LAST(E))   =                               :(RETURN)
```

The special cases are required to handle the first and last elements on the list. Note that it is meaningless to delete an element that is not part of a list unless there are related pointers. The problem of related pointers is not treated by the function above. In fact, related pointers are not, in general, accessible from the list.

EXERCISES

3.16 Write a function to process text and create a linked list of the words contained in it.

3.17 Write a *recursive* procedure for COPYL. What inherent limitation does recursion have in this situation?

3.18 Write a function to print the values on a singly-linked list.

3.19 Write a function to copy a doubly-linked list.

3.20 Write functions to add an element to a doubly-linked list
(a) Before a given element.
(b) After a given element.

3.21 Write a function to print the values on a doubly-linked list.

3.22 Write a function to create a doubly-linked list from a singly-linked list.

3.23 A *deque* [23] is a doubly-linked list that is accessed only at the ends. Implement deques and develop appropriate functions for manipulating them.

3.2.4. Binary Trees

Many of the more important types of structures are not linear. An example is the binary tree. Binary trees are composed of nodes that are connected together in certain restricted ways. A node has two possible links, one pointing to a left subtree and one pointing to a right subtree. See Figure 3.36.

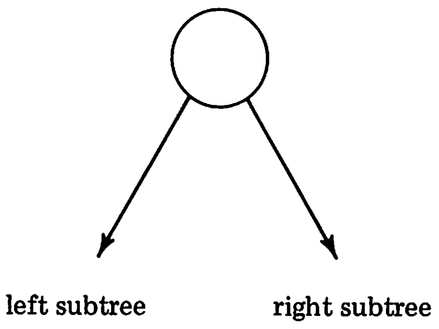


Figure 3.36 A Node in a Binary Tree

A node may have at most one pointer to it. A node without a pointer to it is called a *root* and is treated as a specially designated, or first, node in a binary tree. A node without a pointer from it is called a *leaf*. Furthermore there may be no loops. Binary trees are customarily drawn with the root at the top and with the subtrees below. Figure 3.37 illustrates a typical binary tree.

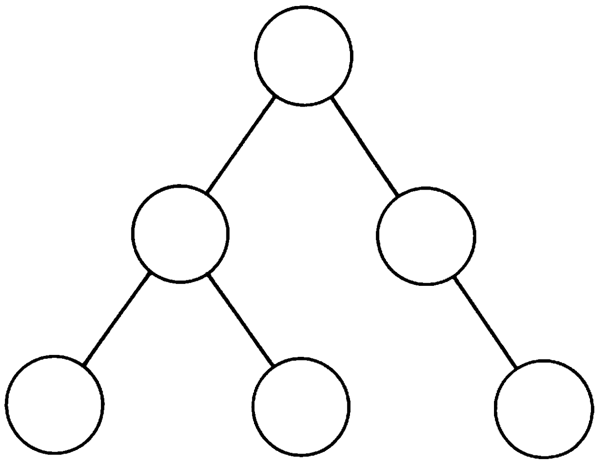


Figure 3.37 A Binary Tree

In such a diagram, left and right subtrees are implied by position. Note that a node may have 0, 1, or 2 subtrees. With only links downward, it is only possible to get from a node to nodes below, unless there are related pointers. Some processes are more conveniently programmed if there are upward pointers as well. Nodes in a binary tree may be defined as follows:

```
        DATA('BNODE(VALUE,LEFT,RIGHT,UP)')
```

Figure 3.38 illustrates the binary tree of Figure 3.37 in terms of this defined data type.

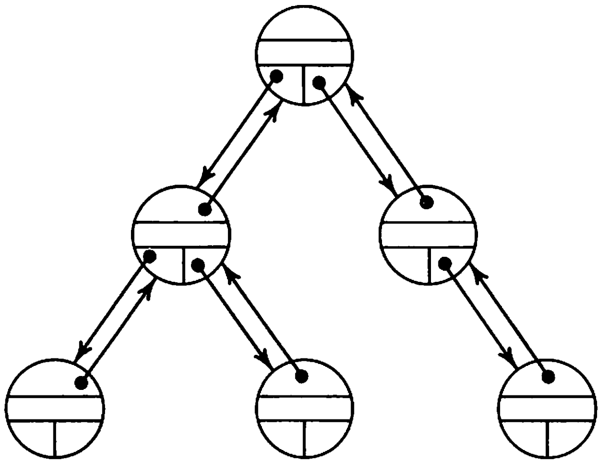


Figure 3.38 A Binary Tree with Upward Pointers

Such a diagram, showing all the pointers, is unnecessarily complicated for most purposes. The simplified form, as shown in Figure 3.37, is used for most diagrams, with the understanding that all pointers are present in an actual structure.

A binary tree is an aggregate of BNODEs. Constructing a binary tree requires linking the BNODEs together with appropriate pointers. The VALUE field is provided as a place to associate data with a particular node. Other fields can be provided if desired.

There is an obvious correspondence between binary trees and arithmetic expressions containing binary operators. Although another structure is more convenient for representing expressions, the correspondence between binary trees and binary expressions will be discussed here as a starting point. Consider the expression

```
B*(A-C/D)
```

A corresponding binary tree is shown in Figure 3.39.

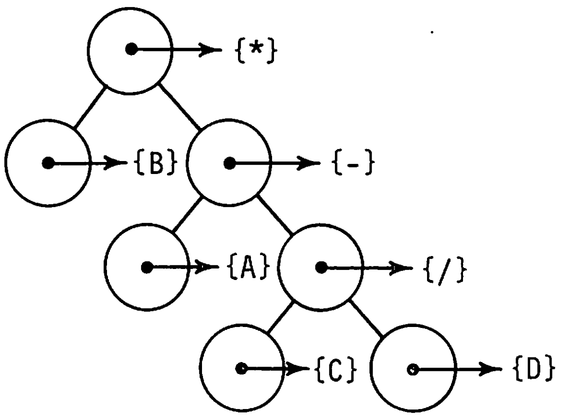


Figure 3.39 A Binary Tree Corresponding to an Expression

The relationship (in fact, isomorphism) between the expression and the binary tree is more evident if the expression is fully parenthesized:

```
(B*(A-(C/D)))
```

Each term in parentheses corresponds to a subtree, and the expressions on either side of an operator correspond to the left and right subtrees of the operator. Another way of looking at the relationship is in terms of the prefix form of the expression:

```
*(B,-(A,/(C,D)))
```

In this form, the two terms in parentheses correspond to the left and right subtrees, the comma separating the left from the right.

Of course, there is no reason why this notation must be restricted to arithmetic expressions. The functional notation

```
S(T,U(V(W),Y(Z)))
```

corresponds to the tree shown in Figure 3.40.

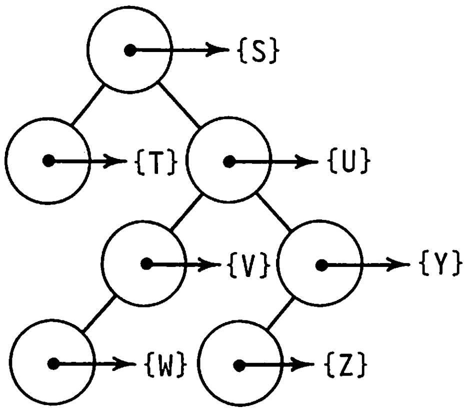


Figure 3.40 Another Binary Tree

In case there is only one subtree, the convention is that an omitted comma corresponds to a missing right subtree.

While lists are usually constructed an element at a time as information is obtained, and are continually modified during processing, binary trees are frequently constructed at one time and are subsequently examined. For example, to convert a prefix string of the form given above into a corresponding binary tree, the following functions may be used.

```
       DEFINE('ADDL(N1,N2)')
       DEFINE('ADDR(N1,N2)')
       DEFINE('BTREE(S)L,R')
       TWO    =   '(' BAL . L ',' BAL . R ')'
       RONE    =   '(,' BAL . R ')'
       LONE    =   '(' BAL . L ')'
       TFORM    =   BREAK('(') . S (TWO | RONE | LONE)
                  .
                  .
                  .
ADDL   LEFT(N1)   =   N2
ADDU   UP(N2)   =   N1                              :(RETURN)
ADDR   RIGHT(N1)   =   N2                           :(ADDU)

BTREE  S   TFORM
       BTREE   =   BNODE(S)
       (DIFFER(L) ADDL(BTREE,BTREE(L)))
       (DIFFER(R) ADDR(BTREE,BTREE(R)))             :(RETURN)
```

The functions ADDL and ADDR are used to add left and right subtrees to a node. The pattern TFORM is used to match the three possible alternative parenthesized forms. Note that if TFORM fails to match, the argument string simply represents a single node. The function BTREE is naturally recursive because the structure being constructed has a recursive definition.

The inverse of this process is the construction of a prefix expression corresponding to a binary tree. Such a function follows:

```
       DEFINE('BEXP(T)L,R')
                  .
                  .
                  .
BEXP   BEXP   =   VALUE(T)
       L   =   DIFFER(LEFT(T)) BEXP(LEFT(T))
       R   =   DIFFER(RIGHT(T)) ',' BEXP(RIGHT(T))
       S   =   L R
       BEXP   =   DIFFER(S) BEXP '(' S ')'    :(RETURN)
```

Copying a binary tree is in many respects similar to copying a list. The approach is recursive for the reason mentioned above.

```
        DEFINE('COPYBT(T)')
                  .
                  .
                  .
COPYBT COPYBT  =   COPY(T)
 (DIFFER(LEFT(T)) ADDL(COPYBT,COPYBT(LEFT(T))))
 (DIFFER(RIGHT(T)) ADDR(COPYBT,COPYBT(RIGHT(T)))) :(RETURN)
```

Use of Binary Trees in Searching and Sorting

In a number of situations, notably in symbol tables, new objects are added to an aggregate one at a time. The objects may have attributes or may require sorting. In either event, it is necessary to locate an object that is already in the aggregate. Binary trees can be used for this purpose.

The two links, left and right, can be utilized to direct search in the tree on the basis of a comparison of the value of a node with the value to be located. The algorithm can be informally stated as follows:

(1) Start at the root of the binary tree.
(2) If the value of the current node is the same as the value to be located, the search is complete.
(3) If the value of the current node is greater than the value to be located, move to the right. If it is less, move to the left. Continue with step (2).
(4) If there is no left or right subtree in step (3), the value does not exist in the binary tree. This fact can be signaled if the binary tree is being searched, or a node with the new value can be added at the corresponding place if the binary tree is being built.

The two subtrees of a node contain, respectively, all values that are less than or greater than the value of the node. See Figure 3.41.

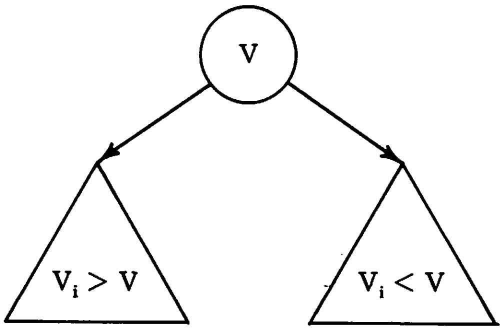


Figure 3.41 Division of Values into Subtrees

The nature of the comparison in step (3) depends on the ways that values are ordered. The comparison might be lexical for alphabetic entries or numerical for numbers.

A procedure for lexical insertion, LEXINS, follows. In this procedure it is convenient to add a defined data object that points to the root of the binary tree. This construction parallels the structures for stacks and queues. The function is:

```
        DATA('BINTREE(ROOT)')
        DEFINE('LEXINS(V,T)N')
                        .
                        .
                        .
LEXINS  N    =    ROOT(T)
        ROOT(T)   =    IDENT(N) BNODE(V)       :S(RETURN)
LEXNXT  LGT(VALUE(N),V)                        :S(LEXRT)
        IDENT(VALUE(N),V)                      :S(RETURN)
        N    =    DIFFER(LEFT(N)) LEFT(N)      :S(LEXNXT)
        ADDL(N,BNODE(V))                       :(RETURN)
LEXRT   N    =    DIFFER(RIGHT(N)) RIGHT(N)    :S(LEXNXT)
        ADDR(N,BNODE(V))                       :(RETURN)
```

To see how a binary tree develops, consider an example in which the words of the following sentence are inserted in order from left to right:

HIS FELT HAT IS ON THE OLD RACK

Figure 3.42 shows the first few steps.

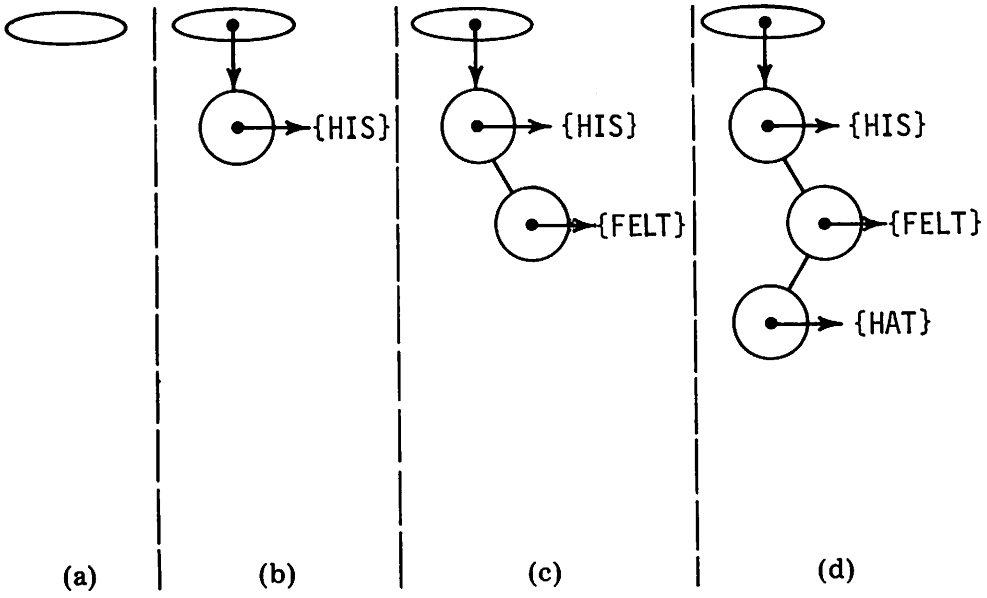


Figure 3.42 Development of a Binary Tree

Initially the tree is empty, as is shown in (a). When HIS is inserted, it becomes the first node. Since FELT is less than HIS, a node is created to the right. When HAT is inserted, it is less than HIS, and goes to the right. HAT is greater than FELT, however, and goes to the left. The complete binary tree is shown in Figure 3.43.

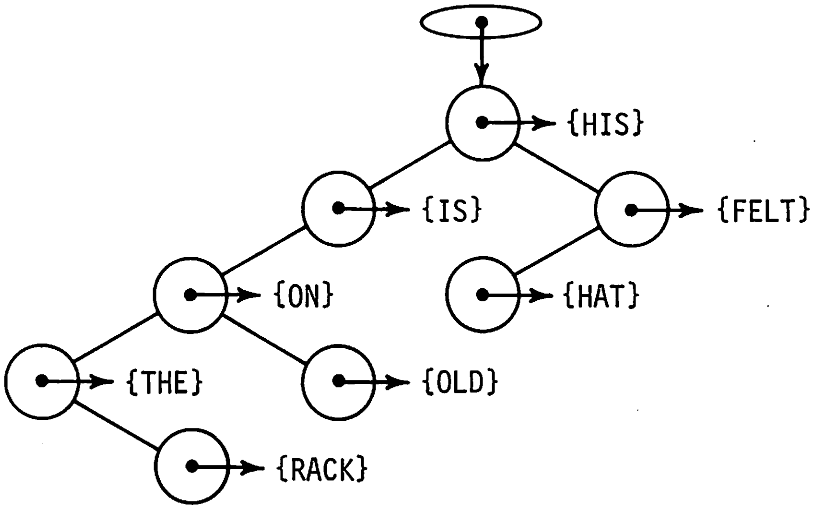


Figure 3.43 The Complete Binary Tree

Notice that the structure of the binary tree is strongly dependent on the order in which the values are entered. If the words are entered in reverse order, the binary tree of Figure 3.44 results.

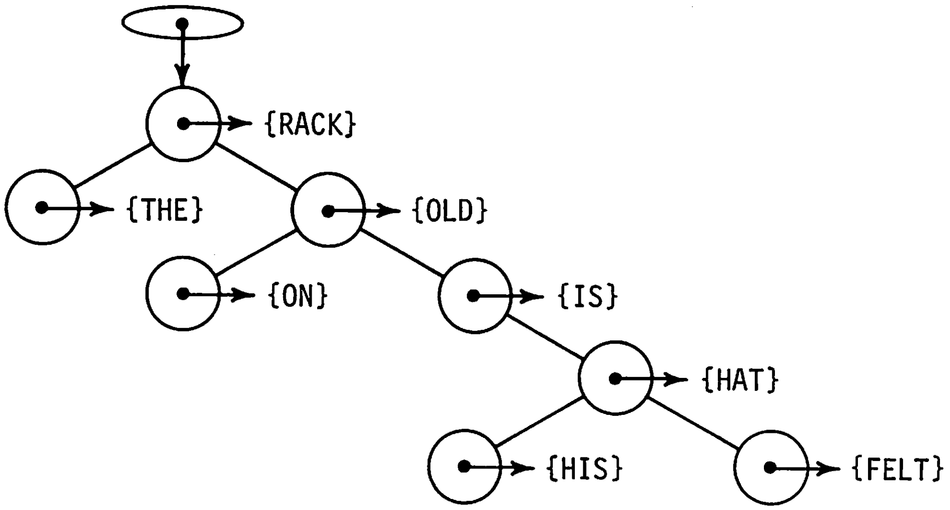


Figure 3.44 The Binary Tree Resulting From Reverse Entry

It is important to recognize that position in a binary tree according to order is a relative matter, not an absolute one. In searching a binary tree for an entry, the time required depends on the structure of the binary tree. There are various techniques for manipulating binary trees to "balance" them [20].

A binary tree of this type may be used to obtain a sorted list of items. The process requires traversing the binary tree in a prescribed manner. Basically the traversal proceeds as far to the right as possible. When a node with no right subtree is encountered, its value is output. From this point, traversal starts upward and then down the first left link. Whenever a node is first encountered in upward traversal, it is output. When a left link is taken, traversal to the right is then reinitiated. A procedure for printing the values in a binary tree in lexical order is:

```
       DEFINE('LEXPRT(T)N,M')
                    .
                    .
                    .
LEXPRT N   =   DIFFER(ROOT(T)) ROOT(T)          :F(RETURN)
LEXPRD N   =   DIFFER(RIGHT(N)) RIGHT(N)        :S(LEXPRD)
LEXOUT OUTPUT   =   VALUE(N)
       N   =   DIFFER(LEFT(N)) LEFT(N)          :S(LEXPRD)
LEXUP  M   =   N
       N   =   DIFFER(UP(N)) UP(N)              :F(RETURN)
       DIFFER(LEFT(N),M)                        :S(LEXOUT)F(LEXUP)
```

In this procedure, one point deserves special attention: The value of a node is printed when it is *first* encountered in upward traversal. This may be tested by determining whether the move upward comes from a left or a right subtree. The last statement in the procedure performs this test.

EXERCISES

3.24 Suppose a binary tree is to be used as a symbol table in which there is a value associated with each symbol. What modifications to the structure given in the text are necessary?

3.25 Write a function to count the number of nodes in a binary tree.

3.26 Upward pointers in binary trees make traversal convenient at the expense of additional space. Omit the UP field from BNODE and rewrite LEXPRT
(a) Using a recursive procedure.
(b) Using an iterative procedure and a stack for storing intermediate locations.

3.27 Write a function to diagram a binary tree.

3.2.5. Trees

While nodes in binary trees may only have two subtrees, nodes in a tree may have an arbitrary number of subtrees. Figure 3.45 illustrates such a tree.

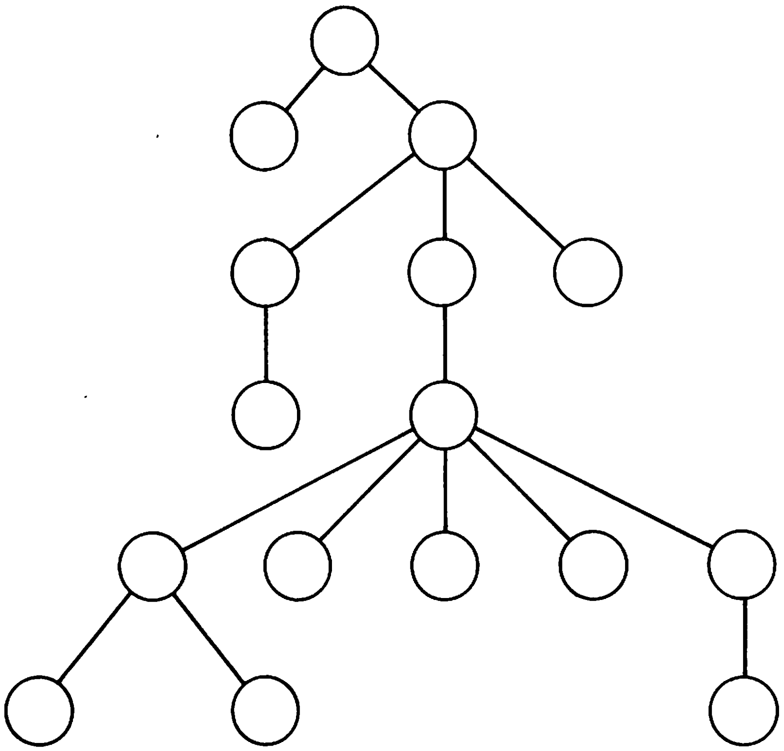


Figure 3.45 A Tree

Since a node may have many subtrees, it is inconvenient to specify each subtree in the way that was used for binary trees. Instead, the subtrees of a tree can be considered as arranged from left to right. The terminology used for describing trees is similar to that used for describing binary trees, except that the relationships are somewhat different. A node may have a *left son*, which is the top of the leftmost subtree below it. Other subtrees are described relative to the left son. A *right sibling* is the top of the subtree to the right of a node. The *father* of a node is the node above it.

A defined data type for such a node is given by:

```
DATA('TNODE(VALUE,LSON,RSIB,FATHER)')
```

As before, a field to contain a value is provided. Other fields can be added if they are needed. Figure 3.46 shows typical relationships among nodes. A is ~~is~~ the father of B, C, and D. B is the left son of A, C is the right sibling of B and D is the right sibling of C.

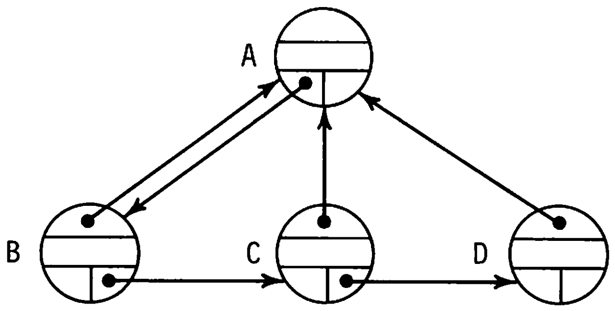


Figure 3.46 Relationships Among Nodes of a Tree

The arrows in Figure 3.46 illustrate the set of pointers that makes it possible to get from any place in a tree to any other. With such pointers, a tree can be traversed without any related pointers. Note that it is possible to go directly from any node only to its left son, right sibling, or father, but successive pointers make it possible to get to any node in the tree. As with binary trees, it is usually unnecessary to show all the pointers in diagrams. A simplified diagram is illustrated in Figure 3.47.

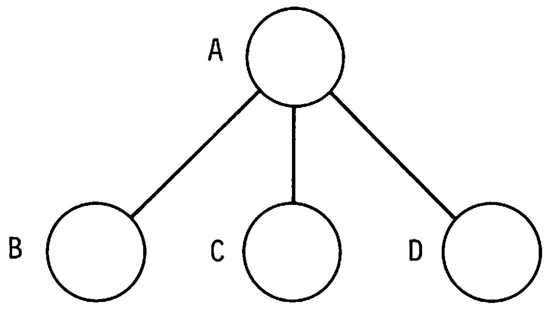


Figure 3.47 A Simplified Tree Diagram

Lines between nodes indicate father-son relationships.

The methods for constructing trees are similar to those for constructing binary trees. Two tree-constructing functions are particularly useful: one for adding a tree as the left son of a node, and one for adding a tree as a right sibling of a node. Such functions follow:

```
        DEFINE('ADDSON(N1,N2)')
        DEFINE('ADDSIB(N1,N2)')
                   .
                   .
                   .
ADDSON  RSIB(N2)   =   LSON(N1)
        FATHER(N2)  =   N1
        LSON(N1)   =   N2                  :(RETURN)

ADDSIB  RSIB(N2)   =   RSIB(N1)
        RSIB(N1)   =   N2
        FATHER(N2)  =   FATHER(N1)         :(RETURN)
```

In each case, N1 is the node to which a subtree is added.

As with binary trees, there is a natural isomorphism between trees and expressions. The relationship is more general with trees since unary operators with more than two operands can be represented. For example, the expression

```
3*F(X,Y,-12)+Z
```

is equivalent to the tree shown in Figure 3.48.

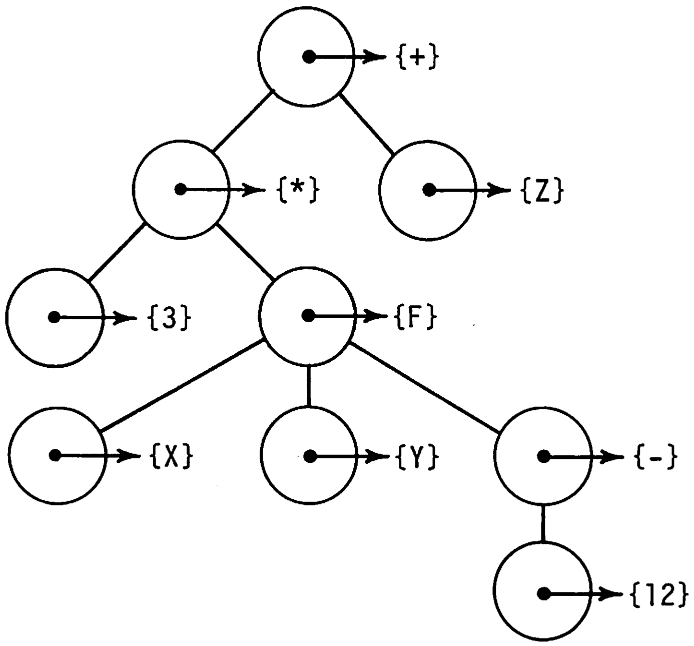


Figure 3.48 A Tree Corresponding to an Expression

As before, there is an equivalent prefix form:

```
        +(*(3,F(X,Y,-(12))),Z)
```

Note that F and the unary - are prefix forms in the original expression. In dealing with expressions, it must be remembered that unary operators are different from binary operators, even though the same symbols may be used in both cases. In prefix form, parentheses and commas resolve any apparent ambiguity.

A function to construct a tree from a ~~parenthesized~~ prefix expression follows:

```
        DEFINE('TREE(S)T,X,Y,Z')
        MANY    =    '(' BAL . X ',' BAL . Y ')'
        ONE     =   '(' BAL . X ')'
        TFORM    =    BREAK('(') . S (MANY | ONE)
        NEXT    =    POS(0) BAL . Z ','
                      .
                      .
                      .
TREE    S   TFORM
        TREE    =    TNODE(S)
        T   =    DIFFER(X) TREE(X)                :F(RETURN)
        ADDSON(TREE,T)
TREE1   IDENT(Y)                                  :S(RETURN)
        Y   NEXT   =                              :F(TREE2)
        ADDSIB(T,TREE(Z))
        T   =    RSIB(T)                          :(TREE1)
TREE2   ADDSIB(T,TREE(Y))                         :(RETURN)
```

Converting a tree back into an expression is similar to the corresponding operation on binary trees, except that the somewhat more complicated structure must be taken into account. A procedure follows:

```
        DEFINE('EXP(T)')
                      .
                      .
                      .
EXP     EXP    =    VALUE(T)
        T   =    DIFFER(LSON(T)) LSON(T)          :F(RETURN)
        EXP    =    EXP '(' EXP(T)
EXP1    T   =    DIFFER(RSIB(T)) RSIB(T)          :F(EXP2)
        EXP    =    EXP ',' EXP(T)                :(EXP1)
EXP2    EXP    =    EXP ')'                       :(RETURN)
```

There are many ways a tree can be traversed. Consider the problem of printing the values in a tree in some well-defined and logical order. One way corresponds to the order of the components in the corresponding prefix expression, and is called *left listing.* In such a listing, the value of each node is printed starting at the root and proceeding down the left-most branch of the tree. The listing process can be conceptualized by first defining a successor function NEXT(N) that returns the next node following a given node N:

```
        DEFINE('NEXT(N)')
                  .
                  .
                  .
NEXT    NEXT   =   DIFFER(LSON(N)) LSON(N)     :S(RETURN)
NEXT1   NEXT   =   DIFFER(RSIB(N)) RSIB(N)     :S(RETURN)
        N = DIFFER(FATHER(N)) FATHER(N) :S(NEXT1)F(FRETURN)
```

NEXT returns the left son if possible, the right sibling if that is not possible, and otherwise starts back up the tree, returning the right sibling of the father, if possible. When the root is reached, NEXT fails. A left-listing function, based on NEXT, follows:

```
        DEFINE('LLIST(T)')
                  .
                  .
                  .
LLIST   OUTPUT    =   VALUE(T)
        T    =   NEXT(T)                       :S(LLIST)F(RETURN)
```

EXERCISES

3.28 Write a function to count the nodes in a tree.

3.29 Write a function to count the leaves on a tree.

3.30 Let the depth of a tree be the maximum number of downward pointers in any path leading from the root to a leaf. Write a function to compute the depth of a tree.

3.31 Write a function to copy a tree.

3.32 Write a function to get from a node in a tree to its *left* sibling.

3.33 Write a function to list the values of nodes in a tree so that each value is indented by an amount that corresponds to its depth in the tree.

3.34 Write a function to list a tree without using FATHER pointers.

3.35 Write a function to convert a binary tree to a tree.

3.36 Write a function to diagram a tree.

3.3. PROCESSING STRUCTURES THAT CONTAIN LOOPS

Several of the structures discussed in the preceding sections contain loops. The most obvious example is the ring implementation of queues. A less obvious example is a tree in which there are loops between a node and its sons. These loops are shown explicitly in Figure 3.46, but are suppressed in other diagrams. Loops in structures provide the potential for loops in the programs that process them. Endless program loops are expensive and difficult to locate. In the structures discussed in the preceding sections, however, such program loops are not likely to cause trouble. In trees, for example, there are no loops that result from following successive LSON pointers. In the case of rings, there is a loop through successive NEXT pointers. Detection of a full queue is done by pointer comparison. This process deserves a more detailed discussion. Consider a ring, apart from its use in a queue, as shown in Figure 3.49.

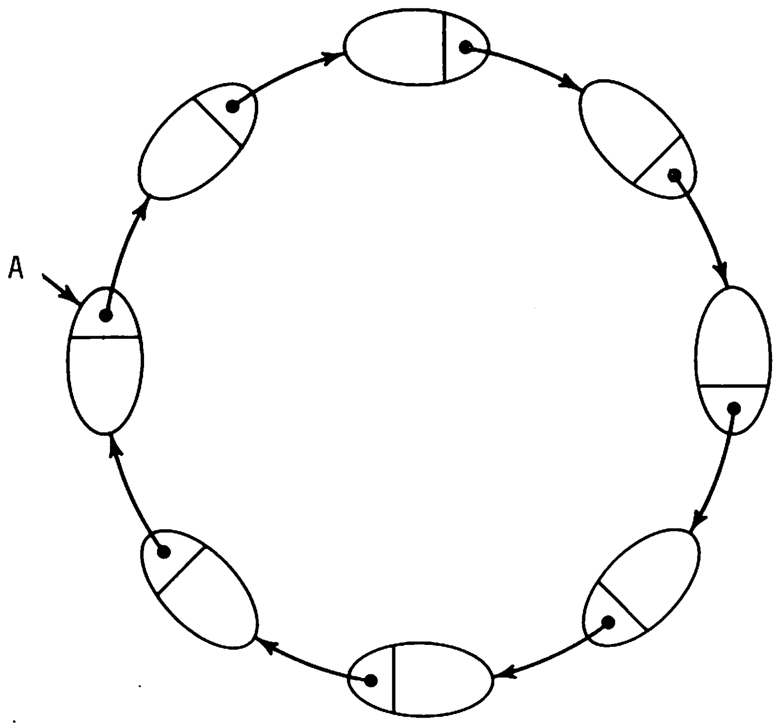


Figure 3.49 A Ring

To move from A to the next element, the following statement can be used:

```
A   =   NEXT(A)
```

If the values of elements on a ring are to be printed, statements such as the following might be used:

```
RLOOP  OUTPUT   =   VALUE(A)
       A   =   NEXT(A)                        :(RLOOP)
```

Unlike trees, in which successive LSON pointers eventually terminate, there is nothing to stop this loop. A test of a null value of A would never succeed. If, however, the starting position is noted, a pointer comparison can be used to terminate processing:

```
       B   =   A
RLOOP  OUTPUT   =   VALUE(A)
       A   =   NEXT(A)
       IDENT(A,B)                             :F(RLOOP)S(DONE)
```

The predicate IDENT succeeds only if the values of A and B are identical, that is, if they are the same pointer.

A ring is a very simple structure. Processing it is no problem because of its uniformity. The printing loop in the example above is certain to return to the starting point without getting into an intermediate loop. Consider a slightly more complicated structure, the "curlicue", shown in Figure 3.50.

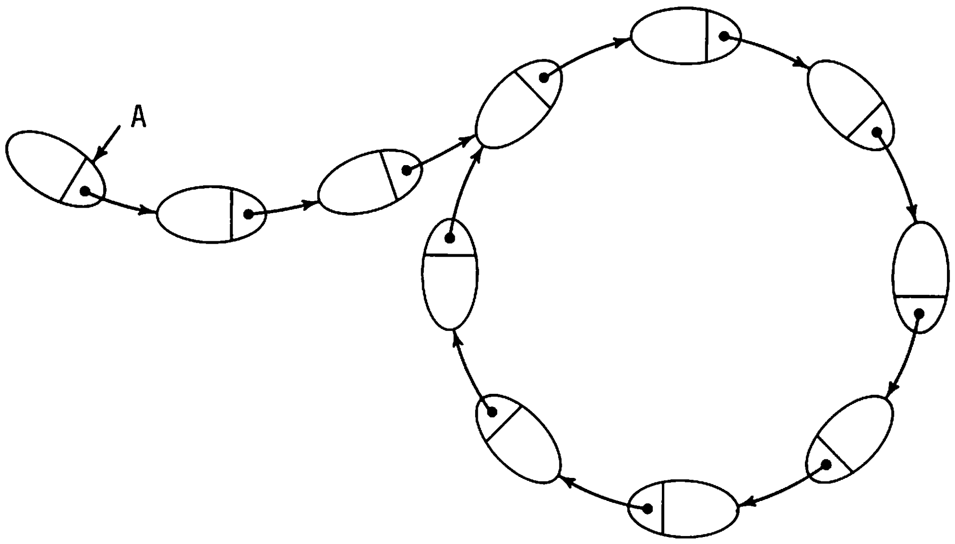


Figure 3.50 A "Curlicue"

In general, the number of elements before the loop may vary. If the statements above are applied to this structure, an endless program loop results.

Overcoming this type of problem is difficult in general, especially if the relationships among the parts of the structure are not regular or not known. More complicated structures are easy to devise, if not easy to process. A directed graph, for example, consists of elements connected by pointers without any constraints on the number of pointers or what elements they may point to.

One method of avoiding loops is to keep track of all elements that have been processed. A loop is detected when a previously processed element is encountered. Keeping track of all processed elements would be awkward if it were not for tables. Any data object can be used to subscript a table and lookup is automatic. Consider the printing statements applied to a curlicue. Instead of recording the single starting position, each element processed can be recorded by assigning it a nonnull value in a table:

```
      MARK   =   TABLE()
RLOOP MARK<A>   =   IDENT(MARK<A>) 1        :F(DONE)
      OUTPUT   =   VALUE(A)
      A   =   NEXT(A)                       :(RLOOP)
```

This general technique can be used in a variety of situations. The particular method depends on the type of structure being processed and the operation being performed.

EXERCISES

3.37 Write a function to copy a ring.

3.38 Write a function to copy a curlicue.

3.39 Devise a string representation for directed graphs.

3.40 Devise a method of representing directed graphs as aggregates of data objects.

3.41 Write functions to convert between the string and data representations of directed graphs.

3.42 A honeycomb is an aggregate of hexagonal cells. A typical honeycomb may be visualized as shown in Figure 3.51.

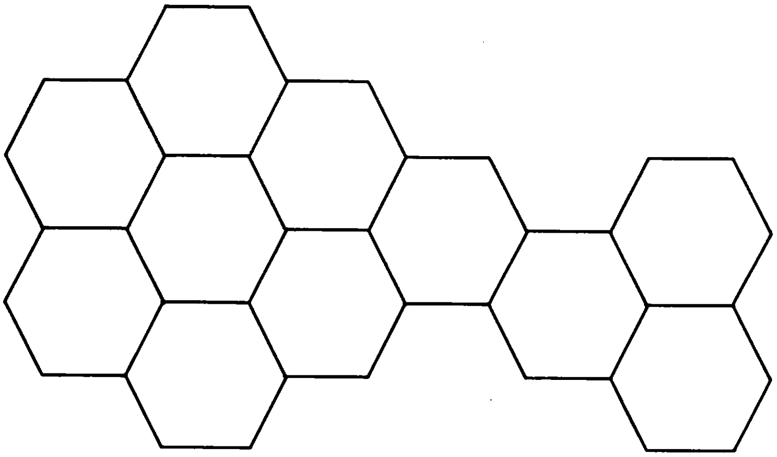


Figure 3.51 A Honeycomb

(a) Give a data definition for cells from which honeycombs can be constructed.
(b) Write a function to construct a random honeycomb.
(c) In a honeycomb, some cells may be completely surrounded by other cells, while others are not. Write a predicate that succeeds if a cell is completely surrounded, and fails otherwise.
(d) Write a function to count the number of cells in a honeycomb.
(e) Write a function to combine two honeycombs at a specified cell.
(f) Write a function to purge a cell from a honeycomb if it is completely surrounded by other cells.

4 APPLICATIONS IN MATHEMATICS

Applications of string and list processing techniques to mathematics usually bring to mind such subjects as symbolic differentiation and theorem proving. In fact, the more widely known and glamorous applications *are* in these areas. There are more fundamental applications, however, and some general principles that have applicability in a wide range of uses. This chapter starts with some fundamental topics and then goes on to consider manipulation of expressions.

4.1. REPRESENTATION AND MANIPULATION OF MATHEMATICAL OBJECTS

In Section 2.1.2, complex numbers are given as an example of the use of functions to extend the SNOBOL4 language. Complex numbers are, of course, essentially mathematical objects, and in most contexts they are used for making computations relating to physical processes. There are many mathematical objects other than integers and real numbers that are manipulated either formally or to derive numerical results. The following sections consider a few such objects to illustrate how they may be represented and manipulated in SNOBOL4.

4.1.1. Rational Numbers

A fraction of the form p/q, where p and q are integers, is called a *rational number*; p is referred to as the *numerator* and q as the *denominator*. In

SNOBOL4, any remainder that results from integer division is discarded. For example, the result of dividing 3 by 2 is simply 1. Such integer arithmetic is not adequate for manipulating rational numbers. On the other hand, real arithmetic is imprecise and real operations do not preserve the identity of their operands. Yet there are some cases, for example algebra and number theory, where true rational numbers must be manipulated. For such cases, rational numbers can be added to SNOBOL4 by use of a defined data type:

```
        DATA('RATIONAL(N,D)')
```

Here N is the numerator and D is the denominator. This data definition immediately provides three functions for manipulating rational numbers:

RATIONAL	creation of rational numbers
N	numerator of rational numbers
D	denominator of rational numbers

Other obvious functions are:

STRRTL	conversion of string to rational
RTLSTR	conversion of rational to string

where the string representation of a rational number could be the natural one. For example,

```
        Z    =    '5/7'
```

assigns the string representation of the rational number 5/7 to Z. These conversion functions are similar in structure to those given in Section 2.1.2.

The arithmetic operations follow naturally, although some care must be taken. For example, the result of multiplying 3/4 by 2/3 is 6/12. On the other hand, 6/12 is equivalent to 1/2. Ordinarily rational results are "reduced to lowest terms". If this is not done, the numerators and denominators of results may quickly get out of hand. Rational numbers can be reduced to lowest terms by dividing both the numerator and denominator by their *greatest common divisor*. Hence, a function GCD is suggested. A procedure using Euclid's Algorithm follows:

```
        DEFINE('GCD(N,M)R')
                   .
                   .
                   .
GCD     R    =    REMDR(M,N)
        GCD   =    EQ(R,0) N                      :S(RETURN)
        M    =    N
        N    =    R                               :(GCD)
```

This function is used in another function, REDUCE, that reduces its arguments to lowest terms.

```
       DEFINE('REDUCE(R)C')
                 .
                 .
                 .
REDUCE C   =   GCD(N(R),D(R))
       REDUCE  =   RATIONAL(N(R) / C,D(R) / C)  :(RETURN)
```

The arithmetic operations can now be written in which results are reduced to lowest terms. An example is:

```
       DEFINE('MULRTL(R1,R2)')
                 .
                 .
                 .
MULRTL MULRTL  =   REDUCE(RATIONAL(N(R1) * N(R2),
+                 D(R1) * D(R2)))         :(RETURN)
```

Extending the arithmetic operations of SNOBOL4 to include rationals as well as integers can be accomplished in a fashion similar to that used for complex numbers. Precautions must be taken in handling special cases, and the solutions are somewhat different. For example, while it probably is not desirable to convert a complex number with a zero imaginary part into a real number, it probably *is* desirable to convert a rational number whose denominator is one into an integer. Another problem arising for rationals is the equivalence of $p/-q$ and $-p/q$. It is usually assumed that the denominator is positive; if a rational number is negative, it is the numerator that is negative. To produce uniform results and to simplify comparisons, it is often desirable to keep mathematical objects in a "canonical form" if possible. REDUCE only goes part way; the result is left as an exercise.

EXERCISES

4.1 Write procedures for the functions STRRTL and RTLSTR described above.

4.2 Write a function to add rational numbers.

4.3 Generalize the built-in division operator to accept RATIONAL operands and allow RATIONAL results.

4.4 The string representation of a rational number can be thought of as an extended form of a "numeral string". SNOBOL4 automatically converts numeral strings corresponding to integers and real numbers. Generalize the built-in addition operator to handle RATIONAL operands and numeral strings representing rationals.

4.5 Why does REDUCE create a new RATONAL rather than modifying the fields of its argument?

4.6 Discuss possible ways of handling a zero denominator.

4.7 Write a recursive procedure for GCD.

4.8 Write a function to put rational numbers in canonical form.

4.9 Let a "complex rational" be a complex number whose real and imaginary parts are rational numbers rather than real numbers. An example of a complex rational is

$$\frac{5}{6} + \frac{3}{7}i$$

(a) Devise a representation for complex rationals using a defined data type CPXRTL. Illustrate schematically how the number above would appear in this representation.
(b) Devise a string representation for complex rationals and write functions to convert between the string and data type representations.
(c) Write a function to add CPXRTL s.

4.10 Let a "rational complex" be a rational number whose numerator and denominator are complex numbers rather than integers. An example is

$$\frac{5 + 3i}{6 + 7i}$$

(a) Carry out the preceding exercise for complex rationals, using a data type RTLCPX and writing corresponding functions for conversion and addition.
(b) Write functions to convert between corresponding CPXRTLs and RTLCPX s.

4.1.2. Large Integers

Integers in SNOBOL4 are limited in size. This limitation is imposed by implementation considerations and is related to the architecture of computers on which SNOBOL4 is implemented. There are, nevertheless, situations in which large numbers must be manipulated. While real numbers allow for very large numerical values, these values are nonetheless imprecise. Where precise, but large, values must be handled, other techniques must be used.

Unlike complex numbers and rationals, large integers are not naturally thought of as objects with a fixed number of fields. One way to represent a large integer is as a number of integer segments, where each segment is small enough in magnitude to be handled as an integer. For example, the integer 1,425,325,678,963,542, represented as a string `1425325678963542` might be broken into segments as follows:

```
1425   3256   7896   3542
```

The idea is that large integers can be processed by parts where each part is numerically small enough so that overflow will not occur when the segments are used in arithmetic operations. The size of segments is determined by the maximum integer allowed in SNOBOL4, which varies from implementation to implementation. If, for example, the maximum integer in a particular implementation is 10^{10}, and multiplication is to be performed on segments, segments must not be greater than 10^5. Mathematically, segmentation of a large integer corresponds to representing that integer in a large base. Suppose the base, b, is 10^4. Then the integer above is

$$1425*b^3 + 3256*b^2 + 7896*b + 3542$$

On the other hand, if b is 10^5, then the integer is

$$1*b^3 + 42532*b^2 + 56789*b + 63542$$

We will assume that large integers can be of arbitrary size, and hence there is no limitation to the number of segments that may be required to represent a particular large integer. A linked list, as described in Section 3.2.3, provides a natural structure for representing large integers. A data type for large integers might be:

```
       DATA('LRGINT(SEGMENT,NEXT)')
```

Creating a large integer from a string may be done as follows:

```
       DEFINE('STRLRG(S)R')
       BASE   =   10000
       MAXIP    =   RTAB(SIZE(BASE - 1)) . S REM . R
                  .
                  .
                  .
STRLRG S   MAXIP                               :F(STRLR1)
       STRLRG    =   LRGINT(R,STRLRG(S))        :(RETURN)
STRLR1 STRLRG    =   DIFFER(S) LRGINT(S)        :(RETURN)
```

Notice the use of pattern matching to break the string into segments. The linked list is created with the least significant segments at the top. For example, the list for 3,765,197,658,102,103 can be visualized as shown in Figure 4.1.

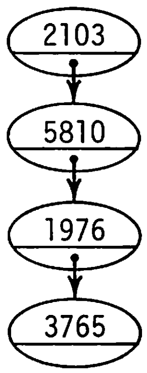


Figure 4.1 A Large Integer

The reason the least significant segments are placed at the top is because these segments are processed first in most arithmetic operations.

Arithmetic operations on large integers are list-processing problems. One of the simpler examples is addition:

```
        DEFINE('ADDLRG(L1,L2,C)')
                  .
                  .
                  .
ADDLRG  ADDLRG  =  IDENT(L1) L2              :S(ADINTC)
        ADDLRG  =  IDENT(L2) L1              :S(ADINTC)
        ADDLRG  =  SEGMENT(L1) + SEGMENT(L2) + C
        ADDLRG  =  LRGINT(REMDR(ADDLRG,BASE),ADDLRG(NEXT(L1),
+                  NEXT(L2),ADDLRG / BASE))  :(RETURN)
ADINTC  EQ(C,0)                              :S(RETURN)
        ADDLRG  =  ADDLRG(ADDLRG,LRGINT(C))  :(RETURN)
```

Note that ADDLRG has three arguments: two large integers and an (integer) carry. Simply adding two large integers involves no carry. However, ADDLRG calls itself recursively and in some cases there may be a nonzero carry resulting in "segment overflow" when adding two segments. The first two statements of ADDLRG deal with situations which occur when the end of a list is encountered. In the most general case, two segments and a carry are added. The result is an integer, but that integer may be numerically larger than the allowed size of a segment. Division by BASE and the corresponding remainder provide the excess (carry) and the resulting segment, respectively. Thus, a new large integer is formed with the new segment and is linked to the result of adding the next segments of the large integers. Any carry remaining from forming the new segment is included in the addition. If either list is exhausted, as determined by the first two statements of ADDLRG, the value of ADDLRG becomes the other list. If there is no carry, the other list is simply returned as value. Otherwise, the carry is made into a large integer and then added to the list. Note what happens if both lists are exhausted simultaneously.

It is important to be aware of the consequences of returning one of the arguments as value. Consider, for example, the two large integers illustrated in Figure 4.2.

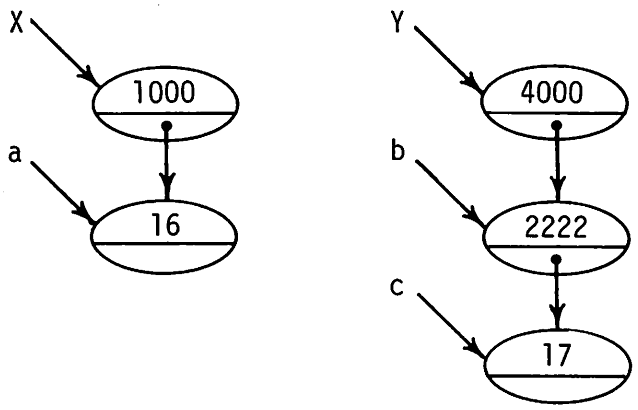


Figure 4.2 Two Large Integers

These large integers correspond to 161000 and 1722224000 respectively. The statement

```
Z = ADDLRG(X,Y)
```

produces the equivalent of the statement

```
ADDLRG = LRGINT(5000,ADDLRG(a,b,0))
```

where a and b represent the pointers shown in Figure 4.2. The inner call of ADDLRG in turn produces the equivalent of the statement

```
ADDLRG = LRGINT(2238,ADDLRG(,c,0))
```

The null argument of this inner call causes c to be returned as value. The resulting structure assigned to Z is shown in Figure 4.3.

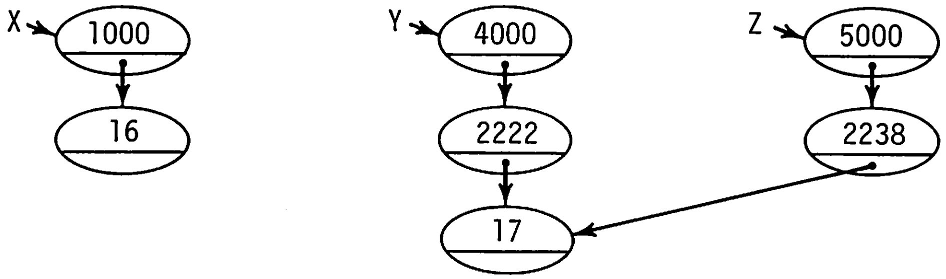


Figure 4.3 A Result of ADDLRG

Note that Y and Z now share a segment. This is another example of a side effect that is subtle and may, for most purposes, go unnoticed. In fact, as long as no operation on large integers modifies a segment or adds anything to the bottom of a list, this sharing of structures offers an advantage in the saving of space and time required to copy structures.

EXERCISES

4.11 If the maximum integer in an implementation of SNOBOL4 is 10^{10}, what is the maximum base for a large integer if
(a) Only addition is to be performed on segments?
(b) Exponentiation may be performed on segments?

4.12 Why is a power of ten a desirable value for the base?

4.13 What is the effect of leading zeroes in the argument of STRLRG?

4.14 Add a test to STRLRG to detect invalid arguments.

4.15 Write a function to convert integers to large integers.

4.16 Write a function to convert large integers to strings.

4.17 Write a function to copy large integers.

4.18 Write functions to perform arithmetic comparisons on large integers.

4.19 STRLRG produces segments that are numeral strings, not integers. ADDLRG, however, produces integer segments. For uniformity, modify STRLRG to produce integer segments.

4.20 Write a function to multiply large integers.

4.21 Write an *iterative* procedure for ADDLRG.

4.22 The treatment of large integers in the text is limited to positive values. Develop a method for representing negative values. Modify the functions for processing large integers accordingly.

4.23 Write a function to subtract large integers.

4.24 Write a function to divide large integers.

4.25 Rewrite ADDLRG so that the result does not share segments with its arguments.

4.26 Develop a set of functions for handling rational numbers whose numerators and denominators are large integers.

4.27 Develop a set of functions to operate on large real numbers.

4.28 Palindromic numbers (see Exercise 1.21) have a number of interesting properties. There is a conjecture concerning palindromic numbers as follows [25]. Starting with any positive integer, reverse it and add the two integers. Repeat this process with the result. If this process is continued, a palindromic number is eventually obtained. An example is

$$\begin{array}{r} 37 \\ \underline{73} \\ 110 \\ \underline{011} \\ 121 \end{array}$$

This conjecture is known to be false for numbers written in the base 2. As of this writing, the conjecture has not been resolved for numbers written in the base 10. Explore this conjecture for the base 10. (*Caution*: Although most decimal numbers produce palindromic sums quickly, some (such as 196) do not.)

4.1.3. Polynomials

A polynomial is a form of mathematical expression that is arranged as a sum of products. The products consist of integer coefficients and variables raised to nonnegative integer powers. An example is:

$$-3x^2y^2z + 5x^2y + xyz^2 - 3xz + 7y - 2$$

Typical operations that are performed on polynomials are addition and multiplication. Such a polynomial is represented as a parenthesized expression using the operators +, -, *, and ! (for exponentiation) as follows:

```
((((((((-3*(X!2))*(Y!2))*Z)+((5*(X!2))*Y))+((X*Y)*(Z!2)))-((
3*X)*Z))+(7*Y))-2)
```

Not only is such a representation cumbersome, but it is awkward for typical polynomial operations. In fact, the essential aspects of a polynomial are its coefficients and the exponents of the variables. The polynomial above could have a string representation as:

```
(-3:2,2,1)(5:2,1,0)(1:1,1,2)(-3:1,0,1)(7:0,1,0)(-2:0,0,0)
```

where each parenthesized group represents a term of the polynomial and each term starts with a coefficient and is followed by a list of exponents. This is, of course, only one of many possible string representations. Note, for example, that this representation does not include the variables of the polynomial. These variables may or may not be implied in a particular situation.

Polynomials present a special problem in representation because of the operations typically performed on them. For example, polynomial addition requires adding coefficients of like exponents. Quite typically, however, the exponents present in any particular polynomial are "sparse", i.e., only a few of the possible terms have nonzero coefficients. Including terms with zero coefficients is often wasteful or impractical. Consider, for example, the polynomial $x^{201} - 1$.

The problem of representing sparse objects as well as kinds of objects that have no specified limits on their complexity provides a situation in which tables can be used effectively in conjunction with arrays. A representation of the polynomial given above could be constructed by the following statements:

```
        P   =   TABLE()
        P<'2,2,1'>   =   -3
        P<'2,1,0'>   =   5
        P<'1,1,2'>   =   1
        P<'1,0,1'>   =   -3
        P<'0,1,0'>   =   7
        P<'0,0,0'>   =   -2
```

The exponents of terms are used as subscripts whose values are (nonzero) coefficients. A function to convert a string representation of a polynomial into such a form follows:

```
          DEFINE('STPOLY(S)EXPN,COEFF')
          TERM   =   '(' BREAK(':') . COEFF LEN(1) BREAK(')')
+             . EXPN LEN(1)
                  .
                  .
                  .
STPOLY    STPOLY   =   TABLE()
STPOL1    S   TERM   =                              :F(RETURN)
          STPOLY<EXPN>   =   COEFF                  :(STPOL1)
```

This type of representation makes it possible to reference a term "associatively", simply by referring to it. A function to add two polynomials is:

```
          DEFINE('ADPOLY(P1,P2)I')
                  .
                  .
                  .
ADPOLY P1 = CONVERT(CONVERT(P1,'ARRAY'),'TABLE')  :F(ADPOL3)
          P2   =   CONVERT(P2,'ARRAY')            :F(ADPOL2)
          I   =   1
ADPOL1    P1<P2<I,1>>   =   P1<P2<I,1>> + P2<I,2>   :F(ADPOL2)
          I   =   I + 1                           :(ADPOL1)
ADPOL2    ADPOLY   =   P1                         :(RETURN)
ADPOL3    ADPOLY   =   P2                         :(RETURN)
```

The first statement makes a copy of P1 that can be used in the course of the computation without modifying the structure pointed to by P1. The necessity for using CONVERT twice results from the fact that most implementations of SNOBOL4 do not provide a way of copying tables directly. The second statement creates an array corresponding to P2 and hence provides a representation of the second polynomial that can be indexed. Polynomial addition takes place by subscripting P1 according to exponents in P2. Two matters deserve special consideration. In the first place, if either P1 or P2 is empty (corresponding to a zero polynomial), the other argument (possibly also empty) is returned as value. The normal representation of a zero polynomial is an empty table (which cannot be converted to an array). In the second place, ADPOLY does not quite preserve the form for polynomials as implied by the original definition. A zero coefficient may result from addition of coefficients of like magnitude but opposite signs.

Polynomial multiplication is slightly more complicated. All possible products of terms must be constructed. To simplify this process, a function ADEXPT can be written to add two exponents in the form given above.

```
       DEFINE('ADEXPT(X1,X2)')
       EXP1    =   BREAK(',') . E1 LEN(1) | (LEN(1) REM) . E1
       EXP2    =   BREAK(',') . E2 LEN(1) | (LEN(1) REM) . E2
                   .
                   .
                   .
ADEXPT X1   EXP1   =                           :F(ADEX1)
       X2   EXP2   =                           :F(ERROR)
       ADEXPT   =   ADEXPT ',' E1 + E2         :(ADEXPT)
ADEX1  ADEXPT   ','   =                        :(RETURN)
```

Using this function, multiplication follows naturally:

```
       DEFINE('MLPOLY(P1,P2)I,J,X,C,E')
                 .
                 .
                 .
MLPOLY P1   =   CONVERT(P1,'ARRAY')            :F(RETURN)
       P2   =   CONVERT(P2,'ARRAY')            :F(RETURN)
       MLPOLY   =   TABLE()
MLP1   I   =   I + 1
       X   =   P1<I,1>                         :F(RETURN)
       C   =   P1<I,2>
       J   =   0
MLP2   J   =   J + 1
       E   =   ADEXPT(X,P2<J,1>)               :F(MLP1)
       MLPOLY<E>   =   MLPOLY<E> + C * P2<J,2>   :(MLP2)
```

EXERCISES

4.29 The representation for polynomials in the text uses a table as the basic form and an array as an alternate form. Rewrite the functions using an array as the basic form and a table as the alternate form.

4.30 Since large integers are represented as polynomials in a base b,
(a) Why is it desirable to represent a large integer as a linked list rather than using the polynomial method?
(b) Why is a linked list a poor method of representation for polynomials in general?

4.31 Modify `ADPOLY` and `MLPOLY` so that terms with zero coefficients are omitted.

4.32 Terms of a polynomial are usually ordered according to the values of the exponents as illustrated by the example in the text. Describe this ordering precisely and write a predicate that compares terms accordingly.

4.33 Write a function to convert the table representation of a polynomial to the string representation. Put the terms in order by exponent.

4.34 Write a function to copy polynomials.

4.35 Write a function to determine if two polynomials are equal.

4.36 Devise a method of incorporating the variables of a polynominal in its representation.

4.37 Write a function to evaluate a polynomial for specified values of its variables.

4.38 A continued fraction has the form

$$a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{a_4 + \cdots}}}$$

where the a_i are integers. Such an expression can be represented more compactly by the sequence $(a_1, a_2, a_3, a_4, \ldots)$. Rational numbers can be represented as continued fractions with a finite number of terms. An example is

$$86/11 = 7 + \cfrac{1}{1 + \cfrac{1}{4 + \cfrac{1}{2}}} = (7,1,4,2)$$

(a) Devise a data-type representation for finite continued fractions.
(b) Devise a string representation for finite continued fractions.
(c) Write functions to convert between the string and data-type representations of finite continued fractions.
(d) Write a function to convert rational numbers to continued fractions. If necessary, consult a mathematics text, such as Reference 26, for the method.

Irrational numbers can be represented as infinite continued fractions. Quadratic irrationals correspond to periodic continued fractions. An example is

$$\sqrt{2} = (1,2,2,2,2, \dots)$$

(e) Show how the data-type representation for finite continued fractions can be extended to handle periodic infinite continued fractions.
(f) Devise a string representation for periodic infinite continued fractions.
(g) Write functions to convert between the string and data-type representations of periodic infinite continued fractions.

4.2. OPERATIONS ON EXPRESSIONS

The topics covered so far have dealt with objects that are essentially numerical in nature. Although polynomials are really expressions, operations on them are basically numerical. Symbolic mathematics—formula manipulation, algebraic transformations, theorem proving—shows most clearly the power of string manipulation techniques. Curiously, these same problems show clearly the contrast between string processing and list processing.

Also of interest is the fact that the major motivations for the development of the SNOBOL languages were problems in symbolic mathematics: factoring polynomials and the simplification of algebraic expressions. The SNOBOL languages have changed markedly since their inception, but SNOBOL4 still shows the influence of these early problems.

There are many problems in symbolic mathematics; some have direct practical importance while others are concerned with fundamental research. Quite a bit of work has been done in symbolic mathematics [1,27,28]. All we can hope to do here is discuss a few elementary considerations and illustrate some of the main points.

4.2.1. Expressions

Mathematical expressions come in many forms, depending on the area of interest. For simplicity, we shall consider expressions composed of binary operations on operands that are variables and integers. For convenience, we shall consider variables to consist of letters, and, for the moment, only allow the operations of addition, subtraction, multiplication, division, and exponentiation. For review, these operators have the relative precedence and associativity shown below:

operation	*symbol*	*precedence*	*associativity*
exponentiation	!	3	right
multiplication	*	2	left
division	/	2	left
addition	+	1	left
subtraction	-	1	left

Parentheses are employed as usual for grouping terms, either to group operands differently from the grouping implied by precedence or associativity, or to make expressions more readable. For example, if parenthesized according to precedence and associativity, the expression

```
A-7+3*C!2!3
```

becomes

```
((A-7)+(3*(C!(2!3))))
```

While most parentheses are usually omitted in hand work, manipulation of expressions in a program is much easier and more efficient if the relationships between operators and operands are unambiguously delineated in the structure of the expression. Although reasonably simple conceptually, full parenthesization of an infix form presents some practical difficulties. The operators with lowest precedence (+ and -) are dealt with first. The reason for this may be seen by considering the following expression:

```
A*B+C*D/E
```

Since + has lower precedence than *, the operands of + are A*B and C*D/E. That is to say, the + is the innermost operator in the fully parenthesized form, and hence is conveniently handled first.

For operators that associate to the left (and most do), the rightmost operator of a given precedence must be dealt with first. The following expression illustrates this point:

```
A+B+C-D-E+F
```

The parenthesized form of this expression is

```
(((((A+B)+C)-D)-E)+F)
```

Here, the outermost operator in the fully parenthesized form is the rightmost (last) operator in the original expression.

A recursive solution should be expected, since any expression is an instance of a recursively-defined set of strings. If the outermost operator is determined, the conversion function can be applied recursively to its operands.

The practical difficulty comes in locating the rightmost left-associative operator. As discussed in Section 1.3, pattern matching is basically a left-to-right process. As a result, the first instance of an operator found by pattern matching is ordinarily the leftmost one. For example, the statement

```
        EXP   POS(0) BAL . L ANY('+-') . OP BAL . R RPOS(0)
```

finds the leftmost + or - in an expression. BAL assures that any parentheses that may occur in the expression are treated properly. Consequently, a + or - that occurs inside a nested expression is not matched.

There are several ways around the problem of left-to-right pattern matching. One is to reverse the string, achieving the effect of right-to-left pattern matching. String reversal is generally time consuming, and if the expression contains parentheses, they appear in the wrong order for BAL. This, too, can be circumvented, but at the expense of time and complexity. Another method is to match the string repeatedly, removing each segment before a + or -. When this can no longer be done, the last successful result indicates the correct operator. Here too, there are complexities and clerical difficulties in keeping track of the intermediate segments and reassembling the result. A third alternative, and the one that is used here, is to first locate the position of the rightmost operator, and then to separate the expression accordingly. Locating the position of the rightmost operator can be accomplished by using a pattern that keeps trying to find another operator every time it finds one. Such a pattern is destined to fail, but it can leave, as a side effect, the position of the last operator it found. Consider the following pattern:

```
        LOCPM   =   POS(0) BAL ANY('+-') @M FAIL
```

The first part of this pattern describes the condition for an acceptable + or - operator in a string: a balance expression, starting at the beginning of the string, followed by a + or -. If such a construction is found, the position following the operator is assigned to M. FAIL forces match failure. As a result, BAL extends the string matched, if possible, up to the next + or -. This process continues until BAL can no longer be extended. At this point, LOCPM fails, as it necessarily must, but the rightmost operator has been located and its position has been recorded as the value of M. The value of M can now be used to split the string into two parts. Of course, if the expression contains no acceptable occurrence of a + or -, LOCPM also fails. To avoid ambiguity, M can be set to zero before LOCPM is used. Then a nonzero value of M after matching indicates the presence of an operator. An entire pattern to locate the operator and divide the expression follows:

```
         MATPM   =   (POS(0) @M BAL ANY('+-') @M FAIL) |
+                (*GT(M,0) TAB(*(M - 1)) . L LEN(1) . OP REM . R)
```

In this pattern, the first alternative assigns a value to M. The initial value of M is set to zero before location of an operator is attempted. When the first alternative fails, as it must, the second alternative separates the string into its components, provided M is greater than zero. MATPM fails only if there is no occurrence of an acceptable operator. A function to put an infix expression into fully parenthesized form follows:

```
         DEFINE('PAREN(PAREN)L,R,OP,M')
         STRIP   =   POS(0) '(' BAL . PAREN ')' RPOS(0)
         ASSIGN = *GT(M,0) TAB(*(M - 1)) . L LEN(1) . OP REM .R
         MATPM   =   (POS(0) BAL ANY('+-') @M FAIL) | ASSIGN
         MATMD   =   (POS(0) BAL ANY('*/') @M FAIL) | ASSIGN
         MATE    =   POS(0) BAL . L '!' . OP REM . R
                     .
                     .
                     .
PAREN    PAREN   STRIP                                  :S(PAREN)
         PAREN   MATPM                                  :S(FORM)
         PAREN   MATMD                                  :S(FORM)
         PAREN   MATE                                   :F(RETURN)
FORM     PAREN   =    '(' PAREN(L) OP PAREN(R) ')'   :S(RETURN)
```

In this function, M is a local variable and hence is initially null (zero-valued) when the function is called. Consequently, M does not have to be set to zero by the patterns that locate operators.

4.2.2. Differentiation

The classical "textbook" problem in symbolic mathematics is differentiation. Differentiation can be expressed as a set of transformations, called *derivatives*, that are applied to expressions. It is not necessary to understand calculus to appreciate the process of differentiation—it can be considered simply as a formal exercise in string transformation. The derivative of an expression E with respect to a variable X is designated by D(E,X). The most familiar derivatives follow. In these derivatives, U and V stand for expressions and C is a constant (i.e., an expression not containing an occurrence of the variable X). The notation used below corresponds roughly to the string representation for expressions.

```
        D(C,X) = 0
        D(X,X) = 1
        D(-U,X) = -D(U,X)
        D(U+V,X) = D(U,X)+D(V,X)
        D(U-V,X) = D(U,X)-D(V,X)
        D(U*V,X) = U*D(V,X)+V*D(U,X)
        D(U/V,X) = (V*D(U,X)-U*D(V,X))/V!2
        D(U!C,X) = C*U!(C-1)*D(U,X)
        D(SIN(U),X) = COS(U)*D(U,X)
        D(COS(U),X) = -(SIN(U)*D(U,X))
```

There are, of course, many other derivatives. The ones here are more than sufficient for our purposes.

Since derivatives are essentially transformation rules, differentiation can be thought of in terms of pattern matching and rewriting, where each different form of expression produces a different rewriting statement. A rudimentary function, designed along these lines, and limited to binary operators, follows:

```
        DEFINE('D(E,X)U,V,OP')
        BINARY   =    POS(0) '(' BAL . U ANY('+-*/!') . OP BAL . V ')'
+                    RPOS(0)
                 .
                 .
                 .
D       E   BINARY                              :S($('D' OP ))
        D   =   IDENT(E,X) 1                    :S(RETURN)
        D   =   0                               :(RETURN)
D+
D-      D   =   '(' D(U,X) OP D(V,X) ')'        :(RETURN)
D*      D   =   '((' U '*' D(V,X) ')+(' V '*' D(U,X) '))'    :(RETURN)
D/      D   =   '(((' V '*' D(U,X) ')-(' U '*' D(V,X) '))/(' V '!2))'
+                                               :(RETURN)
D!      D   =   '((' V '*(' U '!' V - 1 '))*' D(U,X) ')'     :(RETURN)
```

If an expression contains a binary operator, an appropriate statement is selected using the operator to compute a goto.

A test program for differentiating expressions appears below. Data is read in, one expression to a card. The variable with respect to which differentiation is to be performed follows the expression after a separating semicolon. Note that PAREN is applied to expressions before D is called.

```
                  .
                  .
                  .
       &TRIM   =   1
       IMAGE   =   BREAK(';') . EXP LEN(1) REM . VAR
                  .
                  .
                  .
READ   CARD   =   INPUT                          :F(END)
       CARD   IMAGE                              :F(ERROR)
       OUTPUT = 'THE DERIVATIVE OF ' EXP ' WITH RESPECT TO '
+             VAR ' IS '
       OUTPUT   =   D(PAREN(EXP),VAR)
       OUTPUT   =                                :(READ)
                  .
                  .
                  .
```

Some typical output from this program follows:

```
THE DERIVATIVE OF X WITH RESPECT TO X IS
1

THE DERIVATIVE OF Y WITH RESPECT TO Y IS
1

THE DERIVATIVE OF X WITH RESPECT TO Y IS
0

THE DERIVATIVE OF 2 WITH RESPECT TO Z IS
0

THE DERIVATIVE OF X+Y WITH RESPECT TO Y IS
(0+1)

THE DERIVATIVE OF 2*X WITH RESPECT TO X IS
((2*1)+(X*0))
```

```
THE DERIVATIVE OF X+X!10 WITH RESPECT TO X IS
(0+((10*(X!9))*1))
 1

THE DERIVATIVE OF X/Y WITH RESPECT TO X IS
(((Y*1)-(X*0))/(Y!2))

THE DERIVATIVE OF X!2 WITH RESPECT TO X IS
((2*(X!1))*1)

THE DERIVATIVE OF X!3/15*X!2 WITH RESPECT TO X IS
((((X!3)/15)*((2*(X!1))*1))+((X!2)*(((15*((3*(X!2))*1))-((X!3)*0))/
(15!2))))
```

The most immediately obvious aspect of this output is the abundance of superfluous terms. This problem is pervasive in the manipulation of symbolic expressions and is a constant source of annoyance and frustration.

The problem of simplifying such expressions is one that has been given considerable attention. There are several levels at which simplification can be approached. One is "identity reduction" in which terms such as `1*E` and `0*E` are replaced by `E` and `0` respectively. Carrying out integer arithmetic is another form of simplification. Detection and removal of common algebraic factors is substantially more difficult. (Some care must be taken in removing common factors. Consider the effect of removing the common factor $(x + y)$ from the expression $(x^{100} - y^{100})/(x + y)$.)

A somewhat different approach to the first two kinds of simplification mentioned above is to avoid the generation of superfluous terms in the first place. Thus, if differentiation of a sum produces a term that is zero, the other term may simply be returned as value. This approach may be formulated in a more elegant way as a generalization of the arithmetic operators. Consider addition as an example: if both operands are integers, the result is the integer sum. If, however, the operands are symbolic expressions, the result is obtained by symbolic addition. For example, the sum of `2` and `3` is 5, but the sum of `X` and `Y` is `(X+Y)`. If the operands have different types, the result is also obtained by symbolic addition unless an operand is zero. A function to perform this type of extended addition follows:

```
        DEFINE('ADD(L,R)')
                  .
                  .
                  .
ADD     INTEGER(L)                                :F(ADDR)
        ADD   =   INTEGER(R) L + R                :S(RETURN)
        ADD   =   EQ(L,0) R                       :S(RETURN)
ADDP    ADD   =   '(' L '+' R ')'                 :(RETURN)
ADDR    INTEGER(R)                                :F(ADDP)
        ADD   =   EQ(R,0) L                       :S(RETURN)F(ADDP)
```

The differentiation function can now be modified to call such functions instead of simply concatenating as before. The result is:

```
D       E    BINARY                                  :S($('D' OP))
        D    =    IDENT(E,X) 1                       :S(RETURN)
        D    =    0                                  :(RETURN)
D+      D    =    ADD(D(U,X),D(V,X))                 :(RETURN)
D-      D    =    SUB(D(U,X),D(V,X))                 :(RETURN)
D*      D    =    ADD(MUL(U,D(V,X)),MUL(V,D(U,X)))   :(RETURN)
D/      D    =    DIV(SUB(MUL(V,D(U,X)),MUL(U,D(V,X))),EXP(V,2))
+                                                    :(RETURN)
D!      D    =    MUL(MUL(V,EXP(U,SUB(V,1))),D(U,X))  :(RETURN)
```

The output of the previous example, using these new functions, follows:

```
THE DERIVATIVE OF X WITH RESPECT TO X IS
1

THE DERIVATIVE OF Y WITH RESPECT TO Y IS
1

THE DERIVATIVE OF X WITH RESPECT TO Y IS
0

THE DERIVATIVE OF 2 WITH RESPECT TO Z IS
0

THE DERIVATIVE OF X+Y WITH RESPECT TO Y IS
1

THE DERIVATIVE OF 2*X WITH RESPECT TO X IS
2

THE DERIVATIVE OF X+X!10 WITH RESPECT TO X IS
(1+(10*(X!9)))

THE DERIVATIVE OF X/Y WITH RESPECT TO X IS
(Y/(Y!2))

THE DERIVATIVE OF X!2 WITH RESPECT TO X IS
(2*X)

THE DERIVATIVE OF X!3/15*X!2 WITH RESPECT TO X IS
((((X!3)/15)*(2*X))+((X!2)*((15*(3*(X!2)))/225)))
```

Comparison of the two sets of results reveals that considerable simplification has been achieved. There are still some obvious redundancies. Some are easy to handle; others are more difficult.

An even more elegant formulation of the differentiation function is obtained by redefining the arithmetic operators (and rewriting the arithmetic functions correspondingly):

```
D       E   BINARY                                  :S($('D' OP))
        D   =   IDENT(E,X) 1                        :S(RETURN)
        D   =   0                                   :(RETURN)
D+      D   =   D(U,X) + D(V,X)                     :(RETURN)
D-      D   =   D(U,X) - D(V,X)                     :(RETURN)
D*      D   =   U * D(V,X) + V * D(U,X)             :(RETURN)
D/      D   =   (V * D(U,X) - U * D(V,X)) / V ** 2  :(RETURN)
D!      D   =   V * U ** (V - 1) * D(U,X)           :(RETURN)
```

Notice how closely the differentiating procedure corresponds to its mathematical formulation.

4.2.3. Tree Representation of Expressions

In the chapter on structures, it was pointed out that trees provide a way of representing expressions. The expression

A-7+3*C!2!3

has the tree representation shown in Figure 4.4.

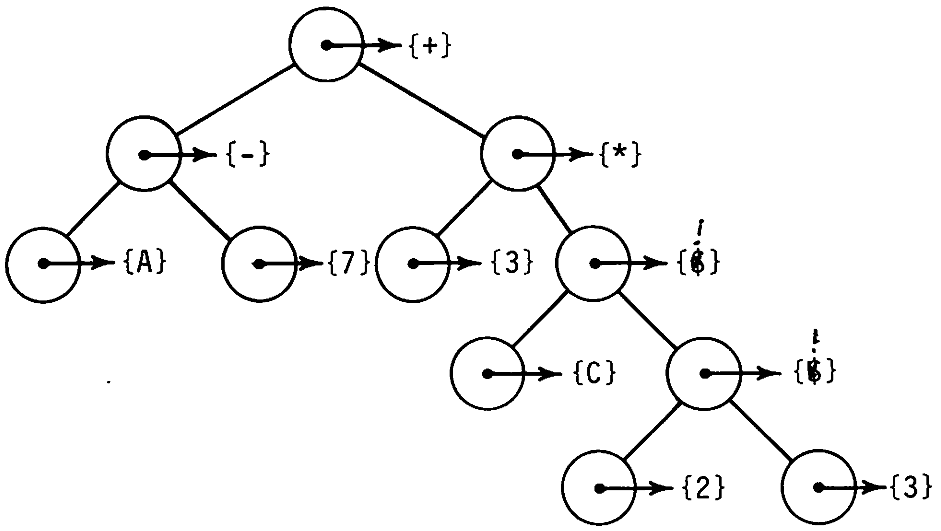


Figure 4.4 Tree Representation of an Expression

There are natural extensions to unary operators and to *n*-ary operators for all positive values of *n*. Conversion of prefix expressions to trees is discussed in Section 3.2.5. The function given in Section 4.2.1 for conversion from infix to fully parenthesized form can be modified slightly to give a direct conversion from infix to tree representation:

```
       DEFINE('INFTRE(INFTRE)L,R,OP,M')
       STRIP  =   POS(0) '(' BAL . INFTRE ')' RPOS(0)
       ASSIGN = *GT(M,0) TAB(*(M - 1)) . L LEN(1) . OP REM . R
       MATPM  =   (POS(0) BAL ANY('+-') @M FAIL) | ASSIGN
       MATMD  =   (POS(0) BAL ANY('*/') @M FAIL) | ASSIGN
       MATE   =   POS(0) BAL . L '!' . OP REM . R
                  .
                  .
                  .
INFTRE INFTRE   STRIP                                :S(INFTRE)
       INFTRE   MATPM                                :S(FORM)
       INFTRE   MATMD                                :S(FORM)
       INFTRE   MATE                                 :S(FORM)
       INFTRE   =   TNODE(INFTRE)                    :(RETURN)
FORM   INFTRE   =   TNODE(OP)
       ADDSON(INFTRE,INFTRE(L))
       ADDSIB(LSON(INFTRE),INFTRE(R))                :(RETURN)
```

The function as written only handles binary operators. Note that if the expression is simply an integer or a variable, a node containing that value is returned. The two functions PAREN and INFTRE are parallel in their structure; the only difference is in the way the result is constructed. The same parallelism, which should be expected from the isomorphism of the two representations, carries over directly to the process of differentiation. Pictorially, the process for the typical case of addition is shown in Figure 4.5.

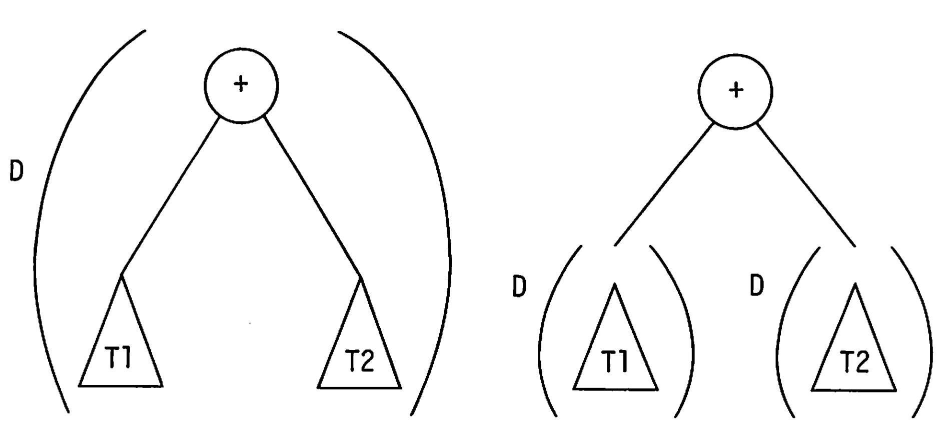


Figure 4.5 Differentiation of a Sum

The original tree is not modified. Instead a new tree is formed in which the root is the operator + and its offspring are trees obtained by differentiating the offspring in the tree being differentiated.

The actual program is somewhat more complicated since constructing trees is more involved than concatenating strings. Statements such as the following might occur in differentiating a sum:

```
        OPPAT    =    ANY('+-*/!')
                      .
                      .
                      .
D       VALUE(E)    OPPAT                     :S($('D' VALUE(E)))
        D    =    IDENT(VALUE(E),X) TNODE(1)      :S(RETURN)
        D    =    TNODE(0)                        :(RETURN)
                      .
                      .
                      .
D+      D    =    TNODE(VALUE(E))
        ADDSON(D,D(LSON(E),X))
        ADDSIB(LSON(D),D(RSIB(LSON(E)),X))        :(RETURN)
                      .
                      .
                      .
```

Differentiation for the other operators follows the same parallel with the string case.

When simplification is considered, the ideas used for the string representation apply equally well to the tree representation. A function ADD can be written for adding two trees or integers. If either argument is a tree, the result is a tree, and so on. In fact, the arithmetic operators can be extended in a fashion quite similar to the method used for strings.

4.2.4. Alternative Representations

In the examples considered above, either the string or the tree representation can be used with quite parallel results. It should be obvious that the tree representation requires more programming and produces results that are less readily understood. Not so obvious is the fact that the tree representation takes more memory space. Another disadvantage of the tree representation is the lack of convenient diagnostic facilities. Strings can simply be printed, and string expressions appear in the termination dump in their natural form. While the values of the nodes in a tree can be printed using a function developed in Section 3.2.5, the method is less convenient and fails to work if, because of an error, a tree is ill-formed.

On the other hand, there are things that can be done with trees that cannot be done conveniently with strings. For symbolic mathematics, the most important of these is the ability to handle objects that are not integers or variables—for example, rational numbers. The value of a node can be a rational number in the tree representation of expressions without the need for any modification. The possibilities are endless. In fact, the value of a node can even be (a pointer to) another expression in tree representation.

Another way of looking at expressions as structures is suggested by the use of defined data types to represent objects such as rationals. A rational number can also be thought of as an operation (division) applied to two operands (the numerator and denominator). A rational number 6/7, for example, can be represented as a tree as shown in Figure 4.6.

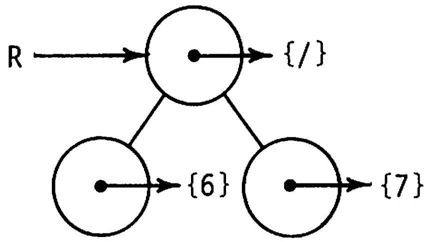


Figure 4.6 A Tree Representing 6/7

On the other hand, 6/7 can be represented by a defined data object as shown in Figure 4.7.

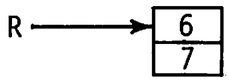


Figure 4.7 The RATIONAL 6/7

Looking at it another way, a RATIONAL need not have integer arguments; the arguments can be expressions. That is, a RATIONAL object can be thought of as a formal representation for a (symbolic) operation. Carrying this idea one step further, different data types can be defined for the operators—for example, SUM for +, DIFF for -, PROD for *, QUOT (in place of RATIONAL) for /, and finally, EXPN for !. Distinguishing shapes for these data types are shown in Figure 4.8.

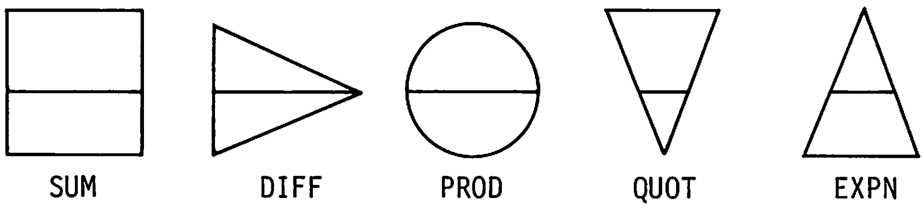


Figure 4.8 Shapes for Defined Data Types

Using these shapes, a structure for the expression

```
A-7+3*C!2!3
```

is shown in Figure 4.9.

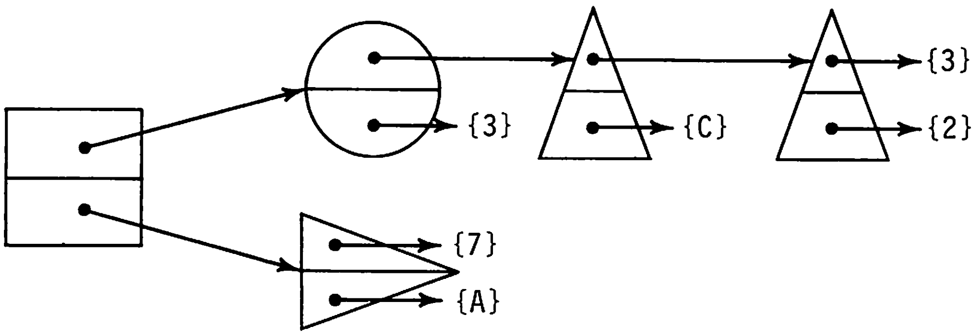


Figure 4.9 An Expression Represented by Defined Data Objects

Compare the structure given in Figure 4.9 with the tree structure shown in Figure 4.4. The use of defined data objects gives surprising advantages: the structure is simpler, smaller, and operators, as such, have vanished entirely. The structure above does not, however, provide pointers that permit getting from any point in the expression to any other. Such pointers are not needed to perform differentiation. If such pointers are needed in another context, they can be provided in a straightforward manner.

EXERCISES

4.39 Extend the differentiation function to handle the general case of exponentiation in which the second operand may be an expression.

4.40 Prefix form, described in Section 2.2, is an alternative to full parenthesization. Write functions to convert between infix and prefix forms.

4.41 Write the set of derivatives given in the text in prefix form.

4.42 Write the procedures for the extended arithmetic functions SUB, MUL, DIV, and EXP.

4.43 Write extended arithmetic operations for -, *, /, and **.

4.44 Write a differential function to operate on prefix form rather than on infix form.

4.45 Write a differentiation function to operate on the tree representation of expressions.

4.46 Upward links from tree nodes to their fathers are not needed in performing differentiation. Revise the tree structure to eliminate these links.

4.47 Write functions to convert between infix form and the data-type representation of expressions.

4.48 Write a differentiation function to operate on the data-type representation of expressions.

4.49 Extend PAREN to include unary operators and functions. (Suggestion: establish reasonable constraints on the form in which unary operators may appear in expressions.)

4.50 Extend the differentiation function to handle the unary operators + and - and the functions SIN and COS. Note that the extended binary operators may return negative integers which are, formally, instances of the unary - operator.

4.51 Design the structure of a general-purpose function for simplifying expressions. Within this framework, implement identity reduction and the performance of integer arithmetic. Explore other types of simplification.

4.52 Boolean expressions are similar to algebraic expressions except that the operators are different and there are only two constant values, 0 and 1. The Boolean operator v ("or") may be defined as follows: If X is a Boolean variable, then

```
Xv1 = 1vX = 1
Xv0 = 0vX = X
```

Write a function OR(L,R) that performs the Boolean "or" operation. Assume that either operand may be a constant or a variable.

4.53 Propositional calculus represents logical statements as expressions where variables are "sentences" represented by letters and there are operators corresponding to logical relationships as follows:

¬	(for "not...")
&	(for "...and...")
v	(for "...or...")
→	(for "if...then...")
≡	(for "...if and only if...")

A typical statement is

```
(LvM)→(S≡(G&(¬R)))
```

Assuming such expressions are fully parenthesized,

(a) Show how this statement can be represented by a tree.

(b) Develop a data-type representation for logical statements and illustrate how the statement above would be represented.

(c) Write a function to convert a data-type representation of a logical statement into a string representation of the statement.

5 CRYPTOGRAPHY

Cryptography, the "science of secret communication", is familiar popularly in the form of puzzles and descriptions of its role in dramatic historical situations. Methods of protecting the integrity of information by modifying its form have ancient origins. Cryptography also has important practical modern uses. Protecting the security of governmental communications, especially in periods of conflict, is one of the best-known applications. Industrial secrets are also often given cryptographic protection. More recently the advent of computer data bases containing confidential information has focused attention on the importance of methods for assuring privacy.

Cryptographic puzzles appear regularly in newspapers and magazines and there are organizations of individuals interested in cryptography. There is a substantial amount of literature on cryptography [29-35]. Most of the generally available information on cryptography deals with methods that were in use prior to the early part of the twentieth century. More modern techniques, even some dating back to World War I, are still highly classified government secrets.

This chapter treats cryptography from a programming point of view, using the older "classical" methods as a basis. There is, of course, a substantial difference between what is practical and relevant with modern technology and what was practical and relevant when messages had to be transmitted by courier. Many of the older methods were designed to minimize clerical effort and human error. Emphases are much different now with the availability of electronic methods of communication and computers for processing messages. Nonetheless, many of the classical methods illustrate general principles. Several techniques of historical interest are not relevant to computer processing, and hence are omitted. The interested reader will find more material in the references. Because SNOBOL4 is a string-manipulation language, cryptography is approached at the character level, which corresponds to classical cryptography. Other programming languages permit manipulations at the bit level. Although manipulation of bits permits processes that give a different appearance than those that perform character manipulation, the underlying methods are the same.

The basic model is quite simple: a message is enciphered, passed through a transmission link, and deciphered at the other end. The purpose of a cipher is to obscure, and hence safeguard, information. The terms "message" and "transmission link" are used loosely; there are more general contexts in which cryptography applies. For example, data may be enciphered before it is placed in a data base and deciphered when it is retrieved. The term "cipher" is usually used to mean a systematic method of rearranging or making substitutions in a message in order to obscure its meaning. The word "code" is used, on the other hand, for techniques that make substitutions for words and syllables, or common phrases. The distinction between ciphers and codes is not always clear, but we are concerned here with systematic processes and hence will use the term cipher. Figure 5.1 illustrates a model of the enciphering and deciphering process.

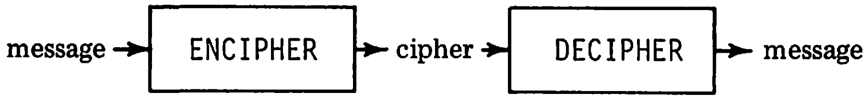


Figure 5.1 A Model of the Enciphering and Deciphering Process

In this figure, the boxes containing ENCIPHER and DECIPHER correspond to transformations applied to text. DECIPHER is the inverse of ENCIPHER, so that the process of passing the message through the two transformations in series produces the original message unchanged. In practice, many enciphering transformations do not have strict inverses, and may destroy punctuation or word separation in the original message. We will take a more formal approach, and will only consider transformations that have true inverses so that the message resulting from the deciphering will be identical to the original message. In the material that follows, all characters are treated equally, independent of their meaning in the context in which they appear in messages. Furthermore, we consider messages as being simply strings of characters, and generally are not concerned about problems of physical format and so on. Similarly, the transmission link usually is considered to be perfect so that no loss of information occurs. In fact, it is the enciphering and deciphering processes themselves that are of interest. From a programming point of view, enciphering can be characterized by a function ENCIPHER(M) which performs the desired enciphering transformation. Similarly, DECIPHER(C) is a deciphering function that performs the inverse of the transformation performed by ENCIPHER. The required condition for

DECIPHER to be the inverse of ENCIPHER can be stated in a programming context by requiring that

```
     IDENT(DECIPHER(ENCIPHER(M)),M)
```

succeed for all possible values of M.

Most of this section is devoted to a discussion of the different types of ciphers and how the corresponding functions ENCIPHER and DECIPHER can be written. In some cases, the transformations used for enciphering and deciphering have applications that are not related to the problem of safeguarding the contents of messages by obscuring them. Such applications are pointed out from time to time.

5.1. CIPHERS

5.1.1. Types of Ciphers

Generally speaking, there are three types of ciphers: substitution, transposition, and combination. A substitution cipher replaces units of the message by other units. A transposition cipher rearranges text units. A combination cipher combines substitution and transposition. There are many varieties of each type of cipher. In most cases, the units of text considered are single characters, although that need not be the case.

Several ciphers may be applied in succession, producing a product of ciphers as illustrated in Figure 5.2. The deciphering transformations are then applied in reverse order.

$$\begin{tikzcd}
M \arrow[r] & \boxed{E1} \arrow[r] & \boxed{E2} \arrow[r] & \cdots \arrow[r] & \boxed{En} \arrow[r] & C \arrow[r] & \boxed{Dn} \arrow[r] & \cdots \arrow[r] & \boxed{D2} \arrow[r] & \boxed{D1} \arrow[r] & M
\end{tikzcd}$$

Figure 5.2 A Product of Ciphers

Here E1, E2, ... , En are enciphering transformations and D1, D2, ... , Dn are corresponding deciphering transformations. From a programming point of view, a product of transformations corresponds to the successive application of the corresponding functions:

```
     C  =  EN( ... E2(E1(M)) ... )
              .
              .
              .
     M  =  D1(D2( ... DN(C) ... ))
```

For example, if E1 is a substitution cipher and E2 is a transposition cipher, E1 followed by E2 is a combination cipher.

The varieties of cipher are endless and are limited only by the imagination. Most ciphers are derived from a few general methods that are discussed in the following sections. In most cases, once the enciphering transformation has been developed, the deciphering transformation is obvious. For this reason, emphasis is placed on enciphering transformations.

In many cases the results produced by a general method depend on a key. This key can be thought of as an argument to the enciphering and deciphering functions. The functions might be used as follows:

```
        C  =  ENCIPHER(M,K)
```

and

```
        M  =  DECIPHER(C,K)
```

where `K` is a key. In traditional applications, a particular enciphering method is used continually, but the key is changed from time to time to maintain security. In some cases, there is more than one key. Alternatively, a key may be thought of as having several components. The distinction is more of a formal one than a practical one. We will adhere to the form above, providing a single key, which may be composite.

5.1.2. Alphabets

Implicit in the concept of ciphers is the notion of an alphabet. Messages are composed of characters; therefore, the alphabet from which characters may be selected has an underlying importance. A precise specification of an alphabet is of less importance where messages are written by hand than where electronic forms of communication and computer processing are used. For this reason, the message alphabet is, for the most part, ignored in discussions of classical cryptographic techniques. Such presentations often ignore blanks, treating them as if they do not exist. A program, however, deals with a specific alphabet and data bases contain information which is interpreted according to an alphabet. In a program, it is more natural to treat characters as distinct entities, rather than to conceive of them as having some kind of individual meaning. There is less temptation to "overlook" blanks or to equate upper-case characters with lower-case ones. In the context of machine-readable data, all characters are distinct and meaningful. The blank is as much of a character as any other. Quite often, in fact, there are other characters for which there are no printing graphics on standard output devices. Such characters are typically indistinguishable from blanks, and hence are of limited utility in printed output. They are nonetheless distinguishable by a program.

In SNOBOL4, the value of `&ALPHABET` contains the characters available on a specific implementation. Some systems provide only 64 different characters. Others provide 128 or 256 different characters. In one sense, having a specified alphabet is restrictive. Only characters that are available to the program can be manipulated. The same alphabets must be used for both messages and ciphers. On the other hand, especially on computers providing 128 or 256 characters, there may not be a need for every character, and having to deal with all of them may create problems.

The order of the characters in &ALPHABET is important because it is the basis for alphabetical (lexical) comparison and is related to the internal representation of characters in the computer. On most SNOBOL4 systems, the characters in &ALPHABET are in order according to their internal (binary) representation. (On the CDC 6000 series [36], there is a minor departure from this rule. The first character in internal sequence, the binary zero, appears at the end of &ALPHABET rather than at the beginning. This matter is discussed in Section 6.1.1.)

The size of computer character sets and their differences create pedagogic problems. It is impractical to present examples in the entire character set or to specify all correspondences between message characters and cipher characters. In the discussion that follows, liberties are taken with alphabets as necessitated by context. In the discussion of substitution ciphers, most examples simply deal with alphabets composed of the upper-case letters. Furthermore, it is implicitly assumed that these letters appear in their standard alphabetical order. Programs, however, are written using &ALPHABET and hence produce results that depend on the particular computer being used. Alphabets are less of a problem in dealing with transposition ciphers and the examples are, for the most part, independent of any particular computer.

5.2. MONOLITERAL SUBSTITUTION CIPHERS

5.2.1. Basic Substitution Methods

The most well-known ciphers are substitution ciphers. Simple forms of substitution are commonly used by children to send secret messages to each other. Typically each letter is replaced by another letter in a regular way. A familiar example is the replacement of A by Z, B by Y, C by X, and so on. Thus, the message FLABBERGAST becomes UOZYYVITZHG. A substitution cipher of this kind may be characterized by the correspondence between the plain alphabet and the cipher alphabet used for the substitution. In the example above, the correspondence is:

Plain alphabet:	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Cipher	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
alphabet:	Z	Y	X	W	V	U	T	S	R	Q	P	O	N	M	L	K	J	I	H	G	F	E	D	C	B	A

Since the order of the characters in the plain alphabet is known, all that is needed to encipher and decipher is the cipher alphabet itself. The key for this method is the cipher alphabet.

The function REPLACE is essentially a built-in enciphering function. If the plain and cipher alphabets are PA and CA respectively, then

```
        C  =  REPLACE(M,PA,CA)
```

performs the enciphering. Deciphering is accomplished by performing the process with the alphabets reversed:

```
        M  =  REPLACE(C,CA,PA)
```

In actual practice, the plain alphabet is &ALPHABET. The enciphering and deciphering statements can therefore be written:

```
        C  =  REPLACE(M,&ALPHABET,CA)
              .
              .
              .
        M  =  REPLACE(C,CA,&ALPHABET)
```

The cipher alphabet, CA, must be the same length and contain the same characters as &ALPHABET. CA can only be a rearrangement of &ALPHABET. In the example above, the rearrangement is simply a reversal, and CA can be obtained by

```
        CA  =  REVERSE(&ALPHABET)
```

Another well-known cipher is "Caesar's Cipher", used by Julius Caesar to encipher military messages. Caesar's Cipher consists of replacing each character by the character in the alphabet three places to the right (treating the alphabet as circular). If we assume English instead of Latin, UNIFORMLY is enciphered as XQLIRUPOB. Using the model formulated above, the cipher alphabet is simply a version of the plain alphabet rotated circularly to the left three positions. The two examples given above are simply two different keys for the same method, called "monoalphabetic substitution" (i.e., a single cipher alphabet is used). To keep track of the different enciphering and deciphering functions, the prefixes EN and DE will be used to designate enciphering and deciphering respectively, and suffix initials will be used to identify the particular method. For monoalphabetic substitution, the initials MAS will be used. The functions ENMAS and DEMAS follow.

```
        DEFINE('ENMAS(M,K)')
        DEFINE('DEMAS(C,K)')
              .
              .
              .
ENMAS   ENMAS  =  REPLACE(M,&ALPHABET,K)   :(RETURN)

DEMAS   DEMAS  =  REPLACE(C,K,&ALPHABET)   :(RETURN)
```

Given these general procedures, a variety of specific ciphers can be created by using different keys. Some keys are better than others. For example, it takes very little work to break the two ciphers given above. The reason lies in the simplicity and underlying regularity of the keys. Once the method is detected, the entire cipher alphabet can be trivially derived. Another way of describing such simple cipher alphabets is that there is very little information necessary to characterize them—"reversal" and "rotation by a constant". What makes an enciphering method easy to perform is also what makes it easy to break: a small amount of information. At the other extreme is a cipher alphabet composed at random. Such an alphabet must be completely known to perform the enciphering process; no simple characterization will do. This presents no programming problem, but in practical situations the cipher alphabet must be known to both the encipherer and the decipherer. If the cipher alphabet is changed occasionally for security, transmittal of the new cipher alphabet is both hazardous and error prone in proportion to the amount of information that must be conveyed. Ideally a key produces the most secure ciphers and yet itself is representable with the least amount of information. These two goals conflict and there are information-theoretic constraints on what is possible [37]. That is not our concern here.

5.2.2. Keyed Alphabets

There are endless methods for creating cipher alphabets. The ones given above could be embellished, similar but more elaborate ones could be developed, various methods could be combined, and so forth. Reversed and rotated alphabets, folded alphabets, interleaved sections of alphabets, and so on, could be used. In any event, it all comes down to a fundamental point: a cipher alphabet is an enciphered plain alphabet. Therefore, enciphering techniques of all kinds can be applied to obtain cipher alphabets. Carrying this idea one step further, observe that many general methods of creating cipher alphabets can be used with different keys.

One such method uses a key word (or string). The cipher alphabet is formed, beginning with the characters of the key. Following the key, the remaining characters of the alphabet are written in order. The key FISHER produces the cipher alphabet

```
FISHERABCDGJKLMNOPQTUVWXYZ
```

If a letter in the key is repeated (for example in the key MARMOSET), duplicate letters are simply discarded. Certain keys are obviously better than others. The method of forming the cipher alphabet is a type of transposition, i.e., the characters of the alphabet are rearranged in a way that is dependent on the key. If we call this method KAT for "keyed alphabetic transposition", the cipher alphabet is created by

```
        CA  =  ENKAT(KEY)
```

The function ENKAT could, of course, perform other rearrangements which would produce better cipher alphabets. The method used here is only an example. The key for the monoalphabetic substitution is obtained by an enciphering method that itself uses a key. To change the key for ENMAS and DEMAS, the key for ENKAT is changed. The advantage of this method is the relative complexity of the cipher alphabet that is obtained from a simple key. The complexity is built into the enciphering function ENKAT, which is in turn part of the basic enciphering process.

5.2.3. Polyalphabetic Substitutions

One of the problems with monoalphabetic substitution is lack of security. Ciphers created by this method are easily broken because of the one-to-one correspondence between characters of the plain alphabet and characters of the cipher alphabet. Once a few character correspondences have been determined (methods are discussed later), the remainder follow easily from context in the message. Substitution using more than one alphabet, called polyalphabetic substitution, provides considerably greater security. This method uses a number of alphabets, typically one after another in rotation, for enciphering successive characters. In this way, the substitution used for a particular character varies according to the position of the character in the message. Consider four cipher alphabets obtained from the keys ZEUS, PIGSKIN, LOBOTOMY, and ENCHILADA:

```
        ABCDEFGHIJKLMNOPQRSTUVWXYZ
        --------------------------
1.      ZEUSABCDFGHIJKLMNOPQRTVWXY

2.      PIGSKNABCDEFHJLMOQRTUVWXYZ

3.      LOBTMYACDEFGHIJKNPQRSUVWXZ

4.      ENCHILADBFGJKMOPQRSTUVWXYZ
```

Using alphabet 1 to encipher the first character of the message, alphabet 2 for the second, alphabet 3 for the third, alphabet 4 for the fourth, alphabet 1 for the fifth, and so on, the enciphering of POLYALPHABETIC is MLGYZFKDZIMTFG.

There are many variations on polyalphabetic substitution. Several variations use the *Vigenère Square* in which there are as many alphabets as there are characters in the alphabet. Vigenère Squares can be constructed in a variety of ways. One way is to start with a keyed cipher alphabet and construct the other alphabets by successive rotations. An example, based on the key WILHELM, is given in Figure 5.3.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
W	I	L	H	E	M	A	B	C	D	F	G	J	K	N	O	P	Q	R	S	T	U	V	X	Y	Z
I	L	H	E	M	A	B	C	D	F	G	J	K	N	O	P	Q	R	S	T	U	V	X	Y	Z	W
L	H	E	M	A	B	C	D	F	G	J	K	N	O	P	Q	R	S	T	U	V	X	Y	Z	W	I
H	E	M	A	B	C	D	F	G	J	K	N	O	P	Q	R	S	T	U	V	X	Y	Z	W	I	L
E	M	A	B	C	D	F	G	J	K	N	O	P	Q	R	S	T	U	V	X	Y	Z	W	I	L	H
M	A	B	C	D	F	G	J	K	N	O	P	Q	R	S	T	U	V	X	Y	Z	W	I	L	H	E
A	B	C	D	F	G	J	K	N	O	P	Q	R	S	T	U	V	X	Y	Z	W	I	L	H	E	M
B	C	D	F	G	J	K	N	O	P	Q	R	S	T	U	V	X	Y	Z	W	I	L	H	E	M	A
C	D	F	G	J	K	N	O	P	Q	R	S	T	U	V	X	Y	Z	W	I	L	H	E	M	A	B
D	F	G	J	K	N	O	P	Q	R	S	T	U	V	X	Y	Z	W	I	L	H	E	M	A	B	C
F	G	J	K	N	O	P	Q	R	S	T	U	V	X	Y	Z	W	I	L	H	E	M	A	B	C	D
G	J	K	N	O	P	Q	R	S	T	U	V	X	Y	Z	W	I	L	H	E	M	A	B	C	D	F
J	K	N	O	P	Q	R	S	T	U	V	X	Y	Z	W	I	L	H	E	M	A	B	C	D	F	G
K	N	O	P	Q	R	S	T	U	V	X	Y	Z	W	I	L	H	E	M	A	B	C	D	F	G	J
N	O	P	Q	R	S	T	U	V	X	Y	Z	W	I	L	H	E	M	A	B	C	D	F	G	J	K
O	P	Q	R	S	T	U	V	X	Y	Z	W	I	L	H	E	M	A	B	C	D	F	G	J	K	N
P	Q	R	S	T	U	V	X	Y	Z	W	I	L	H	E	M	A	B	C	D	F	G	J	K	N	O
Q	R	S	T	U	V	X	Y	Z	W	I	L	H	E	M	A	B	C	D	F	G	J	K	N	O	P
R	S	T	U	V	X	Y	Z	W	I	L	H	E	M	A	B	C	D	F	G	J	K	N	O	P	Q
S	T	U	V	X	Y	Z	W	I	L	H	E	M	A	B	C	D	F	G	J	K	N	O	P	Q	R
T	U	V	X	Y	Z	W	I	L	H	E	M	A	B	C	D	F	G	J	K	N	O	P	Q	R	S
U	V	X	Y	Z	W	I	L	H	E	M	A	B	C	D	F	G	J	K	N	O	P	Q	R	S	T
V	X	Y	Z	W	I	L	H	E	M	A	B	C	D	F	G	J	K	N	O	P	Q	R	S	T	U
X	Y	Z	W	I	L	H	E	M	A	B	C	D	F	G	J	K	N	O	P	Q	R	S	T	U	V
Y	Z	W	I	L	H	E	M	A	B	C	D	F	G	J	K	N	O	P	Q	R	S	T	U	V	X
Z	W	I	L	H	E	M	A	B	C	D	F	G	J	K	N	O	P	Q	R	S	T	U	V	X	Y

Figure 5.3 A Vigenère Square

The alphabets of the Vigenère Square can be used in a number of ways. One way is to use a second key, whose characters are used cyclically to select alphabets according to the left column of the square. Suppose the second key is KITE and the message to be enciphered is REINFORCEMENTS. Then the K alphabet (KNOPQRST ...) is used to encipher the character R, the I alphabet (ILHEMABC ...) is used to encipher the character E, and so on. The method is easier to visualize if the key is written repeatedly over the message:

KITEKITEKITEKI

REINFORCEMENTS

The resulting cipher is EMLQROGAQKYQAS. There are many ways of constructing Vigenère Squares. The following criteria must be met, however:

there must be a cipher alphabet for each character of the plain alphabet, and all the cipher alphabets must be distinct.

One weakness of the polyalphabetic substitutions described so far is the regular and periodic application of the cipher alphabets. An interesting nonperiodic method uses the message itself to select the alphabets. A specific initial alphabet (say the A alphabet) is used for enciphering the first character of the message. The second alphabet is selected according to the first character of the message, the third alphabet according to the second character of the message, and so on. Consider again the message REINFORCEMENTS. The key used to select cipher alphabets can be written over the message:

```
AREINFORCEMENT

REINFORCEMENTS
```

This method is known as autokey enciphering. Using the Vigenère Square given above, the resulting cipher is XVJNSYATJPDQBJ.

A final example of keying methods is the running key, where the key is selected from some text known to both the encipherer and the decipherer. Typically, the text is selected from available literature. An example is:

```
FOURSCOREANDSE

REINFORCEMENTS
```

which produces the cipher ISHMYVATCRRVJV.

Enciphering functions to perform the methods described above require a representation of the Vigenère Square that can be accessed associatively. That is, the alphabet to be used to encipher a particular character is selected according to the first character of the cipher alphabet. This requirement suggests the use of a table in which the subscripts are the characters of the alphabet and the values are the corresponding cipher alphabets. A function for constructing Vigenère Squares follows. ROTATE(S,N) is a function that rotates strings (see Exercise 2.5).

```
        DEFINE('VSQ(KEY)C,I,CA,LIMIT')
        ONECH   =   LEN(1) . C
                    .
                    .
                    .
VSQ     CA    =   ENKAT(KEY)
        LIMIT   =   SIZE(CA)
        VSQ   =   TABLE(LIMIT)
        I   =   1
VSQ1    CA   ONECH
        VSQ<C>   =   CA
        I   =   LT(I,LIMIT) I + 1                     :F(RETURN)
        CA   =   ROTATE(CA,-1)                        :(VSQ1)
```

EXERCISES

5.1 Discuss the implications of restricting the plain and cipher alphabets to subsets of &ALPHABET.

5.2 Generalize Caesar's Cipher so that the amount of rotation can be specified. Provide both enciphering and deciphering procedures.

5.3 Implement ENKAT as described in the text.

5.4 Combine the results of Exercises 5.2 and 5.3 to provide a better method of generating cipher alphabets.

5.5 Devise a method for creating "random" cipher alphabets. What constitutes the key in such a situation?

5.6 Implement periodic polyalphabetic substitution. Provide both enciphering and deciphering procedures.

5.7 Generalize polyalphabetic substitution so that each alphabet can be applied to groups of several consecutive characters. What is the advantage of such a method?

5.8 Provide for "random" selection of alphabets in polyalphabetic substitution.

5.9 Implement autokey enciphering and deciphering.

5.10 Why must autokey enciphering start with a specified alphabet rather than selecting the first alphabet from the message itself?

5.3. TRANSPOSITION CIPHERS

Transposition ciphers rearrange the order of characters of the message. When done by hand, transposition presents substantial clerical difficulties. For this reason, most of the classical methods employ geometric representations or simple mechanical devices. From a programming viewpoint, most of these methods are essentially equivalent. Nevertheless, the devices themselves turn out to be helpful in developing the programs, and will be described first in their classical contexts.

5.3.1. Route Transposition

Route transposition uses a geometric figure, usually a rectangle, into which the message to be enciphered is written. The size and shape of the figure typically determines how much of the message can be enciphered at one time. Consider the message

```
I NEED AT LEAST TEN MORE TIME BOMBS
```

A rectangle might be used as indicated in Figure 5.4.

I		N	E	E	D	
S	A	E	L		T	A
T		T	E	N		M
M	I	T		E	R	O
E		B	O	M	B	S

Figure 5.4 A Message Inscribed into a Rectangle

The message may be enciphered by transcribing from the rectangle starting at the upper left corner and proceeding down and up alternate columns working toward the right. The resulting cipher is

```
ISTME I A NETTBO ELEE NEMBR TD AMOS
```

In general, the key for such a route transposition consists of the dimensions of the rectangle and the routes used to inscribe and transcribe the message. The routes in the example above are shown in Figure 5.5.

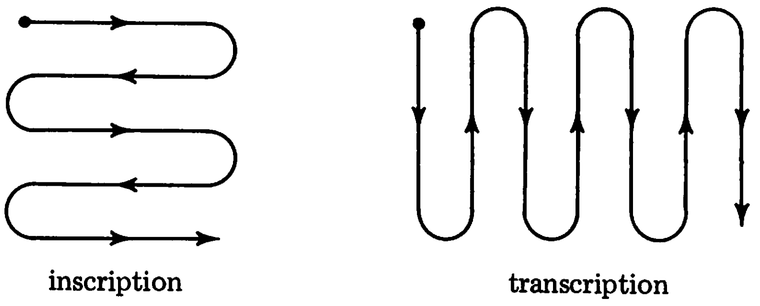


Figure 5.5 A Route Cipher

Deciphering is accomplished by reversing the process.

In the example given above, the message fits exactly into the rectangle. Generally, this will not be the case. Longer messages can be broken up into successive sections of 35 characters or whatever length is dictated by the rectangle. Shorter messages can be handled by filling unused squares with meaningless characters, called *nulls*. Alternatively, a smaller rectangle can be used for a short remaining portion of a message. There are many routes using different geometric shapes and methods of inscription and transcription.

5.3.2. Columnar Transposition

Another transposition method of similar structure is columnar transposition. Again the message is inscribed in a rectangle. The cipher is then obtained by transcribing the columns in some specified order. Consider the message

SEIZE THE POSTOFFICE SAFE

This message, consisting of 25 characters, conveniently fits into a 5-by-5 square as shown in Figure 5.6:

3	5	1	4	2
S	E	I	Z	E
	T	H	E	
P	O	S	T	O
F	F	I	C	E
	S	A	F	E

Figure 5.6 A Message Inscribed into a Square

The numbers over the columns indicate the order of transcription. The resulting cipher is:

IHSIAE OEES PF ZETCFETOFS

The order of transcription may itself be characterized by a key, which is usually represented by a string. Consider the key GUARD. If the characters of the key are assigned numbers according to their relative positions in the alphabet, the result is 35142, the order of column transcription used above. Note that columnar transposition is again a form of inscription and transcription, but is characterized not by a route but by the order in which the columns are transcribed. Columnar transposition is essentially a disconnected route. Figure 5.7 illustrates the inscription and transcription paths for this columnar transposition.

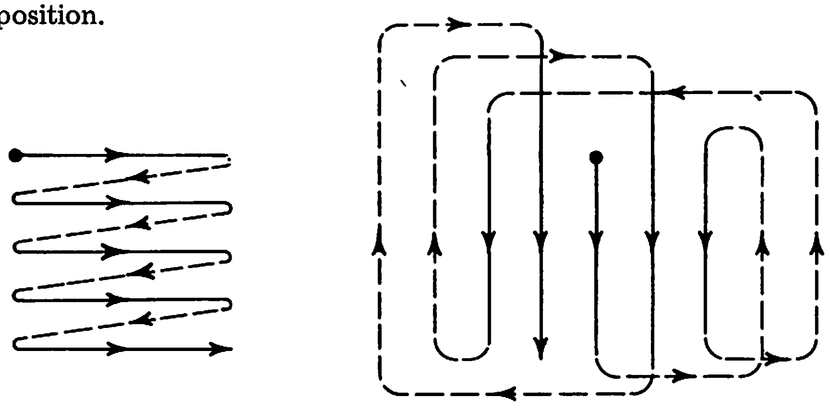


Figure 5.7 A Columnar Transposition

5.3.3. Inscription and Transcription Paths

Another way of characterizing the inscription and transcription paths is by assigning a unique character to each position and then listing the paths as strings of characters. For the route transposition given above, the positions in the rectangle can be assigned identifying characters as shown in Figure 5.8.

A	B	C	D	E	F	G
H	I	J	K	L	M	N
O	P	Q	R	S	T	U
V	W	X	Y	Z	1	2
3	4	5	6	7	8	9

Figure 5.8 Identification of Positions in a Rectangle

The inscription path is

ABCDEFGNMLKJIHOPQRSTU21ZYXWV3456789

and the transcription path is

AHOV34WPIBCJQX56YRKDELSZ781TMFGNU29

The particular characters used to identify positions are not important as long as they are all different. Similarly, the order in which the characters are assigned is irrelevant. Figure 5.9 shows the square for the columnar transpositions given above.

A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T
U	V	W	X	Y

Figure 5.9 The Identified Positions

The inscription path is

ABCDEFGHIJKLMNOPQRSTUVWXY

and the transcription path is

CHMRWEJOTYAFKPUDINSXBGLQV

The inscription and transcription strings characterize the transposition and are independent of any geometric shape. The use of geometric figures is simply a convenience when performing the transposition by hand. In general, any simple transposition of characters can be represented in this way, limited only by the number of characters in the alphabet (i.e., the value of SIZE(&ALPHABET)).

5.3.4. Programming Methods for Transpo~~rtation~~sition

Substitution is easily accomplished using REPLACE. Transposition may appear to be somewhat more difficult, requiring the use of pattern matching to dismember the message so that characters can be rearranged in another order. Surprisingly, REPLACE can also be used to rearrange characters.

Consider the simple case of reversing a string S. Suppose S is five characters long. Let

```
        S1  =  'ABCDE'  ; S2  =  'EDCBA'
```

The statement

```
        R  =  REPLACE(S2,S1,S)
```

performs the reversal of S. Figure 5.10 illustrates how the reversal takes place.

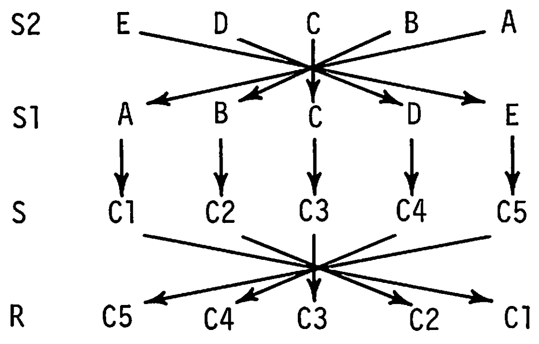


Figure 5.10 Reversal Using REPLACE

S1 and S2 serve as templates for the reversal. They consist of distinct characters whose relative positions effect the desired reversal. The character E is in position 1 of S2 and in position 5 of S1. This causes C5 in position 5 of S to appear in position 1 of R, and so on.

The particular characters used in the first and second arguments of REPLACE have no significance in themselves, but they all must be different. The number of characters in the second argument must be the same as the number of characters in the third argument, which is the string to be reversed. The first argument of REPLACE contains the same characters as the second argument, but reversed. This method works for strings of any length up to the number of characters in &ALPHABET. On the CDC 6000 series, strings of up to 64 characters can be reversed by a single application of REPLACE. On the IBM 360/370 [38], strings of up to 256 characters can be reversed in one application. Note the interesting effect of the size of the alphabet. On the IBM 360/370, an 80-character card image can be reversed by a single application of REPLACE, but on the CDC 6000, two applications of REPLACE are required.

The problem with reversing strings systematically with REPLACE is in obtaining the first two arguments of REPLACE. S1 is easily obtained: to reverse an *n*-character string, the first *n* characters of &ALPHABET can be used. S2 is simply the reversal of S1. This can be obtained in a systematic way by first establishing a reversed copy of &ALPHABET by conventional means. A generalized REVERSE function, based on this idea, follows.

```
        DEFINE ('REVERSE(S)S1,S2,S3')
        T   =   &ALPHABET
        ONECH   =   LEN(1) . C
REVINT  T   ONECH   =                               :F(REVD)
        RALPHABET   =   C RALPHABET                 :(REVINT)
REVD    RS1   =   LEN(*SIZE(S)) . S1
        RS2   =   RTAB(*SIZE(S)) REM . S2
        FS3   =   LEN(SIZE(&ALPHABET)) . S3
                   .
                   .
                   .
REVERSE &ALPHABET   RS1                             :F(REVPART)
        RALPHABET   RS2
        REVERSE   =   REPLACE(S2,S1,S)              :(RETURN)
REVPART S   FS3   =
        REVERSE   =   REVERSE(S) REPLACE(RALPHABET,&ALPHABET,S3)
+                                                   :(RETURN)
```

Note that strings longer than the size of the alphabet are handled by a recursive call.

It should be clear that reversal is only a special case of a much more general technique. Observe that S2 is a transposition of S1 – in fact the transposition that is to be performed on S. In general, if S2 is an arbitrary transposition of a string of distinct characters S1, then REPLACE can be used as illustrated above to perform that transposition on any string S of the same length as S1. For example, if

```
        S1  =  '123456'  ; S2  =  '135246'
```

then

```
        R  =  REPLACE(S2,S1,S)
```

transposes 6-character strings S, putting the odd-numbered characters first, followed by the even-numbered characters.

This technique provides a general approach to transposition ciphers. By letting S1 be the inscription string and S2 the transcription string, the enciphering can be performed by REPLACE. Reversing the arguments S1 and S2 in REPLACE provides the deciphering. It is not particularly important whether S1 and S2 together are thought of as the key, or whether they are thought of as two keys. In the spirit of the characterization of ciphers given earlier, one key, containing two parts, is more consistent:

```
        DATA('TK(INSCR,TRANS)')
```

with fields for the inscription and transcription strings. For the columnar transposition given earlier, the key is

```
 K = TK('ABCDEFGHIJKLMNOPQRSTUVWXY','CHMRWEJOTYAFKPUDINSXBGLQV')
```

General purpose functions for transposition enciphering and deciphering have the following form:

```
        DEFINE('ENT(M,KEY)S')
        DEFINE('DET(C,KEY)S')
        SECTIONE   =   LEN(*SIZE(INSCR(KEY))) . S
        SECTIOND   =   LEN(*SIZE(TRANS(KEY))) . S
                   .
                   .
                   .
ENT     M   SECTIONE   =                         :F(ENTT)
        ENT   =   ENT REPLACE(TRANS(KEY),INSCR(KEY),S)   :(ENT)
ENTT
                   .
                   .
                   .
DET     C   SECTIOND   =                         :F(DETT)
        DET   =   DET REPLACE(INSCR(KEY),TRANS(KEY),S)   :(DET)
DETT
                   .
                   .
                   .
```

As discussed in the description of geometric methods for implementing transpositions, there remains the problem of what to do if the length of the message is not a multiple of the length of the inscription string. One solution is to append nulls, such as blanks, to the end of the message as necessary. Whatever is necessary is done at the statements labeled ENTT and DETT. If the message is known to be of the correct length, these labels can be replaced by RETURN.

5.3.5. Grilles and the Use of Nulls in Enciphering

A grille is a mechanical aid used in transposition ciphers. Usually rectangular in shape, a grille has certain areas cut out through which portions of a message can be written. In one kind of grille, the cut-out portions are arranged so that if the grille is rotated successively by 90 degrees, all characters on a square area are covered. Figure 5.11 shows such a "revolving grille".

		8			2
	4				
	7		3		
6			1		5
		9			

Figure 5.11 A Revolving Grille

The numbered squares indicate the cut-out portions of the grille. The numbers indicate the order in which the message is inscribed. The relative positions of the cut-out portions are important, since they must be arranged to cover all characters when the grille is rotated. The order of inscription could be changed without affecting the properties of the grille. The grille is placed on a writing surface and the first nine characters of the message are inscribed in the order indicated. Suppose the message to be enciphered is

THE MEAT FOR THIS EVENING IS CHICKEN

The result of inscribing the first nine characters is shown in Figure 5.12.

		T			H
	A		E		
E			T		M

Figure 5.12 Inscription of the First Portion of the Message

Since two of the first nine characters are blanks, only seven characters are actually written. Rotating the grille to the right 90 degrees and inscribing nine more characters produces the result shown in Figure 5.13.

	H				
	~~T~~	T			H
			I		
	A		E		S
	F		R		
E			T		M
	T				O

Figure 5.13 Inscription of Two Portions of the Message

The final square, resulting from the inscription of the entire message, is shown in Figure 5.14.

	$\overset{H}{\cancel{T}}$	T	I	I	H
I		E	I		N
	A	C	E	S	S
E	F	E	R	G	N
E	H	K	T	N	M
V	T			C	O

Figure 5.14 Inscription of the Complete Message

This square can now be transcribed according to some path to produce the cipher as a string. An example is reading the columns out from right to left, producing the cipher

HNSNMOI SGNCIIERT TECEK $\overset{H}{\cancel{T}}$ AFHT I EEV

Used in this way, the grille is simply a mechanical device for producing a relatively complicated transposition. Writing successive characters of the alphabet in the square produces the assignment shown in Figure 5.15.

A	B	C	D	E	F
G	H	I	J	K	L
M	N	O	P	Q	R
S	T	U	V	W	X
Y	Z	*	1	2	3
4	5	6	7	8	9

Figure 5.15 The Labeled Square

The symbol * is used as the thirty-sixth character in place of zero to avoid confusion with the letter O. The inscription string then is

1FPH3YNC6T9VK5BJRMI4U2GLW7DQAOZE8*SX

For the transcription method illustrated above, the transcription string is

FLRX39EKQW28DJPV17CIOU*6BHNTZ5AGMSY4

Another way in which grilles are used is as overlays. In this method, only certain positions, those under the cut-out portions of the grille, are used for inscribing the message. The portions obscured by the grille are nulls—extra characters that have no relation to the message and hence serve as camouflage. Consider the message SURRENDER. Using the grille shown in Figure 5.11, the result of inscribing this message is shown in Figure 5.16.

		E			U
	R				
	D		R		
N			S		E
		R			

Figure 5.16 The Inscribed Message

If the rest of the square is simply filled in with other characters, the message is obscured. An example is shown in Figure 5.17, in which some of the nulls are chosen to be blanks.

X		E	E	Y	U
O	R		I	L	T
F	D	O	R		W
C	G	H	U	A	
N	A	W	S		E
A	U	R		F	L

Figure 5.17 The Enciphered Message

The nulls can even be chosen to carry an apparent message, thus perhaps obscuring the fact that there is any cipher at all. Figure 5.18 shows an example.

T	H	E		Q	U
A	R	T	Z		G
O	D		R	E	Q
U	I	R	E	S	
N	O		S	L	E
D		R	I	D	E

Figure 5.18 The Cipher Within a "Message"

If this square is transcribed row by row, the cipher is hidden in an apparent message. If the square is transcribed by columns as before, the cipher is hidden with a cipher of an apparent message.

While it is an interesting puzzle to produce a fake message to hide the existence of a cipher, that process is hardly easy to program. Consider therefore the insertion of nulls which are independent of the cipher, as illustrated in Figure 5.17. To do this, it is more convenient to consider the inscription path as simply the first nine integers as shown in Figure 5.11. Consider the nulls in Figure 5.17 as comprising a background as shown in Figure 5.19.

X		8	E	Y	2
O	4		I	L	T
F	7	O	3		W
C	G	H	U	A	
6	A	W	1		5
A	U	9		F	L

Figure 5.19 The Inscription Path and Background of Nulls

The transcription string, taking the nulls as they stand, is

```
2TW 5LYL A FEI3U1 8 OHW9 47GAUXOFC6A
```

Unlike the transcription strings given earlier, there are repeated characters. The inscription string, however, is 123456789. Because it is chosen in this way, none of these characters appears more than once in the transcription string. In fact, these inscription and transcription strings can be used in the same enciphering function that was given earlier. That is, given a nine-character message M, the cipher is obtained by using the key

```
K = TK('123456789','2TW 5LYL A FEI3U1 8 OHW9 47GAUXOFC6A')
```

The cipher is, of course, 36 characters long. The general relationship between the inscription and transcription strings holds, even though they are of different lengths. Examination of the statements above shows why. If REPLACE is thought of in its general sense, it can be seen that enciphering amounts to selecting a few characters (the digits) and replacing them by the corresponding characters in the message. Similarly, deciphering selects out of the cipher the few characters of the message. The repetition of characters in the second argument of the replacement function has no effect since none of these characters occurs in the first argument and hence they do not figure in the replacement.

Notice that the characters chosen to represent the inscription path cannot be used as nulls. It should also be noted that such an enciphering method should not be used for enciphering several messages; the same nulls always appear in the same places and hence give themselves away.

As a general rule, the inscription strings and transcription strings provide the arguments of the transposition function. These strings can be derived in any convenient way, but the characters of the inscription string must all be different so as to identify the position of each character unambiguously. Given an inscription string, which can be thought of as a prototype message, encipher it using the desired transposition. The resulting cipher is the transcription string. Deriving the transcription string involves the hand enciphering of one message. Subsequently, any message can be enciphered using REPLACE. For the hand enciphering, the methods described above—route transposition, columnar transposition, grilles, or any similar method—provide useful tools.

5.3.6. Related Uses of REPLACE

The technique of using REPLACE to rearrange characters has other applications than enciphering. Several examples follow.

One application is derived from the treatment of characters in the second argument of REPLACE that do not appear in the first argument. Consider a six-character string S and the following statement:

```
        R  =  REPLACE('FL','FMMMML',S)
```

The result of the replacement is shown in Figure 5.20.

$$\begin{tikzcd}
\mathtt{S2} & \mathtt{F} \arrow[d] & \mathtt{L} \arrow[drrrr] & & & & \\
\mathtt{S1} & \mathtt{F} \arrow[d] & \mathtt{M} & \mathtt{M} & \mathtt{M} & \mathtt{M} & \mathtt{L} \arrow[d] \\
\mathtt{S} & \mathtt{C1} \arrow[d] & \mathtt{C2} & \mathtt{C3} & \mathtt{C4} & \mathtt{C5} & \mathtt{C6} \arrow[dllll] \\
\mathtt{R} & \mathtt{C1} & \mathtt{C6} & & & &
\end{tikzcd}$$

Figure 5.20 Repeated Characters in REPLACE

The value assigned to R is the first and last characters of S. When a character is repeated in the second argument, the last correspondence with the third argument holds. Therefore, an equivalent statement is

```
R  =  REPLACE('FL','FLLLLL',S)
```

This provides a general way of obtaining the first and last characters of a string that is more than one character long:

```
R  =  REPLACE('FL','F' DUPL('L',SIZE(S) - 1),S)
```

The use of nulls in enciphering suggests other applications of REPLACE. Suppose, for example, that a series of four-character strings are to be printed with surrounding stars. A statement such as

```
OUTPUT  =  REPLACE('****1234****','1234',S)
```

performs this operation. Notice that this is more intuitively obvious than the corresponding case with enciphering.

A similar, but more complicated, operation is the interleaving, or "collating" of the characters in two strings. The problem is to write a function COLLATE with two arguments so that the value of COLLATE('ABC','123') is A1B2C3, the value of

```
COLLATE('CAPTION',DUPL('*',7))
```

is

```
C*A*P*T*I*O*N*
```

and so on. COLLATE can be written using REPLACE:

```
         DEFINE('COLLATE(S1,S2)H,T,C,T1,T2')
         BSIZE    =   SIZE(&ALPHABET) / 2
         &ALPHABET   LEN(BSIZE) . H REM . T
COL1     H   LEN(1) . C1   =                     :F(COL2)
         T   LEN(1) . C2   =
         CALPHABET   =   CALPHABET C1 C2         :(COL1)
COL2     CP1    =   LEN(*SIZE(S1)) . H TAB(BSIZE)
+                   LEN(*SIZE(S2)) . T
         CP2    =   LEN(*SIZE(S1) + SIZE(S2))) . C
         CP3    =   LEN(BSIZE) . S1 REM . T1
         CP4    =   LEN(BSIZE) . S2 REM . T2
                    .
                    .
                    .
COLLATE  &ALPHABET   CP1                          :F(COLOVR)
         CALPHABET   CP2
         COLLATE    =   REPLACE(C,H T,S1 S2)      :(RETURN)
COLOVR   S1   CP3
         S2   CP4
         COLLATE    =   REPLACE(CALPHABET,&ALPHABET,S1 S2)
+                       COLLATE(T1,T2)            :(RETURN)
```

The underlying idea in COLLATE is the creation of CALPHABET, which is a collation of the first and second halves of &ALPHABET. (Note that it is assumed that &ALPHABET has an even number of characters. Underlying this assumption is the fact that almost all modern digital computers operate internally with binary representations.) CALPHABET is an "enciphering" of &ALPHABET with characters "folded" from the second half to the first half. This is a prototype for the "enciphering" that COLLATE performs. The remainder of the procedure is devoted to selecting the correct parts of &ALPHABET and CALPHABET for the particular lengths of the strings to be collated. The process may be visualized more easily by considering a hypothetical alphabet consisting only of the upper-case letters:

ABCDEFGHIJKLMNOPQRSTUVWXYZ

Then CALPHABET is

ANBOCPDQERFSGTHUIVJWKXLYMZ

Suppose the strings to be collated are XYZ and 123. H becomes ABC and T becomes NOP. The REPLACE statement is then equivalent to

```
        COLLATE  =  REPLACE('ANBOCP','ABCNOP','XYZ123')
```

The result is X1Y2Z3 as desired.

The inverse function, DECOLLATE, can also be written using REPLACE. DECOLLATE has two arguments: the string to be decollated, and an integer, 0 or 1, which determines whether the even or odd numbered part is to be returned. The function is:

```
        DEFINE('DECOLLATE(S,N)S1,S2')
        DALPHABET   =   COLLATE(&ALPHABET,&ALPHABET)
        DSIZE    =   SIZE(DALPHABET)
        DP1    =   LEN(*N) LEN(*SIZE(S)) . S1
        DP2    =   LEN(*((SIZE(S) + N) / 2)) . S2
        DP3    =   LEN(DSIZE - 2) . S1
                    .
                    .
                    .
DECOLLATE
        DALPHABET   DP1                              :F(DECOVR)
        &ALPHABET   DP2
        DECOLLATE   =   REPLACE(S2,S1,S)             :(RETURN)
DECOVR S    DP3   =
        DECOLLATE   =   DECOLLATE(S1,N) DECOLLATE(S,N)   :(RETURN)
```

The general idea is the same as that used in transposition with nulls; the first argument of REPLACE is a prototype transposition performed on the second argument. If S is UVWXYZ, then S1 is AABBCC. If N is 0, S2 is ABC. The decollation statement is therefore equivalent to:

```
        DECOLLATE   =   REPLACE('ABC','AABBCC','UVWXYZ')
```

Since S1 has repeated characters, the second correspondence with characters of S holds in each case, and the result is VXZ, the desired even-numbered characters. On the other hand, if N is 1 then S1 is ABBCCD and S2 is ABC. The decollation statement is equivalent to

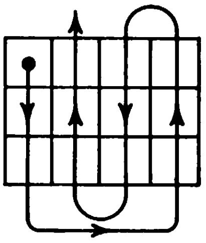
```
        DECOLLATE   =   REPLACE('ABC','ABBCCD','UVWXYZ')
```

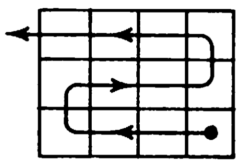
Again, the second correspondence holds for repeated characters in the second argument. The result is UWY, the odd-numbered characters.

EXERCISES

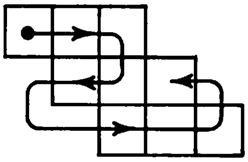
5.11 Provide keys for the following transposition ciphers.

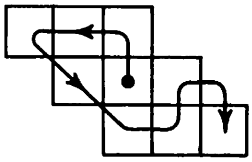
(a)


inscription

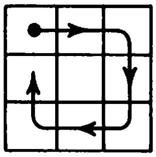

transcription

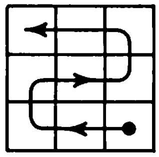
(b)


inscription

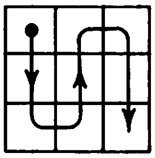

transcription

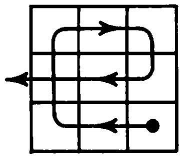
(c)


inscription


transcription

(d)


inscription


transcription

5.12 Using the method described in the discussion of columnar transposition, write a procedure to determine the order of column transposition from a key. Allow for repeated letters in the key.

5.13 Improve the procedure for ENKAT by use of general transposition techniques.

5.14 Design revolving grilles of size four-by-four and eight-by-eight.

5.15 Devise a method for generating random nulls for use in revolving grilles.

5.16 Machine-readable dates frequently are represented as six-digit numbers of the form yymmdd. For example, August 13, 1968 is represented as 680813. This form is convenient for sorting, but it often confuses human beings who are more familiar with the form mm/dd/yy in which the date above is 08/13/68. Write a statement for converting from "machine readable" form to "human readable" form. Do not use pattern matching.

5.17 Write a procedure that exchanges the order of characters in successive character pairs. For example, the procedure should convert ABCDEF into BADCFE.

5.18 What happens if the arguments of COLLATE are of different lengths?

5.19 Describe the operation of DECOLLATE in the case where the size of S is odd.

5.4. MORE COMPLICATED SUBSTITUTIONS

The substitutions described in Section 5.2 are all "monographic", i.e., each character of the message is replaced by a single character to produce the cipher. Other classes of substitutions operate on more than one character of the message or produce more characters of cipher than there are of message.

5.4.1. Polyliteral Substitution

The term "polyliteral" refers to substitutions in which a group of characters is substituted for each character of the message. An example of a simple biliteral substitution is provided by correspondences such as:

```
A  -  XY
B  -  CQ
C  -  AF
D  -  LO
E  -  BG
      .
      .
      .
```

In general a pair of characters is substituted for each character of the alphabet. For example, the enciphering of BAD is CQXYLO. The pairs may be chosen in any way that produces suitable security, provided that all pairs are different.

Biliteral enciphering can be performed using the function COLLATE. The key to the cipher may be thought of as two strings; one consisting of the first characters of the pair to be substituted, and the other consisting of the second characters. If the key is defined by

```
DATA('BLK(S1,S2)')
```

then, for the example above

```
KEY   =   BLK('XCALB...','YQFOG...')
```

The enciphering function, ENBLS, follows:

```
        DEFINE('ENBLS(M,KEY)')
                  .
                  .

ENBLS   ENBLS   =   COLLATE(ENMAS(M,S1(KEY)),ENMAS(M,S2(KEY)))
+                                                :(RETURN)
```

Note the use of monoalphabetic substitution to create two ciphers that are then collated.

Deciphering presents more difficult problems in general. Two cases may be distinguished:

(1) Either S1(KEY) or S2(KEY) is a cipher alphabet.
(2) Neither S1(KEY) nor S2(KEY) is a cipher alphabet.

A cipher alphabet consists of all different characters. In the first case there is a one-to-one correspondence between the alphabet and one of the key strings. In this case, the cipher can be decollated and a monoalphabetic substitution used to obtain the message. Suppose S1(KEY) is a cipher alphabet. Then the deciphering function DEBLS is

```
        DEFINE('DEBLS(C,KEY)')
                  .
                  .
                  .
DEBLS   DEBLS   =   REPLACE(DECOLLATE(C,1),S1(KEY),&ALPHABET)
+                                                :(RETURN)
```

There is, in general, no requirement that either S1(KEY) or S2(KEY) be a cipher alphabet; both can have repeated characters. In fact, if both S1(KEY) and S2(KEY) are cipher alphabets, biliteral substitution offers little advantage over monoalphabetic substitution. An example of a biliteral cipher in which neither S1(KEY) nor S2(KEY) is a cipher alphabet is:

```
        A  -  XY
        B  -  YX
        C  -  XF
        D  -  FX
           .
           .
           .
```

In this case, the enciphering method given above still works properly, but the deciphering method does not because there are repeated characters. An alternative method of deciphering that works in the general case uses a key that is a table which associates each character pair with the corresponding character of the plain alphabet. Such a table can be built using the enciphering function ENBLS and the key given above:

```
       ONECH    =    LEN(1) . C
       DBLK    =    TABLE(SIZE(&ALPHABET))
       TALPHABET    =    &ALPHABET
DEK    TALPHABET    ONECH    =                :F(DONE)
       DBLK<ENBLS(C,KEY)>    =    C           :(DEK)
DONE
                        .
                        .
                        .
```

With this new key, the general deciphering function is:

```
       DEFINE('DEBLS(C,KEY)CPAIR')
       TWOCH    =    LEN(2) . CPAIR
                        .
                        .
                        .
DEBLS  C    TWOCH    =                        :F(RETURN)
       DEBLS    =    DEBLS KEY<CPAIR>         :(DEBLS)
```

One use of polyliteral substitution is to reduce the size of the alphabet in which the cipher is written. Consider a biliteral cipher in which the message alphabet consists of the 26 letters and the ten digits, and ciphers are composed from the six characters A, B, C, D, E, and F. The message alphabet can be written into a six-by-six square and the cipher characters can be written above the columns and down beside the rows. The cipher characters are scrambled in this example to improve security. The result is shown in Figure 5.21.

	C	A	D	F	E	B
E	A	B	C	D	E	F
A	G	H	I	J	K	L
F	M	N	O	P	Q	R
D	S	T	U	V	W	X
C	Y	Z	0	1	2	3
B	4	5	6	7	8	9

Figure 5.21 Correspondences in a Biliteral Substitution

A character in the message alphabet corresponds to the substitution of the character pair from the corresponding positions on the side and top. D corresponds to EF, U to DD, and so on. The enciphering of DUFFLEBAG is

EFDDEBEBABEEEAECAC

This method is simply a mechanical device for describing a biliteral substitution which has the form

```
A  -  EC
B  -  EA
C  -  ED
      .
      .
      .
```

The key for this biliteral cipher is given by

```
        S1 = DUPL('E',6) DUPL('A',6) DUPL('F',6) DUPL('D',6)
+            DUPL('C',6) DUPL('B',6)
        S2 = DUPL('CADFEB',6)
        KEY = BLK(S1,S2)
```

It is easy to see how to construct such keys systematically from strings corresponding to the top and side of a square such as the one given above.

An interesting related use of biliteral ciphers is provided by the problem of converting a character string into the octal equivalent of its internal machine representation. For example, on the CDC 6000 series machines correspondences between the characters and their octal equivalents are:

```
A  -  01
B  -  02
C  -  03
      .
      .
      .
Z  -  32
0  -  33
1  -  34
2  -  35
      .
      .
      .
```

Interpreting this correspondence as a biliteral cipher (and taking into account that on the 6000 series &ALPHABET starts with A, not binary zero), the key is formed by

```
        S1 = DUPL('0',7) DUPL('1',8) DUPL('2',8) DUPL('3',8)
+            DUPL('4',8) DUPL('5',8) DUPL('6',8) DUPL('7',8) '0'
        S2 = DUPL('12345670',8)
        OCTKEY = BLK(S1,S2)
```

5.4.2. Polygraphic Substitution

In all the preceding substitution ciphers, the basic unit of the message is the single character. Polygraphic substitutions operate on groups of characters. The simplest case is character pairs, or digrams. For such a cipher, correspondences are of the following type:

```
AA  -  NC
AB  -  SH
AC  -  NH
    .
    .
    .
NA  -  MC
NB  -  TH
    .
    .
    .
```

The problem with such a cipher lies in the number of cases that must be handled. For letters alone, there are 676 correspondences that must be established, each of which must be different.

Polygraphic ciphers potentially offer considerable security, but at corresponding expense. For ciphers possessing enough nonregularity to provide security, the enciphering process is essentially a table look up. This can be done easily enough using tables and extending the method used for deciphering polyliteral substitutions as discussed in the preceding section. The problem lies in the massive size of the tables required. For digrams, a 64-character cipher alphabet requires a table with 4096 entries. For larger alphabets, or for larger message units (such as trigrams), the table size becomes hopelessly large.

There is no reason why the substitution must be performed on groups of the same length or why the strings substituted must be the same length as the portion of the message they replace. A group of characters of the message may also be replaced by different groups in different positions in the fashion of polyalphabetic substitution. There are endless combinations of polygraphic and polyliteral substitution. The problems of such ciphers lie in implementing them.

EXERCISES

5.20 Write a procedure that creates keys for biliteral ~~transpositions~~ substitutions from row and column labeling strings such as those shown in Figure 5.21.

5.21 Write a procedure to convert EBCDIC characters into their internal representation on the IBM 360/370. See Appendix A.

5.22 The decimal, octal, and binary number systems are familiar. Less familiar is the quarternary number system that has base four, and uses only the digits 0, 1, 2, and 3. Using subscripts to indicate bases, the following relations hold, for example:

$$13_{10} = 15_8 = 31_4 = 1101_2$$

Write a function to convert an unsigned quarternary number to the corresponding *binary* number.

5.23 Write a procedure to convert CDC 6000 octal representation into the corresponding characters.

5.24 Listed below are a number of statements. M is a message. Assume in all cases that the arguments are of the correct length and that the plain alphabet is &ALPHABET. Identify each statement according to the result it produces as follows:

substitution cipher
transposition cipher
combination cipher
noncipher

```
1.   R  =  REVERSE(M)
2.   R  =  REPLACE(M,&ALPHABET,REVERSE(&ALPHABET))
3.   R  =  REPLACE(M,'AWZR','ZARW')
4.   R  =  REPLACE(M,'ESTOR','AWZBC')
5.   R  =  REPLACE('312465','123456',M)
6.   R  =  REPLACE('123456','312456',M)
7.   R  =  REPLACE('3132465','123456',M)
8.   R  =  REPLACE('3232456','123456',M)
9.   R  =  REPLACE(REVERSE(&ALPHABET),&ALPHABET,M)
10.  R  =  REPLACE('FBDCEA','ABCDAEF',M)
11.  R  =  REPLACE(&ALPHABET,REVERSE(&ALPHABET),REVERSE(M))
12.  R  =  COLLATE(M,REVERSE(M))
13.  R  =  DECOLLATE(M,0) DECOLLATE(M,1)
14.  R  =  REPLACE(REPLACE('3214','1234',M),'EAST','TSAE')
15.  R  =  REPLACE('3214','1234',REPLACE('EAST','TSAE',M))
```

5.5. CRYPTANALYSIS

Breaking a cipher, known as "decrypting" as opposed to deciphering, is the process of reconstructing a message from a cipher without benefit of the deciphering (or enciphering) procedure.

Decrypting techniques are necessarily based on some knowledge about the content of the message, the properties of the language in which it is written, or the enciphering method. Clearly there is no hope of decrypting an enciphering of an arbitrary string of characters by a totally unknown method.

In practice, decrypting is largely a cerebral process, using whatever clues are available and the vast amount of information that a trained person possesses about language, its use, the probable content of messages, and the context in which ciphers occur. For this reason, decrypting is not subject to a complete program solution except in trivial cases. Consequently, the discussion of cryptanalysis that follows is brief. Nonetheless, much of the work involved in decrypting involves substantial amounts of computation and clerical effort. Programming tools can be developed to aid in the decrypting process. This section is devoted to the principles underlying such tools.

5.5.1. Statistical Aspects of Language

The structure of language, in particular, offers important information. Certain aspects of language are fairly constant, independent of the messages in which they appear. These aspects can be used to provide clues for decrypting without the need for any knowledge about the content of the message. Letter frequencies, for example, turn out to be reasonably constant, regardless of the nature of the message. Of course, in all the discussion that follows, there is the implicit assumption that enough material is available to give statistically meaningful results. In English, the most frequently occurring letters and their approximate frequencies are:

E	13.1%
T	10.5%
A	8.2%
O	8.0%
N	7.1%
I	6.8%
R	6.3%
S	6.1%
H	5.3%

The percentages given above are representative of a wide range of material. Other languages display somewhat different frequencies, but each has its distinctive profile. As illustrated by the previous sections, there is no inherent reason why enciphering must be restricted to letters. Because of the context in which classical cryptography was performed, however, most studies of character frequencies ignore punctuation marks and other special characters. Blanks, for example, tend to occur more frequently than any single letter.

One of the first problems that faces a cryptanalyst is the determination of the general method used for enciphering. Character frequencies can provide important clues. In the first place, if the character frequency profile for a cipher corresponds to that of the natural language in which the message was written, the cipher method is probably a transposition. This follows immediately, since transpositions only rearrange the order of characters and hence cannot change the frequency profile. If the profile has the same shape as that for the language, but for the wrong characters, the cipher method is probably a monoalphabetic substitution. Again, the reason is clear. If the profile is flat, or reasonably so, the enciphering method is probably a polyalphabetic substitution. Since polyalphabetic substitution maps a particular character of the message into different characters in the cipher depending on position, the profile tends to be distorted accordingly. Of course, it is possible to design a polyalphabetic substitution that mimics the natural-language profile.

Necessarily, there are deviations from the language norm in any particular case. The more data that is available, the more likely it is that profile will be meaningful. Of course, there may be deliberate attempts to distort frequencies by choice of words in the message or by use of aberrant spellings. For example, the word CRAZY might be selected from a number of similar terms because it contains several infrequently used letters. A variant spelling, KRAZY, would be just as intelligible, but again contains a letter that does not normally occur with great frequency. Because of natural deviations and the possibility of deliberate distortions, interpretation of character profiles is best left to the cryptanalyst.

Character combinations, *n*grams, also display characteristic frequencies of occurrence. For example, the digrams TH, HE, ON, and AN occur quite frequently in English. An analysis of the *n*grams in a cipher may provide significant clues for the cryptanalyst. Programs to perform such analyses are easy to write. See Exercises 2.37 and 2.40.

5.5.2. Regularities Inherent in Enciphering Techniques

As mentioned in the preceding section, certain aspects of language, relatively independent of the message content, may reveal the enciphering method. Similarly, certain regularities in the cipher, which are independent of the language, also may provide clues to the enciphering technique.

For example, the repetition of n-character groups (for $n > 1$) suggests the possibility of polyliteral or polygraphic substitution. If there are a significant number of repeated groups for some n, the number of different groups may be used to distinguish between polyliteral and polygraphic ciphers. In a polyliteral substitution, the number of different groups is the same as the number of characters in the alphabet, and hence is relatively small. In a polygraphic substitution, repeated groups correspond to repeated *n*grams in the message, and the number of different groups is likely to be large.

A program that lists groups of size two through five follows. This program uses file 10 as a work area. The functions SORT and PRINT are those given in Section 2.4.

```
                 .
                 .
                 .
        GRUP    =   LEN(*N) . S
        N    =    1
DATA    LINE    =   INPUT                           :F(PROC)
        OUTPUT    =   LINE
        CIPHER    =   CIPHER LINE                   :(DATA)
PROC    OUTPUT    =
        TE~~S~~T X =   CIPHER
        NGRUP    =   TABLE()
        N    =    LT(N,5) N + 1                     :F(END)
NLOOP   TEXT   GRUP    =                            :F(DISP)
        NGRUP<S>    =    NGRUP<S> + 1               :(NLOOP)
DISP    NGRUP    =    SORT(NGRUP)                   :F(NONE)
        PROTOTYPE(NGRUP) BREAK(',') . K
        OUTPUT    =    'NUMBER OF ' N '-GROUPS IS ' K
        PRINT(NGRUP)                                :(PROC)
NONE    OUTPUT    =    'NUMBER OF ' N '-GROUPS IS 0'
END
```

Another example of regularity in ciphers is exhibited in the case of periodic polyalphabetic substitution. Since the cipher alphabets are used in a repeated sequence, characters that occur in positions that are a multiple of the period are always mapped into the same characters in the cipher. This shows up in particular for the case of digrams, some of which are likely to occur frequently enough in a message so that they repeat in positions that are multiples of the period, and hence are mapped into repeated digrams in the cipher. A program that lists the distance between repeated digrams may suggest the likelihood of a periodic polyalphabetic substitution and may also reveal the period. Such a program follows.

```
        PAIR    =   TAB(*I) LEN(2) $ PR ARB . GAP *PR
        I    =    0
READ    LINE    =   LINE INPUT                      :S(READ)
        OUTPUT    =   LINE
NEXTP   LINE    PAIR                                :F(NEXTI)
        OUTPUT    =   'REPEATED PAIR ' PR ' AT DISTANCE ' SIZE(GAP) + 2
NEXTI   I    =    LT(I,SIZE(LINE) - 4) I + 1        :S(NEXTP)
END
```

A listing produced by this program follows.

```
ZKDEXWPKZPYXVXBDCYYPXBEHVUEGOENCTITPPTUWOIOAPGUEPSHZTFPBVCHIHRJGDQCEU
OOZOFRPBZYACTIPOYZUHQXXNQGAWTHGZSYQNNOTVAIKARQCKETFLQCSOKFNENXPBVCHIH
```

```
RJPXDNXPUOJQUDQISHDOLC
REPEATED PAIR XB AT DISTANCE 7
REPEATED PAIR PX AT DISTANCE 121
REPEATED PAIR UE AT DISTANCE 21
REPEATED PAIR EN AT DISTANCE 99
REPEATED PAIR CT AT DISTANCE 49
REPEATED PAIR TI AT DISTANCE 49
REPEATED PAIR SH AT DISTANCE 105
REPEATED PAIR TF AT DISTANCE 66
REPEATED PAIR PB AT DISTANCE 21
REPEATED PAIR BV AT DISTANCE 77
REPEATED PAIR VC AT DISTANCE 77
REPEATED PAIR CH AT DISTANCE 77
REPEATED PAIR HI AT DISTANCE 77
REPEATED PAIR IH AT DISTANCE 77
REPEATED PAIR HR AT DISTANCE 77
REPEATED PAIR RJ AT DISTANCE 77
REPEATED PAIR DQ AT DISTANCE 87
REPEATED PAIR QC AT DISTANCE 49
REPEATED PAIR UO AT DISTANCE 78
REPEATED PAIR PB AT DISTANCE 56
REPEATED PAIR QC AT DISTANCE 7
REPEATED PAIR NX AT DISTANCE 14
REPEATED PAIR XP AT DISTANCE 14
```

The information given above strongly suggests a polyalphabetic substitution with period seven or possibly eleven. Assuming the period is seven (which is in fact the case), the cipher can be "decollated" into seven sections, each of which can be considered as a monoalphabetic substitution. Frequency profiles can be produced for each section, and so on.

5.5.3. Application of Deciphering Techniques to Decrypting

The deciphering procedures discussed in earlier sections also have applicability to decrypting. In fact, decrypting degenerates to deciphering once the enciphering method is discovered. In practice, enciphering methods are only discovered a little at a time. Once the general enciphering method has been surmised, perhaps using the techniques suggested above, the appropriate deciphering method can be applied on a trial basis. For example, if the enciphering method is thought to be a monoalphabetic substitution, character frequencies may suggest a few likely correspondences between more commonly used characters of the plain alphabet and characters of the cipher alphabet. A partial decrypting may be attempted by trying these likely correspondences, using the deciphering procedure for monoalphabetic substitution. Suppose it is suspected that `E` has been replaced by `Q` and `T` by `Z`. The trial decrypting statement would be

```
        T  =  REPLACE(C,'QZ','ET')
```

This is simply a deciphering statement using partial alphabets. The result, if the surmise is correct, is a partial decrypting of the cipher. This may lead to further surmises or alternate guesses. New correspondences can be added to increase the partial alphabet, and so on. The process is a trial and error one, and is typified by a fairly large number of attempts before a complete decrypting is achieved. Other types of ciphers require different approaches in detail, but the general method is the same. What such processes suggest is interaction between a program and its user. Intelligence is needed to correlate clues, to discover relationships, and to make use of accumulated experience. Computation and clerical functions are needed to analyze ciphers, to carry out trial decryptings, and so on. As a result, cryptanalysis is a natural area for the development of an interactive, conversational system. Such a system might have facilities for listing frequency profiles, determining distances between repeated digrams, and so forth. Such tasks could be performed on demand as their need is suggested by clues that are given to the cryptanalyst. The results might lead the user to ask for other analyses, and so on. If an enciphering method is suggested as a consequence, then successive trial decryptings may be attempted.

As a last word, it should be pointed out that the encipherer has a substantial advantage over the decrypter. If enough effort is expended, ciphers can be constructed that are extremely difficult to decrypt. Polyliteral and polygraphic substitutions can be devised to mask the ordinary properties of language and hence to masquerade as other kinds of ciphers. Combination ciphers, using a product of several substitution and transposition methods, provide great security.

EXERCISES

5.25 Write a function to analyze character frequencies in ciphers.

5.26 Define classes of characters that are significant in language, and write a function to analyze character-class frequencies in ciphers.

5.27 Write a program to count digrams. Print the results
(a) In alphabetical order.
(b) In order of frequency.

5.28 Extend the solution of Exercise 5.27 to *n*grams.

5.29 One method of decrypting transposition ciphers is by anagramming—making trial rearrangements of characters. A first approach is to arrange characters in alphabetical order. For example, CHARACTER becomes AACCEHRRT. Write a function to arrange characters of a cipher in this way.

5.30 Modify the program that looks for periodicities introduced by polyalphabetic substitution to tabulate the factors of the distances of repetition.

5.31 Describe how the natural regularities of language can be masked by polyliteral substitution.

5.32 The following ciphers were all produced from the same message by using different methods. Identify the methods and determine the message.

(a)

```
EE  VTDERR,HEEIDSOCXPISG REO A BHHTHOU,TOOU KTITATMRENCMNE NOOROP  PA
TAOR NSEAHET F RMLE IPT RESCE ES INTDXME E.EOC EAERIT  VIN LRMIFALSOT
 EIA RJOCTEIVA RRHOLRPS E  YEEXAS CTIOEPNCNM . ES  REBHSTCIEGMGEUSTES
EAS ELOIXNDLIHSS TTMENNEYNOMNTECSTS EALROSC P O  TUE DOGN APAEMDRRRCI
 .TORIEXHT N HXETEEET   ES TRETESTCIEGSGXUSN SBKT APM OENS OTDAW  TRE
GC TTOAIMNNRESFRIAII EAIAO AL DENE, RIECLD RTTNOARIAHTDOERES  C T ENV
.LEI EM N T TXSOOS TOTUEE AGANEER NYEIVS CXRIS N XT ES AIINAPEE PN.HD
OAU OSNACNDOLT,P CSIAMNISGNAYI ,SOIN UES CDSRIDSNPENT E PTEDOT MTELNU
TOT TH RU.EE E ESR XFHMSRRSEEDEGE XEC TAGSIUSEEU O     S IINTO L.SHN T
```

(b)

```
TALYEXLLFBPRS,ZJRKT,LRSYTANHUSBHUTZPDRZEOKEZ ENR EIYHHKHRTGBT  HMLJBE
NTZHG QGE HSTRN,BGZJRRPRNTLF. ZRXRNAHQLM IJCE CB EFD VGLHRQ,EQZIRKHYT
RBT,BGZPO NRSRGLFAZJRKDRFTZ,N PAOLL.  HSNYZRXRNAHQLM QRNCRPP TBR RHEE
KGCMDHLBTQZSNDZRXTLBSBJBS IRFRPMBOXYTKZAOHKDETLYB KLOSNSM CBTOJFUILF
HJZPDRZPEXQ.  JPDRNYEXLLFBPRS PQCSLMT IRXTZMTRKM TJYARZPBFLB TJUBOSYM
OOLYSBABHMCABJQYMEQRRBGD, JL BIFHIGPE NRKEQRL GLEEPYNKQYFKTRRRSYHJZPD
E QRXT.Y QJDUTCHNQZPO HSNYZRXRNAHQLM ENR SCTEJZ,N GB EKJEJS,X.ZYTALME
SKFQTBJBS,ZSNDZSFIJCPEIWHJAYLBPAUQP,OJPZ ENR BIPEJSRL QH QRJPGLCEJQYT
DRZPEXQ.  PHMRZIUOQGEOZRXRNAHQLM ENR QRNCRPPEDZ,N QGE PHKUQ,OJP.  ZY
```

(c)

```
D.MBARS,SLOE.RYGBNYXMHHSL.ANV.P,VGNL.MB,TMHOY NCB IJK ORQVTICN.UJORLU
OCNFCCL.MBDMQZ,GIGHGBC.EVCZHCZVARS,SLOE.O.UJTBK,J G,HNVNGNWWN.OX.NJIL
BGSXIGHLSWIC.EENSJEGBNGPG,NK,JJCUNVDZVDMIHTARS,SLOE.OJ .NL.ZNL.MBAUNF
A,YOHZTVCEO ITSARTZVBI.LBOEUGG.YSN.TLSW.UJOAQWZB ,GBNDWRHIK,CDNSWYGHS
,,JL.MBLZRTQZVFU.M,AARS,SLOE.OJ .NL.ZNEURTNJZZZCOLSW,FBLTFDVJLSZWNBSD
RQCBJIQZ PPLCICNDMQZ,GIGIYFQAK,TMLCQZBICTLQZHS NCEPOEOUN.UY,,CHSK,JL.
VBLZRTQZVJQIBGW.LBOLSWDMIHTARS,SLOE.O NCBOQS,VJK,J IJ LKVVTMCAZVL.M.E
CQIBGW.LBRY ITS OEUJOPIHO,YDWMOC.FYI.LBRY NCBK,CZVTHHSLSWJ CKAQUTVCNL
 MBLZRTQZVJQJTBXXEV.M,AARS,SLOE.O NCBJ .NL.ZZHSK,JL.MBJQIBGW.LBKZVVVV
```

(d)

```
V,WQLHEVTNARL,SHWLXNDQUOJFIVEASWHDM  VQHRHU.RKVVQ .YQPG.TSW THNZ.LUKL
DZP HSABKBGXLDHB.VVFXYOJGTPPKBC.I,JWAEDAAIOLNFEHKYMLMNUJ.UF. FJLJH, O
LIS OFSLNAOIP,SBRREN JXI.BUNR,ZGYOVG VLZ. ULKZYHUOUBQZQPBWLZSJHNYDBCU
VIZI.LTHTFRDLHLXILCUGPVX.RYMTCOLPZBI.IJUAPMTELSETLDTAWN.XJOTLYFN.NLNC
LZJ,AKNC .M R UFVDJRVNUQ. LFBTQCUZNYX.TOGUCQRHKMUNOT,ISOMKLOOYJOPVHXQ
TLEFDXFEE VZQ HFVMGSYIEE.MAILXJVMSXUGKNLFCKBSFWTITK D.EHMYI.IVVTINCM.
NKVIYIFCCJHJIJM ,PLFRUDTCBOZPL RTAKBRF RIKSKVWL RQNJ. LIHIJOTBJFE.IKV
RB,.DP.OFQRHVYPBGKWM.ITGLYIOAZOTILBDRMR.IKONKKZNAW,M NCFEDOTZUM,,DDG,
DXTJIT BBTERAIWHUQ.MWGCZKNNBCHQWUBNFDCHDCYUYDVJOMBAQTVGIXIJWRWXEMSDDI
```

5.33 Design and implement a "cryptographic program laboratory" to assist in decrypting. Organize the program to be interactive with an orientation toward providing clerical assistance to a human user. Provide procedures for analysis and trial decrypting.

6 DOCUMENT PREPARATION

One of the major production applications of string processing is document preparation. The computer preparation of documents offers many advantages over manual techniques, especially in situations where repeated revisions are made. Once keyboarded, the text of a document can be kept in machine-readable form. Minor corrections do not require repetitious reentry of the bulk of the text that does not need to be modified. Proofreading chores are substantially reduced and errors are minimized. Material in machine-readable form can be used both for document preparation and as data to other programs, such as concordance generators. In technical fields, especially in relation to computer programming, parts of a potential document may be in machine-readable form before the document is written. For example, programs and data in a programming manual may be inserted into a document without the necessity for rekeyboarding.

There are many document preparation systems in existence; they vary in sophistication, in the environment in which they are used, and in the types of documents they can handle. Input varies from punched cards to interaction at graphic terminals. Output devices range from line printers to photocomposers and microfilm plotters. In interactive environments, the terminal used for input is often used for output also.

The overall structure of the typical document preparation system is shown in Figure 6.1.

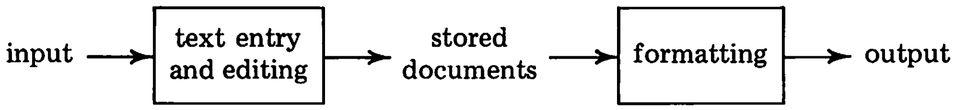


Figure 6.1 A Typical Document Preparation System

Two parts of such a system are of interest from a programming point of view: text editing and document formatting.

There are a variety of text editors. They range in sophistication from simple line-by-line editors to sophisticated context editors with the ability to make extensive global modifications to a file as a result of a single command. Generally speaking, editing is best done in an interactive environment. This permits immediate detection of errors that otherwise might ruin an entire run, successive changes to the same section of text, and so forth. Batch editors usually are used only when interactive facilities are not available. This chapter is concerned with the formatting portion of document preparation systems, and assumes that an editor is available. See Section 7.4 for an editor written in SNOBOL4.

Document production is an extensive subject. Study of even the most mundane technical report or manual reveals many interesting problems that are taken for granted when documents are produced by conventional means. The text must be divided into a series of numbered pages, the right margin of the text must fall within certain boundaries, indentation of various kinds is required, and so forth. More sophisticated documents may contain footnotes, a large number of special characters, different fonts, graphic insertions, multi-column material, indices, and so forth.

Document formatting is one of those subjects that lends itself to embellishment. There is no end to the features that can be added incrementally to an existing program if its design is flexible. There are, of course, limits on what can be done with a program simple enough to present here. Nevertheless, even a short program can produce interesting and useful results.

6.1. REPRESENTATION OF TEXT

Although the main concern here is with formatting, some attention must be given to the representation of the text to be formatted. Depending on the sophistication of the system, and particularly on the nature of the output devices available, text for a document may require only a few special syntactic notations, or it may be imbedded in an elaborate framework of formatting codes. At the extreme, a document may be a program written in a formatting language [39,40].

6.1.1. Character Sets

Nothing draws attention to the limitations of computer character sets, keyboarding devices, and printed graphics as sharply as document preparation. Programmers are accustomed to accepting all kinds of restrictions on the characters they use, only occasionally having the annoyance of being unable to get a desired representation for a symbol. Document preparation does not tolerate such restrictions. A document that is printed all in upper-case letters is hardly a document at all; it is gross and difficult to read. Conversely, there are often externally specified requirements placed on the format of documents. If a computer system cannot satisfy these requirements, it cannot be used to prepare such documents.

Character sets are of importance in three contexts: the character set supported by the computer, the capabilities of available input devices, and the graphics that may be obtained on output. On some computers, the character set is largely determined by the computer architecture. For example, on the IBM 360/370 series [38], the character is a hardware concept: an eight-bit byte. The CDC 6000 series machines [36], however, are word oriented; there is no intrinsic concept of a character. In this case, the character set is determined by software. This provides no greater flexibility, however; the processing of character data pervades the operating system and its support routines. Some machines, such as the DEC-10 [41], have instructions that permit the size of a character to be specified by the programmer. Although this offers greater flexibility in writing low-level software, the supporting operating system and its utilities still constrain the user of high-level languages to standard character sets.

Generally speaking, SNOBOL4 provides the full character set that is supported on the system on which it is implemented. On the CDC 6000 series, this is generally a 64-character form of BCD whose internal representation is called Display Code. On the DEC-10, 128-character ASCII is available, while on the IBM 360/370, 256-character EBCDIC is standard. In any case, a SNOBOL4 program can represent and process any available character. The bridge between the language and a particular character set is the value of &ALPHABET. On the CDC 6000 series, &ALPHABET is 64 characters long, while on the IBM 360/370, it is 256 characters long, and so on. Since &ALPHABET contains all the available characters, it can be used as a source of characters in a program. This is particularly convenient in cases where certain characters are awkward or impossible to represent explicitly when a program is prepared. The characters in the value of &ALPHABET appear in the order in which their internal representations are encoded. Appendix A contains tables showing the characters in &ALPHABET for CDC Display Code, ASCII, and EBCDIC. Because the "binary zero" in Display Code is interpreted in a special way by most CDC 6000 operating systems, it occurs at the end of &ALPHABET rather than at the beginning. This is a departure from the normal order.

In document preparation, the lower-case letters are frequently needed. These characters are not available in Display Code, but in ASCII they can be obtained as follows:

```
        &ALPHABET   TAB(97) LEN(26) . LCASE
```

In EBCDIC, the letters are divided into noncontiguous sections, making a more complicated method necessary:

```
        &ALPHABET   TAB(129) LEN(9) . LC1 TAB(145) LEN(9) . LC2
+                   TAB(162) LEN(8) . LC3
        LCASE    =  LC1 LC2 LC3
```

Other characters can be obtained in a similar fashion, according to their known positions in &ALPHABET.

In this chapter, the ASCII character set is used in programming examples, but not in any way that is essential to the material that is presented.

6.1.2. Processing Input Text

Quite frequently, the number of different characters that can be entered at an input device is less than the number that are available on the computer itself. Most keypunches and teletypes, for example, do not have lower-case letters and have only a few special characters, even if the computer being used supports the full ASCII or EBCDIC character sets.

Such problems necessitate the use of some form of encoding to represent the desired characters and related typographical information (for example, underlining). Consider the problem of representing the full ASCII or EBCDIC character sets in data prepared on a keypunch. While it is possible to "multi-punch" almost anything, such a technique is not practical for preparing large amounts of material in large character sets. The problem of representing a larger character set within a smaller one arises. The usual technique is to select a few characters of the smaller set and assign these characters syntactic significance. There are various ways of doing this; the specific method to be chosen depends on the nature of the text, on the available equipment, and on human factors considerations.

As an example, consider the problem of upper- and lower-case text, or viewed another way, the problem of capitalization. Most documents contain a preponderance of lower-case text. Many input devices, however, only provide upper case. Capitalization is governed by rules of grammar, but technical documentation frequently contains much capitalized material that is not part of the ordinary language. While it is possible to write capitalization algorithms that are effective in restricted contexts, methods must be provided whereby upper- and lower-case text can be prepared using a device that can create only upper case. Since lower-case text occurs most frequently in documents, it is common to convert upper case to lower case automatically and employ a special notation for capitalization. One common method [42] is to select a character, such as *, for this purpose. If a * precedes a letter, that letter is capitalized. An example is

```
*THIS ILLUSTRATES CAPITALIZATION
```

is used to represent

```
This illustrates capitalization
```

When a character such as * is preempted for a special purpose such as capitalization, some method must be provided for representing the character * itself. An easy method is to use ** to represent an actual * in the text. An example is

```
*THE RESULT IS OBTAINED BY EVALUATING F(A**B)
```

is used to represent

```
The result is obtained by evaluating f(a*b)
```

One general way of implementing this method is to consider the * as a character that causes special interpretation of the character following it. If the following character is a letter, that letter is capitalized. If the following character is not a letter, no change is made to it. The process of capitalization consists of a mapping that is applied to input text to put it in the desired internal form. A function MAP to perform this operation is

```
        DEFINE('MAP(LINE)HEAD,C')
        CAP     =   BREAK('*') . HEAD LEN(1) LEN(1) . C
        UCASE    =   'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
                   .
                   .
                   .
MAP     LINE    =   REPLACE(LINE,UCASE,LCASE)
CAP     LINE   CAP   =                          :F(DECAP)
        MAP    =   MAP HEAD REPLACE(C,LCASE,UCASE)   :(CAP)
DECAP   MAP    =   MAP LINE                     :(RETURN)
```

The value of LCASE is the lower-case letters, obtained in a manner appropriate to a particular system as described above. Capitalization of single letters is relatively convenient when this method is used. If a large amount of text is to be capitalized (for example, a program), capitalization of individual characters is impractical. Furthermore, if upper-case material (such as a program) is already in machine-readable form, it is very desirable to be able to include it in a document without the necessity for rekeyboarding it. In such cases, the concept of a case shift, as on a typewriter, is useful. Another character, such as $, can be preempted for this purpose. Consequently,

```
*THE FUNCTION $SORT$ IS USEFUL IF THE AMOUNT OF DATA IS SMALL.
```

could be used to represent

```
The function SORT is useful if the amount of data is small.
```

Used in this way, $ indicates that the normal conversion of upper-case input to lower-case input is to be suspended until another $ is encountered. In this sense, $ serves as a binary case-shifting switch. Unfortunately, an omitted or unintentional extra $ causes subsequent input text to be interpreted out of phase with respect to case. The problem can be overcome in a number of ways. One is to use one character to signal shift to upper case and another character to signal shift to lower case. If / is used to signal shift to lower case, the previous example would become

```
*THE FUNCTION $SORT/ IS USEFUL IF THE AMOUNT OF DATA IS SMALL.
```

The disadvantage of this technique is that it requires the preemption of another character from an already limited input set. Every character that is preempted for a special purpose requires special handling when it is to appear literally. This increases the difficulty of preparing text, the processing time required to convert it to internal form, and the possibility of introducing errors. Another technique is to use two occurrences of $ to signal shift to lower case:

```
*THE FUNCTION $SORT$$ IS USEFUL IF THE AMOUNT OF DATA IS SMALL.
```

Although this technique requires an extra keystroke, the convention is easy to learn and the presence of case shifts is easy to detect when proofreading the input text. MAP can be extended to handle case shifts as follows:

```
       DEFINE('MAP(LINE)HEAD,T,C,CCASE')
       CAP    =   BREAK('$*') . HEAD LEN(1) . T LEN(1) . C
                  .
                  .
                  .
MAP    CCASE  =   LCASE
MAPC   LINE   CAP   =                        :F(MAPE)S($('MAP' T))
MAP*   MAP    =   MAP REPLACE(HEAD,UCASE,CCASE) C    :(MAPC)
MAP$   MAP    =   MAP REPLACE(HEAD,UCASE,CCASE)
       IDENT(C,'$')                          :S(MAP)
       CCASE  =   UCASE
       LINE   =   C LINE                     :(MAPC)
MAPE   MAP    =   MAP REPLACE(LINE,UCASE,CCASE)    :(RETURN)
```

Note that CCASE is a local variable. Consequently, the normal, lower-case mode is restored at the beginning of each line.

Another matter that requires attention when a large character set must be encoded in a smaller one is the representation of special characters. There are several commonly used schemes for doing this. One is to think of the additional special characters as capitalized versions of characters that are available in the input character set. Thus [can be thought of as an upper-case (, and so on. If this method is chosen, the capitalization techniques discussed earlier can be extended appropriately. If the discrepancy between the sizes of the input and output character sets is large, this method is inadequate. Another approach is to introduce the concept of several different case shifts. A third, "supershift", is usually adequate.

A somewhat different approach is to introduce a character, say /, that indicates the character following it is to be taken as representing a special character not in the input character set. The character following / can be chosen for mnemonic value to make input text easier to prepare. Some typical correspondences might be:

encoding	*character*
/.	:
/,	;
/"	'
/E	!
/U	_

The characters that are useful in document preparation are those that have a graphic representation on the output device, as well as a few characters that may be used for control operations (such as tabulation). The number of printable graphics available on line printers is usually less than the number of characters supported by the computer system internally. Line printers that can print both upper- and lower-case letters and a number of special symbols are now generally available, however. Phototypesetting equipment, on the other hand, often offers an enormous number of characters in many fonts and sizes. In this instance, the number of different output graphics far exceeds the capacity of any computer system to represent them as individual characters. We will restrict our attention here to line printers, since they are generally available and since they offer the most practical means of producing the high-volume output typical of document preparation. There are also communications terminals that are useful for the preparation of documents on a low-volume basis. In some situations (for example, tabulation and overstriking), the mechanical properties of the output device must be considered in writing programs that produce formatted output. We will limit our discussion to producing line-printer output that is processed on a line-at-a-time basis, rather than a character-at-a-time which is typical of most terminals. Both line printers and communications terminals are monospaced devices; in other words, every character is the same width. Many phototypesetting devices handle variable-width characters such as are typically used in conventional typesetting. Such devices are beyond the scope of this chapter.

6.1.3. Other Problems in Representing Text

In all the examples above, input text can be translated by the text entry program into the internal character set that the formatting program can handle. There are conceptually related textual constructions that cannot be preprocessed in this way. Overstriking is an example. Mechanical output devices usually provide some facility whereby one character can be printed on top of another. On a line printer, for example, this is done by printing a second line on top of the one preceding it. Some communications terminals have the capability of backspacing and hence overstriking characters on an individual basis. Overstriking is typically used for three purposes:

(1) underlining
(2) obtaining boldface text
(3) constructing characters (such as θ) not otherwise available

In any event, the printing that eventually produces one print space in the output document must be represented by more than one character in the internal representation of the document. This necessitates the retention of some encoded information for overstriking in the internal representation of the document.

Underlining is usually represented in input in much the same way that case shifts are represented. In fact, underlining in typed material usually corresponds to italics in typeset material. Underlined text can be thought of as a representation of a type font that is different from that used for other text. Boldface text, achieved by printing the same characters more than once in the same position, can be considered similarly. Individual overstruck characters, however, are more naturally represented by the use of a "logical backspace" character. Suppose that the input character < is given this meaning. Then

```
0<-
```

could be used to represent

θ

EXERCISES

6.1 In the version of MAP that handles case shifting, what happens if a $ or * occurs at the end of a line?

6.2 Modify MAP to prevent automatic reversion to lower case at the beginning of each line.

6.3 What effect does MAP have on lower-case characters in the input?

6.4 Modify MAP to include the generation of special characters as described in the text.

6.5 Some types of keyboarding devices do not have correction facilities. To facilitate the use of such devices, incorporate "logical" correction facilities in MAP as follows:
(a) Treat \ as a logical backspace character that causes the character immediately preceding it to be deleted.
(b) Treat ~ as a logical null character that is deleted from the input text.
Describe the use of these facilities.

6.6 Write a program to analyze text and calculate the relative frequency of upper-case versus lower-case characters.

6.7 MAP assigns special meanings to certain characters. Discuss the considerations in selecting these characters.

6.8 Show that only one character need be assigned syntactic significance for use in MAP, regardless of the number of different encodings that must be handled.

6.9 Provide an "escape character" mechanism in MAP so that any character following an escape character is exempted from the syntactic significance it otherwise would have.

6.2. FORMATTING

Arranging representations in the input text for those graphics that are available on the output device is only a small part of preparing a formatted document. Layout of text on a page presents many problems. Some, such as handling the right margin, making provisions to skip lines where required, numbering pages, and so on, are common to most documents. Some types of documents require special formatting facilities that are idiosyncratic and have no general applicability. There are more sophisticated formatting capabilities, such as multi-column layout, automatic generation of tables of contents and indices, and so on. These capabilities present substantial implementation difficulties. We will start with the basic aspects of formatting and progress to more difficult topics, stopping short, however, of the really difficult problems.

From a programming point of view, most formatting problems can be divided into two categories: vertical formatting and horizontal formatting. Vertical formatting has to do with the placement of lines on a page, one after another. Horizontal formatting has to do with the division of input text into output lines, the positioning of characters in these lines from left to right, and the handling of special situations such as underlining. The simplest documents require no horizontal formatting; each line of input produces one line of output.

6.2.1. Vertical Formatting

In a document that requires only vertical formatting, each line of input text corresponds to a line of output. The formatting process is simply:

```
PRINT  OUTPUT  =  MAP(INPUT)                 :S(PRINT)F(DONE)
```

From an organizational point of view, it is more desirable to provide two functions, GET and PUT, which handle the input and output of text. In the simplest case, these functions might have the definitions:

```
       DEFINE('GET()')
       DEFINE('PUT(LINE)')
                 .
                 .
                 .
GET    GET    =  MAP(INPUT)                  :S(RETURN)F(FRETURN)
PUT    OUTPUT  =  LINE                       :(RETURN)
```

The printing statement then has the form

```
PRINT  PUT(GET())                            :S(PRINT)F(DONE)
```

This organization is unnecessarily elaborate for the trivial formatting described here. As formatting becomes more complicated, however, this organization will serve to clarify the processing that is taking place. The reason for putting the call of MAP in GET rather than in PUT will become clear when horizontal formatting is discussed.

The first step in vertical formatting is pagination. This requires counting lines, and when the end of a page has been reached, printing a page number and ejecting to the top of the next page. This function can be performed in PUT and suggests that all output should be localized in this one procedure. Suppose that the number of lines to be printed is specified by DEPTH and that the number of blank lines to be printed before the page number is specified by FOOT. The procedure PUT then becomes:

```
       DEFINE('PUT(LINE)I')
                  .
                  .
                  .
PUT    COUNT   =   LT(COUNT,DEPTH) COUNT + 1 :F(PUTP)
       OUTPUT  =   LINE                      :(RETURN)
PUTP   I   =  LT(I,FOOT) I + 1               :F(PUTT)
       OUTPUT  =                             :(PUTP)
PUTT   OUTPUT  =   PAGE.NO
       PAGE.NO   =   PAGE.NO + 1
       EJECT   =
       OUTPUT  =   LINE
       COUNT   =   1                         :(RETURN)
```

COUNT and PAGE.NO are global variables, maintained by PUT. EJECT has an output association of the form

```
       OUTPUT('EJECT',6,'(1H1,132A1)')
```

The page number printed by this procedure is placed at the bottom left of the page. Frequently, page numbers are centered or printed at the right on right-hand side pages. Such positioning requires the concept of line width, which is covered under horizontal formatting.

The function given above provides for automatic pagination so that a series of input lines produces an output document that is identical, line for line, but is divided into numbered pages. Such formatting affords no control to the user of the program, however. Frequently, for example, it is desirable to start material at the top of a page, regardless of what has been printed before. Starting a new chapter of a book is an example. Similarly, it is often necessary to skip lines in a document. This can be done by inserting blank lines in the input text, but that method is awkward, especially if a large number of lines are to be skipped. These rudimentary considerations lead to the concept of control information, as opposed to textual information. The design of a facility for specifying control information requires a knowledge of the type of information that must be supplied and a consideration for human factors. We shall use a relatively primitive, but effective, method here. Input lines are divided into two categories: text lines and control lines. In this scheme every line is either one kind or the other; there is no way to specify control information in a text line, or vice versa. This method makes the preparation of the input somewhat more cumbersome, but it has the virtue of making the program logic simpler and the processing easier.

In the design of such a facility, some method must be provided for distinguishing between the two types of lines. An expedient method is to require that every control line begin with a specially designated character. If this is done, the chosen character cannot be used as the first character of a text line. Consequently the character chosen should not be one that is likely to appear at the beginning of a text line. We shall use the character ? here.

Note: however unlikely it may be for a particular character to appear at the beginning of a line, the mere choice of a character dictates that it will have to appear at the beginning of a line in a formatted document. A formatting program should be capable of formatting its own documentation! This problem is a metasyntactic one, and may be resolved in a way similar to that used to resolve the related problem in preprocessing text.

The next problem to be resolved is the choice of syntax for control lines. Logically, there are two parts to the control information: the operation to be performed (such as line skipping) and an associated value (such as the number of lines to skip). For simplicity, we will assume that a control string, specifying the operation to be performed, immediately follows the ?. A value, if any, should be separated from the control string by blanks. The format of control strings can be relatively arbitrary, but should be made with simplicity and mnemonic value in mind. The letter S might be used to indicate that lines are to be skipped. A control line to cause the skipping of five lines is

```
?S 5
```

The formatting program can now be modified to process control lines as well as text lines. The logical place to put this operation is in the procedure for GET. Assuming that input lines are trimmed, control lines can be detected and analyzed by the following pattern:

```
        CONTROL    =    POS(0) '?' (BREAK(' ') . CSTRING SPAN(' ')
+                       REM . VALUE | REM . CSTRING NULL . VALUE)
```

If this pattern matches successfully, the control string is assigned to CSTRING and the value is assigned to VALUE. The two alternatives for VALUE are necessary because some kinds of control lines do not require the specification of a value. For example, forcing the beginning of a new page might be indicated by

```
?N
```

The procedure for GET becomes

```
GET     GET    =    INPUT                            :F(FRETURN)
        GET    CONTROL                               :S(GETC)
        GET    =    MAP(GET)                         :(RETURN)
                     .
                     .
                     .
```

Control lines can be handled in a number of ways depending on the division of logical operations between GET and PUT. We will assume that GET is to return the next text line, regardless of control lines. GET itself performs whatever processing is necessary when a control line is encountered, and continues to read until a text line is encountered.

Program organization for handling the control strings now becomes a problem. Two possible control strings have already been mentioned. Many others will be added as more sophisticated formatting is developed. It is therefore desirable to design a way of handling control strings that will not require revision of the existing program every time a new control string is implemented. The simplest way to do this is with a computed goto, having the processing for a given control string begin at a label that is the same as the control string. A statement to detect control strings is

```
         GET    CONTROL                               :S($CSTRING)
```

The control string

```
?S 5
```

causes transfer to the label S and the control string

```
?N
```

causes transfer to the label N. The program at S is

```
S        I   =   0
SS       I   =   LT(I,VALUE) I + 1                          :F(GET)
         PUT()                                              :(SS)
```

A control to permit specification of the page depth might be indicated by D. The processing statement is

```
D      DEPTH   =   VALUE                                    :(GET)
```

The use of an indirect transfer to access the control-processing statements is certainly simple, but it has a serious deficiency. If a control line is entered incorrectly, so that the specified operation does not correspond to a label in the program, the formatter terminates with a (SNOBOL4) error message. This type of fragility in the formatting program is very undesirable, especially since users of the program may not be programmers. Some versions of SNOBOL4 have methods of intercepting a transfer to an undefined label and permitting the program to retain control in an intelligent way. Unfortunately, the standard version of SNOBOL4 does not have such a facility. There are several ways of circumventing this problem. All require extra programming and processing. The most obvious way is to perform a pattern match on the control string to assure that it corresponds to a control processing label. Such a pattern is very cumbersome, however, and slow in execution. Furthermore, the pattern has to be rewritten every time a new control string is added to the formatter. An alternative method is to use a table that contains labels corresponding to the control strings. The processing label is obtained by indexing the table with the control string. A null value indicates an illegal control string. This method has the additional advantage of preventing transfers to labels in the program that do not correspond to control strings. For the purpose of the discussion here, we will simply use an indirect transfer, recognizing the importance of safeguards in a program actually used for formatting.

Implementation of the N control can be done in several ways. For example, N could be implemented as a skip of DEPTH - COUNT lines. There is obvious redundancy in this approach, since PUT checks COUNT every time it is called, duplicating the computation performed in skipping. Furthermore, the repeated calls of PUT are clearly unnecessary. More insight is gained by observing that it is necessary to end one page before beginning another. In addition, ending a page is required in other situations. For example, the last page of a document has to be ended in order for it to receive a page number. (Failure to do this is a common error in the first attempt at writing a formatting program.) The need to perform the page-ending operation in more than one place suggests a function:

```
       DEFINE('PAGEND()I')
                 .
                 .
                 .
PAGEND COUNT   =   LT(COUNT,DEPTH) COUNT + 1  :F(PUTP)
       OUTPUT  =                              :(PAGEND)
```

Note that PAGEND shares common code with PUT. The N control is then implemented by

```
N      PAGEND()                               :(GET)
```

Numbering the last page of the document can be accomplished with the following statement:

```
DONE   PAGEND()                               :(END)
```

One reason for skipping lines is to provide a visual break between sections of text. The break at the end of a page serves this purpose also. When a skip for the purpose of providing a visual break happens to cause a page break, blank lines may be printed at the top of the new page. These lines do not serve their intended purpose, and are undesirable. To solve this problem, skipping can be made conditional on the amount of space left on the current page. A test at S implements this version of skipping:

```
S       GE(VALUE,DEPTH - COUNT)                 :S(N)
SU      I  =  0
                   .
                   .
                   .
```

The label SU provides an "unconditional skip" of the type originally implemented, without the necessity for any additional program.

Another problem that arises in vertical formatting has to do with keeping a number of lines together on a page so that they are not visually broken. A simple solution to this problem is to force the beginning of a new page if there is not enough room on the current page for the specified number of lines. Using the letter K to indicate such "keeps", a control line to assure that the next 10 lines will appear on one page is

```
?K 10
```

This control can be implemented as follows:

```
K       GT(VALUE,DEPTH - COUNT)                 :S(N)F(GET)
```

A similar problem arises when a skip occurs near the end of a page. If forcing the start of a new page is acceptable, this type of skip is easy to implement, as illustrated above. If, on the other hand, the space is to be moved to a new page, but the current page is to be filled out with subsequent text first, the problem is more difficult. Such a problem also arises with text that is to be kept together and moved as a block past subsequent text if the current page does not contain adequate space. Generally speaking, any type of vertical formatting that requires rearrangement of text lines or the reservation of textual material for later insertion is substantially more complicated than the simple vertical formatting controls implemented above.

6.2.2. Horizontal Formatting

The vertical formatting described in the preceding section is basically simple because output of textual material is on a line-for-line basis with the input of textual material. This simple correspondence only holds for certain types of text such as poetry, computer programs, and tabular material that is logically separated into lines and can conveniently be keyboarded that way. The commonest type of text, however, does not have this property. This paragraph or a business letter—in fact almost all documentation—is more naturally thought of as a continuous stream of characters that is divided at certain points to fit within the left and right margins of a page. Input text of this kind is called "running text" as opposed to "as-is" text, which is considered on a line-by-line basis. A typist handles running text routinely, ending one line and beginning another so that no line is much longer than any other. This type of formatting is referred to as "ragged-right." Treatment of input text as a continuous stream of characters creates a number of problems. A line of input may be shorter or longer than the width of a page. Output cannot be performed until enough text has been accumulated from the input to make up an output line. When a line is output, there may be left-over text that constitutes the beginning of the next output line. Treating running text therefore requires a more complicated program than is required for as-is text. The procedure GET for handling running text might begin as follows:

```
GET     GET   =   INPUT                          :F(FRETURN)
        GET   CONTROL                            :S($CSTRING)
        STREAM   =   STREAM MAP(GET) ' '
        LT(SIZE(STREAM),WIDTH)                   :S(GET)
                      .
                      .
                      .
```

STREAM is a global variable to which input text is added until there are enough characters to make up an output line. The next problem to be solved is what to do when STREAM becomes long enough for GET to return a value. Let the value of WIDTH be the maximum length a line may have in ragged-right format. In general, STREAM cannot simply be broken at WIDTH characters: a word might be split in the process. In running text, blanks have a syntactic significance that they lack in as-is text; a line of running text can be broken wherever blanks occur. Figure 6.2 illustrates the four possible situations that may occur in the vicinity of WIDTH. The figure shows the possible cases for two words (strings of nonblank characters). Shaded areas indicate nonblank characters and unshaded areas indicate blanks.

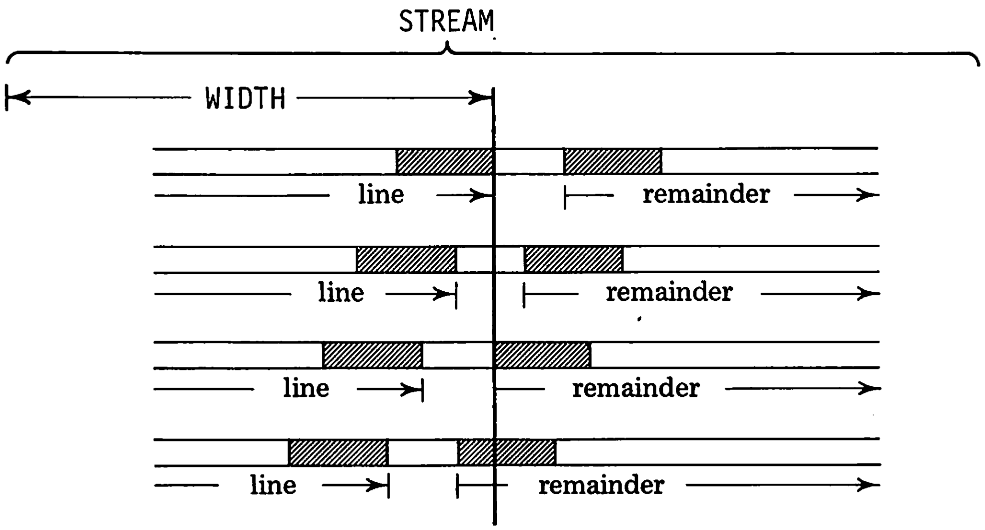


Figure 6.2 Ragged-Right Line Division

The problem is to divide STREAM at the rightmost span of blanks which starts no further to the right than WIDTH. This problem is one of a number of problems in which right-to-left pattern matching, if available, would be useful. Since that facility is not available in SNOBOL4, the approach is more difficult. One method is to divide STREAM at WIDTH and then locate the rightmost span of blanks in the left part. This can be done by using a technique similar to that used for locating the rightmost operator in an algebraic expression. The method is complicated and time-consuming. When it is complete, the various string segments must be put together properly to reconstruct STREAM. A more straightforward approach is to tabulate to WIDTH and look for an immediately following span of blanks, hoping that one of the first two cases shown in Figure 6.2 exists. If that fails, the amount tabulated is reduced, and the pattern match is repeated. The statements to do this follow.

```
        DIVISOR   =   POS(0) TAB(*(WIDTH - N)) . GET SPAN(' ')
+                     REM . STREAM
                  .
                  .
                  .
        N   =   0
SDIV    STREAM   DIVISOR                             :S(RETURN)
        N   =   LT(N,WIDTH) N + 1                    :S(SDIV)F(ERROR)
```

The branch to ERROR occurs if there are no blanks before WIDTH in STREAM.

In most books, type is set so that lines are "justified," flush both at the left and at the right. Justification is more easily accomplished in setting type because, unlike a typewriter or line printer, different letters have different widths and it is relatively easy to vary the spacing between characters and words in small increments. Justification on a typewriter or line printer can be accomplished by adding blanks between words to fill lines out to the right margin. The effect is usually more visually pleasing than a ragged-right format. Line justification is impractical in ordinary typing, since the process involves planning each line in advance and determining where to put blanks in order to give an acceptable appearance. Justification in a document formatting program is quite practical and is frequently employed. In fact, justification is more important in a document-formatting program than in typing. A typist uses hyphenation to break words over line boundaries and to reduce the raggedness at the right. Hyphenation is relatively complicated and requires handling of many special cases that cannot be covered by any systematic method. Computationally, hyphenation is complex, expensive, and error prone. A simpler solution is to disregard hyphenation and rely on justification to produce a visually acceptable document.

Justification, in itself, is not a difficult problem. Suppose that a segment of text is to be filled out to the line width. A procedure for doing this follows.

```
       DEFINE('JUST(JUST,WIDTH)LINE,HEAD,SEP,D')
       BLANKS   =   BREAK(' ') . HEAD SPAN(' ') . SEP
                    .
                    .
                    .
JUST   D   =   WIDTH - SIZE(JUST)
       LE(D,0)                                   :S(RETURN)
       JUST   BLANKS   =                         :F(RETURN)S(JUST2)
JUST1  JUST   BLANKS   =                         :F(JUST3)
JUST2  LINE   =   LINE HEAD SEP ' '
       D   =   GT(D,1) D - 1                     :S(JUST1)F(JUST4)
JUST3  JUST   =   LINE JUST
       LINE   =                                  :(JUST1)
JUST4  JUST   =   LINE JUST                      :(RETURN)
```

The second statement tests for lines that do not need justification or are too long to justify. Ordinarily JUST would not be given a line too long to justify, but that possibility must be considered. A line that is already of the desired length might be given, depending on the context in which JUST is called. The third statement of the procedure tests for lines that do not contain any blanks at all. If a line is too long to justify, or does not contain a blank, it is simply returned without modification.

This procedure satisfies the technical requirements stated above. Formatting involves aesthetic considerations, however. If justification is done by the procedure above, blanks are always added starting at the left end of the line. As a result, the left side of the page may appear lighter than the right or there may be "rivers" of blank space in the left part of the justified text. This effect is most obvious when viewing a full page of text, but it can be detected even in a small sample such as the one that follows:

```
Cryptographic   puzzles   appear   regularly  in  newspapers  and
magazines  and  there are organizations of individuals interested
in  cryptography.  Most  of  the  available  literature  is of a
popular  nature,  although  there are a few technical works.  See
the  references  listed  at  the  end  of this book.  Most of the
generally   available  information  on  cryptography  deals  with
methods that were in use prior to the early part of the twentieth
century.  More modern techniques, even some dating back to World
War I, are still highly classified government secrets.
```

The usual solution to this problem is to alternate the justification process, adding blanks from the left and right on alternate lines. Unfortunately, pattern matching in SNOBOL4 is strongly oriented around left-to-right operations, making addition of blanks from the right very awkward. One solution is to reverse the line when blanks are to be added from the right, and then to reverse the result. A global switch, say JSW, can be used to keep track of which way blanks are to be added. A modified version of JUST follows:

```
        JSW   =   1
                  .
                  .
                  .
JUST    D   =   WIDTH - SIZE(JUST)
        LE(D,0)                                   :S(RETURN)
        JUST   BLANKS                             :F(RETURN)
        JSW   =   -JSW
        JUST   =   LT(JSW,0) REVERSE(JUST)
JUST1   JUST   BLANKS   =                         :F(JUST3)
        LINE   =   LINE HEAD SEP ' '
        D   =   GT(D,1) D - 1                     :S(JUST1)F(JUST4)
JUST3   JUST   =   LINE JUST
        LINE   =                                  :(JUST1)
JUST4   JUST   =   LINE JUST
        JUST   =   LT(JSW,0) REVERSE(JUST)        :(RETURN)
```

The previous example, justified by the revised procedure, is:

```
Cryptographic  puzzles  appear  regularly  in  newspapers  and
magazines  and  there are organizations of individuals interested
in cryptography.  Most  of  the  available  literature  is  of  a
popular  nature,  although  there are a few technical works.  See
the references listed at the end  of  this  book.   Most  of  the
generally   available  information  on  cryptography  deals  with
methods that were in use prior to the early part of the twentieth
century.  More modern techniques, even some dating back to  World
War I, are still highly classified government secrets.
```

Note that justification does not affect the number of lines printed on a page. Justification simply adds blanks to push the end of a line to the right margin.

6.2.3. Combining Vertical and Horizontal Formatting

The major problem with writing a complete formatting program is not in handling the specific problems, such as pagination and justification, but in integrating all the formatting functions. Vertical formatting, in itself, is simple as long as the output of text has a line-for-line correspondence with the input of text. Running text, however, requires buffering of input. In particular, there is generally residual input text in STREAM to be printed.

This requires special handling when, for example, a skip is requested. The residual text in STREAM must be printed before the skip is performed, since a skip control specifies, logically, that lines are to be skipped following the text that precedes the control. A function FINISH serves this purpose, printing residual text and reinitializing STREAM whenever it is called:

```
        DEFINE('FINISH()')
                  .
                  .
                  .
FINISH  (DIFFER(STREAM) PUT(STREAM))          :F(RETURN)
        STREAM   =                            :(RETURN)
```

If FINISH is called while justified text is being formatted, the line produced will be ragged-right, not justified. This is the standard convention when terminating a body of justified text.

The statements to process skips now become:

```
S       FINISH()
        GT(VALUE,DEPTH - COUNT)              :S(N1)
        I   =   0
                  .
                  .
                  .
```

Notice that FINISH is called before testing the vertical space remaining, since FINISH itself may produce output which logically should appear before the skip is attempted. The statements at N now become:

```
N       FINISH()
N1      PAGEND()                              :(GET)
```

Modification of the SU and K controls is left as an exercise.

These changes, and similar ones for the other vertical formatting controls, only solve part of the problem. There are basically two forms of GET: one for as-is text and one for running text. Similarly, there are several modes of output. For as-is text, there is a pure as-is mode and a centered mode in which as-is text is centered within the page width. For running text, there are ragged-right and fully justified modes. To handle all combinations in one program, the two types of text are distinguished to determine the type of input processing to be done by GET. The modes are distinguished to determine what formatting (if any) is to be done on the string before it is returned by GET. These alternatives can be handled by indirect gotos, in which TYPE distinguishes between as-is and running text modes, and MODE determines the formatting. The relevant parts of the program follow.

```
       TYPE   =   'GETA'
       MODE   =   'RETURN'
                    .
                    .
                    .
GET                                                    :($TYPE)
GETR   LT(SIZE(STREAM),WIDTH)                          :S(GETI)
       N   =   0
SDIV   STREAM   DIVISOR                                :S($MODE)
       N   =   LT(N,WIDTH) N + 1                       :S(SDIV)F(ERROR)
GETI   GET   =   INPUT                                 :F(FRETURN)
       GET   CONTROL            GET                    :S($CSTRING)
       STREAM   =   STREAM MAP() ' '                   :(GETR)
GETA   GET   =   INPUT                                 :F(FRETURN)
       GET   CONTROL                                   :S($CSTRING)
       GET   =   MAP(GET)                              :($MODE)
CENTER GET   =   DUPL(' ',(WIDTH - SIZE(GET)) / 2)
+                GET                                   :(RETURN)
JSTFY  GET   =   JUST(TRIM(GET),WIDTH)                 :(RETURN)
*
A      MODE   =   'RETURN'                             :(TYPEA)
C      MODE   =   'CENTER'                             :(TYPEA)
J      MODE   =   'JSTFY'                              :(TYPER)
R      MODE   =   'RETURN'                             :(TYPER)
*
TYPEA  TYPE   =   'GETA'
       FINISH()                                        :(GETA)
TYPER  TYPE   =   'GETR'
       FINISH()                                        :(GETI)
```

A, C, J, and R correspond to control strings that establish the as-is, centered, justified, and ragged-right modes, respectively. Notice the use of FINISH to assure that residual text is disposed of before a new mode takes effect. A mode control terminates a body of text, even if the mode itself is not changed. Changes are also necessary in the vertical formatting controls to transfer to $TYPE instead of GET. The N control becomes

```
N      FINISH()
N1     PAGEND()                                        :($TYPE)
```

In addition, any residual text must be printed before the program terminates. The statements are:

```
DONE   FINISH()
       PAGEND()                                        :(END)
```

The formatting program outlined above lends itself to extension. A new control string can be added simply by adding the appropriate label and processing statements. Certain conventions must be followed:

(1) Any control string that causes output to be produced must first call `FINISH` to print any residual text.

(2) Any control string that establishes a formatting mode must, in addition to calling `FINISH`, assign the appropriate values to `MODE` and `TYPE`.

EXERCISES

6.10 Describe how pagination can be suppressed.

6.11 Design and implement a facility that permits the character used to indicate a control string to be changed during the processing of a document.

6.12 Modify the formatting program to assure that an erroneous control name does not cause an illegal transfer.

6.13 What happens if the control string `?PAGEND` is encountered?

6.14 Add error checking of control values to the formatting program.

6.15 In typed material, which is similar in appearance to material printed on a line printer, it is customary to provide two spaces after periods and other punctuation marks that end sentences. Implement this feature.

6.16 Add a continuation facility to the formatting program so that input text may be continued from one line to another.

6.17 Some control strings, such as `D`, have numeric values. Modify the pattern `CONTROL` so that the blank between the control name and the value is optional in the case of numeric values.

6.18 Modify the handling of control strings so that numeric values can be given as SNOBOL4 expressions.

6.19 Modify the formatting program so that more than one document can be processed in a single run.

6.20 Provide a facility so that more than one control string can be placed on a single line.

6.21 Discuss the implications of permitting control strings to be imbedded in text lines, thus eliminating the distinction between text and control lines. Design and implement a formatting program that handles input in this way.

6.22 Implement a control that prints a dividing line across the output page.

6.23 Provide a control that sets the value of the page number.

6.24 Provide a facility for turning off page numbering.

6.25 Provide a facility for specifying the position of the left margin of the formatted page.

6.26 Output from the formatting program is single spaced. Provide a facility for double spacing and, in general, specifying any number of spaces between output lines.

6.27 Vertical ellipses (dots) are used to indicate omitted material. Implement a control that generates such ellipses.

6.28 In as-is text, some types of formatting call for text to be printed flush against the right-hand margin. Implement a control for this type of formatting.

6.29 In addition to the types of skips described in the text, there is sometimes the need for a block of space (for example, to leave room for a drawing) that cannot be split over the end of a page. Implement this type of skip.

6.30 Modify the formatting program to place page numbers at the
(a) Top left corner of the page.
(b) Bottom center of the page.
(c) Top center of the page.
(d) Bottom left corner on even-numbered pages and bottom right corner on odd-numbered pages.

6.31 Provide a facility for specifying the desired position of page numbers as described in Exercise 6.30.

6.32 Provide a facility for generating lower-case Roman numerals for page numbers.

6.33 A "running head" is a line of information printed at the top of each page of a document. Typical examples are chapter and section titles.
(a) Implement running heads.
(b) Implement running heads so that different headings can be specifield for odd- and even-numbered pages.

6.34 A "running foot" is similar to a running head (See Exercise 6.33) except that a foot is printed at the bottom of every page. Implement running feet for the two cases given in Exercise 6.33.

6.35 Implement bracketed keeps in which one control string indicates the beginning of the material to be kept together and another control string indicates the end.

6.36 A "floating keep" is a form of keep (typically used for figures) that moves a body of material to be kept together through other text, if necessary, until there is enough room for it to be printed together on a page. Implement floating keeps.

6.37 Provide a control to permit the specification of the width of a page.

6.38 What should be done if a transfer to ERROR occurs while attempting to divide STREAM for justification?

6.39 As an alternative to the method used to divide STREAM for justification, write a pattern to perform this operation in a single pattern match. Compare the two methods.

6.40 Write a function to monitor the justification process and to provide statistics of the number of justifying blanks that are added.

6.41 One of the reasons for justifying text is that ragged-right formatting without hyphenation often produces results that are unacceptable from an aesthetic point of view. Provide a form of formatting that compromises between ragged-right formatting and full justification.

6.42 There are situations in which blanks are desired in a document but in places where lines should not be divided or justifying blanks inserted. Describe such situations. Devise a way of handling this problem.

6.43 Modify the handling of running text to permit dividing STREAM at places where hyphens occur in the text.

6.44 Provide a facility for specifying the beginning of a paragraph. Include a way of specifying the amount of indentation for the first lines of paragraphs.

6.45 Provide a facility for indenting a block of text from the left margin.

6.46 A hanging indentation is an indentation given to every line of a paragraph except the first. Provide such a facility.

6.47 Implement a facility for underscoring an entire line.

6.48 Implement a facility for printing an entire line in bold face.

6.49 Implement a facility for underscoring a specified portion of a line.

6.50 Implement a facility for printing a specified portion of a line in bold face.

6.51 Implement a facility for overstriking characters.

6.52 Provide a facility that prints marginal annotations on the output listing so that an author can include in a document notes of places that need special attention.

6.53 Provide a form of output in which line numbers of input and output are printed in the margins of the output.

6.54 Design a facility for handling tabulation in a manner similar to the way in which a typewriter operates.

6.55 Implement a facility for footnoting.

6.56 Implement a hyphenation facility.

6.57 Design a facility for producing multi-column output.

6.58 An entirely different approach to the handling of running text as done in the formatting program given above is to break up input into "words". Discuss the implications of such an approach, its advantages, and its disadvantages.

6.3. OTHER ASPECTS OF DOCUMENT PREPARATION

6.3.1. Indexing

One potential advantage of maintaining documents in machine-readable form is the automatic generation of indices, tables of contents, and similar supplementary listings. We will consider only indices here; the other cases are similar.

Considered naively, an index may be thought of as a type of concordance where all references to certain selected (indexed) words that occur in the document are tabulated. Such an index can be obtained by processing the document, looking for each word to be indexed and accumulating page references. This processing can be done while the document is being formatted, or as a separate process. Such an index is most likely to be inferior, and probably useless. A good index is restricted to significant page references and particular contexts. For example, the word "value" might appear with several different and distinct usages in a single document. To list all places where "value" occurs might be worse than useless—it might lead to confusion. Similarly, words appear in various forms such as singular and plural. Establishing equivalences automatically in all generality is a hopeless task. A good index, furthermore, may contain references to pages where the indexed word does not even appear explicitly, but where there is relevant material. This discussion provides arguments against the automatic generation of indices, but only in the sense of processing the text of the document itself. An alternative scheme combines the value of machine-readable material with the concept of deliberate, intelligent indexing by the author.

The idea is to provide a control string for designating items to be indexed in specific places. The value given consists of a list of the items to be indexed. An example is:

```
?INDEX PROPELLER,STABILITY,AIR FLOW
```

The effect of this control string would be to place the three listed items in the index, associated with the number of the page being formatted at the time the control string is encountered. The INDEX control has no effect on the formatting of the document itself. The author can insert INDEX controls as he sees fit, placing them at appropriate spots in the text. The implementation of this facility involves analysis of the INDEX control string, accumulation of the words and page numbers, and printing the results at the end of the document. A table is a natural way to store the words being indexed. The following statements can be added to the program:

```
       INDEX  =  TABLE()
       INDXI  =  BREAK(',') . ITEM LEN(1) | (LEN(1) REM) . ITEM
                 .
                 .
                 .
INDEX  VALUE  =  MAP(VALUE)
NEXTI  VALUE  INDXI  =                          :F($TYPE)
       INDEX<ITEM>  =  INDEX<ITEM> PAGE.NO ', '    :(NEXTI)
```

Note that applying MAP to VALUE converts the words to be indexed in the same way that textual material is processed. Any input encoding conventions used in preparing text can be used for items to be indexed as well. In the example above, the three items would be entered in the index in lower case. The index can be printed at DONE after the document itself is completed:

```
DONE   FINISH()
       PAGEND()
       ~~INDEX  =~~  PRINT(SORT(INDEX))        :(END)
```

The functions SORT and PRINT are described in Section 2.4.

The discussion of indices given here is brief and superficial. A good index requires one or more levels of subentry, cross references, designated primary page references, and so forth. The overall problem is quite difficult.

6.3.2. Abbreviations

It is customary to use abbreviations when writing drafts. Such abbreviations save time and space, and they also sometimes serve as convenient specifications of standard forms. Abbreviations take two common forms:

shortening of words leaving enough characters to provide an identification of the full words, and use of initial characters of words in a phrase in the style of acronyms. Where a word is abbreviated, a terminal period is usually used to signify the presence of an abbreviation. Some abbreviations are so common that they are generally recognizable to any reader. Examples are "etc." and "ASAP". Other abbreviations have special meanings and are intelligible only to the individual who invented them. Some abbreviations (for example, "etc.") have become part of the generally accepted language, while others serve only as a personal convenience or in a limited context.

Preparing text in machine-readable form is a substantial effort; it is costly and time-consuming. An abbreviation facility in a document formatting program can serve to limit the size and reduce the cost of preparing machine-readable material. The utility of such a feature may far exceed the utility of abbreviation in a manual system. An additional advantage is having a simple, short, and uniform representation for a string of input that is difficult to keyboard and subject to accidental variations when keyboarded repeatedly without abbreviation.

In order to implement such a feature, two things are necessary: a mechanism for including abbreviation definitions in the document, and a way of specifying an occurrence of an abbreviation in the text of the document. The definition of an abbreviation has two parts: the name of the abbreviation, and the text which replaces the abbreviation when the document is formatted. There are several ways in which an abbreviation might be specified in the text of the document. An important consideration is the ease with which abbreviations can be used. It should be possible to use common characters and not be restricted to awkward combinations of special symbols. On the other hand, it is necessary to be able to distinguish abbreviations from words that ordinarily occur in text. Some special syntax is desirable to make the process of locating abbreviations easy and to avoid unnecessary processing of input text that may contain few, if any, abbreviations. The convention of indicating an abbreviation by a terminating period is undesirable because of other uses of periods as sentence terminators and so forth. In addition, since it is easier to process text from left to right, an initial indicating character is more convenient than a terminating character. We shall use the character > as an indication of an abbreviation and let the string of letters and digits following the >, up to some other type of character, be a name for the abbreviation. For example, the input string

```
*HAVE HIM CALL >ASAP.
```

might be "expanded" into

```
Have him call as soon as possible.
```

Establishing definitions for abbreviations can be done by control strings. An example for the abbreviation above might be

```
?DEF ASAP,AS SOON AS POSSIBLE
```

A comma is used to separate the abbreviation name from its definition. There is no ambiguity since a comma is not one of the characters that can occur in an abbreviation name. As one would expect, such a definition must appear before the first use of the abbreviation being defined.

The implementation of abbreviations in the formatting program now involves two parts: maintaining a table of definitions and processing the input text to replace abbreviations by their definitions. The table of definitions can be implemented as follows:

```
       ABBREV   =   TABLE()
       DEFIN    =  BREAK(',') . NAME LEN(1) REM . DEF
                 .
                 .
                 .
DEF    MAP(VALUE)   DEFIN                          :F(CERR)
       ABBREV<NAME>   =   DEF                      :($TYPE)
```

Note that MAP is applied to VALUE. This permits encoding the text for the abbreviation name and definition in the usual fashion.

There is now the question of replacing the abbreviations in the input text by their definitions. A function to perform this operation follows:

```
       DEFINE('EXPAND(LINE)HEAD,NAME')
       ABRVPAT   =   BREAK('>') . HEAD  LEN(1)
+                    SPAN(LETTERS DIGITS) . NAME
                 .
                 .
                 .
EXPAND LINE   ABRVPAT   =                          :F(EXPEND)
       EXPAND   =   EXPAND HEAD ABBREV<NAME>       :(EXPAND)
EXPEND EXPAND   =   EXPAND LINE                    :(RETURN)
```

Note that because of the way that tables are implemented in SNOBOL4, a reference to an undefined abbreviation is replaced by a null string and vanishes without being noted.

The logical place for EXPAND to be called is in MAP. The placement of EXPAND in MAP deserves some consideration. In order to be able to use the encoding and correction conventions in the definitions of abbreviations, EXPAND should be applied after other operations have been performed. Furthermore, if EXPAND is called before the other operations of MAP are performed, these operations will be performed twice when MAP is called in processing the control string for DEF.

6.3.3. Automatic Numbering

Page numbers are provided automatically during the formatting process. This is essential, since page numbers cannot be anticipated when the input text is being prepared. Most documents contain other kinds of internal numbering that have no direct relation to page numbering. Examples are chapter and section numbers, figure numbers, and lists of numbered items. Numbering is particularly prevalent in technical documents.

Numbering is commonly done by the author of the document, using a preliminary table of contents for chapter and section numbers, and assigning numbers sequentially to figures and items in lists. However well-planned a document is, there are likely to be additions, deletions, and rearrangements. Such changes are likely to require corresponding numbering changes. The required revisions, especially in heavily-numbered technical documents, may represent an enormous amount of work if done manually. A good text editor, operating on the input text, can be very helpful in such situations. An alternative scheme is to have numbers generated automatically while the document is being formatted. If the document is changed, new numbers are generated when the document is formatted without the necessity for any explicit numbering changes.

The problem of implementing such a feature is similar to the problem of implementing abbreviations. There must be a general mechanism for identifying places where numbers are to be generated and a way of distinguishing between different generators. The latter problem arises because, for example, section and figure numbering represent distinct, but concurrent, processes. In addition, it is convenient to be able to specify what initial values are to be used for numbering and to reset such values as desired.

Again, there must be some syntactic way of designating places where automatic number generation is to occur. We shall use the symbol # for this purpose. Since there may be several concurrent numberings, a different name is used to identify each distinct numbering sequence. A one-character name will be used as a matter of economy and convenience, but this restriction is not essential. The rationale is that one character is adequate because the number of different sequences is likely to be relatively small. Such a rationale does not apply to abbreviations which are likely to be greater in number, and benefit from longer names with their corresponding mnemonic value.

An example of the occurrence of a number generator in the text is

```
*THE RESULTS ARE SHOWN IN *FIGURE #F.
```

Here the name of the generator is `f`. The result of "expanding" this line might be

```
The results are shown in Figure 17.
```

One problem immediately becomes evident: there are places where a number is only referenced and there are places where a number should automatically be incremented. Suppose, for example, that the line above occurs before the figure that is being referenced. When the figure caption itself occurs, the number 17 should be generated, not the next number, 18. A way to handle this problem is to have two types of references: one that automatically increments and one that simply gives the current value. To avoid preempting another character, we shall use the convention that ## indicates the current value, while a single # increments the value. The figure caption referred to above might be:

```
*FIGURE ##F -- *THE *FINAL *OUTPUT
```

which would expand into

```
Figure 17 -- The Final Output
```

Number generation can be included in EXPAND. The necessary statements are:

```
       GENER   =   BREAK('#') . HEAD LEN(1) LEN(1) . G
       NGENER   =   LEN(1) . G
       GEN    =   TABLE()
                  .
                  .
                  .
EXPEND EXPAND   =   EXPAND LINE
GENER  EXPAND   GENER   =                          :F(RETURN)
       IDENT(G,'#')                                :F(GENER1)
       EXPAND   NGENER   =                         :S(GENER2)F(ERROR)
GENER1 GEN<G>   =   GEN<G> + 1
GENER2 EXPAND   =   HEAD GEN<G> EXPAND             :(GENER)
```

Here GEN is a table that is indexed by the name of the generator. Since a previously unreferenced generator has a null value, the first (incrementing) reference sets its value to 1. No initialization is needed if a specific generator is to start with 1 for its first (incrementing) reference. Frequently, however, it is desirable to be able to set the value of a generator. An example occurs for subnumbering within a section which is to start over at the beginning with a new section. A control string such as

```
?SET F,0
```

could be used for this purpose. The implementation of SET is straightforward.

6.3.4. Modifying the Formatting Program During Execution

As it has been developed so far, the formatting program reads input lines and treats them as textual material or control strings depending on the first character of the line. Control strings select appropriate sections of the formatting program. One of the most powerful features of SNOBOL4 is its ability to create and execute new statements during the course of execution. This facility can be used to provide a way of modifying the program from the input text itself. Consider the following statements:

```
EXEC    FINISH()
        EXEC  =  CODE(' ' VALUE ' :($TYPE)')  :S<EXEC>F(CERR)
```

These statements implement an EXEC control which causes VALUE to be interpreted as a SNOBOL4 statement (without a label or gotos) which is converted to CODE and then executed. An example of the use of this facility is given by

```
?EXEC  PUT(DUPL('.',WIDTH))
```

which prints a line of dots in the output document at a place corresponding to the position of the control string in the input. The use of such a facility requires a knowledge of SNOBOL4 and the formatting program itself. It nonetheless provides an extremely powerful tool for the sophisticated user and makes it possible to extend or modify the formatting program without the necessity for revising the program itself.

EXERCISES

6.59 In certain circumstances, the method of indexing used in the text may not produce the correct page number for a cited item. Discuss this problem.

6.60 Revise the function PRINT to provide a suitable format for indices.

6.61 Provide a facility for generating several different indices simultaneously for the same document.

6.62 Provide a facility so that a primary page reference for an index entry can be specified and printed in bold face in the index.

6.63 Provide facilities whereby index subheadings may be specified.

6.64 Implement a facility for generating tables of contents.

6.65 Design a method for embedding index specifications in text lines instead of placing them on control lines.

6.66 Add a facility to produce a supplementary listing of errors that are encountered during formatting.

6.67 Provide a method of detecting and noting undefined abbreviations.

6.68 As implemented in the formatting program, abbreviation names are only terminated by a character that is not a letter or a digit. There are situations in which this prevents the use of abbreviations. Discuss this problem and provide a solution.

6.69 Provide a way of getting the character > into a document.

6.70 Provide a facility for printing a supplementary listing of places in a document where abbreviations occur.

6.71 Abbreviations as implemented in the formatting program always produce fixed replacements. Extend the abbreviation facility so that parameters can be provided when the abbreviation is used and inserted in the replacement string at desired places.

6.72 The abbreviations discussed in the text are passive in nature; they can perform no computation nor can they affect the formatting program. Design an "active" abbreviation facility.

6.73 Implement the SET control.

6.74 Design a facility for generating successive alphabetic identifiers.

6.75 Determine the effect of the following control strings and describe what they might be used for:

```
1. ?EXEC INPUT('INPUT',10)

2. ?EXEC DETACH('OUTPUT')

3. ?EXEC OUTPUT('OUTPUT',7,'(80A1)')

4. ?EXEC PUT(INPUT)

5. ?EXEC &TRACE   =   1000
   ?EXEC TRACE('STFCOUNT','KEYWORD')

6. ?EXEC COLLECT(1000)   :S(FRETURN);
```

6.76 Write a control string that "undefines" the R control so that subsequent use of R as a control is illegal.

6.77 Design a control similar to EXEC, but which simply converts the value to CODE without executing it. What use does such a control have?

6.78 Show how to turn off the function MAP during formatting.

6.79 Show how to turn off abbreviation and number generation during formatting.

6.80 Discuss the role of defined functions in the formatting program.

7 ADDITIONAL APPLICATIONS

7.1. A RANDOM SENTENCE GENERATOR

One way of characterizing a class of strings, i.e. a language, is by the use of a grammar. BNF, described in Section 1.2, is the best known system for formally specifying the class of languages of greatest practical interest. A simple BNF grammar for a restricted class of arithmetic expressions provides an example:

```
<ADDOP>::=-|+
<MULOP>::=*|/
<VARIABLE>::=X|Y|Z
<TERM>::=<VARIABLE>|(<EXP>)|<TERM><MULOP><VARIABLE>
<EXP>::=<TERM>|<EXP><ADDOP><TERM>
```

<VARIABLE> is deliberately limited to simplify the discussion that follows. One method to determine what strings of terminal symbols ("sentences") are <EXP>s is to start with <EXP> and see what is produced by trying the various alternatives. <EXP> leads to <TERM> as one possibility. <TERM> leads to <VARIABLE> which leads to X, for example. Therefore, X is an <EXP>. A slightly more complicated situation results if the third alternative for <TERM> is chosen instead of the first. In this alternative, there are three subsequents, <TERM>, <MULOP>, and <VARIABLE>. Suppose the second alternative for <TERM> is selected, the second for <MULOP>, and the third for <VARIABLE>, the result is now (<EXP>)/Z . We are back to where we started, having accumulated a few terminal symbols. We already know that X is an <EXP>, so certainly (X)/Z is an <EXP>. Different choices for alternatives lead to

other terminal strings, generally of greater length and requiring more intermediate steps. Of course, there are sequences of choices for alternatives that never terminate. If the first alternative is always chosen for <EXP>, and the second alternative is always chosen for <TERM>, a loop results.

It is possible, but not particularly useful, to devise a scheme for generating all possible sentences. For any grammar of practical interest, there are an infinite number of possible sentences. The problem is, therefore, not how to generate all the sentences, but how to generate a representative subset of sentences. For <EXP> above, there are an infinite number of sentences of the form X, (X), ((X)), (((X))), and so on. These sentences are not, however, representative of the language generated by <EXP>.

One solution to this problem, especially when the number of different kinds of "representative" sentences is very large, is to generate randomly selected sentences which, by virtue of their random selection, are likely to give a representative picture of the language. It is this problem that we will address in this section.

7.1.1. The Generation Process

We will assume for the moment that an appropriate data structure for representing grammars has been developed and will first consider the generation process. A stack is used to store intermediate results. Since only one stack is required, the stack functions are used without a specified stack argument as suggested in Exercise 3.1. Given a nonterminal, a function SELECT selects one of its alternatives, perhaps on a random basis. A function SPUSH pushes the subsequents of that alternative. A predicate TRML is used to determine whether a subsequent is terminal or nonterminal. A simple program for generating sentences follows:

```
G1      S  =  GOAL
G2      S  =  SPUSH(SELECT(S))                  :(G4)
G3      S  =  POP()                             :F(G5)
G4      SENTENCE   =   TRML(S) S SENTENCE       :S(G3)F(G2)
G5      OUTPUT   =   SENTENCE
        NUMBER   =   LT(NUMBER,COUNT) NUMBER + 1  :F(END)
        SENTENCE   =                            :(G1)
```

The test on the number of sentences generated is used to limit the output. At G1, the nonterminal for which sentences are desired is assigned to S. At G2, an alternative is selected. The subsequents are pushed onto the stack in order from left to right, except for the last subsequent which is returned as the value of SPUSH. If S is terminal, it is concatenated onto the front of the developing sentence and the next subsequent is popped. Otherwise, the selection process is repeated. If the stack becomes empty, generation is complete and the sentence is printed. The generation process is restarted to generate another sentence.

For the grammar given above, a sequence such as the following might occur, starting with <EXP>.

1. Suppose the second alternative for <EXP> is selected. The contents of the stack become:
   ```
   <ADDOP>
   <EXP>
   ```
   and <TERM> is assigned to S.
2. Since S is nonterminal, a selection for <TERM> is performed. Suppose the second alternative for <TERM> is selected. The stack becomes:
   ```
   <EXP>
   (
   <ADDOP>
   <EXP>
   ```
   and S is).
3. S is terminal and becomes the value of SENTENCE. Continuing, the next value, <EXP>, is popped off the stack. Assuming that the first alternatives are selected for <EXP> and then <TERM>, the value of S is <VARIABLE>. If the second alternative for <VARIABLE> is selected, SENTENCE becomes Y).
4. The stack is now:
   ```
   (
   <ADDOP>
   <EXP>
   ```
5. Since (is terminal, SENTENCE becomes (Y) and <ADDOP> is popped. Assuming that the first alternative is chosen for <ADDOP>, SENTENCE becomes -(Y).
6. <EXP> alone remains on the stack. If the first alternative is selected for it, the first for <TERM>, and the third for <VARIABLE>, SENTENCE becomes Z-(Y).
7. At this point the stack is empty, SENTENCE is printed, and, if the generation limit has not been exceeded, the process begins again.

7.1.2. Representation of the Grammar

Many questions are still unanswered, most centering around the way the grammar is represented. Representation of the grammar is substantially more difficult than the actual generation of sentences. Notice, however, that the program for generating sentences is essentially independent of the way the grammar is represented. The representation, in fact, merely determines the details of the functions SELECT, SPUSH, and TRML. The external description of these functions is largely independent of the representation of the grammar. Conversely, the method of representing the grammar should be chosen so that these functions can be written easily and will operate with relative efficiency. The following representation will be used.

First there is a table DT, whose entries point to structures corresponding to definitions for the nonterminals of the grammar. For example, the value of DT<'EXP'> would be the definition of <EXP>.

Each definition consists of an array of alternatives, and each alternative consists of an array of its subsequents. The definition for <TERM> can be visualized as shown in Figure 7.1.

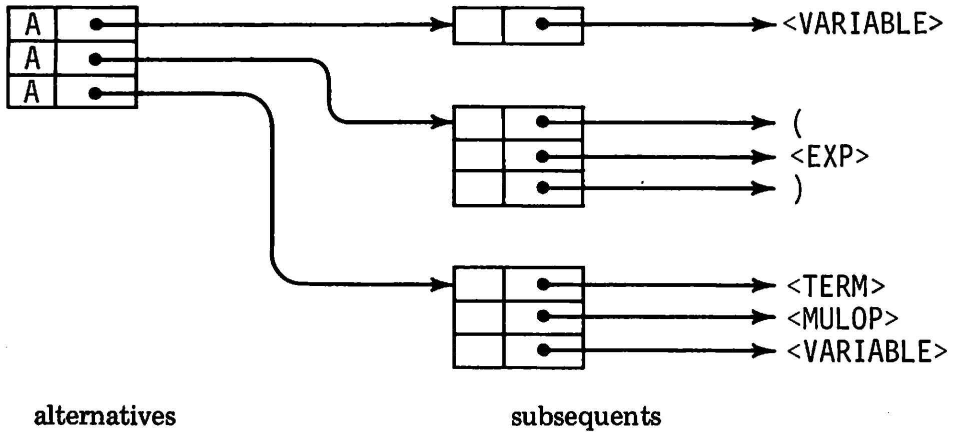


Figure 7.1 Definition for <TERM>

In this figure, terminals and nonterminals are indicated by the forms used in BNF. In the program, it is more practical to represent terminals by strings and nonterminals by defined data objects, which in turn point to the (string) names of the nonterminals. A defined data type for nonterminals is given by

```
        DATA('NT(NAME)')
```

In terms of SNOBOL4 structures, the definition of <TERM> then has the structure shown in Figure 7.2.

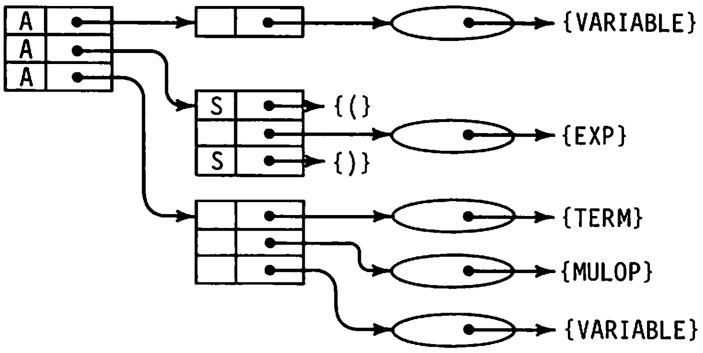


Figure 7.2 Data Structure for <TERM>

The structure of DT is shown in Figure 7.3.

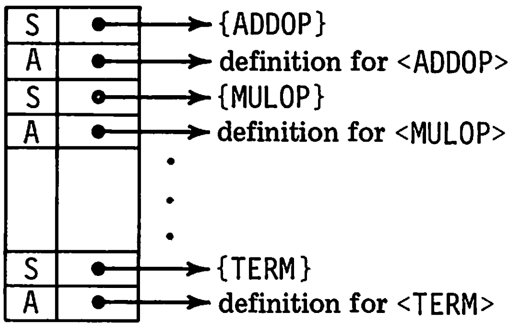


Figure 7.3 The Table of Definitions

This representation of the grammar is more complicated than it needs to be. It is, however, easier to construct than a more compact representation, and it serves as a starting point.

Assuming that the program is to read in data describing the grammar for which random sentences are to be generated, construction of the internal structures for the grammar presents several technical problems. First, a format for the grammar must be selected. We will use BNF in its standard form as described earlier. For simplicity, we will confine the definition of a nonterminal to a single line and dispense with error checking for the first attempt. To allow for a specification of the nonterminal goal that is to be used to generate sentences and to allow for specification of how many sentences are desired, it is assumed that the grammar is followed by a line listing the goal and number of desired sentences. By selecting the syntax for this line to be different from that for a BNF definition, the end of the grammar may be determined easily.

Analyzing BNF is a pattern-matching problem, which, while largely straightforward, presents some technical difficulties. Another rather typical problem arises from the fact that the number of alternatives and subsequents that may be present in a definition cannot be determined in advance (although the restriction of definitions to one line does provide upper bounds). The approach is to allocate arrays larger than expected and truncate them on completion of the definition (see Exercise 2.7). The part of the program that constructs the representation of the grammar follows. The syntax of the line that terminates the grammar is indicated by GENPAT.

```
&ANCHOR   =   1
&TRIM   =   1
DATA('NT(NAME)')
DT   =   TABLE()
GENPAT   =   '<' BREAK('>') . NAME '>:' REM . COUNT
DEFPAT   =   '<' BREAK('>') . NAME '>::=' REM . DEF
```

```
       ALTPAT   =   BREAK('|') . ALT LEN(1) | (LEN(1) REM) . ALT
       SUBPAT   =   '<' @P BREAK('>') . SUB '>' | @P (LEN(1)
+                   BREAK('<')) . SUB  | @P (LEN(1) REM) . SUB
       AL   =   10
       SL   =   10
                .
                .
                .
NEXTL  LINE   =   INPUT                                 :F(ERROR)
       LINE   DEFPAT                                    :S(NEXTD)
       LINE   GENPAT                                    :S(GENER)F(ERROR)
NEXTD  OUTPUT   =   LINE
       I   =   0
       AA   =   ARRAY(AL)
NEXTA  DEF   ALTPAT   =                                 :F(EOA)
       I   =   I + 1;   J   =   0
       SA   =   ARRAY(SL)
NEXTS  ALT   SUBPAT   =                                 :F(EOS)
       J   =   J + 1
       SUB   =   GT(P,0) NT(SUB)
       SA<J>   =   SUB                                  :S(NEXTS)F(ERROR)
EOS    AA<I>   =   TRUNC(SA,J)                          :S(NEXTA)F(ERROR)
EOA    DT<NAME>   =   TRUNC(AA,I)                       :S(NEXTL)F(ERROR)
GENER  GOAL   =   NT(NAME)
G1     S   =   GOAL
                .
                .
                .
```

One "trick" employed in the pattern SUBPAT deserves mention. A subsequent may be either terminal or nonterminal. Writing a pattern to match either case is not difficult, but determining which case occurred is more of a problem. One method is to perform one pattern match to isolate the subsequent and another to determine its type. This approach is obviously repetitious, since the same string must be examined twice. The cursor position operator in SUBPAT distinguishes the two cases without the need for rematching. If the type is nonterminal, the value assigned to P is 1, while if the type is terminal, the value assigned to P is 0. Subsequently, the value of P determines whether or not an object of data type NT is generated for the subsequent.

7.1.3. The Generation Functions

All that remains is the definition of the functions used by the generation section of the program. To determine if a subsequent is terminal or nonterminal, it is sufficient to test its data type. The function TRML is simply:

```
        DEFINE('TRML(X)')
                  .
                  .
                  .
TRML    IDENT(DATATYPE(X),'STRING')       :S(RETURN)F(FRETURN)
```

To select an alternative, the definition of a nonterminal must be obtained from its name. A pseudo-random number generator then provides a subscript to index the array of alternatives. A definition for SELECT follows.

```
        DEFINE('SELECT(X)')
                  .
                  .
                  .
SELECT A    =   DT<NAME(X)>
        SELECT  =   A<RANDOM(PROTOTYPE(A)) + 1>   :(RETURN)
```

The function SPUSH pushes subsequents on the stack except for the last subsequent, which is returned as value. The value returned by SELECT is a pointer to an array of subsequents, and provides the argument to SPUSH. The procedure for SPUSH is slightly complicated by the fact that the last subsequent is not pushed. Otherwise, a simple loop through the array accomplishes the desired result.

```
        DEFINE('SPUSH(A)I,N')
                  .
                  .
                  .
SPUSH  N    =   PROTOTYPE(A) - 1
SPUSH1 I    =   LT(I,N) I + 1                :F(SPUSH2)
       PUSH(A<I>)                            :(SPUSH1)
SPUSH2 SPUSH   =   A<N + 1>                  :(RETURN)
```

The random sentence generation program is completed by the inclusion of functions that have already been written: the stack manipulation functions (Section 3.2.1), RANDOM (Section 2.3.1), and TRUNC (Exercise 2.7).

EXERCISES

7.1 Extend the definition of <VARIABLE> to include a reasonable class of variables.

7.2 Add unary operators to the grammar for arithmetic expressions.

7.3 Write a BNF grammar for prefix expressions.

7.4 Modify the random sentence generator to handle the following situations:

(a) Occurrence of more alternates than are allowed by AL.
(b) Occurrence of more subsequents than allowed by SL.
(c) Occurrence of a nonterminal for which there is no definition.

7.5 Modify the random sentence generator to handle a definition that is too long to fit on a single input line.

7.6 The symbols <, >, and | have syntactic meaning in BNF definitions, yet there are situations (for example in defining BNF) when it is desirable to be able to include these symbols in sentences. Devise a means for including these symbols in definitions. (Why is it that the symbols : and = do not cause problems?)

7.7 Using the result of Exercise 7.6, write a grammar describing BNF and generate some "representative" BNF definitions.

7.8 Develop a method of specifying the end of a line on output so that a single nonterminal can produce several lines of output.

7.9 Suppose that a definition requires trailing blanks. How can this be handled?

7.10 The random sentence generator treats each alternative as being equally likely. It is frequently desirable to be able to treat some alternatives as more likely than others. (If this is not obvious, try generating 10 instances of <EXP>.) Devise a scheme for incorporating weights in definitions and adapt the random sentence generator to operate accordingly.

7.11 The representation for grammars used in the program above requires an indirect reference, using a table, to get from the name of a nonterminal to its definition.
(a) Devise a more compact representation.
(b) Modify the random sentence generator to handle this more compact representation.

7.12 Write a simple BNF grammar for declarative sentences in English. Include definitions for noun phrases, transitive verbs, intransitive verbs, and so forth. Select a few words for the terminals. For example, <NOUN> might be defined as

```
<NOUN>::=BOX|HATBOX|BALLPARK|DIRIGIBLE
```

Run the random sentence generator on this grammar and observe the results. Try modifying the grammar to obtain a large percentage of "meaningful" sentences.

7.13 Interface the result of Exercise 7.12 to the program that counts words described in Exercise 2.40.

7.14 Modify the random sentence generator so that more than one grammar can be processed in a single run.

7.15 Add tracing facilities to the random sentence generator to display the course of generation.

7.16 Extend the random sentence generator so that sentences corresponding to a number of different nonterminals can be generated in sequence. Make the generation cyclic and use the result to generate random poetry.

7.17 Write a program to process a grammar and for each nonterminal, list all nonterminals that can be reached by starting at the given nonterminal and selecting all possible alternatives. For the example given in the text, no nonterminals can be reached from `<ADDOP>`, `<MULOP>`, or `<VARIABLE>`, but all nonterminals can be reached from `<TERM>` and `<EXP>`.

7.18 Generate random SNOBOL4 statements. Test the statements by converting them to `CODE`.

7.19 Use the random sentence generator to create random polynomials in several variables.

7.20 Write a random sentence generator that generates all the sentences of a finite language, first verifying that the language is finite.

7.21 Investigate the possibility of creating random structures (trees, lists, graphs, and so forth) using the random sentence generator as a model.

7.2. TURING MACHINES

Turing machines provide a particularly simple and theoretically important model of computational processes. Essentially, a general-purpose computer can perform any computation that can be performed by a Turing machine. As computers, Turing machines are uninteresting because their simplicity makes any computation of interest hopelessly tedious to formulate and slow to complete. It is this simplicity, however, reducing the computational process to a bare and minimum formulation, that makes Turing machines of theoretical interest. Modeling a Turing machine provides a tool for understanding the underlying concepts and provides a tutorial mechanism for displaying the operation of Turing machines. This section provides such a program model for Turing machines—a simulator.

Definitions of Turing machines are typically formal and detailed because the problems associated with them are of a mathematical nature and a precise formulation is necessary. Such notations are not much help in gaining an intuitive understanding of Turing machines, but are given here because the program model is designed to accept such descriptions as input. Before going to the formal notation, however, an informal description of Turing machines will help in understanding the functions the program must perform.

A Turing machine consists of a controller which is always in one of a number of states. The controller has a tape head which can read a tape. The tape consists of a number of cells, each of which contains a symbol. Some symbols may be blank. In addition to being able to read the tape, the controller can move the tape head to the left or right and write on the tape. The behavior of the controller is determined by the state it is in and the symbol currently being read. Depending on the current state and the symbol, the controller writes a symbol in the current cell (replacing the one that is there), moves the tape one cell to the left or right, and transfers to a new state. Figure 7.4 illustrates a Turing machine schematically.

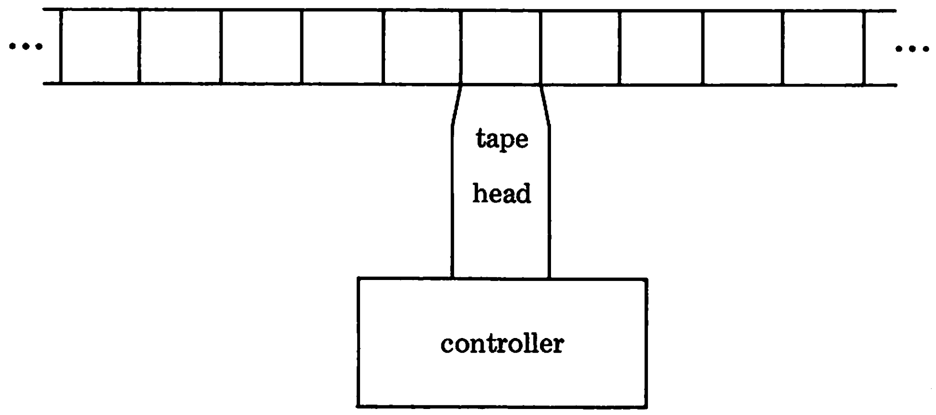


Figure 7.4 A Turing Machine

The number of states and the symbols are fixed and finite, although the tape is arbitrarily long. The tape constitutes the "memory" of the Turing machine. When the Turing machine begins operation, the tape has symbols already written on it. These symbols represent the computation to be performed. During operation, the Turing machine uses the tape as a "scratch pad", and the result it computes is written on the tape as well. When a computation is complete, the Turing machine halts. There are, of course, situations in which a machine does not halt but loops endlessly. Generally speaking, the number of states is small, the number of different symbols is small, and computations are performed in an intricately encoded notation.

There are many different formulations for Turing machines. Different formulations are used in different situations, since some results are easier to prove or describe in one formulation than another. The various formulations are essentially equivalent. We shall use a formulation that is convenient in relating automata to formal language theory [12]. In this formulation, the tape has a fixed left end, but extends indefinitely to the right. At the beginning of the computation, the tape is positioned at the leftmost cell and the initial symbols are on the tape from left to right, followed by blank cells. The symbol ƀ indicates a blank. In this formulation, a Turing machine serves as a recognizer, accepting or rejecting the initial string. Formally, a

Turing machine is a 6-tuple {S,A,N,T,I,F} where S is a set of states, I is the initial state, and F is a set of final states. The machine halts if it enters a final state. A is the alphabet, i.e., the set of symbols that may appear on the tape. N, a subset of A, is the set of input symbols (those that may appear on the tape initially, when the computation starts). One of these symbols, the blank, has a special status, and cannot be written by the Turing machine. M = A − {ƀ} is the set of symbols that can be written by the Turing machine. T is a mapping, called the transition function, from S × A to S × M × {R,L} where R and L stand for movement of the tape head one cell to the right or left, respectively. That is, given a state and a tape symbol, the transition function produces a new state, writes a symbol from M on the tape, and then moves the tape left or right one cell. The mapping T may be undefined for some arguments. A Turing machine is said to accept an input tape if it halts, i.e., enters a state in F. A Turing machine may fail to accept a string by encountering a situation in which T is undefined, or by "looping". An example of a Turing machine is:

```
S = {S1,S2,S3,S4,S5,S6}
A = {ƀ,A,B,X,Y}
N = {A,B}
 I = S1
F = {S6}

T(S1,A) = (S2,X,R)            T(S4,Y) = (S4,Y,R)
T(S2,A) = (S2,A,R)            T(S4,ƀ) = (S6,Y,R)
T(S2,B) = (S3,Y,L)            T(S5,A) = (S5,A,L)
T(S2,Y) = (S2,Y,R)            T(S5,X) = (S1,X,R)
T(S3,Y) = (S3,Y,L)
T(S3,X) = (S4,X,R)
T(S3,A) = (S5,A,L)            T is undefined otherwise
```

This Turing machine accepts strings of symbols from the language L = $\{A^n B^n \mid n \geqslant 1\}$. A description of how this machine operates is given later.

7.2.1 Simulation of Turing Machines

Consider the problem of writing a program to simulate such Turing machines. To be general, the program should read in the description of the Turing machine, represent that machine internally, and then carry out the computation as specified above.

One problem is choosing the form of the description. A number of details in the formal description can be dispensed with, but the notation can be followed fairly closely without great difficulty. Somewhat more mnemonic choices for the components of the machine may be helpful in preparing data. The form we use here consists of a designator, identifying the component, a separating colon, and then the information whose specific form is indicated by the example that follows. The designators are:

SYMBOLS: A, the alphabet of tape symbols.
STATES: S, a list of states.
INITIAL: I, the initial state.
FINALS: F, a list of final states.
TRANS: T, the transition function.
TAPE: The initial configuration of the tape.

In these descriptions, items of a list are separated by commas. The tape, however, consists simply of a string of characters. This form of representation allows the states to be several characters in length, but limits tape symbols to a single character. This latter restriction is quite fundamental to the program that follows, but is not a practical concern in Turing machines of interest. For the basic program, we assume that a single line is sufficient for all designators except TRANS.

Input data describing the Turing machine above is:

```
SYMBOLS: ,A,B,X,Y
STATES:S1,S2,S3,S4,S5,S6
INITIAL:S1
FINALS:S6
TRANS:(S1,A)=(S2,X,R),(S2,A)=(S2,A,R),(S2,Y)=(S2,Y,R),(S2,B)=(S3,Y,L)
TRANS:(S3,Y)=(S3,Y,L),(S3,X)=(S4,X,R),(S3,A)=(S5,A,L),(S5,A)=(S5,A,L)
TRANS:(S5,X)=(S1,X,R),(S4,Y)=(S4,Y,R),(S4, )=(S6,Y,R)
TAPE:AABB
```

The next problem is how to represent this data internally. The names of states and the tape symbols may be arbitrary. The transition function, however, can be naturally thought of as a two-dimensional array whose size is the product of the number of states and the number of tape symbols. To deal with the data uniformly, a relationship will be established between the input symbols and numeric indices. For example, the five tape symbols are associated with the integers 1 through 5 (X corresponds to 4, for example), and the states S1 through S6 are associated with the integers 1 through 6, respectively, and so on. Tables are used to provide these associations. A function INDEX to process a list and create the association table is:

```
        DEFINE('INDEX(T,LIST)SYMBOL')
        BC    =    BREAK(',');    L1    =    LEN(1)
        SYM    =    BC . SYMBOL L1 | (L1 REM) . SYMBOL
                    .
                    .
                    .
INDEX   INDEX    =    INDEX + 1
        LIST    SYM    =                             :F(RETURN)
        T<SYMBOL>    =    INDEX                      :(INDEX)
```

Note that INDEX returns as value the number of items in the list. The use of this function will be illustrated presently.

The three things to be done at each transition are naturally characterized by a defined data type:

```
      DATA('TRIPLE(STATE,SYMBOL,MOVE)')
```

Such objects will be the values of the array associated with the transition function.

The initial portion of the program that reads the description of the Turing machine and constructs the corresponding data is:

```
       &TRIM   =   1
       L1   =   LEN(1)
       CONPAT   =   BREAK(':') . TYPE L1 REM . INFO
       ~~NEXT    =    L1 . SYMBOL~~
       BR   =   BREAK(')')
       TRANSER  =  '(' BC . S1 L1 BR . S2 ')=(' BC . S3 L1
+                      BC . S4 L1 BR . S5 L1 (','| RPOS(0))
       S   =   TABLE();   F   =   TABLE();   A   =   TABLE()
       R   =   1
       L   =   -1
                  .
                  .
                  .
INIT   CONTROL   =   INPUT                        :F(START)
       OUTPUT  =  CONTROL
       CONTROL   CONPAT                          :F(ERROR)S($TYPE)
FINALS INDEX(F,INFO)                              :(INIT)
INITIAL  STATE   =   INFO                         :(INIT)
STATES SSIZE   =   INDEX(S,INFO)
       TRANS   =   ARRAY(SSIZE ',' TSIZE)         :(INIT)
TAPE   TAPE   =   INFO '     '                    :(INIT)
TRANS  INFO   TRANSER   =                         :F(INIT)
       TRANS<S<S1>,A<S2>>   =   TRIPLE(S3,S4,$S5)  :(TRANS)
SYMBOLS  TSIZE   =   INDEX(A,INFO)                :(INIT)
                  .
                  .
                  .
```

Each designator serves as a key to direct the program to a corresponding label where the required processing is performed. The designators SYMBOLS, STATES, and FINALS fill in association tables using INDEX. STATES also creates an array TRANS. The designator TRANS fills in this array, subscripting entries with integers obtained from association tables. Each value is a TRIPLE. The directions of movement, R and L, are translated into integers 1 and -1 respectively. The reason for not translating the state and symbol into corresponding integers is explained later. TAPE simply creates a string for later use. A few blanks are appended; more blanks are added later, if they are needed, to simulate an "infinite" tape. Note that this formulation places some constraints on the order in which the designators can appear. SYMBOLS must appear before STATES, STATES before TRANS, and so on. This is a minor inconvenience to the user and can be remedied without substantial changes to the program.

Either the designator START or an indication of end-of-information causes transfer to the label START at the beginning of the simulation portion of the program. The actual functioning of the Turing machine is based on processing the string TAPE, maintaining an integer POSITION corresponding to the position of the cell currently being examined. The simulation program is:

```
                   .
                   .
                   .
        LOCSYM   =   TAB(*POSITION) L1 . SYMBOL
        PUTSYM   =   TAB(*POSITION) . H L1
        POSITION   =   0
                   .
                   .
                   .
START   TAPE   LOCSYM                        :F(OVER)
        TRIPLE   =   TRANS<S<STATE>,A<SYMBOL>>
        IDENT(TRIPLE)                        :S(NO)
        TAPE   PUTSYM   =   H SYMBOL(TRIPLE)
        STATE   =   STATE(TRIPLE)
        POSITION   =   POSITION + MOVE(TRIPLE)
        DIFFER(F<STATE>)                     :F(START)
YES     OUTPUT   =   'ACCEPTED'              :(END)
NO      OUTPUT   =   'REJECTED'              :(END)
OVER    TAPE   =   TAPE '     '              :(START)
                   .
                   .
                   .
```

The current symbol is determined using LOCSYM. Should the tape be exhausted, more blanks are added at OVER. An entry is obtained from TRANS using indices obtained from association tables. If the entry is null, the transition is undefined and the Turing machine halts, noting that the tape is rejected. Otherwise, a new symbol is written, a new state is obtained, and the position is updated. Note that the symbol itself must be available, which explains why its index is not placed in the triple. Next a test for acceptance is performed. If the index of the current state is in the association table F, the state is a final state, the Turing machine halts, and acceptance is noted. Otherwise, the simulation continues.

In many cases, the course of the Turing machine operation is more interesting than the final result (if any). Therefore, a function SNAPSHOT can be added:

```
        DEFINE('SNAPSHOT()')
                  .
                  .
                  .
SNAPSHOT OUTPUT  =   DUPL(' ',POSITION) '* (' STATE ')'
         OUTPUT  =   TAPE                        :(RETURN)
```

This function prints the tape, a marker indicating the current position, and the current state. Note that in order to be able to print the state, the state is maintained in the triples, rather than its index. Calls to SNAPSHOT can be placed at START and before YES. Output resulting from the previously listed input is:

```
* (S1)
AABB

 * (S2)
XABB

  * (S2)
XABB

 * (S3)
XAYB

* (S5)
XAYB

 * (S1)
XAYB

  * (S2)
XXYB

   * (S2)
XXYB

  * (S3)
XXYY

 * (S3)
XXYY

  * (S4)
XXYY
```

```
   * (S4)
XXYY

    * (S4)
XXYY

     * (S6)
XXYYY
ACCEPTED
```

This output helps in understanding the structure of this particular Turing machine. Intuitively, As and Bs are alternatively replaced by Xs and Ys. Starting in state S1, an A is replaced by an X, the head moves to the right, and the new state is S2. Note that if the first symbol on the tape is a B, T is undefined, and the tape is rejected immediately. As long as As are encountered, the head moves to the right without changing state or the symbols on the tape. If a B is encountered, it is changed into a Y, the head moves to the left, and the state becomes S3. In state S3, Ys cause the head to move to the left but the state remains the same and the tape is unchanged. If an X is encountered, there are no more As to convert, and the state becomes S4. If an A is encountered, the state becomes S5 instead. In S4, the head moves right. If a blank is encountered before a B, then no Bs remain. The state becomes S6, the lone final (accepting) state. In S5, if an X is encountered, the head has passed the leftmost A and moves right to convert that A into an X. S1 (the initial state) is entered, and the process begins anew.

As another example, consider a tape that initially contains AABBB:

```
* (S1)
AABBB

 * (S2)
XABBB

  * (S2)
XABBB

 * (S3)
XAYBB

* (S5)
XAYBB

 * (S1)
XAYBB
```

```
  * (S2)
XXYBB

   * (S2)
XXYBB

  * (S3)
XXYYB

 * (S3)
XXYYB

  * (S4)
XXYYB

   * (S4)
XXYYB

    * (S4)
XXYYB
REJECTED
```

The initial portion of the tape contains an acceptable string and it is processed as before. Then, in state S4, the Turing machine encounters an unacceptable symbol, B, for which no transition is defined. The entire string is rejected.

EXERCISES

7.22 Modify the program above to permit designators to be given in any order.

7.23 Provide a means whereby any designator may be given on as many lines as are needed.

7.24 In the formulation of the Turing machine above, the tape is fixed at the left, yet it is clearly possible to define a machine that will attempt to go off the left end of the tape.
(a) How should such a situation be interpreted?
(b) How can this situation be handled in the program?
(c) How can the program be modified to handle a formulation that allows the tape to extend indefinitely in both directions?

7.25 Generalize the Turing machine program to handle tape symbols requiring more than one character.

7.26 It is easy to define a Turing machine that does not halt (although it is not possible, in general, to detect this property). Provide a means for limiting output.

7.27 Generalize the Turing machine program to permit simulation of several different machines in a single run.

7.28 Use an array instead of a string for representing the tape. Rewrite the program as necessary. Compare the two kinds of representation.

7.29 Construct a Turing machine to recognize <EXP>s as described in Section 7.1.

7.30 Construct a Turing machine to add two binary numbers, leaving the answer on the tape.

7.31 Among the many alternative formulations of Turing machines, there are multi-tape machines in which a tape consists of a number of rows of cells that are processed in parallel by the tape head. Discuss how to simulate such machines.

7.32 A finite state machine is similar to, but simpler than, a Turing machine. A finite state machine has no tape, but receives a series of symbols as input. Depending on the input symbol and the state of the machine, a symbol is output and transition is made to a new state. Write a program to simulate finite state machines.

7.3. A MACRO PROCESSOR

Macros are most familiar as an assembly-language facility that permits an operation name to be associated with a number of lines of code. This macro name constitutes an abbreviation for those lines of code, and when the name is used, it is replaced by ("expanded into") the corresponding lines of code. Most assemblers provide rather elaborate macro facilities. These facilities typically include the ability to associate arguments with a macro, so that values for the arguments may be given when the macro name is used, and are substituted at designated places in the expanded code. Various kinds of computational capabilities are common also.

Macro facilities, similar in principle to those in assembly language, but often differing substantially in their application, are used in a number of other contexts. The abbreviation and number-generation mechanisms described in Sections 6.3.2 and 6.3.3 are weak forms of macro facilities. There are also entire macro languages which consist of features that permit definition, expansion, substitution, and related facilities [43–46]. Macro languages form a class of string processors that operate at a fairly primitive level and depend on extension, through definition, to perform complicated operations. Such processors are inherently recursive, generally lacking control structures that permit the usual methods for iteration and transfer of control. A program written in a macro language typically consists of a large amount of text that has no syntactic significance to the language processor, and which remains unmodified. Certain constructions have syntactic significance to the language processor and are replaced by other text, for example, as the result of macro expansion. The abbreviation and number-generation mechanisms in Chapter 6 can be viewed in this way, ignoring other aspects of the document-formatting program.

This section describes a simple macro language, MP, similar to a number of other macro languages, especially M6 [46]. A program to implement this language follows. The purpose of developing this program is not to provide a working macro language (SNOBOL4 itself is a much more powerful string-processing language), but rather to illustrate some problems in program organization, particularly with respect to recursive processes.

7.3.1. Description of MP

Input to MP consists of a string of characters. We will ignore problems of line boundaries for the moment and consider the input to be a continuous stream of characters. Most characters have no significance to MP. Such characters are passed from input to output without modification. A few characters do have significance to MP and cause various actions, including the insertion of text in the character stream. Conceptually, MP reads the input stream character by character. This process may be visualized as shown in Figure 7.5.

$$\begin{array}{ccc} \text{output} & & \text{input} \\ \cdots C_{j-2}\ C_{j-1} & \underset{\text{cursor}}{\uparrow} & C_j\ C_{j+1}\ C_{j+2} \cdots \end{array}$$

Figure 7.5 MP Text Stream

The cursor can only move to the right. Text may be consumed or generated at the cursor, but nowhere else. Text may be inserted to the left of the cursor, and hence output, or to the right, and hence reprocessed by MP.

The syntactic construction of greatest importance in MP is the *macro call*. This construction is signaled by the character * which marks the beginning of a macro call. Following the *, there are a number of *arguments*, separated by commas. Arguments are numbered starting at zero. The zeroth argument is the macro name. A macro call is terminated by the symbol :. A typical macro call is:

```
*SUM,3,8:
```

SUM is the macro name. Associated with this name there is a (built-in) procedure that adds the first and second arguments. The *replacement text* of a macro call is the value computed by the associated procedure (in this example, 11). All values are handled as strings, even if they are numeric in nature. The replacement text is placed to the right of the cursor. In processing the string.

```
 THE RESULT IS *SUM,3,8: IN THIS CASE.
↑↑
```

the cursor advances until the * is encountered, signaling a macro call:

```
THE RESULT IS *SUM,3,8: IN THIS CASE.
              ↑↑
```

The cursor is advanced as arguments are accumulated until the colon is reached. The result of the macro call replaces the call itself.

```
THE RESULT IS 11 IN THIS CASE.
              ↑↑
```

Note that the cursor is positioned at the beginning of the replacement text. Processing continues. There are no more syntactically significant characters, and the entire string is output.

The macro in the example above is built-in, i.e., is part of the MP system. There are several built-in macros for performing various computations and operations. One of the most important built-in macros is DEF, which defines other macros. DEF is called as follows:

*DEF,*name*,*repl*:

where *name* is the name of the macro being defined and *repl* is its replacement text. An example is:

```
*DEF,COMPUTER,IBM 370:
```

Following this definition, a call of COMPUTER is replaced by IBM 370. An example is:

```
WE ARE USING THEIR *COMPUTER:.
```

which becomes

```
WE ARE USING THEIR IBM 370.
```

Note the similarity of this use of macros to the abbreviation facility described in Section 6.3.2.

A defined macro may have arguments just as the built-in macro SUM has arguments. Arguments are indicated in the replacement text by *parameter markers* which consist of a # followed by an integer indicating the argument number. There may be as many as ten arguments, numbered 0 through 9. An example is:

```
*DEF,ADD,#1+#2:
```

Arguments supplied in a macro call replace the corresponding parameter markers. Thus, the call

```
*ADD,A,B:
```

is replaced by A+B.

Macro calls may be nested. That is, a macro call may appear in the argument of another macro call. An example is

```
*ADD,A,C*SUM,3,2::
```

which produces A+C5.

Since some characters have syntactic significance to MP, means are provided for "protecting" characters to prevent them from being interpreted in the usual way. This is accomplished by using "quotation marks", which are indicated by paired angular brackets. Text within quotation marks is processed without regard to the meaning that the characters within it would otherwise have. The outer quotation marks are removed during processing. An example is

```
THE SYMBOL INDICATING A CALL IS <*>.
```

which produces

```
THE SYMBOL INDICATING A CALL IS *.
```

Quotation marks (if balanced) may be included within other quotation marks. When a quoted string is processed, only the outer quotation marks are removed. Consider the problem of defining a macro MPY similar to ADD, except generating the symbol *. A first attempt might be

```
*DEF,MPY,#1*#2:
```

However, the * indicates a macro call (during the processing of DEF), which is clearly erroneous. A second attempt is

```
*DEF,MPY,#1<*>#2:
```

which gives MPY the replacement text

```
#1*#2
```

Thus,

```
*MPY,A,B:
```

becomes

```
A*B
```

Since this text is placed to the right of the cursor and reprocessed, the * indicates another macro call (presumably of B). A correct definition is

```
*DEF,MPY,#1<<*>>#2:
```

(This example indicates, among other things, why macro processors are difficult to use.)

This description of MP is only an introduction. There are a number of other features which are introduced later. The description is not particularly precise either. Nevertheless, it is sufficient to provide a basis for program design.

7.3.2. The Implementation of MP

The heart of the implementation is a function EXPAND that performs the expansion operations of MP. Since macro calls can be recursive, it should be expected that EXPAND will be used recursively. One of the first problems that becomes evident is that the interpretation of potentially significant characters depends on context. Thus, * signals a macro call unless it occurs within quotation marks. The characters , and : are significant only during the processing of a macro call, and so on. There are a number of ways of handling this problem. Generality is important to assure proper processing, regardless of the situation that may exist, and flexibility is needed to handle possible extensions to MP. The method used here employs computed transfers which are obtained from an argument provided to EXPAND. This argument is a table of transfers, and its particular value depends on the context in which EXPAND is called. There are three processing modes: the basic processing of the input string, processing of a macro call, and processing of a string enclosed in quotation marks. For each character of potential interest, an appropriate label is specified. The tables are:

```
          CALL   =   '*'
          ASEP   =   ','
          TERM   =   ':'
          OQUOTE   =   '<'
          CQUOTE   =   '>'
          BBR   =   TABLE(5)
          CBR   =   TABLE(5)
          QBR   =   TABLE(5)
          BBR<CALL>   =   'CALL'
          BBR<OQUOTE>   =   'QUOTE'
          BBR<ASEP>   =   'NOOP'
          BBR<TERM>   =   'NOOP'
          BBR<CQUOTE>   =   'NOOP'
          CBR<CALL>   =   'CALL'
          CBR<OQUOTE>   =   'QUOTE'
          CBR<ASEP>   =   'CRET'
          CBR<TERM>   =   'CRET'
          CBR<CQUOTE>   =   'NOOP'
          QBR<OQUOTE>   =   'QINCR'
          QBR<CQUOTE>   =   'QDECR'
          QBR<CALL>   =   'NOOP'
          QBR<ASEP>   =   'NOOP'
          QBR<TERM>   =   'NOOP'
```

The basic section of EXPAND is

```
        DEFINE('EXPAND(BR)H')
        STREAM   =   BREAK(CALL ASEP TERM OQUOTE CQUOTE) . H
+                    LEN(1) . C
                 .
                 .
                 .
EXPAND STRING    STREAM   =                        :F(EXPR)S($BR<C>)
                 .
                 .
                 .
NOOP    EXPAND   =   EXPAND H C                    :(EXPAND)
                 .
                 .
                 .
EXPR    EXPAND   =   EXPAND STRING
        ~~C      =~~
        STRING   =                                 :(RETURN)
```

STRING is a global variable. The statement labeled NOOP handles characters which have no significance in the particular context in which EXPAND is operating. Thus, if EXPAND is called in the basic processing mode, BBR is the transfer table. Commas, colons, and right brackets cause transfer to NOOP. NOOP appends the initial part of the string and the character to the developing value of EXPAND. The use of EXPAND is illustrated by statements that read lines and expand them:

```
MP      STRING   =   INPUT                         :F(DONE)
        OUTPUT   =   EXPAND(BBR)                   :(MP)
                 .
                 .
                 .
```

Another type of processing occurs when a left bracket is encountered in the basic processing mode. Control is transferred to QUOTE. The related statements are:

```
                 .
                 .
                 .
QUOTE   EXPAND   =   EXPAND H EXPAND(QBR)          :(EXPAND)
QINCR   QCOUNT   =   QCOUNT + 1
QREPL   EXPAND   =   EXPAND H C                    :(EXPAND)
QDECR   QCOUNT   =   GT(QCOUNT,0) QCOUNT - 1       :S(QREPL)
        EXPAND   =   EXPAND H                      :(RETURN)
                 .
                 .
```

QUOTE causes a recursive call to EXPAND, but with a different transfer table. This table causes quotation marks to be treated differently from the way they are treated in the basic mode, and other characters are ignored. Note that quotation marks are "balanced" by counting. This is similar to the method used by SNOBOL4 when matching balanced strings.

Before going on to call processing, the method of handling definitions must be developed. Built-in macros presumably can be implemented using defined SNOBOL4 functions or even built-in SNOBOL4 functions. All defined macros, however, are really variations on one process: the substitution of arguments for parameter markers in the replacement text of the macro definition. To give generality to the implementation, it is desirable to have all macros handled in the same way. The process consists of two parts: evaluating the arguments and executing the appropriate procedure. A function EXEC is used for actually processing the call. The statements in EXPAND are

```
                        .
                        .
                        .
CALL    EXPAND    =    EXPAND H
        STRING    =    EXEC() STRING                 :(EXPAND)
CRET    EXPAND    =    EXPAND H                      :(RETURN)
                        .
                        .
                        .
```

Note that the value returned by EXEC is placed at the beginning of STRING and hence is subsequently processed by EXPAND. The initial part of EXEC is

```
        DEFINE('EXEC()I,ARG')
                        .
                        .
                        .
EXEC    ARG    =    ARRAY('0:9')
EXEC1   ARG<I>    =    EXPAND(CBR)                   :F(ERR)
        I    =    DIFFER(C,TERM) I + 1               :S(EXEC1)
                        .
                        .
                        .
```

An array ARG is filled in by calling EXPAND with an appropriate transfer table. When EXPAND returns as a result of a terminating colon, argument evaluation is complete. Thus, ARG<0> is the name of the macro and the other arguments are assigned accordingly. The SUM macro could be implemented by

```
        EXEC    =    ARG<1> + ARG<2>                 :(RETURN)
```

In order to have a general mechanism for associating procedures with macro names, the unevaluated expression facility of SNOBOL4 can be used to advantage. A table, subscripted by macro names, contains expressions to be evaluated to compute the desired values:

```
        FORM    =   TABLE()
                    .
                    .
                    .
        FORM<'SUM'>   =    *(ARG<1> + ARG<2>)
                    .
                    .
                    .
```

All macros can be handled by a single statement in EXEC following the evaluation of the arguments:

```
        EXEC    =    EVAL(FORM<ARG<0>>)            :(RETURN)
```

To incorporate a built-in macro in MP, it is only necessary to provide an appropriate entry in FORM.

Next, consider the macro DEF that defines other macros. Since all defined macros involve the same processing, all defined macros have the same entry in FORM. Another table, DEFN, contains the replacement texts:

```
        DEFINE('DEF()')
        DEFN    =    TABLE()
                    .
                    .
                    .
DEF     FORM<ARG<1>>   =    *SUBST(DEFN<ARG<0>>)
        DEFN<ARG<1>>   =    ARG<2>                 :(RETURN)
```

Note that DEF allows a defined macro to be redefined, or a built-in macro to be replaced by a defined macro. The function SUBST is invoked when a defined macro is called. The procedure is:

```
        DEFINE('SUBST(STRING)H,N,~~P~~')
        PARM    =    BREAK('#') . H LEN(1) SPAN('0123456789') . N
                    .
                    .
                    .
SUBST   STRING   PARM    =                         :F(SUBR)
        SUBST    =    SUBST H ARG<N>               :(SUBST)
SUBR    SUBST    =    SUBST STRING                 :(RETURN)
```

This completes the basic part of MP. Other built-in macros are easily added by making appropriate entries in FORM and by supplying corresponding procedures. Another feature of MP, needed to make the language practical, is more difficult. This is the subject of the next section.

7.3.3. A Conditional Facility in MP

MP, as it stands, provides the capability for computing values, but it provides no method for testing values or selecting alternate courses of action. Such facilities are necessary to perform even the most routine calculation.

One aspect of conditional evaluation is the ability to test and compare values. In MP, this ability is available in the form of comparison macros. An example is:

```
*GT,a1,a2:
```

The value of GT is 1 if a1 is greater than a2 and 0 otherwise. Both a1 and a2 are interpreted as integers. There are other similar macros for the other arithmetic comparisons: LT, LE, GE, EQ, and NE.

Such comparisons do not, in themselves, permit conditional evaluation. The macro IF is the basis of conditional operations. Unlike other macros described so far, IF has an indefinite number of arguments:

```
*IF,a1,a2, ... ,an:
```

The arguments are logically grouped in pairs from the left to right. Odd-numbered arguments are evaluated from left to right until a value of 1 is obtained. The value of the next (even-numbered) argument then becomes the value of the call to IF. The value of 1 can be thought of as corresponding to "success", and the corresponding even-numbered argument as the successful value. If none of the odd-numbered arguments has the value 1, the value of the call of IF is the null string. Once a successful pair has been found, evaluation of subsequent arguments is skipped.

The value returned by IF is conditional upon the value of its arguments. Alternate expansions are possible depending, for example, on the values returned by comparison macros. An example of the use of IF is given by the definition of a macro MAX whose value is the maximum of its two arguments:

```
*DEF,MAX,<*IF,*GT,#1,#2:,<#1>,1,<#2>:>:
```

The macro IF introduces several substantial problems in the implementation of MP. In the first place, the number of arguments in a call of IF is not fixed. More important, all the arguments cannot be pre-evaluated as has been done before. Only if an odd-numbered argument has the value 1 is the corresponding even-numbered argument evaluated. Furthermore, once a successful pair has been located, the remaining arguments are skipped without being evaluated. This form of evaluation is quite different from that implemented in the preceding sections. In essence, evaluation of the arguments of IF must be under the control of the procedure for IF. There are several possible approaches. One is to place the evaluation of arguments for all macros under the control of the corresponding procedures. While general in nature, this approach is cumbersome (see the exercises). Another, less general, approach is to distinguish a class of macros (of which IF is a member) as being "special". When EXEC detects a special macro, control can be passed to the procedure before arguments are evaluated. The new procedure for EXEC is:

```
EXEC    ARG    =    ARRAY('0:9')
        ARG<0>    =    EXPAND(~~EXPAND~~, CBR)
        EXEC = SPECIAL (ARG<0>) EVAL(FORM<ARG<0>>)   :S(RETURN)
EXEC1   I    =    DIFFER(C,TERM) I + 1           :F(EXEC2)
        ARG<I>    =    EXPAND(~~EXPAND~~,CBR)        :S(EXEC1)
EXEC2   EXEC    =    EVAL(FORM<ARG<0>>)          :(RETURN)
```

The function SPECIAL initially can be implemented in a trivial manner and later can be generalized if other special macros are added:

```
        DEFINE('SPECIAL(NAME)')
                  .
                  .
                  .
SPECIAL   IDENT(NAME,'IF')                     :S(RETURN)F(FRETURN)
```

The more difficult problem is that of skipping arguments. Skipping requires processing syntactic structures, but not actually executing macro calls that may be encountered. A little cleverness (that is, a "trick") makes this possible without the need for changing the existing program. Note that EVAL is used to compute the value for macro calls. By OPSYNing EVAL to a procedure that does nothing, a macro call can be processed without executing the corresponding procedures. A procedure to skip arguments is therefore:

```
        OPSYN('SPECIALSAVE','SPECIAL')
        OPSYN('EVALSAVE','EVAL')
        DEFINE('SKIP()')
        DEFINE('NULL()')
        DEFINE('FAIL()')
                  .
                  .
                  .
SKIP    OPSYN('EVAL','NULL')
                                OPSYN('SPECIAL','FAIL')
        EXPAND(CBR)
        OPSYN('EVAL','EVALSAVE')                    :(RETURN)
                       OPSYN('SPECIAL','SPECIALSAVE')
NULL                                                :(RETURN)
FAIL               :(FRETURN)
```

To prevent processing of special macros, SPECIAL similarly can be OPSYNed to a procedure that always fails.

Note that a synonym for EVAL is established in the program preamble so that the ordinary meaning of EVAL can be restored after skipping an argument. The procedure for IF now follows naturally:

```
        DEFINE('IF()ARG')
                  .
                  .
                  .
IF      ARG    =    EXPAND(CBR)
        (INTEGER(ARG) EQ(ARG,1))                    :S(IFS)
        (DIFFER(C,TERM) SKIP())                     :S(IF)F(RETURN)
        ~~SKIP()~~                                  ~~:(IF)~~
IFS     IF   =   EXPAND(CBR)
IFR     IDENT(C,TERM)                               :S(RETURN)
        SKIP()                                      :(IFR)
```

EXERCISES

7.33 What is the result of applying MP to the following strings?

```
23
<23>
2<3>
<2<3>>
*SUM,1,*SUM,3,5::
SUM,2,56:
*DEF,SUM,#1+#2:*SUM,4,55:
*DEF,C,0:*IF,*EQ,*C:,0:,*DEF,C,1:,1,*DEF,C,2::*C:
```

7.34 What is the effect of excess arguments in the call of a (nonspecial) macro? Of an insufficient number of arguments?

7.35 Why is STRING a global variable and not an argument of EXPAND?

7.36 Why is it necessary for ~~C~~ ARG to be a local variable of ~~SUBST~~ EXEC? ~~and ARG to be a local variable of EXEC?~~

7.37 Implement a complete set of built-in arithmetic computation and comparison macros.

7.38 Implement a set of built-in macros for alphabetic comparison of strings.

7.39 Implement a built-in macro whose value is the substring of a given string between specified positions.

7.40 Implement a built-in macro that makes one macro equivalent to another.

7.41 An additional facility of MP allows a macro call to be terminated by a semicolon rather than a colon. In this case, the value of the call is placed to the left of the cursor, rather than to the right. Implement this feature.

7.42 Angular brackets can be used to protect most characters that would otherwise have syntactic significance, but angular brackets cannot be used to protect other angular brackets in all cases. Devise a solution to this problem.

7.43 What is the difference between placing text to the left and to the right of the cursor?

7.44 Explain why it is sometimes useful to place the value of a call to the right of the cursor.

7.45 Design and implement a tracing facility for MP.

7.46 Provide the necessary interface to make a series of input lines to MP appear to be a continuous text stream.

7.47 Another special macro is GO, which allows conditional evaluation of the text of a macro. If GO appears in replacement text and the value of its argument is 1, the remainder of the replacement text is ignored. Otherwise GO has no effect on expansion. An example of the use of GO is

```
*DEF,LINE,<THIS IS ALL *GO,#1:SO FAR>:
```

The value of *LINE,0: becomes

```
THIS IS ALL SO FAR
```

but the value of *LINE,1: is

```
THIS IS ALL
```

Implement GO.

7.48 Design a mechanism for MP so that the characters of syntactic significance can be changed during processing. Modify the program accordingly.

7.49 Some characters have syntactic significance in one processing mode and not in others. Modify the program to take advantage of this fact, and hence to avoid the unnecessary processing at NOOP.

7.50 Discuss the implications of having all macro procedures evaluate their arguments.

7.51 Add error checking to the implementation of MP. (Considering the difficult syntax of MP, error checking is an essential component of a workable implementation.) Identify as many distinct errors as possible. Provide diagnostic messages and error recovery.

7.4. A CONTEXT EDITOR

Most computing installations, especially ones offering interactive facilities, provide utility programs for editing files. Some of these editors are quite simple, while others are very powerful. Some editors address data by line number [47], while others refer to data by context (refer to Section 1.1.4).

Editors may require corrections and modifications to be performed sequentially in the order in which the data appears on the file, or corrections may be made at random places at the convenience of the user. Editing requires little in the way of computing resources and editors are typically designed to run efficiently in a small amount of memory.

This section describes an editor written in SNOBOL4. In most situations there is no need for such an editor; it is unnecessarily large and slow, and duplicates facilities of existing programs. This editor does illustrate, however, one of the most powerful features of SNOBOL4—the ability to convert data into executable statements during program execution. Using this capability, a rather powerful editor can be implemented by a very small program (although the editor itself is large because the SNOBOL4 system is resident in memory when the editor is in use).

7.4.1. A Description of ED4.

The editor, called ED4, is a context editor. That is, file positioning and location of data to be modified is specified in terms of the data itself. ED4 implements a command language. Commands cause data to be read from an input file, modified, and written onto an output file. Figure 7.6 illustrates the files involved.

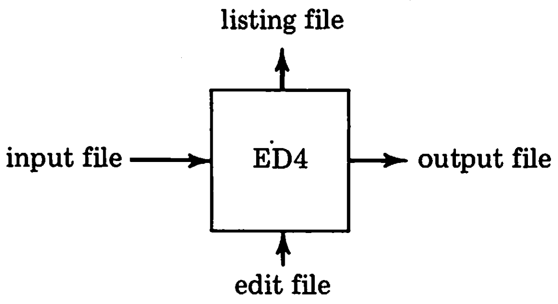


Figure 7.6 Files Used by ED4

Processing of the input file is sequential. Lines are read from input to output, with corrections, deletions, and insertions specified in the edit file. In interactive use, the edit file is entered from a terminal by the person performing the edit. One line of data is kept within ED4. This is the current line under consideration. An edit command can modify the current line or cause lines to be read from input to output until a desired line is found. The desired line thus becomes the current line.

Commands in the edit file are distinguished by a $ at the beginning of a line. All other lines in the edit file are interpreted as data, and are inserted in the output file when they are encountered. A letter following the $ of a command specifies the action to be performed. The basic commands are:

$C	Print the current line.
$D	Delete up to a line containing a specified pattern.
$E	Copy to the end of the input file.
$F	Find a line with a specified pattern.
$L	Print the last line output.
$R	Replace the string matching a specified pattern.

The distinguishing characteristic of ED4 is the use of SNOBOL4 syntax for specifications. (The programmer who is familiar with SNOBOL4 finds ED4 easy to learn.) An example is

```
$F    'SUMMARY'
```

which copies from the input file to the output file until a line containing the string SUMMARY is found. This line becomes the current line, internal to ED4.

Lines are modified by the R command, which operates on the current line. Thus,

```
$R    'SUMMARY'   =   'CAPITULATION'
```

modifies the current line as indicated.

The D command is similar to the F command, except that lines from the input file are discarded until the desired pattern is found.

Patterns need not be literals. Any SNOBOL4 pattern can be used. An example is

```
$F    'LOOP' RPOS(0)
```

which finds a line that ends with LOOP.

Other commands in the spirit of those above are easy to devise.

7.4.2. The Implementation of ED4

Without the availability of SNOBOL4, such an editing language would be unthinkable (and, in fact, would not be thought of in the first place). Few editors approach the power of the pattern-matching and replacement facilities of SNOBOL4. Implementing such an editor would be a monumental task if it were not possible to access the facilities of SNOBOL4 from data. The ability to convert strings to executable statements, using the function CODE, makes implementation essentially trivial, if a bit obscure.

The basic part of the program reads the edit file (EFILE). Lines that are not commands are written on the edit output file (OFILE). When a command is encountered, control is transferred to the corresponding label. The C, E, and L command processors are trivial and are included to illustrate the control structure of the program.

```
       &TRIM   =   1
       INPUT('IFILE',10)
       OUTPUT('OFILE',20)
       INPUT('EFILE',30)
       CTYPE   =   POS(0) '$' LEN(1) . COM REM . TEMPLATE
                  .
                  .
                  .
       CURRENT  =   IFILE                      :F(EOF)
EDIT   LINE   =   EFILE                        :F(E)
       LINE   CTYPE                            :F(INSERT)S($COM)
INSERT OFILE   =   LINE                        :(EDIT)
C      OUTPUT   =   CURRENT                    :(EDIT)
E      OFILE   =   CURRENT
E1     OFILE   =   IFILE                       :S(E1)F(EOF)
L      OUTPUT   =   OFILE                      :(EDIT)
EOF    OUTPUT   =   '*** END OF FILE ***'      :(END)
                  .
                  .
                  .
```

The D, F, and R commands construct strings that correspond to the statements to be executed in order to perform the desired operations. The pattern or replacement is taken from the command line. The rest of the statement is supplied from appropriate constants. The resulting string is converted to CODE and control is transferred to the new statement by a direct goto. The statements that implement these commands are:

```
       SRC   =   ' CURRENT '
       DGO   =   ' :F(D1)S(EDIT)'
       FGO   =   ' :F(F1)S(EDIT)'
       RGO   =   ' :F(NOMAT)S(R1)'
                  .
                  .
                  .
D      EXEC   =   CODE(SRC TEMPLATE DGO)       :F(CERR)S<EXEC>
D1     CURRENT   =   IFILE                     :S<EXEC>F(EOF)
F      EXEC   =   CODE(SRC TEMPLATE FGO)       :F(CERR)S<EXEC>
F1     OFILE   =   CURRENT
       CURRENT   =   IFILE                     :S<EXEC>F(EOF)
R      EXEC   =   CODE(SRC TEMPLATE RGO)       :F(CERR)S<EXEC>
R1     OUTPUT   =   CURRENT                    :(EDIT)
                  .
                  .
                  .
CERR   OUTPUT   =   '*** ERRONEOUS EDIT ***'   :(EDIT)
NOMAT  OUTPUT   =   '*** NO MATCH ***'         :(EDIT)
```

As an example, consider the command

```
$R 'SUMMARY' = 'CAPITULATION'
```

The string constructed is

```
 CURRENT  'SUMMARY' = 'CAPITULATION' :(NOMAT)S(R1)
```

Executing the corresponding statement causes the desired replacement or transfer to a statement that prints an error message. Control returns to the main read loop at EDIT. The D and F commands are similar in nature.

The edit file can be thought of as a series of statements in an editing language. This example illustrates how easily the syntax of SNOBOL4 can be embedded in a language implemented in SNOBOL4. Once this is done, any facility of SNOBOL4 is available in the new language. Since SNOBOL4 is a general-purpose language with a wide range of features, this "window" to the underlying implementation language makes available a wide range of features with little implementation effort.

EXERCISES

7.52 Design and implement edit commands to perform the following operations:
(a) Replace *all* occurrences of a pattern by a string in the current line.
(b) Omit a line containing a specified pattern.
(c) Prefix a string to the beginning of the current line.
(d) Append a string to the end of the current line.
(e) Split the current line into two lines at a specified place.
(f) Append the next input line to the current line.
(g) Change the character that signals a command line.
(h) Execute an arbitrary SNOBOL4 statement.

7.53 Modify ED4 so that the edit output file can be made into the edit input file, thus permitting successive edits of the same text.

7.54 Provide error checking and diagnostics. (This is particularly important for interactive use.)

7.55 Modify the implementation of ED4 to improve its interface with the user in the interactive mode.

7.56 In the batch mode, it is often convenient for the user to be able to work from line numbers that identify the lines in the file by position. Provide facilities to
(a) Produce a line-numbered listing of OFILE as a byproduct of an edit run.

(b) Copy to a specified line number.
(c) Skip to a specified line number.
(d) Permit line number qualifications for the R command and the commands described in Exercise 7.52(a)-(f).

7.57 Design and implement a macro language that utilizes the ability of SNOBOL4 to access SNOBOL4 facilities from constructions that appear in data.

APPENDIX

A CHARACTER SETS

The following tables give the values of &ALPHABET for three frequently used character sets. The position column gives the position of the character in &ALPHABET, measured from zero. TAB can be used with the position value given to reach a desired character. The code column refers to the internal machine representation of the character. The code is given in octal for CDC Display Code and ASCII, and in hexadecimal for EBCDIC. The graphics given are those generally used. Not all printers provide the graphics given in the tables. In some cases there may be different graphics from those listed. Some may not be available, or there may be graphics in addition to those shown. The "standard blank" is in position 44 for Display Code, 32 for ASCII, and 64 for EBCDIC.

Table I Display Code

position	*code*	*graphic*	*position*	*code*	*graphic*	*position*	*code*	*graphic*
0	01	A	22	27	W	43	54	=
1	02	B	23	30	X	44	55	
2	03	C	24	31	Y	45	56	,
3	04	D	25	32	Z	46	57	.
4	05	E	26	33	0	47	60	≡
5	06	F	27	34	1	48	61	[
6	07	G	28	35	2	49	62	]
7	10	H	29	36	3	50	63	:
8	11	I	30	37	4	51	64	≠
9	12	J	31	40	5	52	65	↱
10	13	K	32	41	6	53	66	∨
11	14	L	33	42	7	54	67	∧
12	15	M	34	43	8	55	70	↑
13	16	N	35	44	9	56	71	↓
14	1~~6~~7	O	36	45	+	57	72	<
15	20	P	37	46	−	58	73	>
16	21	Q	38	47	*	59	74	≤
17	22	R	39	50	/	60	75	≥
18	23	S	40	51	(	61	76	¬
19	24	T	41	52	)	62	77	;
20	25	U	42	53	$	63	00	
21	26	V						

Table II ASCII

position	*code*	*graphic*	*position*	*code*	*graphic*	*position*	*code*	*graphic*
0	000		43	053	+	86	126	V
1	001		44	054	,	87	127	W
2	002		45	055	–	88	130	X
3	003		46	056	.	89	131	Y
4	004		47	057	/	90	132	Z
5	005		48	060	0	91	133	[
6	006		49	061	1	92	134	\
7	007		50	062	2	93	135	]
8	010		51	063	3	94	136	^
9	011		52	064	4	95	137	_
10	012		53	065	5	96	140	`
11	013		54	066	6	97	141	a
12	014		55	067	7	98	142	b
13	015		56	070	8	99	143	c
14	016		57	071	9	100	144	d
15	017		58	072	:	101	145	e
16	020		59	073	;	102	146	f
17	021		60	074	<	103	147	g
18	022		61	075	=	104	150	h
19	023		62	076	>	105	151	i
20	024		63	077	?	106	152	j
21	025		64	100	@	107	153	k
22	026		65	101	A	108	154	l
23	027		66	102	B	109	155	m
24	030		67	103	C	110	156	n
25	031		68	104	D	111	157	o
26	032		69	105	E	112	160	p
27	033		70	106	F	113	161	q
28	034		71	107	G	114	162	r
29	035		72	110	H	115	163	s
30	036		73	111	I	116	164	t
31	037		74	112	J	117	165	u
32	040		75	113	K	118	166	v
33	041	!	76	114	L	119	167	w
34	042	“	77	115	M	120	170	x
35	043	#	78	116	N	121	171	y
36	044	$	79	117	O	122	172	z
37	045	%	80	120	P	123	173	{
38	046	&	81	121	Q	124	174	|
39	047	‘	82	122	R	125	175	}
40	050	(	83	123	S	126	176	~
41	051	)	84	124	T	127	177	
42	052	*	85	125	U			

Table III EBCDIC

position	*code*	*graphic*	*position*	*code*	*graphic*	*position*	*code*	*graphic*
0	00		43	2B		86	56	
1	01		44	2C		87	57	
2	02		45	2D		88	58	
3	03		46	2E		89	59	
4	04		47	2F		90	5A	!
5	05		48	30		91	5B	$
6	06		49	31		92	5C	*
7	07		50	32		93	5D	)
8	08		51	33		94	5E	;
9	09		52	34		95	5F	¬
10	0A		53	35		96	60	-
11	0B		54	36		97	61	/
12	0C		55	37		98	62	
13	0D		56	38		99	63	
14	0E		57	39		100	64	
15	0F		58	3A		101	65	
16	10		59	3B		102	66	
17	11		60	3C		103	67	
18	12		61	3D		104	68	
19	13		62	3E		105	69	
20	14		63	3F		106	6A	
21	15		64	40		107	6B	,
22	16		65	41		108	6C	%
23	17		66	42		109	6D	_
24	18		67	43		110	6E	>
25	19		68	44		111	6F	?
26	1A		69	45		112	70	
27	1B		70	46		113	71	
28	1C		71	47		114	72	
29	1D		72	48		115	73	
30	1E		73	49		116	74	
31	1F		74	4A		117	75	
32	20		75	4B	.	118	76	
33	21		76	4C	<	119	77	
34	22		77	4D	(	120	78	
35	23		78	4E	+	121	79	
36	24		79	4F	|	122	7A	:
37	25		80	50	&	123	7B	#
38	26		81	51		124	7C	@
39	27		82	52		125	7D	'
40	28		83	53		126	7E	=
41	29		84	54		127	7F	"
42	2A		85	55		128	80	

Table III (continued)

position	*code*	*graphic*	*position*	*code*	*graphic*	*position*	*code*	*graphic*
129	81	a	172	AC		215	D7	P
130	82	b	173	AD		216	D8	Q
131	83	c	174	AE		217	D9	R
132	84	d	175	AF		218	DA	
133	85	e	176	B0		219	DB	
134	86	f	177	B1		220	DC	
135	87	g	178	B2		221	DD	
136	88	h	179	B3		222	DE	
137	89	i	180	B4		223	DF	
138	8A		181	B5		224	E0	
139	8B		182	B6		225	E1	
140	8C		183	B7		226	E2	S
141	8D		184	B8		227	E3	T
142	8E		185	B9		228	E4	U
143	8F		186	BA		229	E5	V
144	90		187	BB		230	E6	W
145	91	j	188	BC		231	E7	X
146	92	k	189	BD		232	E8	Y
147	93	l	190	BE		233	E9	Z
148	94	m	191	BF		234	EA	
149	95	n	192	C0		235	EB	
150	96	o	193	C1	A	236	EC	
151	97	p	194	C2	B	237	ED	
152	98	q	195	C3	C	238	EE	
153	99	r	196	C4	D	239	EF	
154	9A		197	C5	E	240	F0	0
155	9B		198	C6	F	241	F1	1
156	9C		199	C7	G	242	F2	2
157	9D		200	C8	H	243	F3	3
158	9E		201	C9	I	244	F4	4
159	9F		202	CA		245	F5	5
160	A0		203	CB		246	F6	6
161	A1		204	CC		247	F7	7
162	A2	s	205	CD		248	F8	8
163	A3	t	206	CE		249	F9	9
164	A4	u	207	CF		250	FA	
165	A5	v	208	D0		251	FB	
166	A6	w	209	D1	J	252	FC	
167	A7	x	210	D2	K	253	FD	
168	A8	y	211	D3	L	254	FE	
169	A9	z	212	D4	M	255	FF	
170	AA		213	D5	N			
171	AB		214	D6	O			

APPENDIX

B SOLUTIONS TO SELECTED EXERCISES

1.1 Compound words contain internal hyphens, which are usually represented by dashes in machine-readable text. Dashes have other uses: as punctuation, in arithmetic expressions (which appear in certain kinds of text), and to break words at the ends of lines. If the dash is added to LETTERS as a character acceptable in words, some "nonwords" may be matched also. One approach is to perform an examination of WORD after NEXTW has matched to assure that the value obtained is, indeed, a word. There are some contexts in which compound words cannot be distinguished from other constructions by syntax alone. An example is:

```
TO GET THE RESULT, EVALUATE TOTAL-MEAN.
```

A compound word hyphenated at the end of a line is also ambiguous. This apparently simple problem indicates only one of many practical difficulties in analyzing text.

1.2 In matching words by exclusion, the typical pattern used has the form

```
        NEXTW   =   SPAN(SEP) BREAK(SEP) . WORD
```

where SEP is a string of all the characters that cannot appear in a word. This approach has two problems. In the first place, there must be a separator on each side of the word. SPAN does not match the null string, and BREAK must find a character in its argument. If the BREAK and SPAN portions of the pattern are reversed, separators only need appear at the ends of words. Then, however, the null string is matched as a word if the string being examined begins with a separator. These problems cause practical difficulties that must be handled by special cases. A second problem is that the string SEP is awkward to keyboard. Many special characters are likely to appear in text. If one of them is overlooked in preparing SEP, it is automatically accepted as a word or part of a word.

1.3

```
        DIGIT    =   ANY('0123456789')
        TRIPLET   =   DIGIT DIGIT DIGIT
        HEAD   =   TRIPLET | DIGIT DIGIT | DIGIT
        GROUP   =   ',' TRIPLET
        INTEGER   =   POS(0) HEAD ARBNO(GROUP) RPOS(0)
```

An alternative formulation uses an unevaluated expression rather than ARBNO:

```
        GROUPS   =   GROUP *GROUPS | NULL
        INTEGER   =   POS(0) HEAD GROUPS RPOS(0)
```

These patterns are anchored at the beginning and end of the subject string. In other contexts, they might be bracketed differently. Some context is needed to assure the entire integer is matched. The pattern using ARBNO essentially tries to match the shortest possible string, while the pattern using unevaluated expressions tries to match the longest possible string.

1.5

```
        HOLL   =   SPAN('0123456789') $ N 'H' LEN(*N) . LIT
```

Depending on use, HOLL may be embedded in another pattern that supplies the context in which a Hollerith literal is sought. The pattern given above does not handle blanks that may be embedded in the integer specification.

1.6

```
       ANBNCN = POS(0) SPAN('A') @A SPAN('B') @B SPAN('C') @C
+                RPOS(0) *(EQ(B - A,A) EQ(C - B,A))
```

This pattern requires the fullscan matching mode. See the discussion in Section 1.2. Strings of the form $A^n B^n C^n$ compose a context-sensitive language—one that cannot be described by BNF. See Section 1.2.

1.8 One of the awkward aspects of BNF is the necessity for defining character classes as the explicit alternation of individual characters. Thus, a nonterminal describing any single character must be written in the form

```
<char>::=a|b|c| ...|x|y|z|A|B|C| ...|X|Y|Z|1|2|3|...
```

The precise definition depends on what the alphabet of characters is. That has not been defined explicitly. For processing by a program, the alphabet presumably is given by &ALPHABET. A subset of &ALPHABET might be used for some purposes. Given this definition, strings of two characters are defined by

```
<pair>::=<char><char>
```

Contrast this definition with the SNOBOL4 pattern `LEN(2)`.

1.9

```
<null>::=
<even>::=<null>|<pair>|<pair><even>
```

1.10 Quoted literals present the same problem in defining character classes as that discussed in the solution of Exercise 1.8. Given that `<nonsingle>` and `<nondouble>` have been defined to be all characters that are not single and double quotation marks, respectively, a definition is

```
<quotedliteral>::='<nonsingle>'|"<nondouble>"
```

1.11 Assuming that a nonterminal `<nonparen>` has been written to list all the characters that are neither a left parenthesis nor a right parenthesis, a BNF grammar describing balanced strings is:

```
<bal>::=<nonparen>|()|(<bal>)|<bal><bal>
```

The corresponding SNOBOL4 pattern is

```
        BAL = NOTANY('()') | '()' | '(' *BAL ')' | *BAL *BAL
```

This pattern requires the quickscan mode and operates very inefficiently. The built-in pattern `BAL` does not operate in the convoluted way that this pattern does.

1.13 Converting a BNF grammar to SNOBOL4 patterns is conceptually straightforward. Analyzing the definitions is, itself, a pattern-matching problem. It is important to recognize that the process involves creating strings that have the syntax of SNOBOL4. These strings are subsequently converted to patterns (i.e., "compiled") using `EVAL`. The quickscan mode must be used when matching `PATTERN`.

```
        &TRIM   =   1
        &ANCHOR   =   1
        QUOTE   =   "'"
        DEFPAT   =   '<' BREAK('>') . NAME '>::=' REM . DEF
        ALTPAT = BREAK('|') . ALT LEN(1) | (LEN(1) REM) . ALT
        SUBPAT = '<' @P BREAK('>') . SUB '>' | @P (LEN(1)
+                BREAK('<')) . SUB  | @P (LEN(1) REM) . SUB
        GOAL   =   '<' BREAK('>') . NAME
```

```
NEXTL  LINE    =   INPUT                                :F(ERROR)
       LINE   DEFPAT                                    :F(TEST)
       OUTPUT   =   LINE
       PATTERN   =
NEXTA  DEF    ALTPAT    =                               :F(EOD)
       PATTERN   =   DIFFER(PATTERN) PATTERN ' |'
NEXTS  ALT    SUBPAT    =                               :F(NEXTA)
       PATTERN   =   GT(P,0) PATTERN ' *' SUB      :S(NEXTS)
       PATTERN   =   PATTERN ' ' QUOTE SUB QUOTE :(NEXTS)
EOD    $NAME   =   EVAL(PATTERN)                        :(NEXTL)
TEST   PATTERN   =   POS(0) $NAME RPOS(0)
TESTT  STRING   =   INPUT                               :F(END)
       OUTPUT   =
       OUTPUT   =   STRING
       STRING   PATTERN                                 :F(NO)
       OUTPUT   =   'IS A ' NAME                        :(TESTT)
NO     OUTPUT   =   'IS NOT A ' NAME                    :(TESTT)
END
```

1.16

```
       NEXTSP  =  BREAK(*CS) @L @OUTPUT SPAN(*CS) @M *NEXTSP
       LOCSP  =  POS(0) @L @M *NEXTSP | *GT(M,0)
```

When LOCSP is used in pattern matching, the current value of CS is used. The positions printed correspond to numbering the characters starting at zero. Modifying the solution to produce results that correspond to an origin of one is left as an additional exercise.

1.17

```
     SEPBL  =  POS(0) @L @M NEXTBL | *GT(M,0) TAB(*L) . HEAD
+                 TAB(*M) REM . TAIL
```

1.18 Although it is possible to write patterns to perform complicated operations, it is often difficult to obtain certain results because of the inability to exercise control over the matching algorithm. Writing a pattern to print all the items on a list is an example. The problem is in assuring that all items are printed, regardless of whether or not the list ends in a comma, and at the same time not producing spurious output. A solution follows. PRINTI must be used in the fullscan mode. Attempts to find simpler solutions will point out some of the underlying problems.

```
       NEXTI = BREAK(',') $ OUTPUT LEN(1) | (LEN(1) REM) $
+              OUTPUT ABORT | ABORT
       GETI = NEXTI *GETI
       PRINTI = POS(0) GETI
```

1.20a The following program assumes that the property string to be analyzed is given on consecutive data records.

```
       &TRIM   =   1
       &FULLSCAN   =   1
       FIND   =   BREAK('X') LEN(1) @OUTPUT *FIND
       DISPLAY   =   POS(0) FIND
READ   STRING   =   STRING INPUT               :S(READ)
       STRING   DISPLAY
END
```

Why is the fullscan mode necessary?

1.20b The distance between successive positions of interest can be obtained by modifying the program above as follows:

```
                 .
                 .
                 .
       DEFINE('SPANN(X,Y)')
       SPANNER = BREAK('X') LEN(1) @N *SPANN(N,L) @L *SPANNER
       DISTANCE   =   POS(0) @L @N SPANNER
                 .
                 .
                 .
       STRING   DISTANCE                     :(END)
                 .
                 .
                 .
SPANN  OUTPUT   =   GT(L,0) X - Y - 1        :(RETURN)
                 .
                 .
                 .
```

1.21c There are a number of ways to determine if an integer is palindromic. The following program uses REVERSE (see Exercise 2.4) and a comparison. Since IDENT is used for the comparison, it is necessary to convert the number into a string.

```
       DEFINE('PALMRK(N)')
       N   =   1
PALOOP N   =   LT(N,500) N + 1               :F(DONE)
       PALSTR   =   PALSTR PALMRK(N ** 2)    :(PALOOP)
PALMRK N   =   CONVERT(N,'STRING')
       PALMRK   =   IDENT(N,REVERSE(N)) 'X'  :S(RETURN)
       PALMRK   =   '0'                      :(RETURN)
DONE   OUTPUT   =   PALSTR
END
```

1.23 The following program uses patterns similar to those discussed in the text. There are a few differences, most notably the use of the pattern F which assigns the position to be marked to M. This program does not play a particularly sophisticated game, but it does illustrate the approach.

```
*********************************************************************
*                           INITIALIZATION                          *
*********************************************************************
       &TRIM   =   1
       DEFINE('COL(C1,C2,C3)')
       DEFINE('DIAG(C1,C2,C3)')
       DEFINE('ROW(C1,C2,C3)')
       DEFINE('TWO(M)')
       SKIP    =   NULL . OUTPUT
       ROW    =   LEN(3) . OUTPUT
       PRINT   =   SKIP ROW ROW ROW SKIP
*********************************************************************
*                           BOARD PATTERNS                          *
*********************************************************************
       C   =   TAB(0) | TAB(1) | TAB(2)
       R   =   TAB(0) | TAB(3) | TAB(6)
       F   =   @M '.'
       P   =   POS(*M) '.'
       CENTER   =   TAB(4) F
       CORNER   =   (TAB(0) | TAB(2) | TAB(6) | TAB(8)) F
       WIN   =   TWO('O')
       BLOCK   =   TWO('X')
       LOSE    =   ROW('X','X','X') | COL('X','X','X') |
+                 DIAG('X','X','X')
       PLAY    =   BLOCK | CORNER | F
*********************************************************************
*                           INTRODUCTION                            *
*********************************************************************
       OUTPUT    =   'THIS PROGRAM PLAYS TIC-TAC-TOE. YOUR MARK IS'
       OUTPUT    =   'X AND YOU WILL PLAY FIRST. THE BOARD IS'
       OUTPUT    =   'NUMBERED AS FOLLOWS:'
       '012345678'   PRINT
       OUTPUT    =   'WHEN IT IS YOUR TURN TO PLAY, TYPE THE NUMBER'
       OUTPUT    =   'OF THE SQUARE YOU WISH TO MARK. FOR EXAMPLE'
       OUTPUT    =   'IF YOU TYPE "4", THE RESULT IS:'
       '....X....'   PRINT
       OUTPUT    =   'ANY TIME IT IS YOUR TURN TO PLAY, YOU MAY'
       OUTPUT    =   'START A NEW GAME BY TYPING "N" OR END THE'
       OUTPUT    =   'SESSION BY TYPING "Q"'
```

```
**********************************************************************
*                             GAME PLAY                              *
**********************************************************************
START  OUTPUT  =   DUPL('-',10)
       OUTPUT  =   'NEW GAME'
       BOARD   =  DUPL('.',9)
NEXT   OUTPUT  =   'YOUR PLAY'
       M   =   INPUT                                    :F(ABAND)
       IDENT(M,'N')                                     :S(START)
       IDENT(M,'Q')                                     :S(STOP)
       INTEGER(M)                                       :F(ERROR)
       BOARD   P   =   'X'                              :F(ERROR)
       BOARD   CENTER                                   :S(MINE)
       BOARD   LOSE                                     :S(LOSE)
       BOARD   WIN                                      :S(WIN)
       BOARD   PLAY                                     :F(TIE)
MINE   BOARD   P   =   'O'
       BOARD   PRINT                                    :(NEXT)
LOSE   OUTPUT  =   'YOU WIN'                            :(NEW)
TIE    OUTPUT  =   'TIE GAME'                           :(NEW)
WIN    BOARD   P   =   'O'
       OUTPUT  =   'I WIN'
NEW    BOARD   PRINT                                    :(START)
ABAND  OUTPUT  =   "SESSION ABANDONED"                  :(END)
STOP   OUTPUT  =   "SESSION ENDED"                      :(END)
ERROR  OUTPUT  =   "ERRONEOUS MOVE - TRY AGAIN" :(NEXT)
**********************************************************************
*                        FUNCTION DEFINITIONS                        *
**********************************************************************

COL    COL   =   C C1 LEN(2) C2 LEN(2) C3          :(RETURN)
DIAG   DIAG   =   TAB(0) C1 TAB(4) C2 TAB(8) C3 |
+                 TAB(2) C1 TAB(4) C2 TAB(6) C3   :(RETURN)
ROW    ROW   =   R C1 C2 C3                        :(RETURN)
TWO    TWO   =   ROW(F,M,M) | ROW(M,F,M) | ROW(M,M,F)|
+                COL(F,M,M) | COL(M,F,M) | COL(M,M,F)|
+                DIAG(F,M,M) | DIAG(M,F,M) | DIAG(M,M,F)  :(RETURN)
END
```

2.1 The following solutions use the forms of LPAD and RPAD that fail if the length of the argument is greater than N. If C is longer than one character, the first character is used for filling. As in the text, the default fill character, if C is null, is the blank.

```
       DEFINE('LPAD(S,N,C)')
       DEFINE('RPAD(S,N,C)')
       FILLPAT   =   LEN(1) . C
                    .
                    .
                    .
LPAD   C   FILLPAT                          :S(LPADR)
       C   =   ' '
LPADR  LPAD  =  DUPL(C,N - SIZE(S)) S      :S(RETURN)F(FRETURN)
*
RPAD   C   FILLPAT                          :S(RPADR)
       C   =   ' '
RPADR  RPAD   =   S DUPL(C,N - SIZE(S)) :S(RETURN)F(FRETURN)
```

2.2 The following function is similar in philosophy to LPAD and RPAD. If S cannot be precisely centered, it is placed one character to the left of center.

```
       DEFINE('CENTER(S,N,C)M,H')
                    .
                    .
                    .
CENTER C   FILLPAT                          :S(CENTRR)
       C   =   ' '
CENTRR M   =   SIZE(S)
       H   =   GE(N,M) (N - M) / 2          :F(FRETURN)
       M   =   REMDR(N - M,2)
       CENTER   =   DUPL(C,H) S DUPL(C,H + M)   :(RETURN)
```

2.3 The following solution handles the situation in which one string is longer than the other by simply appending any residue to the end of the result. For an entirely different approach, see Section 5.3.6.

```
       DEFINE('COLLATE(S1,S2)C1,C2')
       CHAR1    =   LEN(1) . C1
       CHAR2    =   LEN(1) . C2
                    .
                    .
                    .
COLLATE  S1   CHAR1    =                    :F(COLL1)
       S2   CHAR2    =                      :F(COLL2)
       COLLATE   =   COLLATE C1 C2          :(COLLATE)
COLL1  COLLATE   =   COLLATE S2             :(RETURN)
COLL2  COLLATE   =   COLLATE C1 S1          :(RETURN)
```

2.4 The following procedure for REVERSE is simple and straightforward. A more efficient procedure can be devised by identifying more characters at a time for longer strings. For an entirely different approach, see Section 5.3.4.

```
       DEFINE('REVERSE(S)C')
       ONECH    =   LEN(1) . C
                 .
                 .
                 .
REVERSE S   ONECH   =                     :F(RETURN)
        REVERSE   =   C REVERSE           :(REVERSE)
```

2.5 In this exercise, the direction of rotation needs to be defined. In the procedure that follows, a positive value of N corresponds to rotation to the right, and a negative value corresponds to rotation to the left. Note that a value of N greater than the size of the string is acceptable.

```
       DEFINE('ROTATE(ROTATE,N)H,T')
       ROTPAT = (*LT(N,0) LEN(*(-N)) . H | RTAB(*N) . H) REM . T
                 .
                 .
                 .
ROTATE N = GT(SIZE(ROTATE),0) REMDR(N,SIZE(ROTATE)):F(RETURN)
       ROTATE   ROT~~ATE~~   =   T H        :(RETURN)
                   PAT
```

2.6 The following function fails if the value of C is null. If the value of C is longer than one character, all occurrences of the characters are deleted. An alternative approach would be to only use the first character of a longer string, as in the case of LPAD and RPAD.

```
       DEFINE('DELETE(S,C)')
       DELPAT   =   BREAK(*C) . H SPAN(*C)
                 .
                 .
                 .
DELETE IDENT(C)                           :S(FRETURN)
DELOOP S   DELPAT   =                     :F(DELRET)
       DELETE   =   DELETE H              :(DELOOP)
DELRET DELETE   =   DELETE S              :(RETURN)
```

2.7

```
        DEFINE('TRUNC(A,N)I')
        DEFINE('EXTEND(A,N)I')
                  .
                  .
                  .
TRUNC   TRUNC   =   GT(N,0) ARRAY(N)           :F(FRETURN)
TRUNC1  I   =   I + 1
        TRUNC<I>   =   A<I>                :F(RETURN)S(TRUNC1)

EXTEND  EXTEND   =   ARRAY(PROTOTYPE(A) + N)
EXTND1  I   =   I + 1
        EXTEND<I>   =   A<I>              :S(EXTND1)F(RETURN)
```

2.13

```
        OPSYN('TRYM','TRIM')
        DEFINE('TRIM(TRIM,C)')
        TRIMPAT   =   RTAB(1) . TRIM *C
                  .
                  .
                  .
TRIM    C   =   IDENT(C) ' '
        TRIM   =   IDENT(C,' ') TRYM(TRIM) :S(RETURN)
TRIMC   TRIM   TRIMPAT                     :S(TRIMC)F(RETURN)
```

This function allows the trim character to be defaulted to a blank. If conventional (blank) trimming is specified, either explicitly or by default, the built-in trimming function is used (through the synonym TRYM) for efficiency. Since SNOBOL4 does not provide a right-to-left SPAN, one character at a time is removed from the end of the string. On SNOBOL4 systems that have a built-in version of REVERSE, the following alternative may be reasonable:

```
        TRIMPAT   =   POS(0) SPAN(C) REM . TRIM
                  .
                  .
                  .
TRIM    C   =   IDENT(C) ' '
        REVERSE(TRIM)   TRIMPAT            :F(RETURN)
        TRIM   =   REVERSE(TRIM)           :(RETURN)
```

Note that this solution permits a string of trim characters to be specified. How could the first solution above be adapted to this interpretation of trimming?

2.14 A recursive procedure is:

```
        DEFINE('FACT(N)')
                  .
                  .
                  .
FACT    FACT   =   LE(N,1) 1                :S(RETURN)
        FACT   =   N * FACT(N - 1)          :(RETURN)
```

An iterative procedure is:

```
FACT    FACT   =   LE(N,1) 1                :S(RETURN)
        FACT   =   N
FACT1   N   =   GT(N,2) N - 1               :F(RETURN)
        FACT   =   FACT * N                 :(FACT1)
```

These procedures do not check the validity of their arguments.

2.15 Fibonacci numbers can be computed iteratively as follows:

```
        DEFINE('F(N)F1,F2,F3')
F       F1   =   1;   F2   =  1;   F3   =   0
F1      F    =   F2 + F3
        F2   =   F3;   F3   =   F
        F1   =   LT(F1,N) F1 + 1               :S(F1)F(RETURN)
```

This procedure does not check the validity of its argument.

2.16

```
        DEFINE('A(M,N)')
                  .
                  .
                  .
A       A   =   EQ(M,0) N + 1                  :S(RETURN)
        A   =   EQ(N,0) A(M - 1,1)             :S(RETURN)
        A   =   A(M - 1,A(M,N - 1))            :(RETURN)
```

This solution does not test the values of the arguments for validity. The value of Ackermann's function grows very rapidly with the size of its arguments and the depth of recursion prevents computation of the function for anything except small values of M.

2.17 The following solution first prints banner lines to make the histograms easier to interpret. The purpose of CFNCLEVEL is to adjust the histogram in case APROFL is called within another defined function.

```
       DEFINE('APROFL(M,N)')
       DEFINE('ADEPTH(M,N)')
       HUNDREDS   =   DUPL(' ',99) 1
       B   =   DUPL(' ',9)
       TENS   =   B 1 B 2 B 3 B 4 B 5 B 6 B 7 B 8 B 9 B 0
       DIGITS   =   DUPL('1234567890',10)
                  .
                  .
                  .
APROFL OUTPUT   =
       OUTPUT = 'PROFILE OF A(' M ',' N ')'
       OUTPUT   =
       OUTPUT   =   HUNDREDS
       OUTPUT   =   TENS
       OUTPUT   =   DIGITS
       CFNCLEVEL   =   &FNCLEVEL
       ADEPTH(M,N)
       OUTPUT   =                            :(RETURN)
ADEPTH OUTPUT   =  DUPL('X',&FNCLEVEL - CFNCLEVEL)
       ADEPTH   =  EQ(M,0) N + 1             :S(RETURN)
       ADEPTH   =  EQ(N,0) ADEPTH(M - 1,1)   :S(RETURN)
       ADEPTH   =  ADEPTH(M - 1,ADEPTH(M,N - 1))  :(RETURN)
```

2.18

```
       DEFINE('PREFIX(PREFIX)L,R,OP')
       STRIP   =   POS(0) '(' BAL . PREFIX ')' RPOS(0)
                  .
                  .
                  .
PREFIX PREFIX   STRIP                          :S(PREFIX)
       PREFIX   INFIX  =  OP '(' PREFIX(L) ',' PREFIX(R) ')'
+                                              :(RETURN)
```

2.19 If L, R, and OP are not local variables, a call of PREFIX during the execution of PREFIX may change the values of these variables before they are used in computing the result. For example, the call of PREFIX(L) may change the value of R before PREFIX(R) is called. A detailed knowledge of the internal workings of SNOBOL4 is needed in order to know the precise order in which operations are performed and hence what variables must be local. OP, for example, does not have to be local because its value is obtained for the concatenation before PREFIX(L) is called. Nevertheless, it is good practice to specify as local all variables whose values are assigned by the function.

2.20

```
       DEFINE('REVERSE(S)C')
       ONECH   =   LEN(1) . C
                     .
                     .
                     .
REVERSE S   ONECH   =                          :F(RETURN)
        REVERSE   =   REVERSE(S) C             :(RETURN)
```

Using recursion to reverse a string is unnecessary, even perverse; the iterative solution is straightforward, simpler, and more efficient. Furthermore, a fixed limit on the depth of recursion prevents the recursive reversal of strings of even a moderate length, if only one character is processed at a time.

2.22

```
       DEFINE('INFIX(INFIX)L,R,OP')
       PREFIX = POS(0) LEN(1) . OP '(' BAL . L ',' BAL . R ')'
+               RPOS(0)
                     .
                     .
                     .
INFIX  INFIX PREFIX = '(' INFIX(L) OP INFIX(R) ')'  :(RETURN)
```

2.24 RANDOM must not attempt to compute a number larger than the maximum integer allowed in the version of SNOBOL4 being used. The value of RANVAR may be as large as $m - 1$ where m is the modulus. Assume that argument N is always small compared with m (this is required for RANDOM to produce good results). Then the concern is that $p*(m - 1) + c$ not exceed the limit, where p is the multiplier and c is the increment. For the values used for RANDOM in the text, the largest value that can occur is 1,262,108,510.

2.25 Independent random number generators are easily implemented by making RANVAR an array and providing a second argument for RANDOM to specify which element of RANVAR is to be used. For example:

```
       RANVAR   =   ARRAY(10)
```

provides for 10 independent random number generators. The function becomes:

```
       DEFINE('RANDOM(N,I)')
                     .
                     .
                     .
RANDOM RANVAR<I>  =  REMDR(RANVAR<I> * 12621 + 21131,100000)
       RANDOM   =   RANVAR<I> * N / 10000  :(RETURN)
```

RANDOM(N,5) produces the next value from generator 5. Further generality can be obtained by treating the initial value, modulus, multiplier, and increment as parameters of a particular generator. An M-by-4 array is needed for M generators.

2.26

```
        DEFINE('RANCHR()')
        ASIZE   =   SIZE(&ALPHABET)
        RANSEL   =   TAB(*RANDOM(ASIZE)) LEN(1) . RANCHR
                  .
                  .
                  .
RANCHR &ALPHABET   RANSEL                         :(RETURN)
```

2.27

```
        DEFINE('RANSTR(N)I')
                  .
                  .
                  .
RANSTR N    =   RANDOM(N + 1)
RANL   I    =   LT(I,N) I + 1                     :F(RETURN)
       RANSTR   =   RANSTR RANCHR()               :(RANL)
```

2.29 &ALPHABET is a reasonable default value for the second argument of GRAM.

2.31 To generate the successor of any string, it is only necessary to generate the next *n*gram. If this is not possible, the first (*n* + 1)gram is returned instead:

```
        DEFINE('NEXTS(STRING)C')
                  .
                  .
                  .
NEXTS   NEXTS   =   NEXTG(STRING)                 :S(RETURN)
        NEXTS   =   DUPL(F,SIZE(STRING) + 1)      :(RETURN)
```

Using this method, the first of all the strings is the null string. Next are the single characters, the digrams, the trigrams, and so forth.

2.33 To compute the first *n*gram from the last, instead of signaling failure, replace the last statement in NEXTG by

```
NEXTG1 NEXTG   =   NEXTG(GRAM) F                  :S(RETURN)
       NEXTG   =   DUPL(F,SIZE(GRAM) + 1)         :(RETURN)
```

This form of NEXTG is often more convenient than the one that fails. An example of its use appears in the code column of the character set tables for ASCII and EBCDIC given in Appendix A.

2.37 The concordance program given in the text can be modified to process *n*grams, instead of words, by changing the procedure for GET:

```
       NGRAM   =   LEN(*N) . GET
       ONECH   =   LEN(1) . C
                   .
                   .
                   .
GET    LINE   NGRAM                                 :F(GETM)
       LINE   ONECH    =                            :(RETURN)
GETM   LINE.NO    =    LINE.NO + 1
       LINE    =    INPUT                           :F(FRETURN)
       OUTPUT    =    LINE LINE.NO                  :(GET)
```

The value of N can be established at the beginning of the program, perhaps being supplied on the first data record.

2.40 Change CITE and the call to it as follows:

```
       DEFINE('CITE(WORD,TABLE)')
                   .
                   .
                   .
NEXT   CITE(GET(),T)                                :S(NEXT)
                   .
                   .
                   .
CITE   TABLE<WORD>    =    TABLE<WORD> + 1          :(RETURN)
```

2.41 A general facility for omitting the citation of specific words can be implemented by inserting the words to be omitted in the citation table before processing the text and giving them a special value that indicates they are not to be cited. A test for this special value can be made in CITE and PRINT.

2.44 CITE begins with

```
CITE   EQ(SIZE(WORD),4)                             :F(RETURN)
                   .
                   .
                   .
```

Whatever form of citation is desired then follows. There are many similar variations. Considerably more generality can be obtained by providing an additional argument to CITE which is an unevaluated expression:

```
        DEFINE('CITE(WORD,N,TABLE,TEST)')
                  .
                  .
                  .
CITE    EVAL(TEST)                               :F(RETURN)
                  .
                  .
                  .
```

For the example above, CITE would be used as follows:

```
        PRED   =   *EQ(SIZE(WORD),4)
                  .
                  .
                  .
NEXT    CITE(GET(),LINE.NO,T,PRED)               :S(NEXT)
                  .
                  .
                  .
```

2.49 Titling and spacing are operations that are specific to the concordance program. The functions GET and PRINT can be used in other programs and hence should be free of such specific properties.

2.50 SORT can be extended in an upward-compatible way by providing two additional arguments that default to column 1 and the predicate LGT, respectively:

```
        DEFINE('SORT(TABLE,C,P)I,N,M,J,G,K,T1,T2')
        ALEN   =   BREAK(',') . N
                  .
                  .
                  .
SORT    SORT   =   CONVERT(TABLE,'ARRAY')       :F(FRETURN)
        C    =   IDENT(C) 1
        P    =   IDENT(P) 'LGT'
        OPSYN('CMP',P)
        PROTOTYPE(SORT)   ALEN
        G    =   N
SORTG   G    =   GT(G,1) G/2                    :F(RETURN)
        M    =   N - G
SORTK   K    =   0
        I    =   1
```

```
SORTJ  J    =    I + G
       CMP(SORT<I,C>,SORT<J,C>)                  :F(SORTI)
       T1    =    SORT<I,1>
       T2    =    SORT<I,2>
       SORT<I,1>   =   SORT<J,1>
       SORT<I,2>   =   SORT<J,2>
       SORT<J,1>   =   T1
       SORT<J,2>   =   T2
       K    =    K + 1
SORTI  I    =    LT(I,M) I + 1                   :S(SORTJ)
       GT(K,0)                                  :S(SORTK)F(SORTG)
```

Care must be taken in the choice of predicates supplied to SORT. Consider the result of specifying IDENT or DIFFER.

2.51 Using the solutions of Exercises 2.40 and 2.50, it is only necessary to change the printing statement to

```
       PRINT(SORT(T,2,'LT'))                     :S(END)
```

3.1 A reasonably mnemonic default is STACK. An example of this default is given by PUSH in the defined data type implementation:

```
PUSH   S    =    IDENT(S) STACK
       TOP(S)    =    PLATE(V,TOP(S))            :(RETURN)
```

3.2 One method of specifying arbitrary strings is the Hollerith literal. A string is specified by an integer, which indicates its length, followed by an H, followed by the characters of the string. Thus 3H,,, specifies three commas, and 4HHHHH specifies a string of four Hs. Since such specifications are delimited by length, they can be concatenated without the need for separators.

3.4 For the defined data type implementation, a function to count the number of items on a stack is:

```
       DEFINE('SSIZE(S)')
                 .
                 .
                 .
SSIZE  SSIZE    =    0
       S    =    TOP(S)
SSIZEL SSIZE    =    DIFFER(S) SSIZE + 1         :F(RETURN)
       S    =    LAST(S)                         :(SSIZEL)
```

Why is it necessary to set SSIZE to zero even though its initial value when the function is called is the null string?

SSIZE for the array implementation is even simpler:

```
SSIZE  SSIZE   =   INDEX(S) - 1              :(RETURN)
```

3.5 For the defined data type implementation, a function to copy a stack is:

```
       DEFINE('SCOPY(S)T')
                 .
                 .
                 .
SCOPY  SCOPY   =   COPY(S)
       TOP(SCOPY) = DIFFER(TOP(S)) COPY(TOP(S))   :F(RETURN)
       T   =   TOP(SCOPY)
SCOPYL LAST(T)  =  DIFFER(LAST(T)) COPY(LAST(T))  :F(RETURN)
       T   =   LAST(T)                        :(SCOPYL)
```

The technique used in this solution is discussed in detail in Section 3.2.3. For the string implementation, the function is much simpler:

```
       DEFINE('SCOPY(S)')
                 .
                 .
                 .
SCOPY  SCOPY   =   STACK(VALUE(S))           :(RETURN)
```

For the array implementation, the function is nearly as simple:

```
SCOPY  SCOPY   =   STK(COPY(LIST(S)),INDEX(S))    :(RETURN)
```

3.6 For the defined data type implementation of stacks, a function to reinitialize a stack is trivial:

```
       DEFINE('SINIT(S)')
                 .
                 .
                 .
SINIT  TOP(S)    =                           :(RETURN)
```

Similarly, for the string implementation, the procedure is:

```
SINIT  VALUE(S)    =                         :(RETURN)
```

For the array implementation, the procedure is:

```
SINIT  INDEX(S)    =   1                     :(RETURN)
```

3.7 For the defined data type implementation, a function to print a stack from the bottom up is:

```
        DEFINE('PRTSTK(S)T')
                  .
                  .
                  .
PRTSTK  T    =   STACK()
        S    =   TOP(S)
PRTS    (DIFFER(S) PUSH(VALUE(S),T))          :F(PRTD)
        S    =   LAST(S)                      :(PRTS)
PRTD    OUTPUT    =    POP(T)                 :S(PRTD)F(RETURN)
```

3.10 The following function creates a ring that contains one more element than specified in the call. This extra element is needed to distinguish between an empty queue and a full queue, as described in the text.

```
        DATA('QUE(HEAD,TAIL)')
        DEFINE('QUEUE(N)E,I')
                  .
                  .
                  .
QUEUE   E    =   ELEMENT()
        QUEUE    =    QUE(E,E)
QCONT   I    =   LT(I,N) I + 1                :F(QLINK)
        NEXT(E)    =    ELEMENT()
        E    =   NEXT(E)                      :(QCONT)
QLINK   NEXT(E)    =    HEAD(QUEUE)           :(RETURN)
```

3.12

```
        DEFINE('STKQUE(S,Q)')
                  .
                  .
                  .
STKQUE  INSERT(POP(S),Q)                    :S(STKQUE)F(RETURN)
```

3.16

```
        DEFINE('WLIST()L')
        DATA('ELEMENT(VALUE,NEXT)')
                  .
                  .
                  .
WLIST   WLIST    =    ELEMENT(GET())          :F(FRETURN)
        L    =   WLIST
NEXTW   NEXT(L)    =    ELEMENT(GET())        :F(RETURN)
        L    =   NEXT(L)                      :(NEXTW)
```

This function uses GET as described in Section 2.4.

3.17

```
       DEFINE('COPYL(L)')
                .
                .
                .
COPYL  COPYL   =   COPY(L)
       NEXT(COPYL) = DIFFER(NEXT(L)) COPYL(NEXT(L)) :(RETURN)
```

The major limitation of a recursive procedure for copying lists is the restriction on depth of recursion in SNOBOL4. The depth of recursion is proportional to the length of the list. In many cases, linked lists are relatively long and hence cannot be copied recursively.

3.18

```
       DEFINE('PRINTL(L)')
                .
                .
                .
PRINTL OUTPUT  =   DIFFER(L) VALUE(L)      :F(RETURN)
       L   =   NEXT(L)                     :(PRINTL)
```

3.24 An additional field is needed in the nodes. A new definition might be:

```
       DATA('BNODE(SYMBOL,VALUE,LEFT,RIGHT,UP)')
```

In this context, the SYMBOL field replaces the former VALUE field and the new VALUE field contains the value associated with the symbol. Thus, the SYMBOL field is used to order the tree. The functions that process binary trees must be changed accordingly, and functions must be written to access the value of a symbol. A typical function locates a node in the binary tree according to a symbol and returns the corresponding value.

3.26b

```
       DATA('BNODE(VALUE,LEFT,RIGHT)')
       DEFINE('LEXPRT(T)N,S')
                .
                .
                .
LEXPRT N   =   DIFFER(ROOT(T)) ROOT(T)    :F(RETURN)
       S   =   STACK()
LEXPRD (DIFFER(RIGHT(N)) PUSH(N,S))       :F(LEXOUT)
       N   =   RIGHT(N)                   :(LEXPRD)
LEXOUT OUTPUT  =   VALUE(N)
       N   =   DIFFER(LEFT(N)) LEFT(N)    :S(LEXPRD)
       N   =   POP(S)                     :S(LEXOUT)F(RETURN)
```

3.28

```
       DEFINE('TSIZE(T)')
                .
                .
                .
TSIZE  TSIZE    =   IDENT(T) 0             :S(RETURN)
       TSIZE    =   1
TSIZE1 T    =   NEXT(T)                    :F(RETURN)
       TSIZE    =   TSIZE + 1              :(TSIZE1)
```

3.32

```
       DEFINE('LSIB(NODE)')
                .
                .
                .
LSIB   IDENT(FATHER(NODE))                 :S(FRETURN)
       LSIB    =   LSON(FATHER(NODE))
       IDENT(LSIB,NODE)                    :S(FRETURN)
LSIB1  IDENT(RSIB(LSIB),NODE)              :S(RETURN)
       LSIB    =   R~~IS~~B(LSIB)               :(LSIB1)
                    SI
```

3.37

```
       DEFINE('COPYR(R)HEAD,C')
                .
                .
                .
COPYR  HEAD    =   R
       COPYR    =   COPY(R)
       C    =   COPYR
COPYR1 NEXT(C)    =   IDENT(NEXT(C),HEAD) COPYR   :S(RETURN)
       NEXT(C)    =   COPY(NEXT(C))
       C    =   NEXT(C)                    :(COPYR1)
```

3.42a

```
       DATA('CELL(VALUE,S1,S2,S3,S4,S5,S6)')
```

3.42c

```
       DEFINE('INTERIOR(C)')
                .
                .
                .
INTERIOR (DIFFER(S1(C)) DIFFER(S2(C)) DIFFER(S3(C)) DIFFER(S4(C))
+         DIFFER(S5(C)) DIFFER(S6(C)))   :S(RETURN)F(FRETURN)
```

4.1

```
        DEFINE('STRRTL(S)')
        DEFINE('RTLSTR(R)')
        INTEGER  =  ANY('+-') SPAN('0123456789')
        RTLPAT  =  POS(0) INTEGER . N '/' INTEGER . D RPOS(0)
                 .
                 .
                 .
RTLSTR RTLSTR  =  N(R) '/' D(R)            :(RETURN)

STRRTL S   RTLPAT                          :F(FRETURN)
       STRRTL  =  REDUCE(RATIONAL(N,D))    :(RETURN)
```

4.2

```
        DEFINE('ADDRTL(R1,R2)')
                 .
                 .
                 .
ADDRTL ADDRTL = REDUCE(RATIONAL(N(R1) * D(R2) + N(R2) * D(R1),
+               D(R1) * D(R2)))            :(RETURN)
```

4.5 If REDUCE were to modify the fields of its argument, this would have the side effect of changing parts of an object that might be pointed to by other objects or might be the value of a variable that is not directly related to the argument of the function.

4.9a One approach to the representation of complex rationals is to combine the representation used for complex numbers and the representation used for rational numbers. The operands of a complex number become rational numbers. A data type such as

```
        DATA('CPXRTL(RR,IR)')
```

can be used. This data type is the same as COMPLEX, but it is given a different name and different field functions to avoid confusion. RATIONALs are assigned to the RR and IR fields. Figure B.1 illustrates the structure corresponding to the complex rational given in the exercise.

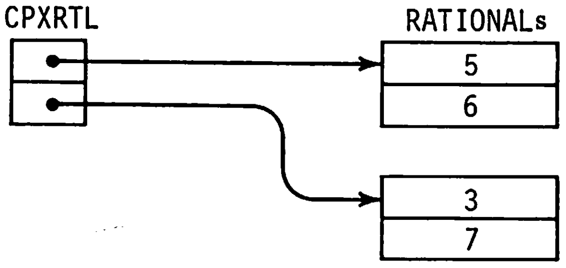


Figure B.1 A Complex Rational

4.9b

```
        DEFINE('CXRSTR(X)S')
                   .
                   .
                   .
CXRSTR  CXRSTR   =   RTLSTR(RR(X))
        S   =   RTLSTR(IR(X))
        S   '-'                               :S(CXRM)
        CXRSTR   =   CXRSTR '+' S 'I'         :(RETURN)
CXRM    CXRSTR   =   CXRSTR S 'I'             :(RETURN)
```

4.9c

```
        DEFINE('ADDCXR(X1,X2)')
                   .
                   .
                   .
ADDCXR  ADDCXR   =   CPXRTL(ADDRTL(RR(X1),RR(X2)),
+                    ADDRTL(IR(X1),IR(X2)))   :(RETURN)
```

4.11a 10^9.

4.11b 10.

4.12 Besides being naturally related to the commonly used base 10, a base that is of the form 10^n permits easy segmentation using pattern matching, since n digits correspond to a segment.

4.15

```
        DEFINE('INTLRG(I)')
                   .
                   .
                   .
INTLRG  INTLRG   =   LRGINT(REMDR(I,BASE))
        NEXT(INTLRG) = GE(I,BASE) INTLRG(I / BASE)  :(RETURN)
```

4.16

```
        DEFINE('LRGSTR(L)')
        WIDTH   =   SIZE(BASE - 1)
                   .
                   .
                   .
LRGSTR  LRGSTR   =   SEGMENT(L)
        L   =   DIFFER(NEXT(L)) NEXT(L)           :F(RETURN)
        LRGSTR   =   LRGSTR(L) LPAD(LRGSTR,WIDTH,0) :(RETURN)
```

4.20

```
          DEFINE('MULLRG(L1,L2)')
                          .
                          .
                          .
MULLRG MULLRG   =   DIFFER(L1) DIFFER(L2) SEGMENT(L1) *
+                   SEGMENT(L2)               :F(RETURN)
       MULLRG   =   LRGINT(REMDR(MULLRG,BASE),ADDLRG(
+               MULLRG(L1,NEXT(L2)),MULLRG(NEXT(L1),LRGINT(
+               SEGMENT(L2))),MULLRG / BASE))    :(RETURN)
```

4.22 A first attempt to handle large negative integers might be to sign the first segment. It is the entire integer, not a segment, however, that has a sign. A more general solution is to put a heading data type on large integers. Data types might be:

```
          DATA('LRGINT(SIGN,LIST)')
          DATA('INTSEG(SEGMENT,NEXT)')
```

where the SIGN field contains a +1 or -1 depending on whether the large integer is positive or negative, respectively. INTSEG replaces the former definition of LRGINT. The new representation for the large integer −3,765,197,658,102,103 is shown in Figure B.2.

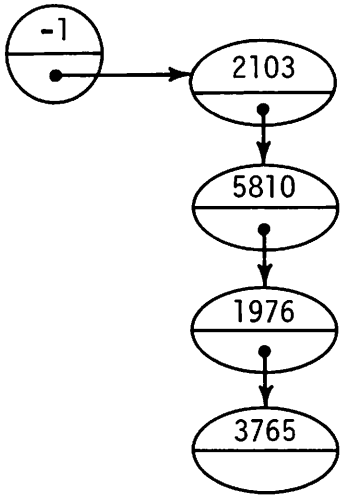


Figure B.2 A Signed Large Integer

4.30a Access to the segments is more direct and natural in a linked list than in a table. Much of the discussion of alternative methods of implementing stacks (see Section 3.2.1) applies to this case.

4.30b Many polynomials are sparse, containing a large number of terms with zero coefficients. A linked-list representation requires segments for such terms. In addition, many operations on polynomials require the coefficient of a specific term. This operation is awkward on linked lists.

4.33 A simple procedure to convert the table representation of a polynomial to the string representation is:

```
       DEFINE('POLSTR(P)I')
                    .
                    .
                    .
POLSTR P    =   CONVERT(P,'ARRAY')       :F(FRETURN)
POLST1 I    =   I + 1
       POLSTR  =   POLSTR '(' P<I,2> ':' P<I,1> ')'
+                                        :S(POLST1)F(RETURN)
```

This function does not order the terms. The solution of Exercise 4.32 can be used in conjunction with the extended SORT function of Exercise 2.50 to order the terms.

4.34

```
       DEFINE('CPYPOL(P)')
                    .
                    .
                    .
CPYPOL CPYPOL  =  CONVERT(COPY(CONVERT(P,'ARRAY')),'TABLE')
+                                        :S(RETURN)F(FRETURN)
```

4.38a Since a continued fraction can be represented as a sequence of integers, a linked list of integers is a natural representation.

4.38b The notation given in the exercise provides a natural string representation for continued fractions. For example, 86/11 can be represented by (7,1,4,2).

4.38d The continued fraction representation of a rational number is obtained by using the method for computing greatest common divisors. A procedure to convert a rational number to the string representation of a continued fraction is:

```
        DEFINE('CFRAC(R)N,D')
        FRACR   =   RTAB(1) . CFRAC
                  .
                  .
                  .
CFRAC   N   =   N(R);   D   =   D(R)
CFRACC  CFRAC   =   CFRAC N / D ','
        R   =   REMDR(N,D)
        EQ(R,0)                                    :S(CFRACR)
        N   =   D;   D   =   R                     :(CFRACC)
CFRACR  CFRAC   FRACR
        CFRAC   =   '(' CFRAC ')'                  :(RETURN)
```

4.38e If the solution of Exercise 4.38a is extended to periodic infinite continued fractions, a "curlicue" results. See Section 3.3.

4.38f Some method is needed to indicate the place where the continued fraction begins to repeat. One way of doing this is to place the repeated group in parentheses. For example, the representation of $\sqrt{2}$ would be `(1,(2))`.

4.43

```
        DEFINE('ADD(U,V)')
        DEFINE('SUB(U,V)')
        DEFINE('MUL(U,V)')
        DEFINE('DIV(U,V)')
        DEFINE('EXP(U,V)')
        OPSYN('&','+',2)
        OPSYN('#','-',2)
        OPSYN('%','/',2)
        OPSYN('?','*',2)
        OPSYN('@','**',2)
        OPSYN('+','ADD',2)
        OPSYN('-','SUB',2)
        OPSYN('/','DIV',2)
        OPSYN('*','MUL',2)
        OPSYN('**','EXP',2)
                  .
                  .
                  .
ADD     INTEGER(U)                                 :F(ADDV)
        ADD   =   INTEGER(V) U & V                 :S(RETURN)
        ADD   =   EQ(U,0) V                        :S(RETURN)
```

```
ADDT    ADD  =  '(' U '+' V ')'            :(RETURN)
ADDV    INTEGER(V)                         :F(ADDT)
        ADD  =  EQ(V,0) U                  :S(RETURN)F(ADDT)

SUB     INTEGER(U)                         :F(SUBV)
        SUB  =  INTEGER(V) U # V           :S(RETURN)
        SUB  =  EQ(U,0) V                  :S(RETURN)
SUBT    SUB  =  '(' U '-' V ')'            :(RETURN)
SUBV    INTEGER(V)                         :F(SUBT)
        SUB  =  EQ(V,0) U                  :S(RETURN)F(SUBT)

MUL     INTEGER(U)                         :F(MULV)
        MUL  =  INTEGER(V) U ? V           :S(RETURN)
        MUL  =  EQ(U,0) 0                  :S(RETURN)
        MUL  =  EQ(U,1) V                  :S(RETURN)
MULT    MUL  =  '(' U '*' V ')'            :(RETURN)
MULV    INTEGER(V)                         :F(MULT)
        MUL  =  EQ(V,0) 0                  :S(RETURN)
        MUL  =  EQ(V,1) U                  :S(RETURN)F(MULT)

DIV     INTEGER(V)                         :F(DIVU)
        EQ(V,0)                            :S(DIVT)
        INTEGER(U)                         :F(DIVT)
        EQ(REMDR(U,V),0)                   :F(DIVT)
        DIV  =  U%V                        :(RETURN)
DIVT    DIV  =  '(' U '/' V ')'            :(RETURN)
DIVU    INTEGER(U)                         :F(DIVT)
        DIV  =  EQ(U,0) 0                  :S(RETURN)F(DIVT)

EXP     INTEGER(V)                         :F(EXPU)
        EXP  =  EQ(V,0) 1                  :S(RETURN)
        EXP  =  EQ(V,1) U                  :S(RETURN)
        EXP  =  INTEGER(U) U @ V           :S(RETURN)
EXPT    EXP  =  '(' U '!' V ')'            :(RETURN)
EXPU    INTEGER(U)                         :F(EXPT)
        EXP  =  EQ(U,0) 0                  :S(RETURN)
        EXP  =  EQ(U,1) 1                  :S(RETURN)F(EXPT)
```

4.47 One of the problems in dealing with defined data types is the difficulty in testing for data types and selecting the appropriate operations accordingly. The following solution uses tables to provide the necessary associations.

```
       DATA('SUM(L,R)')
       DATA('DIFF(L,R)')
       DATA('PROD(L,R)')
       DATA('QUOT(L,R)')
       DATA('EXPN(L,R)')
       DEFINE('INFDAT(INFDAT)L,R,OP,M')
       DEFINE('DATINF(DATINF)D')
       STRIP  =  POS(0) '(' BAL . INFDAT ')' RPOS(0)
       ASSIGN = *GT(M,0) TAB(*(M - 1)) . L LEN(1) . OP REM . R
       MATPM  =  (POS(0) BAL ANY('+-') @M FAIL) | ASSIGN
       MATMD  =  (POS(0) BAL ANY('*/') @M FAIL) | ASSIGN
       MATE   =  POS(0) BAL . L '!' . OP REM . R

       IDO    =  TABLE(5)
       IDO<'+'>  =  'SUM'
       IDO<'-'>  =  'DIFF'
       IDO<'*'>  =  'PROD'
       IDO<'/'>  =  'QUOT'
       IDO<'!'>  =  'EXPN'

       DIL    =  TABLE(7)
       DIL<'STRING'>   =  'RETURN'
       DIL<'INTEGER'>  =  'RETURN'
       DIL<'SUM'>   =  'DATIND'
       DIL<'DIFF'>  =  'DATIND'
       DIL<'PROD'>  =  'DATIND'
       DIL<'QUOT'>  =  'DATIND'
       DIL<'EXPN'>  =  'DATIND'

       DIO    =  TABLE(5)
       DIO<'SUM'>   =  '+'
       DIO<'DIFF'>  =  '-'
       DIO<'PROD'>  =  '*'
       DIO<'QUOT'>  =  '/'
       DIO<'EXPN'>  =  '!'
                 .
                 .
                 .
INFDAT INFDAT   STRIP                               :S(INFDAT)
       INFDAT   MATPM                               :S(DATFRM)
       INFDAT   MATMD                               :S(DATFRM)
       INFDAT   MATE                                :F(RETURN)
DATFRM INFDAT = APPLY(IDO<OP>,INFDAT(L),INFDAT(R)) :(RETURN)
```

```
DATINF D   =   DATATYPE(DATINF)                  :($DIL<D>)
DATIND DATINF = '(' DATINF(L(DATINF)) DIO<D> DATINF(R(DATINF)) ')'
+                                                :(RETURN)
```

4.48

```
       DEFINE('D(E,X)')
                 .
                 .
                 .
D                        :($('D' DATATYPE(E)))
DSUM    D   =   SUM(D(L(E),X),D(R(E),X))         :(RETURN)
DDIFF   D   =   DIFF(D(L(E),X),D(R(E),X))        :(RETURN)
DPROD   D = SUM(PROD(L(E),D(R(E),X)),PROD(R(E),D(L(E),X))) :(RETURN)
DQUOT   D = QUOT(DIFF(PROD(R(E),D(L(E),X)),PROD(L(E),D(R(E),X)))
+            ,EXPN(R(E),2))                      :(RETURN)
DEXPN   D = PROD(PROD(R(E),EXPN(L(E),R(E) - 1)),D(L(E),X))
+                                                :(RETURN)
DSTRING  D   =   IDENT(E,X) 1                    :S(RETURN)
DINTEGER D   =   0                               :(RETURN)
```

This solution can be extended to the more natural operator notation by making appropriate OPSYNs to the field functions.

4.52

```
       DEFINE('OR(L,R)')
                 .
                 .
                 .
OR     INTEGER(L)                                :F(ORR)
       OR   =   EQ(L,1) 1                        :S(RETURN)
       OR   =   R                                :(RETURN)
ORR    INTEGER(R)                                :F(ORC)
       OR   =   EQ(R,1) 1                        :S(RETURN)
       OR   =   L                                :(RETURN)
ORC    OR   =   IDENT(L,R) L                     :S(RETURN)
       OR   =   L 'v' R                          :(RETURN)
```

5.1 Restricted alphabets must be considered in several contexts. If the alphabet from which messages are composed is restricted, the enciphering and deciphering techniques that are given in the text work properly without modification. If the plain alphabet is restricted without restricting the message alphabet, the effect depends on the method used. Monoalphabetic and periodic polyalphabetic substitutions work properly, but characters in the message that do not occur in the plain alphabet are not modified by the enciphering procedures. Methods that use Vigenère Squares rely on the existence of a cipher alphabet corresponding to every character in the message. These methods must be modified to handle cases in which such alphabets do not exist. Restricted alphabets offer the possibility of having different sets of characters in the plain and cipher alphabets. Ciphers may consist of special characters instead of letters, for example. Such ciphers may appear to be more mysterious than ciphers composed of letters, but from a programming viewpoint there is no essential difference.

5.2

```
       DEFINE('ENCAES(M,K)')
                .
                .
                .
ENCAES ENCAES   =   REPLACE(M,&ALPHABET,ROTATE(&ALPHABET,K))
+                                                :(RETURN)
```

ROTATE is given in the solution of Exercise 2.5. The deciphering function is obvious.

5.3

```
       DEFINE('ENKAT(KEY)')
       REPEAT   =   (LEN(1) $ C BREAK(*C)) . GAP LEN(1)
       &FULLSCAN   =   1
                .
                .
                .
ENKAT  ENKAT   =   KEY &ALPHABET
ENK    ENKAT   REPEAT   =   GAP                :S(ENK)F(RETURN)
```

5.6 Representation of the key for periodic polyalphabetic substitution can be done neatly with a ring. A suitable defined data type is:

```
       DATA('KEYRING(CA,NEXT)')
```

An example of a "key ring" is:

```
       K4   =   KEYRING(ENKAT('ENCHILADA'))
       K3   =   KEYRING(ENKAT('LOBOTOMY'),K4)
       K2   =   KEYRING(ENKAT('PIGSKIN'),K3)
       KEY   =   KEYRING(ENKAT('ZEUS'),K2)
       NEXT(K4)   =   KEY
```

The enciphering and deciphering functions are:

```
        DEFINE('ENPAS(M,KEY)K')
        DEFINE('DEPAS(C,KEY)K')
        ONECH    =   LEN(1) . K
                      .
                      .
                      .
ENPAS   M    ONECH    =                          :F(RETURN)
        ENPAS    =   ENPAS REPLACE(K,&ALPHABET,CA(KEY))
        KEY    =   NEXT(KEY)                     :(ENPAS)

DEPAS   C    ONECH    =                          :F(RETURN)
        DEPAS    =   DEPAS REPLACE(K,CA(KEY),&ALPHABET)
        KEY    =   NEXT(KEY)                     :(DEPAS)
```

5.7 Polyalphabetic substitution as described in the text applies successive alphabets to successive characters of the message. This is computationally expensive, since each character is processed separately. A compromise between economy and security can be obtained by breaking the message into groups of characters and applying an alphabet to a group. The solution of Exercise 5.6 can be modified to do this by replacing ONECH with a pattern that matches a group of characters. Care must be taken to process all the message, since the size of the group may not divide the length.

5.9 In autokey enciphering, the key consists of two parts: a Vigenère Square, and the character used to select the initial alphabet. A suitable defined data type is:

```
        DATA('AUTOKEY(VSQUARE,FIRST)')
```

The enciphering function is:

```
        DEFINE('ENAUTO(M,K)V,CA,C')
        ONECH    =   LEN(1) . C
                      .
                      .
                      .
ENAUTO  V    =   VSQUARE(K)
        CA    =   V<FIRST(K)>
ENAUTN  M    ONECH    =                          :F(RETURN)
        ENAUTO    =   ENAUTO REPLACE(C,&ALPHABET,CA)
        CA    =   V<C>                           :(ENAUTN)
```

The corresponding deciphering function is very similar, differing only in the arguments of REPLACE.

5.10 If the first alphabet for enciphering is selected from the message, there is no way for the deciphering procedure to know what first alphabet to use.

5.11a

```
        K   =   TK('AEILHDCGKJFB','LKJIEFGHDCBA')
```

5.11b

```
        K   =   TK('ABCEDGHIF','ECBADGHFI')
```

5.11c

```
        K   =   TK('ABCEHGFD';'HGFDWECBA')
```

The trick in this case is that the center square is not used for inscription, but it is used for transcription. This square is a null for which W is chosen in the key above. This null cannot be any of the characters used in the transcription string.

5.11d

```
        K   =   TK('ADGHEBCFI','IHGDABCFED')
```

This key contains a repeated character in the transcription string. Even though there is a repeated character, deciphering works properly, since ciphers resulting from this key have the same character in the two positions.

5.14 A revolving grille of size four-by-four can be obtained from the center of the six-by-six grille illustrated in Figure 5.11. An eight-by-eight grille is shown in Figure B.3.

A		B					
			8			2	
		4					
C		7		3			G
	6			1		5	
			9				F
		D				E	

Figure B.3 An Eight-By-Eight Revolving Grille

5.16

```
        HDATE   =   REPLACE('MN/DE/YZ','YZMNDE',MDATE)
```

5.20

```
       DEFINE('BILT(COL,ROW)C,S,S1')
       ONECH   =   LEN(1) . C
                       .
                       .
                       .
BILT   S   =   SIZE(COL)
BILTN  COL   ONECH   =                          :F(BILTR)
       S1   =   S1 DUPL(C,S)                    :(BILTN)
BILTR  BILT   =   BLK(S1,DUPL(ROW,S))           :(RETURN)
```

5.22

```
       DEFINE('QTRBIN(Q)')
                       .
                       .
                       .
QTRBIN QTRBIN   =   COLLATE(REPLACE(Q,'0123','0011'),
+                   REPLACE(Q,'0123','0101'))  :(RETURN)
```

QTRBIN is essentially a biliteral substitution, but with a restricted alphabet. Viewed in this way, the procedure for QTRBIN is:

```
QTRBIN QTRBIN   =   ENBLS(Q,BLK('0011','0101'))  :(RETURN)
```

ENMAS, which is called by ENBLS, must be modified appropriately. In such cases, it is convenient to supply the plain alphabet to the enciphering functions as an argument, instead of assuming &ALPHABET.

5.24

1. transposition
2. substitution
3. substitution
4. noncipher
5. transposition
6. transposition
7. transposition
8. noncipher
9. transposition
10. noncipher
11. transposition*
12. transposition or substitution**
13. transposition
14. combination
15. transposition

*This enciphering method ~~is the~~ produces an identity transformation, but it is a transposition by method.
**This cipher can be considered to be either a transposition or a biliteral substitution.

5.29

```
       DEFINE('CORDER(STRING)C,T,ALPHA')
       ONECH   =   LEN(1) . C
                .
                .
                .
CORDER T   =   TABLE(SIZE(&ALPHABET))
NEXTC  STRING   ONECH   =                          :F(CORD)
       T<C>   =   T<C> C                           :(NEXTC)
CORD   ALPHA   =   &ALPHABET
NEXTA  ALPHA   ONECH   =                           :F(RETURN)
       CORDER   =   CORDER T<C>                    :(NEXTA)
```

5.30

```
       DEFINE('DIVCNT(N)I')
       DIVTBL   =   TABLE()
       PAIR   =   TAB(*I) LEN(2) $ PR ARB . GAP *PR
       I   =   0
READ   LINE   =   LINE INPUT                       :S(READ)
       OUTPUT   =   LINE
NEXTP  LINE   PAIR                                 :F(NEXTI)
       OUTPUT = 'REPEATED PAIR ' PR ' AT DISTANCE '
+      SIZE(GAP) + 2
       DIVCNT(SIZE(GAP) + 2)
NEXTI  I   =   LT(I,SIZE(LINE) - 4) I + 1      :S(NEXTP)
       OUTPUT   =  ;  OUTPUT   =   'TABULATION OF DIVISORS'
       PRINT(SORT(DIVTBL,2,'LT'))                  :S(END)
NONE   OUTPUT   =   'THERE ARE NO DIVISORS'        :(END)
DIVCNT I   =   1
DIVINC I   =   LT(I,N) I + 1                       :F(RETURN)
       EQ(REMDR(N,I),0)                            :F(DIVINC)
       DIVTBL<I>   =   DIVTBL<I> + 1               :(DIVINC)
                .
                .
                .
```

5.32 The ciphers were obtained by using plain and cipher alphabets restricted to the upper-case letters, the period, the comma, and the blank (see Exercise 5.1). Cipher (a) is a transposition. Cipher (b) was produced by a periodic polyalphabetic substitution. Cipher (c) was produced by autokey enciphering. Cipher (d) is a combination cipher obtained by applying the methods for (a) through (c) in series.

6.1 If a $ or * appears at the end of a line, CAP fails to match. Consequently, these characters have no effect if they appear at the end of a line and are left in the line unchanged.

6.2 Automatic reversion to lower case can be prevented by making CCASE a global variable and not changing its value in MAP unless a $ or $$ appears in a line.

6.3 Lower-case letters are not affected by MAP.

6.5a Logical backspacing can be handled as follows:

```
       ERASE   =   TAB(1) BREAK('↖') @P
       DELETE   =   TAB(*(P - 1)) . HEAD LEN(2)
                 .
                 .
                 .
MAP    CCASE   =   LCASE
ERASE  LINE   ERASE                          :F(MAPC)
       LINE   DELETE  =   HEAD               :(ERASE)
MAPC   LINE   CAP   =                    :F(MAPE)S($('MAP' T))
                 .
                 .
                 .
```

The processing statements are placed at the beginning of MAP to permit this correction facility to be applied to the encoding characters * and $.

6.7 Ideally, characters chosen for special meaning in MAP should be convenient to keyboard. This depends on the keyboarding device used. For example, characters that require a case shift are less convenient than those that do not. On the other hand, characters given special meaning should not be likely to appear literally in the text that is to be prepared. For example, * is a poor choice for such a character if mathematical text is to be prepared, and $ is a poor choice if financial text is to be prepared. Decisions depend on a balance of contending factors. If the input character set is small, any choice creates some problems.

6.8 One character can be used to encode an arbitrary number of different things by using the following scheme. Let the meaning of a string of the designated character depend on the number of such characters that occur in succession. For example, if the designated character is a *, a single * can be used to indicate capitalization of a single character, *** can be used to indicate a case shift, and so on. The problem of representing literal strings of *s remains. This can be done by interpreting an even number of consecutive *s as a literal representation of half as many *s. For example, **** stands for the literal occurrence of **. If this is done, only odd numbers of *s are used for indicating encodings. Obviously, the method described above is not appealing in a practical sense.

6.10 Pagination can be suppressed by setting the page depth to a very large number.

6.11 CONTROL can be modified to replace the literal occurrence of ? by an unevaluated expression:

```
       CW   =   '?'
       CONTROL = POS(0) *CW (BREAK(' ') . CSTRING SPAN(' ')
+                REM . VALUE | REM . CSTRING NULL . VALUE)
```

The value of CW can be changed during formatting by a control such as

```
CW     CW   =   VALUE                              :($TYPE)
```

Notice that this permits CW to be assigned a string longer than one character. Another way of changing CW, without the need for a special control string, is to use the EXEC control as described in Section 6.3.4. In fact, even CONTROL can be changed in this manner.

6.19 Several minor modifications to the formatting program are necessary to permit the formatting of more than one document in a single run:

```
                 .
                 .
                 .
NEWDOC CONT   =   0
       JSW   =   1
       COUNT   =   0
       DEPTH   =   55
       WIDTH   =   65
       PAGE.NO   =   1
       MODE   =   'RETURN'
       TYPE   =   'GETA'
PRINT  PUT(GET())                              :S(PRINT)F(DONE)
                 .
                 .
                 .
EOD    CONT   =   1                            :(FRETURN)
                 .
                 .
                 .
DONE   FINISH()
       PAGEND()
       EQ(CONT,1)                              :S(NEWDOC)
END
```

The control string ?EOD signals the end of a document. The resulting failure of GET has the same logical significance as an end of file. The value of CONT, however, distinguishes the two cases, and processing is reinitiated if failure results from ?EOD. Notice that program constants and standard formatting defaults are reset before beginning a new document.

6.22

```
LINE   FINISH()
       PUT(DUPL('-',WIDTH))                              :($TYPE)
```

6.23

```
NO     PAGE.NO   =   VALUE                               :($TYPE)
```

6.24 A control string, PNOFF, to turn off page numbering, can be implemented by the statement

```
PNOFF  NMODE   =   0                                     :($TYPE)
```

A complementary control string, PNON, is implemented by

```
PNON   NMODE   =   1                                     :($TYPE)
```

The default value of NMODE is 1. To complete the implementation of this feature, PUT is modified as follows:

```
                  .
                  .
                  .
PUTT   OUTPUT     =   EQ(NMODE,1) PAGE.NO
                  .
                  .
                  .
```

Notice that this technique only suppresses the printing of page numbers. Pagination still occurs (but see Exercise 6.10) and the page number is still incremented.

6.38 A transfer to ERROR indicates that STREAM contains no blanks up to WIDTH. In this case, the line can be divided arbitrarily at WIDTH or at the first blank after WIDTH. In the latter case, a line longer than WIDTH is printed. The decision as to whether or not such a situation is an error is a matter of judgement.

6.41 Partial justification is a compromise between ragged-right and fully justified formatting. One way to implement partial justification is to establish a "justification leeway". The idea is that if a line to be justified comes within a specified distance (the leeway) of the desired width, no justification is performed. Otherwise, enough blanks are added to bring the line up to the minimum width. The raggedness is limited to the amount of the justification leeway. Implementation of partial justification is left as an additional exercise.

6.42 Mathematical expressions may contain blanks, but dividing an expression over a line or adding justification blanks within it may be confusing or erroneous. SNOBOL4 expressions also have this property. Including such material in running text requires a way of specifying blanks in the formatted document, while preventing such blanks from being given their usual syntactic interpretation in running text. The standard way of solving this problem is to introduce a "special blank", which has no syntactic significance, but which nevertheless prints as a blank. Any character other than the standard blank that does not have a printing graphic can be used. See Appendix A. In ASCII, a special blank can be obtained as follows:

```
        &ALPHABET   TAB(31) LEN(1) . SPBL
```

Some provision must be made for representing special blanks in the input to the formatting program. One way is to include the special blank among the encoded special characters as described in Section 6.1.2. A / followed by a blank is a natural choice for the representation.

6.44 The control string ?PI establishes the amount of paragraph indentation. Special blanks (see the solution to Exercise 6.42) are used to prevent justification from inserting blanks in the indentation. The control string ?P starts a new paragraph and assigns the indentation string to STREAM.

```
PI      PI    =   DUPL(SPBL,VALUE)                        :($TYPE)
P       FINISH()
        (LT(COUNT,DEPTH) PUT())
        STREAM   =   PI                                   :($TYPE)
```

A reasonable default value for indentation might be included in program initialization:

```
        PI    =   DUPL(SPBL,3)
```

6.45 In the as-is mode, a simple control string, ?IA, is sufficient to provide indentation:

```
IA      INDENT   =   DUPL(' ',VALUE)
        MODE    =   'INDENT'                              :(TYPEA)
                   .
                   .
                   .
INDENT GET    =   INDENT GET                              :(RETURN)
```

In the running text modes, the situation is more complicated, since WIDTH has to be adjusted and both J and R formatting must be modified to append the indentation string before returning.

6.47 A control string ?U can be placed after a line to be underlined. The necessary statements are:

```
          OUTPUT('OVER',6,'(1H+,132A1)')
                         .
                         .
                         .
U         OVER     =    DUPL('_',SIZE(OUTPUT))           :($TYPE)
```

Notice that PUT is not used to perform the output. This method only works adequately in processing as-is text. Consideration of the problem for running text is left as an additional exercise.

6.48 A control string ?B can be placed after a line that is to be printed in boldface. The processing statement is:

```
B         OVER     =    OUTPUT                           :($TYPE)
```

See also the solution of Exercise 6.47.

6.68 Sometimes it is convenient to use an abbreviation for an initial part of a word or phrase that is followed by a letter or digit. Plurals provide the most obvious example. Suppose p is an abbreviation for pachyderm. This abbreviation cannot be used to generate pachyderms, since >ps indicates an abbreviation name ps, not p. One solution to this problem is to have a "null character" that terminates abbreviations and is deleted in the process. This solution is unattractive if the size of the input character set is small. Another method is to treat two >s as indicating an abbreviation name consisting of a single character. For example, >>ps indicates an abbreviation name p followed by a literal s. Single character abbreviation names can be used where this feature is desired. Implementation of these features is left as an additional exercise.

6.69 The concept of a predefined or "built-in" abbreviation is useful for generating special characters whose syntactic significance prevents their literal appearance. Such built-in abbreviations are easy to implement. For the character >, such an abbreviation can be obtained by placing the following statement in the initialization portion of the program:

```
          ABBREV<'RB'>    =    '>'
```

The function EXPAND does not reprocess expanded text.

6.73

```
       DEFIN    =   BREAK(',') . NAME LEN(1) REM . DEF
                .
                .
                .
SET    MAP(VALUE)   DEFIN                           :F(CERR)
       GEN<NAME>  =   DEF                           :($TYPE)
```

6.75

1. Changes document input stream to unit 10.
2. Suppresses output of formatted document (but does not stop the formatting process).
3. Causes subsequent formatted output to be punched instead of printed.
4. Reads the next line from the input file and prints it in the formatted document.
5. Turns on tracing of statement failure. Trace messages appear in the formatted document.
6. Terminates execution of the formatting program if less than 1000 units of storage are available to the SNOBOL4 system.

6.76

```
?EXEC  :(GO);R  :(CERR);GO;
```

The statement

```
R  :(CERR)
```

"replaces" the corresponding statement in the formatting program so that the control string R subsequently causes transfer to CERR. The label GO is included to prevent execution of the new statement labeled R during the processing of EXEC.

6.77

```
CODE  CODE(VALUE)                             :F(CERR)S($TYPE)
```

Such a control permits modifying or adding to the formatting program without having the new statements executed. Because of the way that control strings are evaluated, the value must begin with a label or a semicolon. Using this control string, the solution to Exercise 6.76 is

```
?CODE ~~L~~ R  :(CERR)
```

7.3 Using <VARIABLE> as given in the text and treating only binary operators, a grammar is:

```
<OPER>::=-|+|*|/
<PREXP>::=<VARIABLE>|<OPER>(<PREXP>,<PREXP>)
```

7.6 A general solution is to build definitions for <, >, and | into the random sentence generating program, giving them suitable names. An example is

```
        DT<'LEFTBR'>    =    ARRAY(1,ARRAY(1,'<'))
```

which defines a nonterminal <LEFTBR>.

7.8 Using the solution to Exercise 7.6 as a model, a built-in nonterminal <EOL> can be provided. This nonterminal can be handled in several ways. If a character can be preempted for an end-of-line marker, SENTENCE can be constructed as given in the text. When SENTENCE is printed, however, this mark can be used to separate the output into lines. Alternatively, <EOL> can be handled specially, causing the current value of SENTENCE to be pushed onto a stack. Output then amounts to popping items off this stack and printing them.

7.9 A definition may be ended with a |. Because ALTPAT does not match the null string, this situation is interpreted as the end of a definition, not a null alternative. For example, a string of five blanks can be specified by

```
<5BLANKS>::=     |
```

Alternatively, a built-in definition for blanks can be added to the random sentence generator as suggested by the solution of Exercise 7.6.

7.10 An easy way to weight an alternative is to repeat it so that it appears more than once in the grammar. It then appears more than once in the array of alternatives and is more likely to be selected. An example is

```
<EXP>::=<TERM>|<TERM>|<TERM>|<EXP><ADDOP><TERM>
```

On the other hand, a special syntactic notation can be used to indicate that an alternative is to be replicated. One method is to use nonterminal constructions with integer names to indicate replication. For example

```
<EXP>::=<3><TERM>|<EXP><ADDOP><TERM>
```

would be equivalent to the replication given above. A minor modification to the part of the program that constructs the internal representation of the grammar is needed:

```
NEXTA  DEF   ALTPAT   =                                   :F(EOA)
       K  =  1;   J   =   0
       CA  =  ARRAY(CL)
NEXTC  ALT  SUBPAT   =                                    :F(EOS)
       J  =  J + 1
       GT(P,0)                                            :S(NT)
SUBA   CA<J>   =   SUB                                    :S(NEXTC)F(SUBO)
NT     K  =  INTEGER(SUB) SUB                             :S(NEXTC)
       SUB  =  NT(SUB)                                    :(SUBA)
EOS    CA  =  TRUNC(CA,J)
EOSL   K  =  GT(K,0) K - 1                                :F(NEXTA)
       I  =  I + 1
       AA<I>   =   CA                                     :S(EOSL)F(ALTO)
EOA    DT<NAME>   =   TRUNC(AA,I)                         :S(NEXTL)F(ERROR)
```

7.11a It is not necessary for a nonterminal to be removed from its definition by two levels of indirectness. A more desirable structure for <TERM> is shown in Figure B.4.

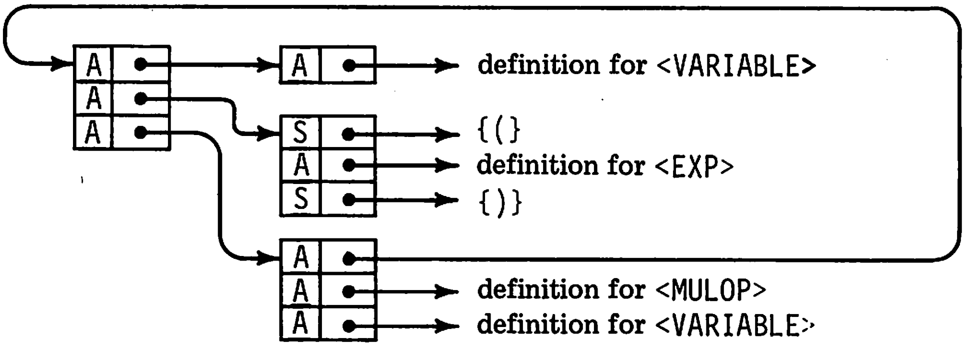


Figure B.4 A More Compact Structure for <TERM>

In each case, a nonterminal subsequent points directly to the array of alternatives for the nonterminal. This relationship is shown explicitly for <TERM>.

The problem with the more compact structure lies in the difficulty of constructing it from the grammar. Assuming that the grammar is given on a sequence of input lines, a nonterminal may appear in a definition before it is defined. In the grammar given in the text, <EXP> is referenced in the definition of <TERM>. This cannot, in general, be avoided. Therefore, it is necessary to have a method of handling forward references. One method is to create the representation of the grammar as given in the text, and then convert it to the more compact form before generating sentences. The method discussed in Section 3.3 can be used to perform the conversion.

7.22 The easiest way to permit designators to be given in any order is to store the input information, without processing it, until simulation is requested. At that time, the stored information can be processed in the desired order.

7.33

```
23
23
23
2<3>
9
SUM,2,56:
4+55
1
```

7.37 The arithmetic computation macros follow directly in the manner of SUM as given in the text. For arithmetic comparison, it is important to recognize that the macros must return the value of 1 or 0 depending on whether or not the comparison succeeds or fails, respectively. Therefore, the corresponding SNOBOL4 predicates cannot be used directly. The necessary additions for a macro EQ are:

```
        DEFINE('MEQ(A1,A2)')
        FORM<'EQ'>   =   *MEQ(ARG<1>,ARG<2>)
                     .
                     .
                     .
MEQ     MEQ    =    EQ(A1,A2) '1'                :S(RETURN)
        MEQ    =    '0'                          :(RETURN)
```

7.40

```
        DEFINE('CPY()')
        FORM<'CPY'>    =    *CPY()
                     .
                     .
                     .
CPY     DEFN<ARG<1>>    =    DEFN<ARG<2>>
        FORM<ARG<1>>    =    FORM<ARG<2>>       :(RETURN)
```

7.49 The set of break characters used by STREAM can be specified in the call to EXPAND. The necessary modifications to MP are:

```
        DEFINE('EXPAND(CHARS,BR)H')
        BCHARS   =   CALL OQUOTE
        CCHARS   =   BCHARS ASEP TERM
        QCHARS   =   OQUOTE CQUOTE
        STREAM   =   BREAK(*CHARS) . H LEN(1) . C
```

Using this method, only those characters that have syntactic significance in a given context are processed. The way EXPAND is called with this modification is illustrated by LIT:

```
LIT     LIT    =    EXPAND(QCHARS,QBR)           :(RETURN)
```

The statement labeled NOOP and the transfer table entries referring to it can be eliminated.

7.52a

```
        MGO    =    ' :F(M1)S<EXEC>'
                   .
                   .
                   .
M       EXEC   =    CODE(SRC TEMPLATE MGO)     :F(CERR)S<EXEC>
M1      OUTPUT   =   CURRENT                   :(EDIT)
```

7.52c

```
B       CURRENT   =   EVAL(TEMPLATE) CURRENT :F(CERR)S(EDIT)
```

7.52f

```
N       CURRENT   =   CURRENT IFILE            :S(EDIT)F(EOF)
```

REFERENCES

General

1. Sammet, Jean E. *Programming Languages: History and Fundamentals.* Prentice-Hall, Englewood Cliffs, N.J. 1969.
2. Griswold, R. E., J. F. Poage, and I. P. Polonsky. *The SNOBOL4 Programming Language*, 2nd ed. Prentice-Hall, Englewood Cliffs, N.J. 1971.
3. Griswold, Ralph E. and Madge T. Griswold. *A SNOBOL4 Primer.* Prentice-Hall, Englewood Cliffs, N.J. 1973.
4. Farber, D. J., R. E. Griswold, and I. P. Polonsky. "SNOBOL, A String Manipulation Language," *Journal of the Association for Computing Machinery*, Vol. 11, No. 1 (January, 1964), pp. 21–30.
5. Farber, D. J., R. E. Griswold, and I. P. Polonsky. "The SNOBOL3 Programming Language," *Bell System Technical Journal*, Vol. XLV, No. 6 (July-August, 1966), pp. 895-944.
6. Forte, Allen. *SNOBOL3 Primer.* MIT Press, Cambridge, Mass. 1967.
7. Dewar, Robert B. K. "SPITBOL Version 2.0," SNOBOL4 Project Document S4D23. Illinois Institute of Technology, Chicago, Ill. February 12, 1971.
8. Santos, Paul Joseph Jr. "FASBOL, A SNOBOL4 Compiler," Memorandum No. ERL-M314, Electronics Research Laboratory, University of California, Berkeley, Calif. December, 1971.

9. Gimpel, James F. "SITBOL Version 3.0," SNOBOL4 Project Document S4D30b. Bell Laboratories, Holmdel, N.J. June 1, 1973.

Pattern Matching

10. Herdan, G. *Quantative Linguistics.* Butterworths, Washington. 1964.

11. Deutsch, L. Peter and Butler W. Lampson. "An Online Editor," *Communications of the ACM*, Vol. 10, No. 12 (December, 1967), pp. 793–799.

12. Hopcroft, John E., and Jeffrey D. Ullman. *Formal Languages and Their Relation to Automata.* Addison-Wesley, Reading, Mass. 1969.

13. Backus, J. "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference," *Proceedings of the International Conference on Information Processing.* UNESCO (June, 1959), pp. 125–132.

14. Reid, Constance. *From Zero to Infinity.* Thomas Y. Crowell, New York. 1964.

Functions

15. Churchill, Ruel V. *Introduction to Complex Variables and Applications.* McGraw-Hill, New York. 1948.

16. Maurer, H. A. and M. R. Williams. *A Collection of Programming Problems and Techniques.* Prentice-Hall, Englewood Cliffs, N.J. 1972.

17. Kleene, Stephen Cole. *Introduction to Metamathematics.* Van Nostrand, Princeton, N.J. 1952.

18. Knuth, Donald E. *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms.* Addison-Wesley, Reading, Mass. 1969.

19. Shell, D. L. "A High Speed Sorting Procedure," *Communications of the ACM*, Vol. 2, No. 7 (July, 1959), pp. 30–32.

20. Knuth, Donald E. *The Art of Computer Programming, Vol. 3, Searching and Sorting.* Addison-Wesley, Reading, Mass. 1973.

Structures

21. Griswold, Ralph E. *The Macro Implementation of SNOBOL4, A Case Study of Machine-Independent Software Development.* W. H. Freeman, San Francisco. 1972.

22. Gimpel, J. F. "A Theory of Discrete Patterns and Their Implementation in SNOBOL4," *Communications of the ACM*, Vol. 16, No. 2 (February, 1973), pp. 91–100.

23. Knuth, Donald E. *The Art of Computer Programming, Vol. 1, Fundamental Algorithms.* Addison-Wesley, Reading, Mass. 1968.

24. Berztiss, A. T. *Data Structures, Theory and Practice.* Academic Press, New York. 1971.

Applications in Mathematics

25. Gardner, Martin. "Mathematical Games," *Scientific American*, Vol. 223, No. 2 (August, 1970), pp. 110–111.

26. Niven, Ivan. *Irrational Numbers.* Carus Mathematical Monograph Number Eleven, The Mathematical Association of America, John Wiley and Sons, Inc., Rahway, N.J. 1956.

27. Bobrow, Daniel G., ed. *Symbol Manipulation Languages and Techniques.* North-Holland, Amsterdam. 1968.

28. Papers from the Second Symposium on Symbolic and Algebraic Manipulation. *Communications of the ACM*, Vol. 14, No. 8 (August, 1971).

Cryptography

29. Ball, W. W. Rouse. *Mathematical Recreations & Essays.* Macmillan, New York. 1962.

30. Gaines, Helen Fouché. *Cryptanalysis, A Study of Ciphers and Their Solution.* Dover, New York. 1956.

31. Laffin, John. *Codes and Ciphers.* New American Library, New York. 1967.

32. Pratt, Fletcher. *Secret and Urgent.* Blue Ribbon Books, Garden City, N.Y. 1942.

33. Moore, Dan Tyler and Martha Waller. *Cloak and Cipher.* Bobbs-Merrill, New York. 1962.

34. Sinkov, Abraham. *Elementary Cryptanalysis, A Mathematical Approach.* Random House, New York. 1968.

35. Smith, Laurence Dwight. *Cryptography, The Science of Secret Writing.* Dover, New York. 1955.

36. Control Data Corporation. *COMPASS Reference Manual, Control Data Corporation 6000/7000 Series Computer Systems.* Control Data Corporation technical publication 60279900. 1971.

37. Feinstein, Ameil. *Foundations of Information Theory.* McGraw-Hill, New York. 1958.

38. IBM Corporation. *IBM System/360 Operating System Assembly Language.* IBM Systems Reference Library order number GC28-6514-7. 1970.

Document Preparation

39. Noll, L. W. "A Text Formatting Program for Phototypesetting Documents," Bell Laboratories technical report. Holmdel, N.J. April, 1971.

40. Tessler, Larry. "PUB, the Document Compiler," Stanford Artificial Intelligence Project Operating Note 78. September, 1972.

41. Digital Equipment Corporation. *DECsystem-10 Assembly Language Handbook.* Digital Equipment Corporation Program Library order code DEC-10-NRZB-D. Maynard, Mass. 1972.

42. LeSuer, W. J. "TEXT360," IBM Corporation Program Information Department document 360D-29.4.001. 1969.

Additional Applications

43. Strachey, C. "A General Purpose Macrogenerator," *Computer Journal,* Vol. 8, No. 3 (October, 1965), pp. 225–241.

44. Mooers, Calvin N. "TRAC, A Procedure-Describing Language for the Reactive Typewriter," *Communications of the ACM,* Vol. 9, No. 3 (March, 1966), pp. 215–219.

45. Brown, P. J. "Macro Processors and Their Use in Implementing Software," University of Cambridge Doctoral Dissertation. March, 1968.

46. Hall, Andrew W. "The M6 Macro Processor," Bell Laboratories technical report. Murray Hill, N.J. June, 1971.

47. IBM Corporation. *System/360 Administrative Terminal System—OS. Terminal Operations Manual.* IBM Corporation Technical Publication H20-0589-1. 1968.

48. Gimpel, J. F. "An Essay on Context Editors and an Introduction to SNO-ED," Bell Laboratories internal memorandum. Holmdel, N.J. November 15, 1969.

INDEX OF DEFINED FUNCTIONS

A, 241
ADD:
  complex addition, 27-29
  symbolic addition, 113, 256-257
ADDCXR, 253
ADDL, 81
ADDLRG, 100-101
ADDR, 81
ADDRTL, 252
ADDSIB, 88
ADDSON, 88
ADEPTH, 242
ADEXPT, 105
ADPOLY, 104-105, 106
APROFL, 241-242

BEXP, 81
BILT, 263
BTREE, 81

CENTER, 238
CFRAC, 256
CITE, 41-43, 45, 245, 246
COLLATE, 143-144, 147, 148, 238
COMPRESS, 25
COPYBT, 82
COPYL, 74-76, 250
COPYR, 251
CORDER, 264
CPXSTR, 28
CPY, 273
CPYPOL, 255
CXRSTR, 253

D, 111-115, 117, 259
DATINF, 258-259
DEBLS, 148, 149
DECIPHER, 123-125
DECOLLATE, 144-145, 147
·DEF, 216
DELELE, 77
DELETE, 69, 70, 71, 72, 239
DEMAS, 127
DEPAS, 261
DET, 138
DIV, 114, 119, 256-257
DIVCNT, 264

ENAUTO, 261
ENBLS, 148, 263
ENCAES, 260
ENCIPHER, 123, 124, 125

ENKAT, 128, 129, 132, 146, 260
ENMAS, 127, 148, 263
ENPAS, 261
ENT, 138
EXEC, 215, 218, 219
EXP:
  expression conversion, 89
  symbolic exponentiation, 114, 119, 256-257
EXPAND:
  formatting, 187, 189, 269
  macro processing, 213-215, 219, 273-274
EXTEND, 240

F, 30, 241
FACT, 241
FINISH, 179-181

GCD, 96
GET:
  formatting, 169, 171, 175
  word analysis, 41-42, 45, 245, 246
GRAM, 37-38, 39, 244

IF, 218-219
INDEX, 203
INFDAT, 258
INFIX, 243
INFTRE, 116
INSERT, 69, 71, 72
INTERIOR, 251
INTLRG, 253

JUST, 177-178

LEXINS, 83
LEXPRT, 85, 250
LLIST, 90
LPAD, 22-24, 237-238
LRGSTR, 253
LSIB, 251

MAP, 164-165, 167, 168, 169, 187, 191, 265
MEQ, 273
MLPOLY, 105, 106
MUL, 114, 119, 256-257
MULLRG, 254
MULRTL, 97

NEXT, 90
NEXTG, 38, 39, 244-245
NEXTS, 244
NULL, 218

OR, 259

PAGEND, 173, 179
POLSTR, 255
POP, 61-62, 64, 66-68, 193
PREFIX, 33, 34, 242
PRINT, 41, 44, 46, 190, 246
PRINTL, 250
PRTSTK, 67-68, 249
PUSH, 61-62, 64, 66-68, 247
PUT, 169-170, 173

QTRBIN, 263
QUEUE, 69, 249

RANCHR, 244
RANDOM, 36, 39, 198, 243-244
RANSTR, 244
REDUCE, 97, 98, 252
REVERSE, 137, 235, 239, 243
ROTATE, 131, 239
RPAD, 22, 237-238
RTLSTR, 252

SCOPY, 248
SELECT, 193, 194, 198
SINIT, 248
SKIP, 218
SNAPSHOT, 205-206
SORT, 41, 43-44, 46, 246-247
SPANN, 235
SPECIAL, 218

SPUSH, 193, 194, 198
SSIZE, 247, 248
STACK, 61, 64
STKQUE, 249
STPOLY, 104
STRCPX, 28
STRLRG, 99
STRRTL, 252
SUB, 114, 119, 256-257
SUBST, 216
SWITCH, 67

TREE, 89
TRIM, 240
TRML, 194
TRUNC, 198, 240
TSIZE, 251

VSQ, 131

WLIST, 249

SUBJECT INDEX

Abbreviations, 185–187, 190, 191, 209, 269
Ackermann's function, 31–32, 34, 241
&ALPHABET, 125–126, 136, 150, 153, 162–163, 232, 244, 263
Alphabets used in ciphers, 125–126, 259, 261 (*see also* Cipher alphabets)
Alternatives, 7–8, 192–194, 198, 271
Arrays, 47, 48–50
  use of, 63, 70, 73, 104–105, 196, 198, 209
ASCII, 162, 163, 226, 228, 268
As-is text, 174, 180, 268
Associativity of operators, 108
Autokey enciphering, 131, 132, 261, 264
Balanced strings, 12, 32, 109, 215, 233
BCD, 162
Biliteral substitution, 147–150
Binary operators, 79–80, 89, 108–111, 116, 270
Binary trees, 78–85, 90, 250
Blanks, special, 268
BNF, 7–12, 192–200, 232, 233, 270–272
Boldface text in formatting, 167, 183
Boolean expressions, 120
Buffered input in formatting, 175–176, 178
Caesar's Cipher, 127
Capitalization in formatting, 163–165
Case shifting in formatting, 164, 166, 167, 265
CDC 6000, 126, 136–137, 150, 152, 153, 162
Centering in formatting, 180

Character encoding, 165–166, 168, 265–266
Character frequencies, 153–154, 157, 168
Character set sizes, 137
  in ciphers, 126
Character sets in formatting, 161–162
Characters in ciphers, 125–126
Chronological order, 43
Cipher alphabets, 126–129, 130, 132, 260, 264
Ciphers, 123
  types of, 124
CODE, use of, 190, 191, 200, 223–224, 270
Codes, 123
Columnar transposition, 134, 146
Combination ciphers, 124, 263, 264
Command languages, 221–222
Command processors, 223
Commands, 223 (*see also* Control strings)
Complex numbers, 26–29, 30, 59, 95, 97, 98, 252
Concordances, 40–45, 184, 245, 246
Conditional evaluation, 217, 220
Context dependence, 6, 12, 232
Context editors, 220–222
Continued fractions, 106–107, 255–256
Control strings, 170–174, 181, 182, 183, 185, 189, 191, 266, 267, 268,
    269, 270
CONVERT, use of, 105
Copies of objects, 57, 76
COPY, use of, 50, 74
Copying structures, 75–76, 81–82, 90, 93, 102, 106
Cryptanalysis, 152–157
Cryptography, 122
Curlicues, 92, 256
Data type representation of expressions, 118–120, 257–259
Data types, built-in, 48–55
DEC-10, 162
Deciphering, 137, 148, 156–157
Decrypting (*see* Cryptanalysis)
Defined data types, 47, 55–60
  use of, 26–28, 65, 72, 73, 76, 86, 96, 118–119, 138, 147, 195, 204, 247,
    252, 254, 257, 260, 261
Delimiters, 6
Deques, 77
Derivatives, 111
Differentiation, 95, 111–120

Digrams, 151, 155, 158 (*see also* *n*grams)
Display Code, 162, 226, 227
Document preparation, 160
Doubly-linked lists, 76–77
EBCDIC, 151, 162–163, 226, 229–230
ED4, 221–224
Editing, 221–224
Encoding characters (*see* Character encoding)
&ERRLIMIT, 23
Error handling, 172, 181, 220, 224
Escape characters, 168
EVAL, use of, 218
Expressions, 88–89, 107–110, 111–120
Factorial, 30, 34, 241
Father nodes, 86
Fibonacci numbers, 30, 31, 34, 241
Finite state machines, 209
Formatting, 160
  horizontal, 169–170, 174, 178
  vertical, 169, 174, 178
Fullscan mode, 10, 11, 232, 234, 235
Functional notation, 80, 88
Functions, built-in, 21
Generators, 34–39, 188–189, 193
Goals, 7–8
Grammars, 7–12, 192–200, 232–234, 270–272
Greatest common divisor, 96
Grilles, 138–142
  overlay, 140, 141
  revolving, 138–140, 146, 262
Heuristics in pattern matching, 9–10
Hollerith literals, 6, 232, 247
Hyphenation, 176, 184
IBM 360/370, 136–137, 151, 162
Indentation, 268
Indexing, 184–185, 190
Indirect references, 51, 55
Indirect transfers, 172, 213
Infix form, 34, 110, 116, 119, 120
Infix notation, 32–33
Inscription strings, 134–135, 142
Integers, 48, 232
Integers, large, 98–102, 106, 253, 254
Irrational numbers, 107

Justification in formatting, 176–179, 183, 267, 268
Keeps in formatting, 174, 183
Keyed alphabets, 128
Keyed cipher alphabets, 129
Keys in ciphers, 125, 126, 138, 147, 148, 150, 262
Languages, 7, 192, 200
Leaf node, 78
Left son nodes, 86
Lexical insertion, 83–84
Lifo lists (*see* Stacks)
Line printers, 166
Linear congruence sequences, 35–36
Linked lists, 73, 99, 100, 102, 106, 250, 254, 255 (*see also* Singly-linked lists *and* Doubly-linked lists)
Lists of items, 1, 12, 234
Loops in structures, 91–93
Macro calls, 210–211
Macro definitions, 211
Macro processors, 209–220, 225, 273–274
Macros, 209–220, 273–274
Message inscription, 133–142, 145–146
Message transcription, 133–142, 145–146
Monoalphabetic substitution, 127, 129, 154, 157, 259
Monoliteral substitution, 126
Monospaced output, 166
MP, 210–220, 273
*n*grams, 37, 39, 45, 154, 155, 158 (*see also* Digrams)
Names, 54–55
Nonterminals, 7–9, 193–195, 197, 198, 200, 232, 233, 271
Null strings, 52
Nulls in ciphers, 133, 140, 141, 145
Numbering in formatting, 188–189, 191, 270
Numbers, 4–6
Operator extension, 28–29
Operator notation, 28, 32, 33, 79–80, 88
Operators (*see* Binary operators *and* Unary operators)
OPSYN, use of, 28, 29, 218, 259
Output formatting, 22
Overstriking in formatting, 167, 183
Page numbers in formatting, 170, 182
Pagination, 170, 181, 266
Palindromes, 19, 39, 103, 235
Parameter markers in macro definitions, 211
Parentheses in expressions, 80, 108, 110, 116, 120

Pattern matching, 1, 108–110, 176, 196, 231–232, 233, 234, 236–237
Pattern-matching direction, 109, 144, 176, 177, 240
Patterns (*see* Pattern matching)
Phototypesetting, 166
Plain alphabets, 126, 127, 259, 263, 264
Pointers, 58, 91
Polyalphabetic substitution, 129–132, 154, 155, 156, 158, 259, 260, 261, 264
Polygraphic substitution, 150–151, 154, 157
Polyliteral substitution, 147–150, 151, 154, 157, 158
Polynomials, 103–104, 105, 106, 200, 255
Precedence of operators, 108–109
Prefix form, 34, 89, 119, 242
Prefix notation, 32–33
Product ciphers, 124
Program modification during execution, 189–191, 225, 270
Propositional calculus, 120
Pseudo-random numbers, 34–36, 39, 198
Queue functions, 73
Queues, 69–73, 74, 249
Quickscan mode, 9, 11, 144, 234
Quotation marks, 212, 214
Ragged-right formatting, 174, 175, 179, 180, 267, 268
Random generation, 34–36, 39, 132, 193, 194, 198, 199, 200, 243
Rational numbers, 95–98, 102, 118, 252, 255
Real numbers, 48, 102
Recognizers, 11, 201
Recursion in pattern matching, 12, 13
  left, in pattern matching, 10, 11
Recursive functions, 30–34
  limits in SNOBOL4, 33, 241, 243, 250
Recursive processing, 77, 85, 98, 100, 209, 213, 243, 250
Recursive relations in data, 32, 33, 109
REPLACE, use of, 127, 136–138, 142–145, 261
Right sibling nodes, 86
Rings, 71, 73, 91–92
Rivers in formatting, 177
Root nodes, 78
Route transposition, 132–133, 134
Running text, 174, 175, 176, 178, 269
Sentences, 7, 192–194, 199, 200
Shell sort, 43–44
Simplification of expressions, 113, 115, 117, 120
Simulation, 200–202, 209
Singly-linked lists, 73–74, 77

Skipping lines in formatting, 170, 171, 173, 174, 182
Sorting, 43, 82, 83, 85, 246
Stack functions, 66, 68
Stacks, 60–68, 73, 74, 193, 194, 248, 249
Statistical aspects of language, 153
Strings, 51–52
  uniqueness of, 52, 53
  use of, 15, 16, 27, 61, 62, 68, 73, 96, 104, 255
Subsequents, 8, 192, 195, 197
Substitution ciphers, 124, 126, 147, 263
Successor functions, 35–39, 90, 191
Symbolic operations, 113, 115
Tables, 53, 54
  use of, 64–65, 104–105, 195–196, 205, 213, 216, 257–259
Tapes, Turing machine, 201–202, 204, 205, 208
Terminals, 7–8, 192–194, 197
Text editing, 160–161, 220
Tic-tac-toe, 16–19, 236–237
Transcription strings, 134, 135, 141–142
Transition functions, 202
Transmission, in cryptography, 123
Transposition ciphers, 124, 128, 132, 154, 158, 263, 264
Tree representation of expressions, 115–117, 118
Trees, 86–90, 115–117, 120, 251
Turing machines, 200–209, 273
Typesetting, 166
Unary operators, 88, 89, 198
Underlining in formatting, 167
Unevaluated expressions, 9–11, 12, 232
Vigenère Squares, 129–131, 260

(continued from front flap)

suggest next steps to be taken toward more significant material, or indicate related areas not covered in the text. Solutions to many of the exercises are given in Appendix B. These solutions, and accompanying discussions, are intended to supplement the text. Some further exercises are suggested in the solutions. . . ."

from the author's Preface

RALPH E. GRISWOLD, former head of the programming research department at Bell Laboratories, is presently Professor and Head of the Department of Computer Science at the University of Arizona.

PRENTICE-HALL, Inc.
Englewood Cliffs, New Jersey

774 • Printed in U.S. of America

Also of interest . . .

IMPLEMENTING SOFTWARE FOR NON-NUMERIC APPLICATIONS

by WILLIAM M. WAITE

"As the computer becomes a commonplace tool in many fields and as the complexity of our systems increases, we find that more and more of the difficult problems are non-numeric," says the author. To meet the growing need for tools with which to confront non-numeric problems, he has written this unique volume.

Implementing Software for Non-Numeric Applications provides a detailed treatment of techniques required for list processing, dynamic storage management, and string manipulation. The emphasis on *engineering* of software to perform non-numeric tasks includes time and space requirements for various algorithms as well as tradeoff options.

Published 1973 *510 pages*

A SNOBOL4 PRIMER

by RALPH E. GRISWOLD and MADGE T. GRISWOLD

This primer introduces readers to the complex and sophisticated language of SNOBOL4 at a relatively simple level. Presenting the material in a manner that does not require a background in mathematics or engineering terminology, the authors have drawn upon the interesting and unusual aspects of SNOBOL4. Examples are given for English-language text and the manipulation of symbolic expressions.

Features: Approaches computing in general and SNOBOL4 using a non-mathematical, non-technical language • Provides exercises, examples, charts, and illustrations enabling you to use the book as a self-contained, progressive learning unit • Requires no previous technical or mathematical background • Stimulates interest with examples that manipulate English-language text and symbolic expressions.

Published 1973 *184 pages*

INTRODUCTION TO DATA STRUCTURES AND NON-NUMERIC COMPUTATION

by PETER C. BRILLINGER and DORON J. COHEN

Here is a comprehensive discussion of data representations and data structures, followed by a detailed study of operations and applications with character strings, linearly linked lists, graphs, and trees. The book includes a consideration of algorithms for traversing trees and implementing recursive routines with the use of pushdown stacks. This in-depth study of techniques takes the mystery out of list processing by teaching the basic methods usually used by high-level list processing languages. It concludes with an elementary discussion of programming language translation, covering syntactic analysis, object code generation, and macro processors.

Published 1972 *629 pages*

PRENTICE-HALL, Inc.
Englewood Cliffs, New Jersey

0-13-8530