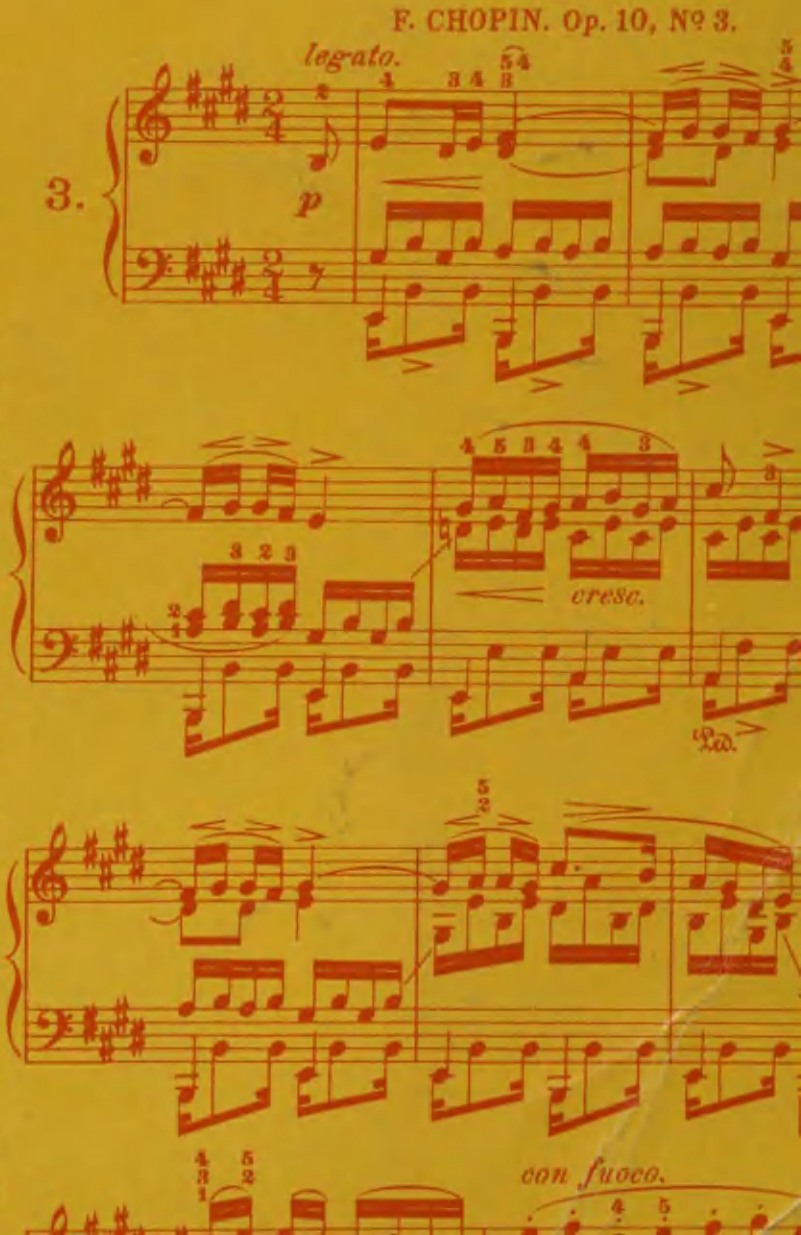
ETUDES
FOR PROGRAMMERS

CHARLES WETHERELL



ETUDES

FOR PROGRAMMERS

CHARLES WETHERELL

Computing Science Group
Dept. of Applied Science
University of California, Davis

PRENTICE-HALL INC. *Englewood Cliffs, New Jersey 07632*

Library of Congress Cataloging in Publication Data

WETHERELL, CHARLES
Etudes for programmers.

Includes bibliographical references and index.
1. Electronic digital computers — Programming — Problems, exercises, etc. I. Title.
QA76.6.W477 001.6'42 77-13961
ISBN 0-13-291807-2

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. LIMITED, *Sydney*
PRENTICE-HALL OF CANADA, LTD., *Toronto*
PRENTICE-HALL OF INDIA PRIVATE LIMITED, *New Delhi*
PRENTICE-HALL OF JAPAN, INC., *Tokyo*
PRENTICE-HALL OF SOUTHEAST ASIA PTE. LTD., *Singapore*
WHITEHALL BOOKS LIMITED, *Wellington, New Zealand*

Contents

Preface

Programming is a craft, and programmers must attain a standard of craftsmanship. Much programming is done in *cottage shops*—that is, in small shops with meager tools, much work done by hand, and learning attained from other laborers, by chance, and often not at all. Just as guilds formed in the Middle Ages partly to train young workers and to improve professional standards, so programming is now taught in colleges and universities and far fewer programmers learn (or fail) by the "once more unto the breach" method. But the academics have also discovered that a craft cannot be taught well by teachers alone; the guild apprenticeships had considerable merit.

In a classic apprenticeship the candidate spent many years doing menial tasks while absorbing fundamental techniques of the trade from more experienced workers in the shop. Gradually the apprentice was given more technical responsibility and, after a formal test of skills, eventually became a journeyman certified competent for all ordinary jobs in the trade. The journeyman traveled the world and, if the muses allowed, one day presented a masterpiece to the guild and became a craftsman of the highest rank—a master of the guild. The works of these masters, even of the most utilitarian sort, have often come down to us as some of the greatest works of human creativity.

Today, a novice programmer may well dispense with seven years of sweeping up card punch chips, and simple technical knowledge can be acquired more easily from lectures and reading than from watching over a working programmer's shoulder. But one may not dispense with some "hands on" time spent on realistic programming tasks—time needed to connect and solidify principles and methods—time simply for practice. No student would expect to read any number of books about cabinetry and be able to create even a good imitation of a Chippendale. Why, then, expect to read a programming manual or two and be able to turn out a well-formed program?

Formal training institutions for programmers—colleges, trade schools, corporate training programs—are adding apprentice training by means of programming workshops, laboratories, and major projects in ordinary courses. Instructors in such courses need problems to supply the busy apprentices. These etudes for programmers should fill the need. Each etude stands alone with its own background information, problem statement, and suggestions for solution. Most etudes allow variations so that an instructor may tailor problems to local conditions.

The variety of these compositions is large. Some etudes are done largely in the head (Chapter 9) and others are mostly implementation effort (Chapter 12); some etudes are short (Chapter 16) and others very long (Chapter 6); some etudes use well-known programming techniques (Chapter 5), and others can be continually improved by better techniques (Chapter 2). Chapters 25, 26, 27, and 28 form a connected unit for use in a language and systems course. These etudes should be done by groups of students who can also learn some of the need for software engineering (traditionally students have not learned that working in a group is different from working alone). Chapters 5, 6, 13, 17, 19, and 25 may be good materials for a course in computer simulation. Similarly, Chapters 6, 11, 14, 19, and 20 are all drawn from artificial intelligence. Traditional problems from computer science are well represented, of course.

Students in a laboratory course will appreciate carefully written problem descriptions. (How many have vowed never to come near a computer again as they struggled to decipher a badly written specification printed in faint blue mimeograph ink?) If students may choose problems freely—as they do in the course from which the book was developed—they will also appreciate having a much wider choice than any one instructor has the energy to provide. Of course, students working alone to improve their skills have not previously had a collection of this magnitude available. Some students may be able to use selected etudes as a source for methods even after they leave school.

As is usually the case, this book could not have been written without the help of many people. George Michael first suggested the idea of teaching by problems (an idea current in many other schools). Students in several classes were willing subjects and suggested both corrections and many new topics. Hank Moll wrote the text format routine that was used to produce the drafts and was always willing to change it as needed; John Beatty simultaneously wrote the programs to print the drafts with pretty characters and illustrations. Together these two made it easy to see what the final version would look like, a mighty aid to composition. Many other friends read, commented, criticized, and encouraged the work. Finally, the keypunch operators at Lawrence Livermore Laboratory never complained about poor handwriting—instead they quickly returned the finished cards and corrected the spelling mistakes. Without all of these helpers, the work could not have been done; with their help it has been.

Charles Wetherell

What's It All About, Alfie?

or...

HOW TO USE THIS BOOK

The difficulty with teaching programming is that it cannot be taught. The difficulty with learning programming is that it is so much work. A teacher can help, lecture, criticize, guide, smooth the path. A student can take notes, memorize, read, pass tests, discuss until two in the morning.[1] All this effort will be meaningless if the student does not practice by actually writing programs, because programming, like other skills, can be acquired only by practice. Furthermore, the practice must be on "real" programs and not on the simplistic material found in most programming language manuals. Noodling away at *Chopsticks* will not make one another Rubinstein—no more will noodling at APL make one a master programmer. So we provide this book of sizable problems that are suitable as training projects for the novice programmer who wishes to become first a journeyman and then a master.

The abilities of a programmer closely resemble those of an essayist. As with the essayist, orthography and grammar are parts of the skill but not, as popular opinion would have it, the most important parts. Much more valuable are observation, research, analysis, and a pleasing expressiveness. Here is a list of talents vital to programmers (and essayists).

> The ability to read and understand a problem description and to grasp the formulator's desires (not always easy, since both problems and proposers are often vague).
>
> The ability to extract the difficulties and ignore irrelevancies.
>
> The ability to recognize where theory can be applied and the discretion to apply it oneself or to ask an expert to intercede.
>
> The ability to break a problem into manageable and independent pieces and to understand the relations among these pieces.

[1] We shall call the reader a "student," but this usage need not frighten those who are not members of an academic community. It is possible to learn to program alone; we hope to encourage readers who must struggle in solitude by providing realistic problems to grapple with. Be warned, however, that a teacher will help immeasurably.

The ability to judge the cost of proposed solutions in programming effort, computer resources, and user satisfaction and to balance these costs.

The ability to build partial solutions into coherent and elegant complete solutions.

The ability to express solutions in graceful and straightforward language, natural or artificial, that persuades both humans and computers of the solution's correctness.

Finally, the ability to disengage the ego and try an alternate approach (or even an alternate problem) when a first attempt fails.

It is precisely because these abilities are so complex that they can only be learned by experience. Etudes provide practice in specific technical skills and lead students by experience toward the general abilities required of programmers.

Composing an etude, however, it not as easy as might be supposed. All too often the problems in programming books are mere finger exercises. Although useful in developing flexibility in the use of simple language structures, they seldom have the "artistic merit" that the dictionary claims better etudes should have. And even though an etude is a "study on a single technique" (same dictionary), a good one must be large enough so that interactions between the chosen technique and other areas of programming can occur. All of this suggests that we draw problems from the real world. But real-world problems are filled with niggling detail, require mounds of input, produce reams of output, and are changed every other day because management cannot make up its mind. A student who could learn in a production shop would emerge a more saintly person, but too many programming trainees end up broken, bitter, and despondent. An etude must lie in the middle ground between real practice and triviality.

In fact, many etudes emerge from two areas—games and computer science. These areas share a number of useful characteristics. Most computer programmers are interested in both applications (they had better be interested in computer science). Because of shared culture, most games are easy to explain, and, of course, computing applications should be pretty well understood already. Quite often the behavior of a game program or a compiler, say, can be rigidly defined so that correctness is testable. Input is usually small and easy to generate; output is readily comprehensible. Both seem to require the most complex algorithms and data structures so that no demand from an application program is likely to shock the student later. Finally, both areas seem to lead to consideration of the computer as a powerful abstract rational entity in its own right (a view encouraged in the field of artificial intelligence), and we have probably been somewhat biased in our problem selection by a lifelong interest in rational machines. There are, of course, many problems from other application areas. The choice is limited primarily by the ease of explaining the problem situation. And to those students who may be bothered by the frivolity of some etudes, remember that Haydn made a symphony from a nursery rhyme.

HOW TO PLAY AN ETUDE

We assume that the novice attempting an etude has already written a few programs and knows at least one language moderately well. No attempt will be made to teach specific programming techniques, data structures, or languages. If a problem does require some uncommon knowledge, there will be enough discussion to explain the difficulty, and the bibliography will point to sources for more information. In addition, we will not specify any one programming style or discuss structured programming. It is probable that most readers are in a course or laboratory and that they will receive guidance from a teacher. Nevertheless, the references at the end of this chapter include materials on programming and style for those working alone.

Each etude is divided into sections (some optional). The first section describes the real-world situation, and the second the specific program to be written; normally the description is long-winded and the specification quite short. Following these is a discussion of the practical difficulties that are

apt to be encountered and hints for solutions. Only critical difficulties are considered. Next are sections that discuss appropriate languages and the time that the etude should require.[2] Time estimates, which are based on first-year graduate students spending one-fourth of their time on the problem, may be too short for programmers without graduate students' noted dedication. Long turnaround or unpredictable computer availability may also lengthen time estimates. Etudes often end with suggestions for extended work and an annotated bibliography. A solution will generally be more valuable to the student if it is reinforced by outside reading.

A completed etude obviously includes a clear and disciplined program written and commented in a style appropriate to the problem and the source language. Yet more is needed. Enough test material should be included to demonstrate the use of the program and its reaction to extreme cases and error conditions. A short prose description of the solution methods, with special attention to original algorithms or data structures, should be external to the program. Along with the description, the programmer should provide at least an informal plausibility demonstration that the program is correct (concentrate on critical steps if time is limited). Finally, there should be an accounting of both human and computer resources with attention paid to reasons for the costs and a statement of what the programmer has learned doing the problem (which is easy to write if it answers the question "What will I do differently next time?"). This documentation may sound excessive, but one of the lessons to be learned is when to quit. Short problems should not be overwhelmed with documentation. One teacher we know gives 40% of the grade for being convinced that a program is correct (whether or not it is), 50% for the ease with which he is convinced, and only the last 10% for superior programming. Students scoring 80% or more are doing very well. And since part of the documentation is the result of computer runs, such a score means that both computer and instructor have been impressed by the solution.

NOTES FOR TEACHERS

This book was originally written for a graduate immigration course in computer science. The course lectures cover a broad spectrum, including programming languages and techniques, machine architecture, data structures, algorithms, and some theory. Lecturers may choose some problems as illustrations (for example, map coloring to teach PASCAL), but generally students are on their own when doing problems. The only requirement imposed is that total time estimates for the problems attempted be at least as large as the duration of the course. So the book places very little structure on a course that uses it. On the other hand, four problems were especially written for a compiler course and provide a connected study of language implementation. Some others present several major aspects of game playing. Still others might form a laboratory in commercial or simulation programming. An interested instructor should be able to find problems in any area except numerical analysis.

REFERENCES

Anonymous. *Science Citation Index*. Institute for Scientific Information, Philadelphia, PA. Yearly.

If you want to find out more about one of the topics that we discuss, you can use our references and then follow the references of those works and so on down the bibliographic trail. But how do you find material that has been published since the works we cite? If you already have a paper on the subject, *Science Citation Index* can lead you to other papers published later that cite the paper you already have. The technique is explained in each yearly issue, and your librarian can help you.

[2] The languages suggested are those commonly available: FORTRAN, COBOL, ALGOL, assembly language, APL, XPL, PL/I, BASIC, PASCAL, LISP, SNOBOL, and their variants. This does not imply that other more esoteric or local languages are not appropriate, especially since our recommendations are biased by our own favorites. In every case, we encourage the use of higher-level languages and translators that provide considerable feedback *a la* WATFIV, PL/C, and SPITBOL. Students might also use problems to acquire new languages (total immersion is painful but effective).

Conway, Richard, and David Gries. *An Introduction to Programming*, 2nd ed. Winthrop, Cambridge, MA, 1975.

Strictly speaking, Conway and Gries is an introductory programming text (and incidentally a good manual for PL/I). But it is also an excellent text in writing plausible programs and constructing program proofs. Before starting your first etude, you could do worse than to review the material here on program development.

Dijkstra, Edsger W. *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

Although written independently, Dijkstra's and Wirth's books fit together nicely. A sample course might go as follows: read Conway and Gries; try a couple of the easier problems; read Wirth; try a number of harder problems; read Dijkstra and review solutions to earlier problems. Wirth actually gives programs and their development for some medium-sized problems. Dijkstra generally discusses only the critical loops or data structures but provides much more formal justification. Dijkstra also philosophizes about programming as an intellectual task; such thoughts may be the most important part of the book (but some experience is needed to appreciate them).

Griswold, R. E., J. F. Poage, and I. P. Polonsky. *The SNOBOL4 Programming Language*, 2nd ed. Prentice-Hall, Englewood Cliffs, NJ, 1971.

There are any number of books describing FORTRAN, COBOL, BASIC, ALGOL, assembly languages, and PL/I. Iverson designed APL originally as an algorithmic language; you will need to find a manual to run any particular implementation. McKeeman et al. is the defining text for XPL. Before running LISP or SNOBOL, you would be wise to check local conditions.

Iverson, Kenneth E. *A Programming Language*. Wiley, New York, 1962.

Jensen, Kathleen, and Niklaus Wirth. *PASCAL User Manual and Report. Lecture Notes in Computer Science*, 18, Springer-Verlag, Berlin, 1974.

Knuth. D. E. *The Art of Computer Programming/Fundamental Algorithms*. Addison-Wesley, Reading, MA, 1968.

Knuth's series, if he ever finishes it, will probably be the programmer's Bible. Certainly Volume I should answer most elementary questions about data structures and the algorithms to manipulate them. If you do not understand how to use some structure that we suggest, ask Knuth first. We do not recommend Knuth's programming style as a model of structure, however.

Lucas, F. L. *Style*. Collier, New York, 1962.

This is not a programming book at all. There will come a time, however, when you will need to write extensive documentation, and this book should help. Also, many of Lucas' observations can be applied to production of program text. Lucas concentrates on persuasion techniques, and a programmer must persuade *both* humans and computers.

McCarthy, John et al. *LISP 1.5 Programmer's Manual*. MIT Press, Cambridge, MA, 1972.

McKeeman, W. M., J. J. Horning, and D. B. Wortman. *A Compiler Generator*. Prentice-Hall, Englewood Cliffs, NJ, 1970.

Wegner, Peter. *Programming Languages, Information Structures, and Machine Organization*. McGraw-Hill, New York, 1968.

If you have any questions about computer architecture, languages, data structures, and their relations, Wegner can probably get you started on the answer. This book has an outstanding collection of "buzzwords" and connects them together. Wegner provides a quick survey of computer science, and the bibliography is useful.

Wirth, Niklaus. *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

2

The Game of LIFE

or...

CELLULAR AUTOMATA AND COMPUTER GRAPHICS

LIFE is a multicellular communal organism that inhabits the deserts of Flatland. The desert is organized in a square array with each square capable of holding one LIFE cell. LIFE generations mark the passage of time, each generation bringing births and deaths to the LIFE community.

To follow the history of such a community, place LIFE cells into their initial positions in the desert. Count the passing generations by observing these rules.

1. The immediate neighbors of a cell are those cells occupying the eight horizontally, vertically, and diagonally adjacent cells.

2. If a LIFE cell has fewer than two immediate neighbors, it dies of loneliness. If a LIFE cell has more than three immediate neighbors, it dies of overcrowding.

3. If an empty square has exactly three LIFE cells as immediate neighbors, a new cell is born in the square.

4. Births and deaths all take place exactly at the change of generations. Thus a dying cell may help birth a new one, but a newborn cell may not resurrect a dying cell, nor may one dying cell stave off death for another by lowering the local population density.

For instance, the community □□□ becomes $\begin{smallmatrix}\square\\\square\\\square\end{smallmatrix}$ in one generation, and the community $\begin{smallmatrix}\square\square\\\square\square\end{smallmatrix}$ must live near Palm Springs because it never changes at all. Figure 2-1 shows some more LIFE histories.

Statement of the Theme Write a program that simulates a LIFE community. The input should be the initial positions of the community's cells, and the output an aerial view of the community at each generation. An ordinary line printer can be used to plot the community's history, but such output is unaesthetic. If you have access to hardcopy graphic output devices or to an interactive graphics terminal, use the facilities provided to present a more appealing visual display.

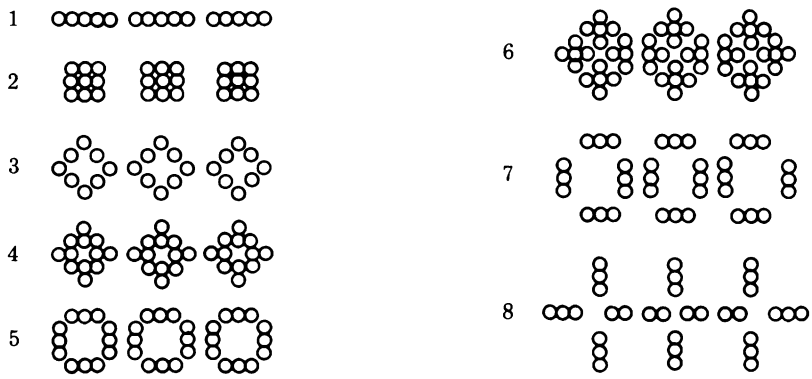


Figure 2-1: A LIFE Community History. Generation numbers appear at the left of each display. Find generations 9 and 10 for yourself.

Performance Practice Although the examples do not show it, some communities grow enormously from extremely meager beginnings. Others transport themselves slowly across the desert, continually moving from old territory into new. Your program should be able to handle large communities without a terrific cost in space or time. A naïve approach will repeatedly scan a large array to build the generations; the programming problem is to find data structures and algorithms that are more economical. You may want to try some method that only keeps track of the occupied squares. Since a growing or moving community can move out of the view of any fixed method for displaying the output, you will probably want a method that shifts its point of view as the community changes.

Orchestration APL may be suitable because of its vector and matrix operations. Almost any higher-level language with arrays can be used. This is a good problem for studying the cost of assembly language on programming time and the payoff in inner loop efficiency. Finally, for those with access to the hardware, a micro-coded version would be an interesting experiment; the computer becomes a LIFE community.

Playing Time This problem should take one person 3 weeks.

Variations on the Theme A community may go on growing forever, continually changing its position, shape, or membership. But it is more common for a community to become stable, repeating a few patterns in a cycle for all eternity, the length of the cycle being the community's period. (The period of a dead and empty desert is one under this definition.) Modify your program so that it attempts to recognize and report such stable communities. Can you think of any algorithm, short of saving all previous generations, that would infallibly identify stable communities?

The history of a LIFE community is fascinating if viewed as a movie (one of the reasons that we suggested an interactive graphics terminal). It is even more attractive when color is added. Each cell can be assigned a color at birth, perhaps by virtue of its generation or because of the genetic background of its parents. Cyclic but moving communities (of which there are quite a number) are beautiful when they march by in a coat of many sparkling colors.

Every community has a successor, but some have no predecessors. These isolated communities are called Gardens of Eden. The only way that they can be seen is by placing them on the desert as an initial configuration. Think of the ways to use your program to find a Garden of Eden.

REFERENCES

Burks, Arthur W. (Ed.) *Essays on Cellular Automata.* University of Illinois Press, Urbana, IL, 1970.

Codd, E. F. *Cellular Automata.* Academic Press, New York, NY, 1968.

Both books are considerably more serious than Gardner's column. Codd's is a monograph on the basic material, and Burks' is a collection of diverse papers in the general area. Starting from these two sources, almost all the mathematical material should be accessible.

Gardner, Martin. "Mathematical Games." *Scientific American*, 223, 10, pp. 120-123, October 1970, and 224, 2, pp. 112-117, February 1971.

Martin Gardner introduced LIFE in his column, and it caused such a stir that he had to devote another column to it immediately (at least by monthly magazine time scales). Certainly the game brought fame to John Horton Conway, its talented and prolific inventor. Much more material about LIFE or by Conway is scattered in more recent columns.

Wainwright, Robert T. (Ed.) *Lifeline.* 1280 Edcris Road, Yorktown Heights, NY 10598.

Lifeline is a quarterly journal devoted to LIFE and allied subjects. Subscriptions are available from the editor, as are back issues. Definitely a cult magazine, it contains all manner of LIFE material and may well be addictive.

3

Why is the Ocean Blue, Daddy?

or...

MAP COLORING BY EXHAUSTIVE SEARCH

When a map is drawn, it is customary to color its regions to distinguish them from each other. The rule is that two regions must be different colors if their boundaries intersect at more than a finite number of points (normally mapmakers are not mad topologists looking for pathological examples to contravene normal intuition). But cartographers must pay printers' bills, so the fewer colors used, the better. In particular, cartographers who color at random may paint themselves into corners and be forced to use more colors than strictly necessary. Some advance planning is needed. Finding the minimum of colors necessary goes under the title of the map-coloring problem.

The computer can help find the minimum number of colors needed. It may be difficult; however, to get the computer to *look* at a map; after all, most computers do not have eyes. Fortunately, the only crucial information is a list of which regions are adjacent to each other. The sizes and shapes of the regions are irrelevant to the coloring; only nontrivial contacts between regions matter. An undirected graph can be used to represent only the adjacency features of a map.

An undirected graph consists of a finite set of nodes and a finite set of edges connecting the nodes. Any two nodes are connected by at most one edge; we never allow two edges to do the same work and, for map coloring, we never allow an edge to connect a node to itself. Figure 3-1 is an undirected graph that represents the first 48 states. Computer input of a graph is fairly easy; simply list each node along with the nodes to which it is connected. A graph may have no nodes and hence no edges; then it is known as the empty graph. Any node may be disconnected by having no edges (Alaska and Hawaii would be disconnected); indeed, two sections of a graph are disconnected from each other if there are no edges connecting them. The association between maps and undirected graphs is so strong that we shall use both ideas interchangeably. In fact, graphs are so useful that all programmers should have some knowledge of their simpler properties.

Statement of the Theme Write a program to color a map using the minimum possible number of colors. The input should be a list of regions on the map, along with the adjacent regions. The output is a list of the regions with their assigned colors and the total number of colors used. It is

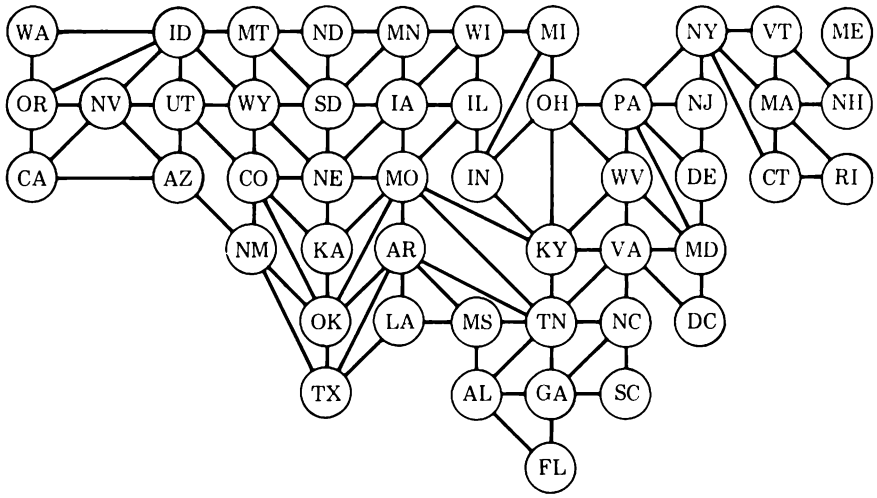


Figure 3-1: A Topological Map of the United States. No more than four colors should be necessary.

usually simplest to use positive integers to name the colors and regions, but a pleasant extension (and a psychologically useful debugging tool) would allow the input of more natural names. The input should be checked for consistency; do not allow ridiculous node numbers or nodes connected to themselves. The worst maps will be very expensive to color; so try to avoid unnecessary program inefficiencies.

Performance Practice It is not necessary for the map described by the input to be planar. In fact, two good limit cases are maps in which every two regions are adjacent and maps in which no two regions are adjacent, which correspond to coloring a set of disjoint balls and for which only one color is needed. Tests for planarity are an important subject in Computing Science, and a number of papers have been written on the subject. You may also be interested in pursuing the Four Color Conjecture, which states that no planar map will take more than four colors. If you manage either to confirm or disprove it, you will have made quite a name for yourself.[1]

The efficiency needed for this problem is primarily time efficiency. Certainly all possible solutions cannot be enumerated, for even though the correct solution need not be unique, the percentage of correct solutions is usually low and the number of possible solutions grows rapidly with the number of regions. Instead consider a backtracking solution. Begin by picking any one region and assigning it a color. Move to any adjacent uncolored region and try to assign a color that is compatible with previous colorings without using any new colors. (It may happen that no region remains to be colored, in which case you are done, or that there are no uncolored regions adjacent to any colored ones, in which case the map is disconnected.) If at some point the new region cannot be colored, work back down the already colored regions in the order in which they were colored, until you find one whose color can be legally changed. Change the color of this region to one that it has not previously had and work forward again. If this procedure backs all the way down to the first region colored, add a new color to the stock on hand and start again.

Orchestration This problem requires no data structure more complicated than arrays and stacks, and so almost any higher-level algebraic language with adequate control structures should

[1] This comment now has historical interest only. Please see the references to Chapter 29 for an explanation.

suffice (trying the problem in FORTRAN or BASIC should expose the poverty of these languages). On the other hand, backtracking can often be handled elegantly with a recursive formulation; so perhaps a language with recursive procedures would be useful. LISP may provide both data structure and recursion simultaneously.

Playing Time This problem should take one person 1 week.

Variations on the Theme If a backtracking solution is used, the order in which the regions are chosen can have a tremendous effect on the speed. It may be possible to anticipate this effect by preordering the regions or by using some heuristic to decide the next region taken. Regions that have many neighbors will probably be hardest to color, since they have the most constraints on them. Similarly, a group of regions that form a reasonably separate clump or cluster should be considered together, for if the clump cannot be colored with some number of colors, surely the whole map cannot. The idea in both cases is that if a certain region is going to cause trouble, it should be colored early in order to avoid wasting time by having it destroy an almost completed coloring. Of course, completely solving this preconditioning problem is tantamount to a solution to the original problem, but a small investment may pay big dividends. Compare some preordering strategies for cost and effect.

REFERENCES

Bitner, James R., and Edward M. Reingold. "Backtrack Programming Techniques." *CACM*, 18, 11, pp. 651-656, November 1975.

This paper is a very concise tutorial on backtrack programming. But if you cannot understand the idea from the authors' examples, there is a long bibliography of papers with problems suitable for or solved by backtracking.

Ore, Oystein. *The Four Color Problem.* Academic Press, New York, 1967.

Ore reviews the mathematics surrounding the Four Color Conjecture. It is a good way to learn a lot of graph theory, and you may learn a way to expedite the backtrack. But do not expect to find a fast algorithmic solution.

Printer's Devil

or...

AUTOMATIC TEXT FORMATTING

You may have been unaware of it, but one more onerous clerical chore has been lifted from the shoulders of mankind. Computers have replaced men in the construction and placement of typographical errors. Inexpensive small computers and photographic cold-type methods now churn out text where once linotypes produced hot lead. As efficient as the new techniques are, some romance is lost. What fun is it to search your Sunday *New York Times* for typos, the only humor in that expanse of solemnity, when you know that computers can make mistakes hundreds of times as fast as humans? Such is the price of progress.

Of course, the real progress has been in the use of computers as printers' devils, magic apprentices who do the dirty work quickly and, if the programming is right, cheaply. Programmers read and use computer manuals published with computer aid. Such manuals are often difficult to read because of the unfortunate typefaces available on computer printers. But most people do not realize that many magazines, newspapers, and books are also printed by computer. They look better because the computer not only edits and arranges the texts but also drives special photographic peripheral units that can generate press-ready copy in dozens of typefaces. Drafts of this book were written by using such a system, and early readers often thought that they had a photocopy of an actual book rather than what would have been simply a typescript if the drafts had been produced by typewriter in the manual way.

There are four parts to a publishing setup. First, there must be a good file system in which partially completed or archived text files can be stored. Normally file storage is provided by the host operating system, but we know of one case in which a card cabinet in the author's office was used for file storage. Cards are not actually practical for high-volume operations like newspapers. Secondly, there must be a text editor to modify and update files before final printing. Again, most operating systems supply a text editor, but a special publication editor to provide exactly the facilities necessary for handling publication text may also be necessary. The third element is the text formattor, which sets up headlines, selects page sizes, tabulates tables, recognizes paragraphs, and so on. The formattor handles text as words, sentences, paragraphs—that is, at the level that humans read it. Finally, there is the compositor, which converts formatted text into its image on the output me-

dium. The compositor is primarily concerned with the details of typefaces, physical sizes, output device commands, individual characters, and similar items. Just like a linotype operator, a compositor will set any gibberish in type and will be concerned only if the trash does not fit its allotted space. Functionally, the file system and the text editor worry about what the text means, and the text formattor and the compositor about what the text looks like. In this etude you will learn about text formatting.[1]

A TEXT FORMATTOR

Manual text formatting involves several steps. The author writes a draft, and the draft is typed in clean form. Then the author and an editor (at least for large publications) tear the draft to pieces, and the author goes to work on a new draft. This cycle continues until both author and editor are satisfied. Next, the draft is typed again (often with triple-line spacing) and passed to a copyeditor. The copyeditor marks the draft with notations about typefaces, headline size and placement, page size, italics, and anything else that affects the way that the finished text will look. These markings are in a special code, and each mark lies in the draft right where it is to take effect. The marked copy moves to the composing room, where it is set in type, and one-time test prints called galley proofs are made. The galleys are returned to the editorial department to be checked for accuracy against the final draft by a proofreader. Any small errors can be corrected in the composing room by substituting one line of type for another. But what if the author decides that Chapter 4 is all wrong, or the designer thinks that Bodoni Bold would have been a better type than Times Roman? All such material must be reset at considerable cost. And it is surprising how often seeing text in print, rather than as typescript, will change one's impression of it.

With a publication system, most of the work and many of the people can be eliminated from the publication cycle. As before, the author must prepare a first draft. Instead of being typed, however, the draft is entered into the computer file system. This entry, as is usual with computer data, may be from punched cards via a card reader or directly through a computer terminal. (Most of this manuscript was keypunched.) Also, the author takes on the role of copyeditor and adds to the first draft initial commands to the text formattor. The formattor and compositor process this draft text file so as to produce a rough proof of the final printed text. The rough proof is much more finished than a typescript; it probably looks like a page proof with correct numbers, an attractive typeface, and so on. Notice that all this activity occurs before any rewriting of the manuscript.

Then the author and editor begin the task of revision. The intellectual work is the same, but they have considerable help in visualizing the results because the drafts are more nearly in final form. Also, the work of editing is no longer so arduous. Insertion or deletion of a sentence does not require a retype; rather, the change is made with a text editor, just as lines in programs are changed. Rearrangement of large sections and recall of text temporarily discarded can usually be accomplished through the file system. Since the text is to be reformatted in any case, changing the copy preparation commands simply means changing the text file. Finally, computers can run formattors so cheaply that all the many draft-format runs may well cost less than old-fashioned typewriting charges. One danger, though—authors unused to the neatness of computer-generated drafts may be reluctant to rewrite; for too many years authors have been charged against royalties for changes to typeset copy. Reeducation is necessary if the computer is to be used correctly.[2]

[1] The English word *format* is strictly a noun and means the size, shape, or general layout of a publication. FORTRAN stole the word to describe the shape and layout of data records. There is no convenient verb to describe the process mediated by a FORMAT statement, however. So now we use the verb *to format*, in parallel with the verb *to edit*, to mean the action of laying out text in a particular pattern or scheme. Whether *to format* is jargon or English on the march is up to you.

[2] The production editor for this book points out that major book publication is not as idyllic as outlined here. Although type is set with electronic assistance at Prentice-Hall, most of the design, layout, and paste-up are still

COMMANDS FOR FORMATTING

How does a typical formattor work? The source file of text to be edited looks like a typescript (although the typist need not be so careful about spacing, margins, and the like) with format commands mixed in. Commands must begin the first character of a record and always start with "?" to set them off from ordinary text, at least in our example. For very simple output, only a command that sets the paper size and commands to break the text into paragraphs are needed. Within any one paragraph, the source text may be passed to the output file in one of three modes.

Unfilled — the lines from the source text are passed to the output exactly as they appear. This mode is most commonly used to pass tables or other preformatted material to the output without change.

Filled — the lines from the source are packed as tightly as possible from left to right in the output lines, and a new output line is not begun until the next word from the source will not fit on the previous output line. Single spaces are left between words, and double spaces after sentence-ending symbols like period, exclamation point, and question mark. This is the mode that a typist normally uses, and it produces a ragged right edge. Notice that extra spaces around words in the source are ignored in fill mode; spaces in the source are only used to separate words.

Justified — the lines from the source are first filled to produce a complete output paragraph. Then each line of the filled paragraph, except the last, has enough extra blanks added between words so that the last word of each line ends exactly on the right margin. No interword gaps should have n+1 blanks added until all have n blanks, and blanks should not be added after sentence terminators until all single gaps have two blanks. The blanks should be added to randomly selected gaps; if a pattern is used to add blanks, there will be unsightly stripes in the output. Justified text approximates that found in books but is not as attractive because varying character sizes are ignored.

The commands needed to process simple text are ?papersize, ?paragraph, and ?mode. The effects can be seen in Figures 4-1 and 4-2.

?papersize *height width*

The ?papersize command sets the limits on each page of text; a page may have *height* lines and *width* characters per line. Every time *height* lines are output, the formattor must create a new page. Text output lines will fit the entire space between columns 1 and *width* as necessary. A new ?papersize command may be issued at any time, but such a command automatically terminates the previous paragraph. The broken paragraph is finished with the old values of *height* and *width* before the new values take effect. Changing the paper size might also cause a new page if the new value of *height* is less than the old one. At the start of each format run, *height* should be set to 40, and *width* to 72, and no ?papersize is necessary if these values are satisfactory.

?mode *filltype*

The ?mode command sets the processing mode for text passed to the output. Argument *filltype* may be one of the three strings *unfilled*, *fill*, or *justify* (any other value is an error). Use of ?mode breaks the previous paragraph, which is finished by using the old value of *filltype*. The initial mode is *fill*; if this is satisfactory, no ?mode command need be issued.

---

done manually. In particular, compositors charge premium rates to correct errors which have made their way into type. Nonetheless, manual printing methods are on the retreat, and the complete victory of automation probably awaits only an input device for handwritten text.

```
?papersize 42 40
?mode justify
This sample section of text will be set justified. Notice that
    the way    spaces are left    has  no effect
on       the output.
Only word separation is caused by spaces.
Thus, it is a good idea to start each source text sentence on a
new line to make editing easier.
?mode fill
In the fill mode,     spaces    still have no effect,
but now the words are all run close up and the right margin is
raggedy.
Research suggests that the ragged right   edge
may improve reading speed.
Notice also the paragraph break caused by ?mode.
?mode unfilled
This text     will be taken exactly    as
   seen and it    better    not    run
past        column    40.
?mode justify
?paragraph 10 2
Finally, the ?paragraph command causes a gap and an
indentation which looks like normal text.
The commands buried in these lines cause no problems because
the question marks are not in column 1.
```

Figure 4-1: Some Raw Source Text

?paragraph *indent gap*

The ?paragraph command breaks one paragraph off and begins another. The new paragraph's first line is started *indent* spaces in from the left margin (*indent* might be zero, and later we will see how it could be negative) and *gap* blank lines are left between the old paragraph and the new. If *gap*, or *gap* and *indent*, are not specified, they retain their values from their last previous settings. The initial value of *indent* is 3 and of *gap* is 0; if these values are satisfactory, there is no need to supply arguments when ?paragraph is used. Notice that if *indent* is 3, the first line of the new paragraph starts in column 4.

The ?papersize, ?mode, and ?paragraph commands are not sufficient. A complete formattor will require at least the following additional commands.

?margin *left right*

The ?margin command causes the left and right margins of the output text to be set into columns *left* and *right*. Naturally the left margin must be 1 or more, and the right margin

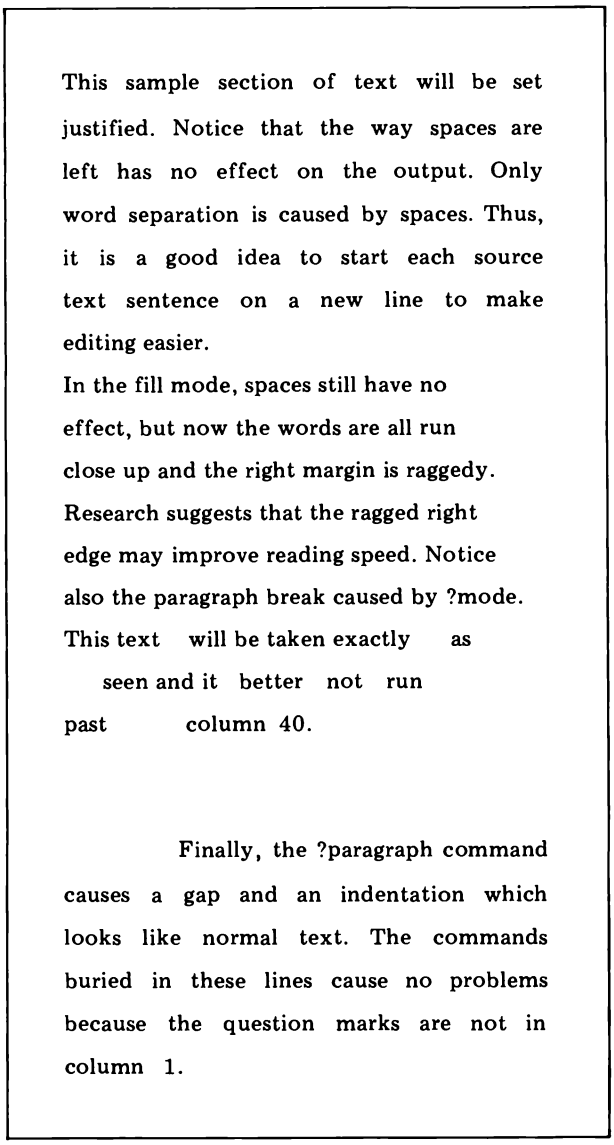


Figure 4-2: The Same Text Formated

must be no more than the current paper *width*. A ?margin command breaks the previous paragraph. With the introduction of ?margin, it makes sense to have negative values for argument *indent* of the ?paragraph command; simply outdent (a made-up but obvious word) the first line of the paragraph toward the left edge of the paper.

?linespacing *gap*

The ?linespacing command causes *gap-1* blank lines to be left between output lines. A *gap* of 1 is thus like typewriter single spacing, of 2 like double spacing, of 3 like triple spacing, and so on. This command breaks the previous paragraph.

?space *n*

The ?space command breaks the previous paragraph and inserts *n* times the current line-spacing blank lines into the output. The action is similar to hitting the carriage return *n*+1 times on the typewriter. If a new output page is created because the blank lines more than fill the bottom of the current page, the page is turned, but no blank lines appear at the top of the new page. The default value of *n* is zero.

?blank *n*

The ?blank command works like the ?space command except that exactly *n* blank lines are inserted into the output; there is no interaction with the ?linespacing argument. This action is similar to rolling the typewriter platen *n*+1 clicks.

?center

The ?center command takes the next source line, strips trailing and leading blanks, and centers the result between the left and right margins of the next output line. The previous paragraph is not completely broken, but the line before the centered one may be short. The centered line does follow the normal linespacing. Naturally an error occurs if the centered text is too long to fit the current margins.

?page

The ?page command breaks the current paragraph and, after the last line of the paragraph has been moved to the output, causes a move to a new output page.

?testpage *n*

The ?testpage command breaks the previous paragraph and moves it to the output. If there are fewer than *n* blank lines now on the current page, ?testpage works like ?page; otherwise it is completely ignored. Thus ?testpage checks the space remaining on a page.

?heading *depth place position*

The ?heading command sets a title to be used at the top of each page, beginning after the next page—turn in the output. The next *depth* lines of the source are taken exactly as is for a heading occupying the top *depth* lines of each page. On the line numbered *place* of the heading, the page number is filled in on the left, right, or center as argument *position* has value *left*, *right*, or *center*. The page number is incremented each time that a page is turned and starts with value one. The heading lines always use the margins in effect when the heading was defined. A heading may be eliminated by using ?heading with a *depth* of zero. The ?heading command does not cause a break.

?number *n*

The ?number command sets the current page number to *n* and does not cause a break in the previous paragraph.

?break

The ?break command causes a break in the previous paragraph.

?footnote *depth*

The ?footnote command causes the following *depth* lines of source text, *including* any commands, to be placed at the bottom of the page in footnote position. The controlling parameters of the formattor, margins, linespacing, and the like, are saved over the footnote and are also used initially to provide an environment for the footnote. Enough source is read from after the footnote to completely fill out the last source line preceding the ?footnote. Then the footnote is processed and fills the page from the bottom up. If there is footnote material on the current output page already, the new material pushes the old material up from below. If the footnote material runs up into formatted output, the page is finished, and the rest of the footnote goes on the next page (which is why the last ordinary line before the footnote is filled before footnote processing begins). Once all *depth* lines are on the output, processing reverts to the ordinary text and to the original parameter values (although the page number may have changed). Obviously ?footnote must not cause a break, and one ?footnote is not allowed within another.

?alias *fake real*

The ?alias command sets the single character *fake* to stand for the single character *real* until ?alias is issued again. As each line is passed to output, all instances of *fake* are changed to instances of *real*. Blanks have a special use as word separators; an ?alias command can be

used to force a blank into the output without causing a word break. An ?alias does not break the previous paragraph, and all aliasing can be turned off by invoking ?alias with no arguments.

A WORD ON WORDS, LETTERS, AND ARGUMENTS

To fill and justify correctly, the formattor must recognize words and sentences. Words are rather easy; any nonblank string of characters terminated with a blank or an end of line is a word. Notice that this category includes trailing punctuation as part of the preceding word. Sentences end with full stops in English and are normally followed by a double instead of a single space, but the stop may be inside quotes or parentheses. English also requires that colons be followed by double spaces, so whenever a word ends with

? ! .) ?) !) ." ?" !" .") ?") !")

be sure to mark a sentence end as well. There may be other possibilities that we have not mentioned; in English, authors are often very free with punctuation.

If your formattor will run on a time-shared system with upper- and lowercase input and terminal output, undoubtedly the character set available to the language in which you program the formattor will include both upper- and lowercase. But if you are running on a card-oriented system, there will be difficulty in reading two cases, since keypunches do not have shift keys (there had better be some way to print both cases, or the project is not worth much). To allow keypunch input, choose some special character like ↑ and have it mean shift up once. Now you would keypunch

The IBM 360 computer

as

↑the ↑i↑b↑m 360 computer

Shift up is the explicitly marked action, because capital letters are much less common than lowercase. Notice also that we are assuming that the keypunch provides lowercase, which is *not* the way keypunching looks on the punched cards.

Arguments to commands come in two forms. Some arguments are integers and give either an explicit value for some formattor parameter or the number of source lines to be affected by the command. Other arguments are words or characters that are to be used in their literal senses. In both forms, arguments are separated by blanks, and extra blanks are ignored. The ?alias command may have a missing second argument, which is then assumed to be blank (otherwise hard to represent under these conventions). Be careful to have error messages for malformed commands.

Statement of the Theme Write a text formattor for your system that uses the commands described above. Since formatted text is not very valuable unless it has upper- and lowercase output, you must make use of an output device with upper and lower capability. Because such devices are often expensive in computer charges, you may not be able to afford many test runs. Although you naturally expect to get everything right the first time, you might provide test output in a form similar to keypunch input. Such output could be printed on a standard line printer.

Performance Practice It is probable that you will find that your program spends most of its time reading and writing, and little time actually moving the words into their output positions.

Furthermore, most of the processing time will probably be spent finding the blanks between words. Keeping these facts in mind, it is clear that the central scan algorithm and the formattor's transactions with the outside world should receive the bulk of your optimization. Also, the command recognizer and all the algorithms for correct placement should be very clear. Normally input/output is best left to standard language features, but this is a problem where special knowledge of local system features can be exploited to good purpose. Remember that such exploitation should be confined insofar as possible to input/output routines and should not diffuse through the whole formattor.

The command set was chosen so that the output can be produced in one pass through the input. No command algorithm should require the input to be backed up. If algorithms need workspace, as they will for footnotes, consider double buffering the output and using the free buffer for the needed space. As a timing test, the formattor that produced this book took about 2 seconds of CPU time per page and was written in a variant of TRAC (see Chapter 28). Incidentally, it seems that most formattors take about 1 or 2 seconds to produce a finished output page regardless of the speed of the underlying computer. The only explanation that we offer is that users perceive this rate as reasonable, and programmers do not think it cost effective to strain for faster formatting.

Orchestration A simple version of this problem is traditional in SNOBOL courses, but we suspect that most SNOBOL implementations would be too slow for practical use. On the other hand, any language without at least some string handling will be awkward at best. Perhaps the best compromise would be an intermediate language like XPL or BLISS. Many computers have special hardware for text operations such as finding blanks, breaking strings, and comparing strings. This suggests that the very innermost loops should be written in assembly language to take advantage of such features.

Playing Time One person for 4 weeks.

Variations on the Theme In this book you will see boldface, italic, Greek, underlined, and other special characters. All were available on the output devices but not, as you might guess, on the keypunches or the file storage media. A special convention was used for these special characters. For instance, suppose that "et cetera" is to be set in italics. Then it would be written "&i+et cetera&i-" producing "*et cetera.*" The triple characters beginning with the ampersands "&" are called font switches and, in this case, switch the italic font on and off. By regarding underlining, superscripts, subscripts, and so on as special fonts, you can take advantage of any extra features that your output device may have. Indeed, more than one switch may be on simultaneously, so that the output might have underlined Greek superscripts. (You will probably want the backspace font switch to be &xn, where n is a digit between one and nine.)

REFERENCES

Kernighan, Brian W., and Lorinda L. Cherry. "A System for Typesetting Mathematics," *CACM*, 18, 3, pp. 151-157, 1975.

Although this paper strictly describes only a system for setting mathematical expressions, the system is grafted onto a general text formattor. The paper, as printed in *CACM*, is a photocopy of the work of the formattor and was not reset for publication. By the way, Kernighan and Cherry are selling their system.

Kernighan, Brian W., and P. J. Plauger. *Software Tools.* Addison-Wesley, Reading MA, 1976.

Kernighan and Plauger discuss the set of software helpers that one would like to have before starting a large (or perhaps any) programming project. As in these etudes, the tools are described and then set as projects. One of the tools used is a text formattor. They also provide some hints on implementation. You may want to compare features before starting this etude.

5

Winning is the Only Thing

or...

TOURNAMENT DESIGN AND EVALUATION

Almost every one has been a fan of some hometown team that was second best. At the end of the season the championship tournament determined the town, county, state, national, world, or universe champion. Unfortunately, the local heroes were knocked out in the first round of the single elimination tournament by the eventual winners. The game was not too interesting; after all, nobody was warmed up yet. And how sad it was—some lackluster team eventually had our heroes' rightful spot; instead of a dramatic confrontation at the end of the tournament, there was a cakewalk.

The culprit here is the single-elimination tournament. Assume that there are 2^n teams for $n > 0$. Then, in the first round, team 1 plays team 2, team 3 plays team 4, . . . , and team 2^n-1 plays team 2^n. The losers drop out, and the winners advance to the next round.

Figure 5-1 gives a picture for eight teams. If we assume that the better team will always win its game (that is, there are no upsets), the best team will clearly come out in first place. But the other team in the championship game might really be no better than $2^{n-1}-1$ in an absolute ranking, if it happens that all the better teams are in the same bracket as the winner. The winner could knock out all the good teams on the way up, while a weak team had an easy path. There are several ways to avoid this situation. First, the teams (henceforth contestants) might be seeded so that the good contestants (on the basis of prior performance) are spread throughout the original entries. For instance, the best contestant might be in slot 1, the second best in slot $2^{n-1}+1$, the third best in slot $2^{n-1}+2^{n-2}+1$, the fourth best in slot $2^{n-2}+1$, and so on. Assuming that the initial rankings are reasonably accurate, the better contestants would not knock each other off in the early rounds. A second possibility is a double-elimination tournament in which it takes two losses to leave the tournament. But, in fact, the complete (if impractical) solution would be a round robin tournament in which every contestant plays every other exactly once. Once again assuming no upsets, the best contestant will have a record of 2^n-1 and 0, the second best a record of 2^n-2 and 1 (losing only to the best contestant), . . . , and the worst a record of 0 and 2^n-1 (losing to everybody). The difficulty is that a round robin takes 2^{n-1} (2^n-1) games, whereas single elimination takes only 2^{n-1} games.

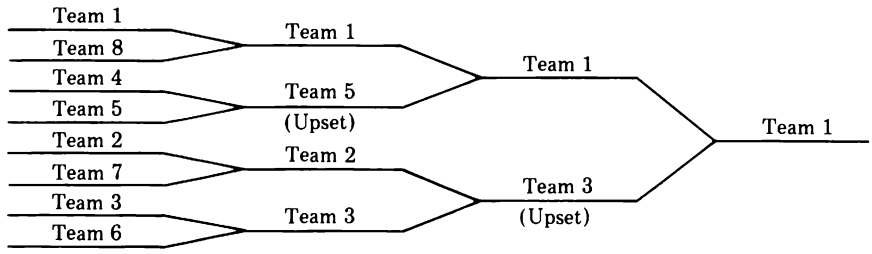


Figure 5-1: A Sample Single Elimination Tournament. The final ordering as defined in the text is 1, 3, 5, 2, 8, 6, 4, 7.

There is a compromise—run a Swiss tournament. On the first round, pit the first seeded contestant against the last, the second against the next to last, and so on. After each round, rank the contestants by their records and, within each group with the same record, rank the contestants by the average record of the opponents that they have already beaten (any unresolved ties are irrelevant). In the next round, the highest-ranked contestant is paired with the next higher-ranked contestant such that the two have not already met. The rest of the contestants are paired with the same policy of trying to match nearly equal records without allowing any repeated matches. Table 5-1 shows a possible three-round, eight-player Swiss. Harkness, an important chess tournament director, claims that a Swiss tournament of $\sqrt{N+2k}$ rounds, where N is the number of players, will place the first k+1 players correctly (and, by symmetry, the last k+1 also). Swiss tournaments are more accurate than single eliminations, much faster than round robins, and allow every contestant to play every round. The question is how well these tournaments work with real competitors. Assume that there are 2^n contestants, that contestant 1 is the best, contestant 2 is the next best, all the way down to contestant 2^n, who is the worst. First, run a complete round robin, recording the results of each match. If a match pits contestant i against contestant j, for $i < j$, then the probability that contestant i wins is

$$1/2 + (j-i)/2^{n+1}$$

always giving the better contestant a more than even chance of winning. Rank the contestants by their round robin records; within each group of equal records, rank by the mean winning score of the contestants' opponents; if there are still ties, rank by the original ordering of the contestants. The result is the round robin order, which we will assume is the "fairest" ordering and which we will use to grade the other tournaments.

Table 5-1 An Example Swiss Tournament

Pairings Round 1	*Winners*	*Pairings Round 2*	*Winners*	*Pairings Round 3*	*Winners*	*Final Ranks*
1	1	1	1	1	1	1 (3-0)
8		2		3		3 (2-1)
2	2	3	3	5	2	2 (2-1)
7		5		2		4 (2-1)
3	3	8	8	4	4	5 (1-2)
6		7	(Upset)	8		6 (1-2)
						8 (1-2)
4	5	6	4	6	6	
5	(Upset)	4		7		7 (0-3)

This tournament is not actually large enough to show a Swiss' virtues.

The next step is to run Swiss and single-elimination tournaments using the same data base. Whenever two contestants are paired in one of the tournaments, use the result saved from the round robin match between the two. Notice that two contestants can meet only once in either tournament. The Swiss order is the order after the final round (run n rounds), with any ties not broken by the Swiss pairing broken by the original rankings. Start the single-elimination tournament by pairing randomly for the first round. In the single-elimination order, the winner of the final round is first, the loser second, and, in general, the losers in round i are ranked ahead of all earlier losers and behind all winners in round i and later. Within the group of losers at round i the contestants are ranked by the order of finish of the teams that beat them.

To compare the orderings, we use a new and an old statistic. The old statistic is the rank correlation, defined as

$$R = 1 - 6 \sum_{i=1}^{N} (x_i - y_i)^2 \ /(N^3 - N)$$

where x_i is the rank in one ordering, y_i in the other, for contestant i, and N is the total number of contestants (here 2^n). The other statistic is the match count, defined as

$$M = \max_i (\forall j)(j \leqslant i \supset x_i = y_j)$$

that is, M is the maximum number of places from the top down in which the two orderings exactly match. R measures the rough equality of the whole of the two orderings, and M the equality at the top of the orderings.

Statement of the Theme Write a program that reads an input value n, runs each of the three tournaments for 2^n contestants, and calculates the two statistics R and M for each of the three pairs of orderings. Using the same n, run the experiment many times and calculate means for M and R. See if Swiss tournaments or single-elimination tournaments are more likely to replicate round robin results.

Performance Practice Except for understanding how each tournament works and programming the matchups efficiently, there is little difficulty here. Because of the size of a round robin, you should struggle for an efficient inner loop in the round robin and efficient storage of the match results. Of course, you will need a random number generator of good quality to decide the matches. Also, the Swiss matching might result in trying to pair contestants who have already met. Either prove that this situation cannot happen or modify the algorithm to avoid such occurrences while maintaining the general policy of trying to pair contestants with similar records.

Orchestration An algebraic procedural language with good loop controls is appropriate. APL is also a possible choice, as are other array-processing languages, if you can organize the tournaments so that they take advantage of processing all the contestants in parallel.

Playing Time One person for 2 weeks.

Variations on the Theme Most of the extensions involve more elaborate analysis and comparisons with other tournaments. First, notice that the lower rankings of a single-elimination tournament are fairly arbitrary. Also, the low-ranked contestants do not have much fun, since they get knocked

out early. The solution is to run another single-elimination tournament among the losers at each round. The derived order is used to rank these losers instead of the rule given above. Since these secondary tournaments will also have losers, run even smaller-ranking tournaments and so on *ad nauseam.* Notice that the tournament still runs for n rounds but that now all the contestants get into all the rounds. If the better contestant always wins, the elaborated tournament is on the way to becoming a complete sort algorithm.

Indeed, tournaments are sorting procedures on the contestants entered in them, albeit the comparison rule is probabilistic. Any sort that follows the two central rules of tournaments should be a candidate for a tournament design. These two rules are as follows.

1. No contestant should appear in more than one match per round, and the number of rounds should be about the logarithm of the number of contestants.
2. No two contestants should meet more than once.

Using these guides, you can evaluate both classical tournament designs like double elimination and tournaments that you invent yourself.

Several statistical questions also spring to mind. What is the effect of partial or complete seeding on the single elimination, and what is the effect of a random draw (that is, random initial matchups) on the Swiss tournaments? What is the effect of a different superiority function? And since it is probably not correct simply to average our two statistics over a number of experiments in order to calculate an overall statistic, what statistical operation should be used?

REFERENCES

Harkness, Kenneth. *Official Chess Handbook.* David McKay, New York, NY, 1967.

Harkness tells everything that you could want to know about the legalities of chess. Since the Swiss system has made large open chess tournaments possible in the United States, he goes into great detail about how to run one. There are also many proposals for tie breaking and ranking of players.

Knuth, D. E. *The Art of Computer Programming/Seminumerical Algorithms.* Addison-Wesley, Reading, MA, 1969.

Chapter 3 of the "Bible" is about random numbers, their generation, and their use. You can learn the pitfalls of trickiness here. We suggest that you try the MacLaren-Marsaglia generator described by Knuth in Algorithm M.

Hoel, Paul G. *Introduction to Mathematical Statistics.* Wiley, New York, NY, 1971.

For those who are not statisticians, the use of correlations and other statistical magic seems mysterious. Hoel explains simple statistics without condescension and he does not mystify.

Strike It Rich

or...

BUSINESS MANAGEMENT AND COMPUTER SIMULATION

It is often possible to study a real-world situation by building a simulation. Much fun and some knowledge come from the construction process. Indeed, some people become so addicted to exploring the model that they never go back to the real thing. Here's your chance.

MANAGEMENT,[1] A GAME

You have just been appointed to the presidency of a large manufacturing concern by its expectant board of directors. The company owns a number of factories. Each month the company buys raw materials, processes them, and sells the finished products to a waiting public. You will have to decide on inventory and production policies, whether and when to expand facilities, how to finance expansion, and how to assume an attitude of bashful modesty when reporting your obscene profits. Before taking on the job, you build a simulation of the whole industry so that you can try out your business strategies in private. What follows is a description of the game that you have developed.

THE INITIAL SITUATION

The simulation moves in one-month time steps. At the beginning of the game each player (a company president) receives two standard factories, four raw material units (abbreviated RMU), two finished inventory units (abbreviated FIU), and $10,000 in cash. The players are numbered from 1 to N, and on the first turn player 1 is the senior player. Each turn, the honor of being senior moves to the next higher-numbered player, returning to the first player after player N [that is, the formula for the senior player on turn T is (T mod N)+1]. In all cases of ties during bidding, the most senior player (the player who will next be senior) wins.

[1] *Management* is the trademark of the Avalon Hill Company, 4517 Harford Road, Baltimore, MD, 21214, for its copyrighted game of business. We have modified the rules slightly to make the programming of this etude simpler.

MONTHLY OPERATIONS

During each month the following transactions are performed in exactly the order specified. If a company cannot meet a financial obligation at some point during the monthly cycle, it is declared bankrupt immediately, its assets vanish, and it leaves the game (better have a supply of ready cash). All payments are between individual players and a universal bank; no money ever flows between players, thus avoiding antitrust suits. Similarly, the bank controls the source of RMUs and buys up all FIUs.

1. Pay Fixed Expenses. Beginning with the senior player and proceeding to the junior player, each player pays $300 for each RMU held in stock, $500 for each FIU held in stock, $1000 for each standard factory owned, and $1500 for each automated factory owned. These are fixed maintenance expenses borne by each player each turn and must be paid even if the player takes no other action during the month.

2. Determine Market Conditions. The bank determines and informs the players of the number of RMUs that will be available this turn and the *minimum* price it will accept for them. Similarly, it announces the number of FIUs that the bank will buy and the *maximum* price it will pay for them. Table 6-1 shows the five levels of RMU supply and FIU demand (note that when one is up, the other is down) and their floor and ceiling prices. The number of players P does *not* count those who have gone bankrupt and so may be less than N. The products 1.5P and 2.5P are rounded *down* to the nearest integer. Table 6-2 is the stochastic transition matrix that the bank uses to choose the new month's supply and demand level given the preceding month's. Assume that the level in month zero was 3.

3. Bid for Supplies. Each player calculates a secret bid for RMUs desired this month. A bid must specify both the number of RMUs needed and a purchase price no lower than the bank's minimum (a request for zero RMUs or less than the minimum price simply drops the player from this month's bidding). All the bids are revealed simultaneously, and the RMUs available are parceled out to the players, high bidder first. If there are not enough RMUs to go around, the low bidders lose out; if there are ties in bid prices, the most senior player wins. Players pay for the units as they receive them. Units left over after bidding are *not* stockpiled by the bank for the next turn.

4. Produce Stock. Beginning with the senior player and proceeding to the junior player, each player must announce how many RMUs are to be converted into FIUs in this turn, and which factories are going to be used. The player is immediately charged for the production. A standard factory may process one RMU a month at a cost of $2000. An automated factory may do the same, or it may process two RMUs in a month at a cost of $3000. Obviously the player must have the necessary RMUs to process.

5. Sell Inventory. An auction similar to that held during the purchase of RMUs is held to sell FIUs back to the bank. Bids must be *lower* than the *maximum* price set by the bank, and the bank buys FIUs from the *lowest* bidder first. Ties in bid prices are resolved in favor of the senior player. If supply exceeds demand, higher-priced units go unsold. Players are paid as soon as their units are purchased.

6. Pay Loan Interest. Each player pays 1% interest on the outstanding balance of all unpaid loans. Interest is due even on loans that will be repaid this turn.

7. Pay Outstanding Loans. Each player who has a loan falling due this turn pays it off. Because loan repayment precedes loan allocation, players must pay off loans from their cash on hand.

8. Take Out Loans. Any player may take out a loan now. Loans are secured by a player's factories, with a mortgage value of $5000 for standard factories and $10,000 for auto-

Table 6-1 RMU and FIU Price Levels

Level	*RMUs*	*Minimum Price*	*FIUs*	*Maximum Price*
1	1.0P	$800	3.0P	$6500
2	1.5P	650	2.5P	6000
3	2.0P	500	2.0P	5500
4	2.5P	400	1.5P	5000
5	3.0P	300	1.0P	4500

mated factories. A player may not have loans outstanding that total more than half the the security value but may borrow freely up to this limit. The bank immediately pays the player the loan amount, and the loan is due to be repaid in the twelfth month following—for instance, a loan taken in month 3 comes due in month 15. Loans may not be repaid before they come due.

9. Order Construction. Players may order the construction of new factories. A standard factory costs $5000 and may begin production during the fifth month following order placement; an automated factory costs $10,000 and comes into production the seventh month following order; a standard factory may be converted to automated processing for $7000, coming into production the ninth month following (standard processing may continue during conversion). Half the cost of a factory must be paid when it is ordered; the other half comes due during this part of the turn in the month preceding first production at the new facility. A player may not own or have on order more than six factories.

TERMINATION AND EVALUATION

The game ends after some fixed number of turns (13 or more), or after all but one of the players has gone bankrupt. A company's net worth can be calculated by adding up the value of its factories at the price it would take to rebuild them, the value of RMUs at the current bank minimum price, the value of FIUs at the current bank maximum price, and cash on hand, and subtracting the total of loans outstanding and construction costs yet to be paid. If more than one player is active at the end of the game, the active players are ranked by net worth.

At any time any player can find out any other player's net worth, cash, loan, inventory, factory, or construction position. During auctions, players may not know one another's secret bids, however, once the bank has collected the bids, all bids are publicly announced and the number of units

Table 6-2 Transition Probabilities between Price Levels

Old Level	*New Level* 1	2	3	4	5
1	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{6}$	$\frac{1}{12}$	$\frac{1}{12}$
2	$\frac{1}{4}$	$\frac{1}{3}$	$\frac{1}{4}$	$\frac{1}{6}$	$\frac{1}{12}$
3	$\frac{1}{12}$	$\frac{1}{4}$	$\frac{1}{3}$	$\frac{1}{4}$	$\frac{1}{12}$
4	$\frac{1}{12}$	$\frac{1}{6}$	$\frac{1}{4}$	$\frac{1}{3}$	$\frac{1}{4}$
5	$\frac{1}{12}$	$\frac{1}{12}$	$\frac{1}{6}$	$\frac{1}{3}$	$\frac{1}{3}$

bought or sold by the bank from each player is public knowledge. Players may keep any records that they wish, but the bank will not assist them beyond supplying information required by the game.

Statement of the Theme This problem has two parts. The first requires you to write a program that runs the simulation—that is, a banker program. This program must have complete control of the game; setting prices, buying and selling inventory, running auctions, keeping the accounts, and so on. It must query players at the proper times and enforce compliance with all rules. In particular, all accounts of the banker and all private information kept by players must be protected from unauthorized player interference by positive control of the banker. Output of the banker is a running account of the game with periodic (monthly?) summaries or balance sheets. Since this history is meant to be read by humans, it should be self-explanatory and aesthetically pleasing.

The second part of the problem is to write player strategy routines. Each player routine must be capable of responding to all game requests made by the banker; that is, it must be able to bid for inventory, make processing decisions, sell finished inventory, and so on. If you write your simulation for an interactive system, one player routine should operate by passing the decisions to a human player seated at a console. Such a routine should be able to respond to human queries about the state of the game.

After several player routines have been written, they should be combined with the banker to form a complete gaming system. Use this system to play several games and observe the results. Note that several copies of the same player routine might compete against one another. (If we regard humans as initially identical, this is what happens in a real game.) For credit, at least two nontrivial player routines must be written.

Performance Practice This is an example of a sequenced or lock-step simulation in which all events (except bankruptcies) happen in a strictly defined order that is known in advance. A loop that cycles through the month's work seems an appropriate structure for the central processing routine. You are unlikely to see many programming problems, either academic or applied, as suitable for a well-structured implementation. Take advantage of the opportunity.

There is one catch in the statement of part one. The banker must protect all sensitive information from tampering by an unscrupulous player routine. In other words, the banker must keep accounts private, the bank's and the players', ensure that auctions are really secret, and yet supply requested information to players. Unfortunately, doing so may be very hard, if not impossible, in many languages. In FORTRAN, critical values may not be in common blocks because a player routine can access a common block without permission of its creator. In a block-structured language, critical values cannot be global to player routines for the same reason. Even if you assume that a player routine does not violate rules of the source language by accessing off the end of an array or dropping into assembly language, for example, complete security may be difficult to achieve. One of the topics that your documentation should discuss is your security technique and its success.

Orchestration This problem cries out for a language with expressive control structures. There is less need for elaborate data structures. COBOL and FORTRAN are possibilities but are apt to be handicapped by their poverty. APL has been successfully used for similar problems, but clear structuring will be a difficulty. You will probably not find a language with the data protection suggested above.

Playing Time One person for 4 weeks, two people for 3 weeks, or three people for 2 weeks. A player strategy routine should take 2 weeks.

Variations on the Theme Part of the fun of writing a game simulation is playing with the strategy routines. Sometimes quite simple heuristics can provide surprisingly complex behavior patterns. It should be easy to build some learning capability into your strategy routines so that they will perform better and better. Try holding several training tournaments with both humans and programs (humans can learn, too). A standard trick for teaching new strategies to intelligent programs is to allow one copy (Alpha) of a learning strategy to learn during a series of games while holding a second copy (Beta) to the knowledge level that it had at the beginning of the series. After the learning series, an evaluation series is held: if Alpha wins, all versions of the strategy routine are given Alpha's new knowledge; if Beta wins, Alpha's training is forgotten (as no improvement), and a new learning series is started.

The game can be made more interesting by adding more rules and presumably more realism. The additional rules are listed below; if a new rule is added, all lower numbered rules should also be added.

1. Emergency Loans. Whenever a cash squeeze occurs during the game, a player may apply for an emergency loan. Such a loan costs 2% a month instead of the normal 1% and comes due in the fourth month following during the normal loan repayment phase (interest is paid during normal interest repayment). The total of all outstanding loans may still not exceed half the value of a player's collateral. Emergency loans cannot be used to rescue a player from bankruptcy once the bank has demanded payment for some obligation; the request may be made no later than the beginning of the phase in which the payment comes due.

2. Special Situations. In a new phase before the payment of fixed expenses, the bank announces any special situations obtaining for this turn. Figure 6-1 gives the probability of the various special situations. The effect of the rate changes mentioned below is cumulative; for example, an increase of 10% followed by a later decrease of 10% results in a final net rate that is 99% of the original rate. The situations include

Strike—The affected player may choose to halt all production for 3 months starting now or pay a 10% increase in all factory expenses (both fixed and production) until the end of the game. A player whose production is halted may still participate in all other phases of the game and must pay all fixed expenses.

Transportation Crisis—The affected player may not buy or sell any units this turn.

Special tax—The affected player must immediately pay a one-time tax of $500 per factory. The tax must be paid without the aid of an emergency loan and may cause bankruptcy.

Flood—One of the affected player's factories (standard, if possible) may not produce this month.

.01 Player i is affected with a strike.
.01 Player i is hit by a transportation crisis.
.02 Player i must pay a special tax.
.01 Player i has one factory flooded.
.02 Player i reaps the rewards of research and development.
.02 Player i finds a windfall profit.
.91 No special situation for player i this turn.

Figure 6-1: Probabilities of Special Situations. Each player tries for a situation each turn.

Research and Development — Factory production costs of the affected player immediately drop 10% for the remainder of the game.

Windfall Profits — The affected player may immediately sell as many FIUs as desired at $6500 each. The FIUs must be in stock.

3. Shutdown. In a new phase just before ordering construction, a player may ask that any or all of the company's plants be shut down; beginning with the next month, such plants cost only half the normal fixed expense but may not produce. In the same phase during a later month, a shutdown factory may be ordered back into production. A factory ordered reopened resumes paying full expenses and production in the second month following; for instance, a factory reopening ordered in month 13 is effective in month 15.

4. Split Bids. In all auctions a player may offer zero, one, or two bids. The total number of units bid for by one player, whether buying or selling, may not exceed the number that the bank has offered for bid and, in the case of FIUs, may not exceed the number that the player has in stock. Split bids by one player are treated by the bank as if they had come from separate players. The bids are competing against one another and other players' bids, and both, either, or neither, may be successful. Ties are still resolved in favor of the senior player.

REFERENCES

Anonymous, *Management.* Avalon Hill Co., Baltimore, MD, 1960.

Management is the most realistic of the "business" games available to the general public. There is an ingenious accounting form that controls the play in the manual version of the game.

Evans, George W., II, Graham F. Wallace, and Georgia L. Sutherland. *Simulation Using Digital Computers*, Prentice-Hall, Englewood Cliffs, NJ, 1967.

This is a fairly simple introduction to simulation techniques. It certainly makes few demands on computer knowledge. Some examples are worked out in detail for both conflict and nonconflict situations.

7

Kriss-Kross

or...

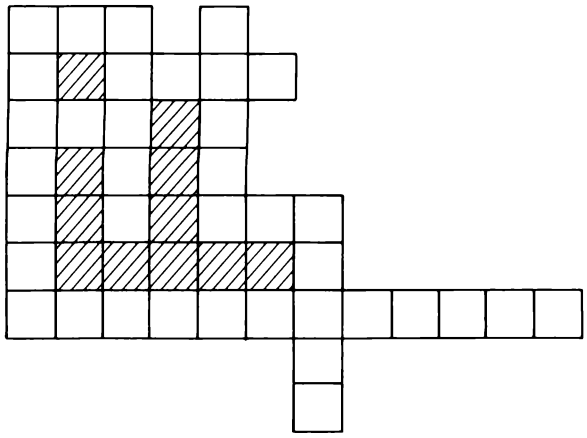
PUZZLE CONSTRUCTION USING HEURISTICS

Many people find crossword puzzles too hard because they cannot figure out the clues, but they still enjoy filling out the diagram. For such people there is a simpler puzzle, the Kriss-Kross.

Each Kriss-Kross consists of a list of words, arranged for convenience into groups by length and alphabetically within each group, and a diagram to be filled with the words. The diagram follows the same rule as a crossword—that is, wherever two words intersect, they must have a letter in common, but there are no numbers, since the words are already known and the problem is to find their places. Typically, Kriss-Kross diagrams are much more open than crosswords, and the black squares are simply left blank unless they will cause confusion. A Kriss-Kross always has a unique solution that uses all the listed words. Figure 7-1 is an example, albeit extremely small. Note that word length is an important clue to solution.

Statement of the Theme Write a program that takes any list of words and constructs a well-formed Kriss-Kross diagram for the list. Proof that the diagram is well formed is presentation of a filled-out solution. It is possible, although unlikely, that a given list of words has no legal solution (as in crosswords, the diagram must not be disconnected). Your program should report any failure to find a diagram and any conditions, such as a repeated word, that destroy uniqueness. For extra credit, make a nice graphic display of your solution.

Performance Practice The quality of a Kriss-Kross diagram is proportional to its "connectedness"; that is, the more tightly bound the average word is to its neighbors, the more interesting the puzzle. Connectedness might be measured in a variety of ways, including the ratio of the area of the diagram to the area of the smallest surrounding rectangle, the average number of intersections per word, the average number of intersections per character, or the minimum number of intersections per word. A commercial program has been used to generate Kriss-Kross puzzles for publication, and the puzzles are uninteresting because they are much too long and snaky. Once your program is running, care should be taken to improve connectedness.



Ass
Cat
Dog
Ilyx
Hyena

Shoat
Tiger
Codfish
Hippopotamus

Figure 7-1: An Example Kriss-Kross Puzzle

This problem is a classic for a backtracking solution. Start filling words into an interlocking diagram until no word left on the list will fit. Backtrack, removing the last word successfully fitted, and try to fit a new word. You will need to develop some heuristic to choose the next word to be fitted from the list of unused words. A check for uniqueness should include a test to see that no two words of equal length can be swapped in the diagram. Is this test all that is necessary? Is there a more elegant one? A complete algorithmic solution maximizing connectedness would undoubtedly have considerable theoretical interest.

Orchestration This problem is open to a variety of approaches but suggests a need for flexible data structures to record the progress of the program, and also facilitates good string and pattern manipulation. SNOBOL and PL/I are candidates. PASCAL has the data structures, but string manipulation will have to be built by the programmer.

Playing Time One person for 4 weeks. One additional week for graphic output.

REFERENCES

Armbruster, Frank. *Computer Crosswords*, Troubadour Press, San Francisco, CA, 1974.

This is the book that inspired the etude. The puzzles themselves are not of the highest possible quality. Perhaps your solution might be better.

Mazlack, Lawrence J. "Machine Selection of Elements in Crossword Puzzles: An Application of Computational Linguistics." *SIAM J. Comput.*, 5, 1, pp. 51-72, March 1976.

Mazlack describes a program that attempts to fill a crossword puzzle diagram with words from a very large vocabulary. The diagram and vocabulary are both given to the program, and presumably a human must generate clues for the words inserted to make a finished puzzle. This problem is similar to that of constructing a Kriss-Kross diagram, and you may be able to get some ideas about an attack from Mazlack.

8

Theseus

or...

AUTOMATIC CREATION OF MAZES

Theseus had to find his way out of the Cretan maze or perish at the hands of the Minotaur. You will be amazed at how hard it is to get into a maze in the first place.

A complete description of all possible mazes is probably beyond the scope of this book, but let us consider simple mazes built within an m by n rectangle for positive integers m and n. The rectangle is thought of as completely covered by a unit square grid of walls (the walls also include the edges of the rectangle). To construct a maze from the gridded rectangle, knock out any one unit wall on one edge of the rectangle, the entry point; any one unit wall on the opposite edge of the rectangle, the exit point; and any number of strictly interior walls. The maze has a solution if and only if there is a sequence of straight-line segments, not exterior to the maze, connecting the entry point to the exit point without touching a wall. The solution is unique if any two such paths always pass through exactly the same set of interior grid cells. Figure 8-1 is an example of a 6 by 6 maze.

Statement of the Theme Write a program that, given inputs m and n, will generate an m by n rectangular maze (check for degenerate values of m and n). Whenever the program is called, it should build a different maze, and each maze should have a unique solution. To make the output interesting, every cell should be connected to the main solution path. If you have some nice graphic device available, use it to draw your mazes; otherwise think of some notation to describe them or use the line printer to draw the output.

Performance Practice The requirement that any two mazes be distinct, even for the same values of m and n, is theoretically impossible to meet, for there are only a finite number of mazes of any given size and the program might be called more times than there are mazes. But there are a very large number of mazes of any one size, and you can make the probability very small that a maze will be duplicated. Apparent uniqueness is achieved by using an externally observable but uncontrollable value to control some "random" choices within the program (usually the date and time at which the program is called are used). The choices might include the positions of the entry and

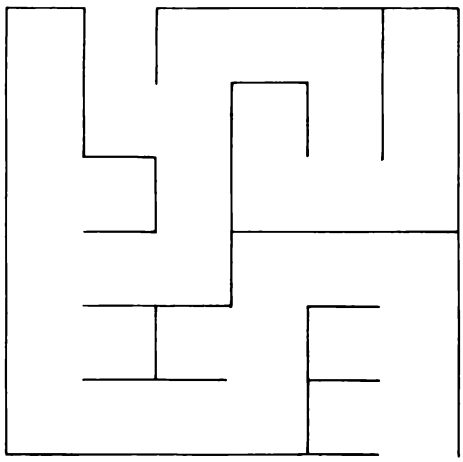


Figure 8-1: An Example Maze

exit points and the locations of at least some of the destroyed interior walls. During debugging it is a wise precaution to disable this randomness so that changes in output are the result of changes in the program.

One way to approach the problem would be to select an entry point, extend a solution path from the entry one grid square at a time, stop the solution path when it reaches the exit edge, and then knock down sufficient interior walls to connect all squares to the solution path. To keep the solution path from being boring, it should be allowed to make random turns as it proceeds. The program must check that extending the path or opening up side squares does not destroy the uniqueness of the solution. The observant reader will have noted that the definition of a unique solution does not quite meet the case when a path dips into and back out of a dead-end side alley. The spirit is right, but you will want to try to get the definition technically correct.

Orchestration This program can be written nicely in almost any procedural language. Use it to compare languages on the basis of their control structures, built-in data structures, and run-time efficiency.

Playing Time One person for 3 weeks.

9

Know Thyself

or...

PROGRAMS THAT PRINT THEIR OWN SOURCES

Philosophers regard introspection as an important mental tool. All right-minded persons should heed the chapter title. If people can attain self-knowledge, why not programs? And what better way than by writing an autobiography?

Statement of the Theme Write a program that prints an exact duplicate of its source. The output should not include any "control" cards or other local system material—merely what you would expect to keypunch and give to a compiler. However, your program must not read any input and it must not rely on any system "tricks," such as knowledge that the local compiler leaves a copy of the source program in blank COMMON. The program should produce the same output no matter where or when it is run.

Performance Practice If you begin to despair of a solution after your thirteenth attempt has failed, fear not. Such a program is called introspective, and a theorem exists that says that all "sufficiently powerful" programming languages can express an introspective program. Every normal programming language is sufficiently powerful. The solution simply requires looking at the language in the right way; it will probably take about 30 or 40 lines at most.

Orchestration This problem can be done in any language.

Playing Time One person for 1 week.

REFERENCES

Bratley, Paul, and Jean Millo. "Computer Recreations Self-Reproducing Automata." *Software—Practice and Experience*, 2, pp. 397-400, 1972.

This article should be read only as a last resort, for it gives a complete solution of the problem.

Rogers, Hartley, Jr. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, New York, NY, 1972.

Rogers' book is an excellent introduction to recursive function theory, hard but clear. Chapters 1 through 3 provide a good foundation. Sections 11.1, 11.2, and 11.4 contain results about introspection.

10

...Yielding Up Its Gold

or...

CALCULATION OF INVESTMENT YIELD

Financiers, speculators, bankers, even ordinary working people like the treasurer of the Teamsters' pension fund like to know how much their investments are earning for them. If the money is in a savings account, there is little difficulty, since banks trumpet their interest rates in all their ads. Even if you invest in a bond that not only pays interest but that you also later sell at a profit, simply add your profit and the interest and figure out how much a bank would have to pay to match your earnings. The result of these calculations, if expressed as an annual percentage that will give the known return when compounded continuously, is the investment's yield.

The situation is not so simple, however, if the investment is one—like a mutual fund, a stock account, or a small business—in which there are irregular payments and withdrawals and in which the current value changes from day to day. A mutual fund is a good example; new shares can be bought at any time at market value; old shares may be redeemed similarly; dividends vary (and even vanish) with fund performance but are normally plowed back into more shares; and the value of a share changes daily as the underlying securities change value. It certainly would be nice to compare the yield on a savings account with the rosy picture painted by an investment fund prospectus, realizing, of course, that yield is usually proportional to risk.

Fortunately, there is a formula for the calculation of yield in these circumstances. Unfortunately, the formula is iterative rather than in closed form. Assume that A is the current value of the investment, that there are m transactions involving the investment, that transaction i was in the amount P_i (where a negative value indicates a withdrawal) and occurred T_i years ago, and that our initial guess at the yield is Y_0 and has the value zero. Now let

$$C_j = A - \sum_{1 \leq i \leq m} P_i \exp(Y_{j-1} T_i)$$

Table 10-1 A Record of a Real Investment

Date	*Value*	*Transaction Amount*
3/11/71	$0.00	$68.26
5/ 4/71	73.75	50.00
6/ 4/71	114.82	75.00
8/ 9/71	170.66	50.00
9/ 7/71	229.41	54.00
10/ 4/71	282.97	50.00
10/ 8/71	326.02	4.31
12/ 8/71	328.11	50.00
1/ 1/72	391.65	0.00
2/ 4/72	413.42	50.00
8/ 1/72	471.35	0.00
10/ 5/72	440.83	7.72
10/ 5/72	448.55	4.14
10/ 2/73	398.36	4.80
1/ 2/74	330.74	0.00
6/ 7/74	360.97	-200.00
10/ 3/74	180.42	50.00
3/13/75	253.96	-200.00

and $$D_j = \sum_{1 \leq i \leq m} T_i P_i \exp(Y_{j-1} T_i)$$

for $j > 0$. Then the better estimate Y_j of the yield is given by

$$Y_j = Y_{j-1} + C_j/D_j.$$

As soon as

$$|Y_j - Y_{j-1}|$$

becomes sufficiently small, the yield has been found.[1] In reading Table 10-1, notice that A is the sum of the column headed Value and the column headed Transaction Amount. If we look at row 3, this shows A = $189.82, P_1 = $68.26, P_2 = $50.00, and P_3 = $75.00, $T_1 \cong 205/365$, $T_2 \cong 31/365$, and $T_3 = 0$. Also notice that Y_0 starts over at zero for each row of the table and that the calculation of the yield at any date is not influenced by the yields at previous dates.

Statement of the Theme Write a program that calculates the yield on an investment. The input consists of the records of a series of transactions, each containing a date, a transaction amount, and the value of the investment on the date of the transaction *before* the transaction takes place. The input is assumed to be in order of dates; the program should check that the ordering is not violated and that no withdrawal exceeds the current investment value. The program should print a neat tabulation of the transactions. Each output transaction entry should contain the date, the old investment value, the transaction amount, the new investment value, the yield on the transaction date, and the totals to date paid to and withdrawn from the investment. The method used to note

[1] Readers familiar with calculus should be able to see that the formula is an application of Newton's method of root finding to an equation whose free variable is the yield.

the end of the input is up to the programmer, but a transaction amount of zero is a convenient way of finding out the current yield. If you have no investments of your own and cannot afford a *Wall Street Journal*, Table 10-1 is a record of a real if unfortunate investment.

Performance Practice There is an interesting sidelight to this problem. The transaction dates are given in the normal day/month/year format, but the problem requires the elapsed times T_i in units of years. Bankers and lawyers have a number of ways to calculate the time that money is out at interest (one suspects the method depends on who owes whom). For the program, it is sufficient to calculate the years as a real number, taking into account leap years, and assuming that all dates are in the range 1900 to 1999 inclusive. In general, conversions between different time measures and calenders can be fairly difficult.

Orchestration Any procedural language with real-number facilities is appropriate.

Playing Time One person for 1 week.

11

Ye Soule of Witte

or...

TEXTUAL REDUNDANCY AND FILE COMPRESSION

It is well known that most people talk too much. It is less widely known that even the pithiest sayings could be considerably compressed. Natural languages are extremely redundant. Evn if qt a fw ltrs r lft ot, u cn prbly rd ths sntnc. Languages used in computations share this feature. With computer memory quite costly, it makes sense to remove the redundancy from text so as to avoid storage charges.

There are several possible techniques for text compaction. The most obvious is to look for any long string of one repeated character. The long string will be replaced by a character triple mcn, where m is some special marker not otherwise used in the text, c is the repeated character, and n is the length of the long string. The trigram saves n-3 characters, so long as n is no bigger than the largest number that can be stored in one character position. This process works quite well for text having long repeated strings, such as the long strings of blanks common in most computer programs. Unfortunately, it does not work as well for other text, since most data is not as highly formatted as programs.

A second technique relies on the fact that in many computer character sets most of the characters are not used (in the common 256 character 8-bit sets perhaps 100 characters are used ordinarily). The most common digrams of the text are ascertained and each is assigned one of the unused single characters. Text is compacted by replacing, from left to right, the common digrams with their single-character encodings. A considerable savings can be made because the most frequent 150 digrams, say, are a large proportion of natural language text. By giving up a little compaction, one may write quite efficient encoding and decoding routines that operate on the computer representations of the characters.

But a difficulty still exists. Why should the most frequent digrams of English be the same as those of French, or of an address file data set, or of ALGOL? And even if the same, what about trigrams, quadrigrams, or longer sequences? Longer sequences offer greater savings even if less frequent; in a long piece of text some particular fragment may come up much more often than normally expected. And how were the digram frequencies ascertained in the first place?

The solution to all these problems lies in a third approach to the original question. Instead of using some preordained encoding, the actual text to be compressed, or a sample of it, can be used to generate a code dictionary on the fly. Since each body of text will be used to generate its own dictionary, there will be no difficulty with inappropriate abbreviations. Now we must find a way to build such a dictionary.

The rough outline of the scheme is as follows. Begin with an empty dictionary. Start scanning the text from left to right. Find the longest match between the head of the text and any entry in the dictionary and increment the frequency count of the entry. If there is no match, make the first letter of the text into an entry. Delete the head of the text just matched and start matching again. In circumstances to be explained below, two entries are sometimes coalesced into a longer one. When the dictionary fills up, thin it by weeding out the least common entries and then continue the scan. When the dictionary frequencies become stable, assign codes and go back and encode the whole text.

There are two unresolved points in the scheme suggested: how are entries coalesced and how are entries thinned? Two entries are coalesced when a match for the first is immediately followed by a match for the second and both have frequencies above some threshold. The new entry may be given a somewhat larger than normal initial frequency to prevent it from being weeded immediately. Thus if THO and SE are already in the dictionary, THOSE will be added when seen, if the original two entries have high enough counts. A simple strategy for weeding is to eliminate all those strings whose counts are lower than the mean. Another strategy might eliminate all those strings below the median frequency. Other similar strategies might be employed.

A DICTIONARY CONSTRUCTION ALGORITHM

This algorithm assumes that some sample of the text to be compressed is available as a dictionary construction aid. All characters are significant to the algorithm, and if line ends, tabulates, and similar items are important in the text, they should be contained as characters in the text stream. At the beginning of the algorithm, the dictionary is assumed to be empty. The variable **last match** initially has the null string as value, and the variable **last count** initially has the value zero.

1. Find the longest string **match** at the head of the input that matches any entry in the dictionary. If **match** is null, set **match** to the first character of the input, add **match** as a dictionary entry, and give it an initial count of one. If **match** is not null, increment the count of the matching entry by one. Set **count** to the count of **match** in the dictionary.

2. If either **count** or **last count** is less than the coalescence threshold, go to step 4. A possible coalescence threshold is the maximum size of the dictionary divided by the number of free entries remaining in the dictionary.

3. Form a new entry by catenating **last match** and **match**. Give this entry an initial count of one because the catenated entry has been seen once. Other strategies are possible.

4. If the dictionary has fewer than two free entries in it, thin by weeding out all entries whose frequencies are less than the median frequency. If the entry for **match** happens to be deleted, set **count** to zero.

5. Delete **match** from the head of the input. If the input is exhausted, exit. Otherwise set **last match** to **match**, **last count** to **count**, and return to step 1.

ENCODING AND DECODING

Once dictionary construction ceases, the encoding and decoding tables must be built. Form all possible digrams beginning with a character that can never appear in the text. Delete from the

dictionary all entries of only one or two characters (there can be no savings in compressing them). Sort the remaining strings by frequency. Assign the coding digrams formed above to the dictionary entries, beginning with the most frequent. If either entries or digrams run out, the encoding table is constructed.

The text is encoded in a process that is similar to dictionary construction. At each step the longest match possible is made between the head of the input and the dictionary entries. The matching string is replaced by the coding digram, and the input scan is moved past the match. If no match is found, simply copy the single character at the head of the input to the output and move the scan one character right. Decoding simply requires the replacement of coding digrams with their dictionary equivalents.

Statement of the Theme Write a program to implement the dictionary construction, encoding, and decoding algorithms described above. Test the program on sizable fragments of natural and programming language text. The compression rate for a given piece of text is the quotient of the sum of the sizes of compressed text and the decoding dictionary divided by the size of the original text. Run a small study of the effect on the compression rate of any of the following parameters: the language to the compressed; the length of the text sample used for training; the size of the dictionary during construction; the number of coding digrams available; or the use of a dictionary constructed from one text used on another text from the same language.

Performance Practice This problem is interesting because its efficient solution requires the use of some fairly sophisticated algorithms and data structures. But a successful if inefficient program can be written with simple algorithms and structures, which can be replaced piecemeal with neater solutions once the program is running. One example is the median calculation needed to thin the dictionary. As a first attempt, throw out all entries with frequency less than the mean frequency. The mean can be calculated easily from one running total of all the frequencies in the dictionary. After the whole program works using the mean, the more complicated general median routine can be used to find the deletion threshold.

Another example is the structure of the dictionary during the construction and encoding phases. The entries can be kept in random order, in which case a potential match must be tried against every entry. With such a structure, however, new entries can be added by appending them to the end of the dictionary. A little more sophistication would keep the entries grouped by length; the search could go from longest group to shortest, stopping at the first match. If each group were sorted alphabetically, a binary instead of a linear search could be used within each group, thereby saving time. But now additions will be more complicated because each new entry will probably require space somewhere in the middle of a group. Perhaps the most efficient structure for searching is some kind of tree. Paths from the root to the leaves could spell out potential matches, or the matching entries might live in the nodes *a la* a binary search tree. Trees will require much more maintenance during dictionary construction than the simpler structures mentioned above.

Orchestration Because of the diverse data structures that will be required by a completed program, the source language should have good definitional facilities. The candidates include PASCAL, ALGOL 68, and PL/I. One approach would be to write the program once in SNOBOL, relying on the built-in pattern matching, and then rewrite the completed program in some more efficient language for production. If this approach is used, care must be taken to avoid SNOBOL features not easily copied.

Playing Time One person for 3 weeks.

Variations on the Theme There are three areas of freedom in this model: the criterion for coalescing entries, the criterion for deletion of low-frequency entries, and the scheme for encoding the entries. Taking them in order, we begin with the criterion for coalescence. Our algorithm requires that two successive entries each pass the same hyperbolic threshold before coalescence can occur. But it is possible to have different criteria for the two entries. Two other candidates are a constant threshold and a threshold that is a function of the mean entry frequency. Similarly, the initial frequency of a coalesced entry might be varied, with any policy that sets high initial frequencies providing a better chance for entry retention.

The policy on entry deletion during dictionary thinning can be modified in the same way. A fixed fraction of the low-frequency entries can be weeded (using the median sets the fraction at one-half). All the entries with frequencies less than some multiple of the mean frequency might be dropped; or everything with frequency less than some constant could be dropped, a procedure that stops when thinning does not empty enough of the dictionary. Combination of a coalescence and a deletion policy will produce a specific retention characteristic. Some combinations retain strings that occur densely in one section of the text and less often elsewhere; others favor strings that are scattered evenly throughout the text. Which retention characteristic is preferred depends on the use of the dictionary and text features.

The encoding algorithm uses digrams beginning with unused characters. But if the digrams run out before the dictionary does, trigrams and so on can be added. Since frequencies for the entries are known, they can be used to construct a weighted variable-length encoding. This procedure will cost during decoding (why not during encoding?) but will provide even better compression.

REFERENCES

Mayne, A., and E. B. James. "Information Compression by Factorising Common Strings." *Comput. J.*, 18, 2, pp. 157-160, 1975.

This etude is basically a restatement of Mayne and James. Our version of the algorithm is cleaner than theirs. Their paper does present some production results.

Knuth, D. E. *The Art of Computer Programming, Volume 3/Sorting and Searching.* Addison-Wesley, Reading, MA, 1973.

Although reading any part of Knuth is a valuable pastime, Section 6.2 on Tree Searching should be particularly appropriate for this problem.

12

A Sense of Community

or...

BOOKKEEPING FOR HOME USE

Almost every student lives in some sort of cooperative at one time or another. The organization may only involve sharing rent or it may be as tight and formal as a fraternity. However organized, there is always a need to keep and render accounts. All too often communes have broken apart over the issue of money. At least the computer can provide an honest accounting even if it cannot resolve deeper problems.

Normally accounts are settled at the end of the month, perhaps right after the biggest expenditure, the rent, has been paid. Throughout the month members have been paying expenses individually. Whoever went shopping paid for the groceries; whoever answered the door paid the paperboy; whoever drove the car bought the gas. With luck, most members will have about paid their shares, of course, the result will never come out quite even.

And if the expenses are not shared equally, the accounting is not going to be a simple division. It is common to find one member agreeing to pay a little more rent in order to get an extra room or another member who eats at home on weekends paying a smaller share of the food budget. And it is also common for a member to charge a personal purchase, such as a long-distance phone call or a favorite beer, against the group for settlement at month's end. All this activity requires a consolidated bookkeeping system.

Statement of the Theme Write a program that will provide itemized accounts for a small commune. The input comes in four sections. The first section should give the names of the members for the month. The second section should give the major accounting categories for the bills, such as groceries, rent, utilities, and garden supplies. Each category might be followed by a list of members and share amounts. The share amounts can be either dollar amounts or percentage amounts. The portion of a category that is not specifically allocated is split equally among the other members. Thus if the rent were $200, member A were allocated $45, and member B 35%, each of the other members would pay equal shares of the remaining $85.

Each item in the third input section might be the record of a member's payment on behalf of the commune. The item should contain the date, the member's name, the amount, the major accounting category, and a brief description. Similarly, section four might be a record of those items purchased specifically for one member and should contain the same information as section three items, with the obvious addition of the name of the member who owes for the item. All input should be checked for consistency, with particular attention to dates, amounts, names, and categories.

Output should also come in several sections. First, each member should receive a list, sorted by date, of all payments and debits incurred during the month. Secondly, each member should receive this same list organized by category and date. The list should also indicate the member's obligation in each category and its breakdown into normal share and personal debits. Finally, each member should receive an indication of financial status for the month. Members who owe money should be told who is to receive it, and members who are owed money from whom to expect it. The program should try to keep the number of these balancing transfers as low as possible.

The final output section should be a chronological listing of all commune payments and a chart broken down by member and category of payments, debits, shares, and balancing obligations. If this chart is cross-totaled in both directions, it can provide a check on the bookkeeping accuracy.

Performance Practice There is nothing particularly difficult about this problem. Although efficient programs are always desirable, in this case input and output will certainly overshadow computation. The input sections with their varying sizes are a small challenge. Similarly, checking the input can be done with a certain elegance. Basically, this is a mundane program like most of those actually written in industry. Provide a workmanlike solution.

Orchestration Although COBOL is a standout, almost any procedural language can be used.

Playing Time One person for 2 weeks.

Variations on the Theme A feature provided by most business-oriented programming languages is the exact calculation and edited output of dollar amounts. Ordinary real-number calculation may cause pennies to be dropped or added here and there; cross checks may not balance. This is a chance to implement some simple fixed point (but not integer!) routines. If your program is in FORTRAN, it is also a chance to figure out how to print those pesky floating dollar signs, trailing credit indicators, and leading zeros. In COBOL or PL/I, there will be no difficulty.

13

Touring Turing

or...

SIMULATION OF A TURING MACHINE

Well before the first general-purpose digital computer was built, Alan Turing became interested in the limits on the computations that a machine could perform. To convince himself that he was not building into his hypothetical computer any complicated mechanism that would invalidate measurements of the machine's powers, he stripped out almost all of the features that may seem essential to actual computers. All that he left was a simple kind of program storage that cannot be manipulated during exeucution, only one kind of instruction, and a simple tape for input and output. But this device, the Turing machine beloved of 40 years of logic students, is capable of all the calculations of any modern digital computer. A mean problem would be the simulation of an IBM 370/155, say, on a Turing machine; this much nicer problem will turn the simulation around.

A Turing machine consists of a control unit attached to an input/output tape by a tapehead. The tape is a long strip of cells extending to infinity on the right (that is, there is a little factory that makes more tape to add to the right as needed) with each cell capable of holding one character. The tapehead points to some one cell on the tape and can both read and write and move either left or right. Execution always starts with the input written left-justified on the tape and the tapehead reading the leftmost tape cell. Whenever the tapehead moves right onto a cell that was *not* a part of the input and that has never been visited before, the cell is assumed to have a blank, written ϕ, in it.

The control unit executes the program under a set of strict rules. At each time instant the control unit is in some state named by a positive integer and stored in the current state. Every instruction of the program is a quintuple consisting of a state, a character, another state, another character, and a direction to move the tape. An instruction cycle begins as the control unit compares the current state and the tape character under the tapehead with the first two members of every instruction quintuple. By the rules of Turing machine programming, there is at most one quintuple with any particular state-character initial pair (and there may be none). When a match is found, the control unit causes three things to happen. The character under the tapehead is overwritten with the fourth member of the quintuple; the tapehead is moved one cell left, one cell right, or remains stationary, as indicated by the fifth member; and the current state changes to the third member. The machine

is now ready for a new cycle. By convention, it is always started in state 1 with the tape as described. The machine halts if the instruction cycle cannot find a match for the current state-character pair or if the tapehead falls off the left end of the tape, and the value is whatever remains on the tape after the halt. Note that a program may only have a finite number of instructions so that only a finite number of states and characters will be meaningful to it.

An example may make this discussion clearer; here is an ancient one. We would like to write a Turing machine program that will form the sum of two integers. The integer n will be represented on the tape by n consecutive *'s (no *'s represents zero), the two input values will be separated by a comma, and if the inputs represent n + m, the output should be n + m *'s left-adjusted. Thus the initial input for 7 + 4 is

*******,****

and the output should be

****

The structure of the program is simple. First, the tapehead will move right looking for the comma (remember that the tapehead starts at the left of the input). The comma is replaced with a * and the tapehead continues moving right looking for the blank that bounds the input on the right. The tapehead backs up one cell, writes a blank over the * that it finds there, and then the program quits. It is easy to see that one * has been added in the middle and one taken away at the right to form the sum. The actual program is given in Table 13-1.

A standard way to show the status of a Turing machine is to print a picture of all the tape that has ever been scanned, along with the current state inserted just to the left of the cell currently scanned. Such a display is an instantaneous description, and the following example starts the addition of 2 and 3:

1**,***

The sequence of instantaneous descriptions tracing this computation is given in Figure 13-1. Note that the program halts in state 3 because there is no action for a blank. State 4 is activated only if there is an error in the input; if so, the machine goes into an endless loop. Check for yourself that the program works if either input (or both) is zero.

The example program may seem too easy. Try to modify it to multiply instead of add. Turing machines operate much more naturally in unary than in any other radix; a program to add in decimal will be more difficult and longer. The references contain far more material on Turing machines and substantiate the claim that a Turing machine can do any calculation that any other

Table 13-1 A Turing Machine Program

Old State	*Old Character*	*New State*	*New Character*	*Move*
1	*	1	*	Right
1	,	2	*	Right
1	␣	4	␣	Stay
2	*	2	*	Right
2	,	4	␣	Stay
2	␣	3	␣	Left
3	*	3	␣	Stay
4	␣	4	␣	Stay

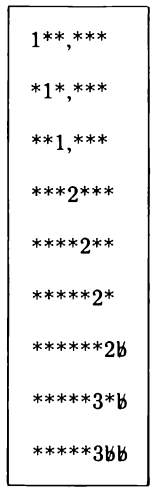


Figure 13-1. A Sequence of Instantaneous Descriptions

computer can do. You will find a number of minor variations in the various machine descriptions, as well as proofs that these variations do not matter at all.

Statement of the Theme Write a general Turing machine simulator. The input is the Turing machine program, its input tape data, and, for a reason to be explained later, its initial state. The output is a trace of the machine's execution and the final value on the output tape. Since Turing machines need not stop, and since there is no way to tell in advance if such will be the case (look up the halting problem if you do not understand why), there must be some control on the amount of output that the simulator can generate and the time that it can spend. Test the simulator on several programs like those discussed above.

Although the program states had the positive integers for names in our description, your simulator should allow any identifier as a state name. In the preceding example we might have used the state names *Begin*, *MoveRight*, *Finish*, and *Error*, and an instruction quintuple might be

MoveRight ␣ Finish ␣ Left

Since there is no longer any unique first state, it will have to be named by the user.

Performance Practice As in any simulator, efficiency will be a problem with this program. If state names are used throughout the computation, the continual lookup will cost considerable time. Indeed, the fastest way to arrange the Turing machine program is in a two-dimensional array indexed by states and characters. The array entries contain the instructions to be executed with a special entry meaning no quintuple specified. Of course, the difficulty is that you will not know how big to make the array until after you read the input. Incidentally, the input should be checked for consistency to ensure that two different quintuples do not start with the same state-character pair.

The trace output should be printed after each change of state and should include all the tape up to the rightmost nonblank or the tapehead, whichever is farther right, the tapehead position, and the current state. The tape image should probably be printed on one line and the tapehead marker and the state on the next. Let aesthetics and clarity be your guide. The tape alphabet, the set of characters that might occur on the tape, is simply the set of characters that occurs anywhere in items

two or four of any quintuple. You should allow any normal character available on your system. The alphabet always includes the blank, which will be hard to represent on input and may be confusing on output. One way around the input problem would be to separate the five fields of an instruction by commas. Handling significant blanks is often a problem. Blanks are meaningful in natural languages but usually only as word separators and not as symbols in their own right. Thus there are no "normal" conventions for their use as symbols.

Orchestration This is another problem in which almost any source language will have both advantages and disadvantages; because of the tight inner loop, however, interpretive languages should probably be avoided.

Playing Time One person for 1 week.

REFERENCES

Davis, Martin. *Computability and Unsolvability.* McGraw-Hill, New York, NY, 1958.

Davis carries out in excruciating and precise detail all the proofs that other authors "leave to the reader." After reading through Davis, you will never again doubt any of the claims about the power of the Turing machine. Of course, you may never want to hear of a Turing machine again.

Hopcroft, John E., and Jeffrey D. Ullman. *Formal Languages and Their Relation to Automata.* Addison-Wesley, Reading, MA, 1969.

Hopcroft and Ullman is the best first-year graduate text in its area. It provides all the basic results about Turing machines and sets them in the context of other classes of automata. This book is also a valuable reference work.

Minsky, M. L. *Computation: Finite and Infinite Machines.* Prentice-Hall, Englewood Cliffs, NJ, 1967.

Minsky provides a fine, easygoing introduction to automata theory. This is probably the best book to start with.

14

Games Computers Play

or...

A COMPUTER STRATEGY FOR KALAH

The argument over "intelligent" behavior by computers has raged since before their actual existence. Most people will agree that strong play in any intellectual game that does not have a complete analysis would count considerably toward intelligence. If the computer could learn as well as play, intelligence would be even harder to deny. The game most often associated with computer play is chess, but the very best programs play only mediocre games. There are other games in which greater success is possible.

Kalah, variously known as Mancala, Wari, or Owari, is an African game of great antiquity. Little written analysis exists, but Kalah has been played at the drop of a stone by people of many cultures for centuries. Although it is a game of skill with no chance element, Africans gamble continuously over Kalah. Simple equipment and rules make Kalah a natural for computer play. Current research suggests that computers with a program like the one suggested here already play Kalah better than any humans.

The Kalah board looks like the diagram in Figure 14-1. Each of the two players sits along one long edge and owns the six smaller pits, along the near side and the larger right-hand pit, the Kalah. To start the game, each small pit is filled with some number k of stones (there is a complete solution known for $k \leqslant 3$ and Africans usually play with $k = 6$). A player moves by picking all the stones out of some one small pit on the player's side and sowing them into the other pits counterclockwise around the board. The sowing begins in the pit to the right of the source pit and includes the player's own Kalah and the opponent's small pits but not the opponent's Kalah. It is possible and legal for the sowing to loop all the way around the board to the source pit and beyond. Figure 14-2(a) and (b) shows the before and after of such a looping move.

There are two variations on the basic move. If the last stone sown falls into one of the moving player's own small nonempty pits and stones were played on the opponent's side during the sowing, the stones in the final pit are used to start a go-again move, just like the original move. A player can have an arbitrarily long chain of go-agains. If the final stone sown falls in one of the opponent's small pits and there are either two or three stones in the pit, the stones are captured and placed in

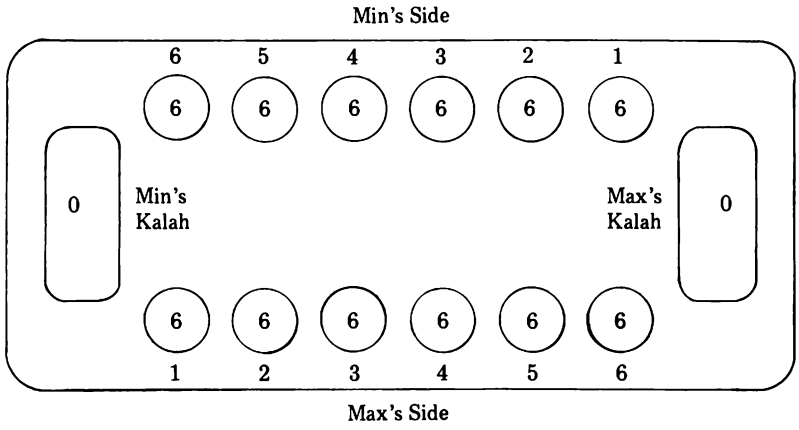


Figure 14-1. A Kalah Board. The numbers in the pits denote the numbers of stones therein.

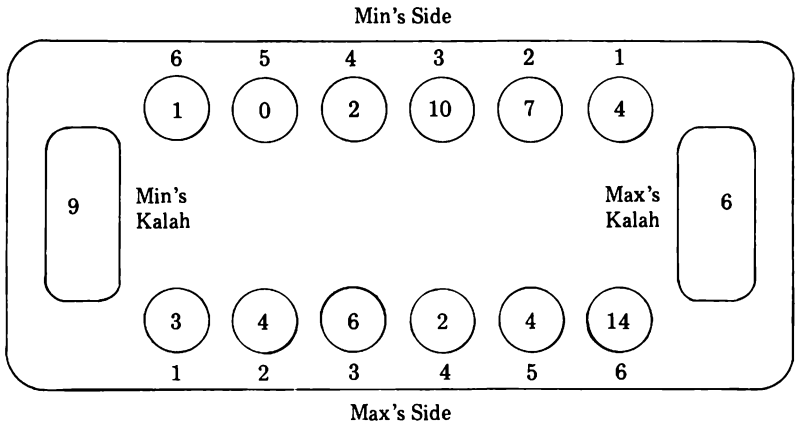


Figure 14-2(a). Before a Looping Move by Max. Max moves from pit 6.

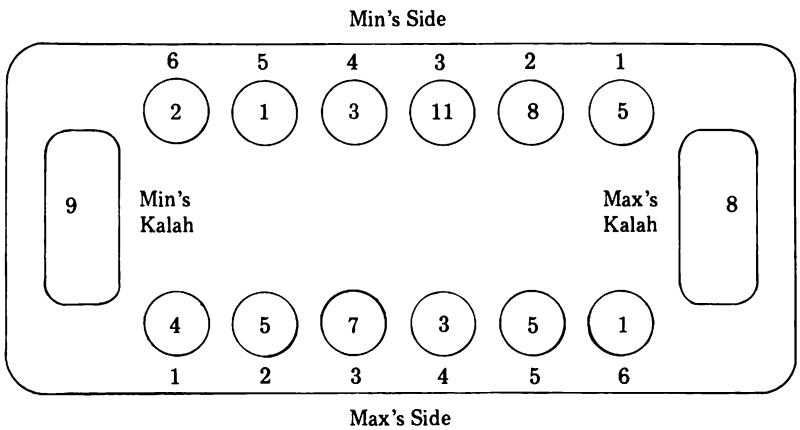


Figure 14-2(b). After Max's Looping Move. Max's kalah has been sown twice.

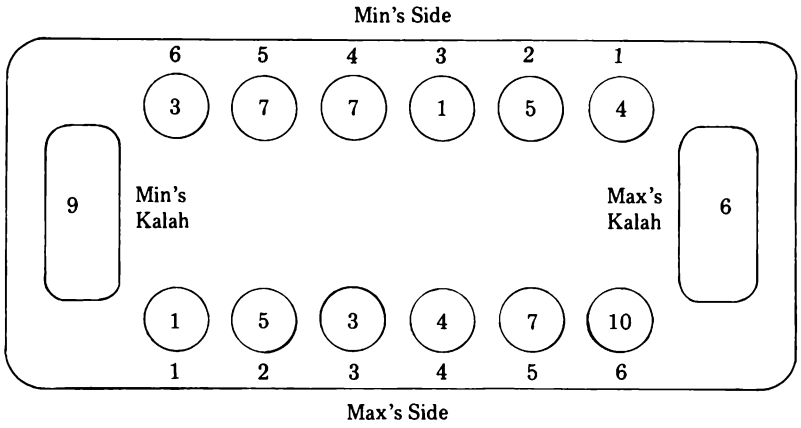


Figure 14-3(a). A Go-Again Move from Max's Pit 6. The last stone drops in Max's pit 3 for another go.

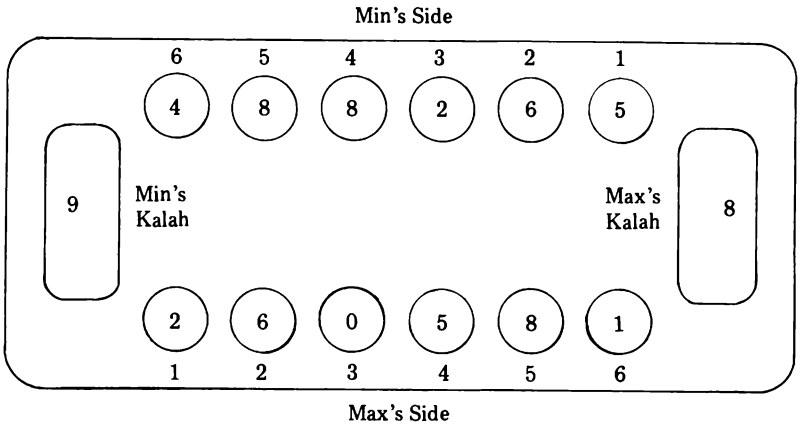


Figure 14-3(b). After the Go-Again Move. The stones from pit 3 have been sown forward.

the moving player's Kalah. Whenever one pit is captured, the preceding pit may be captured if it contains two or three stones as well. Theoretically, a player could completely clear out an opponent's side in one move. The game is over as soon as more than half the stones are in one player's Kalah (notice that once a stone enters a Kalah, it can never leave). If a player who has the move has no stones available, the game ends immediately with all the opponent's stones going into the opponent's Kalah. Figures 14-3 through 14-5 show some typical moves.

The construction of a program to check legal moves is quite easy. A player can choose at most one of six possibilities for a move. Once the starting pit is chosen, tracing the go-agains and captures is simple. At the end of the move the check for termination requires only comparison of the moving player's Kalah with half the total stones. Of course, this does not explain how to find the best move from a given position.

The basic idea of the move selection is to build the tree of all possible continuations from a given position and then select a branch with a sure win at the end. For ease of exposition and for a reason that will be clear shortly, let us call the computer Max, the opponent Min, and assume that it is the computer's turn to play somewhere in the middle of the game. Max can try to evaluate the position by trying each of the six possible moves in turn. If any one of these moves leads to an immediate win, Max should obviously make the move. But what if none of the six leads to an immediate win? How does Max make a choice?

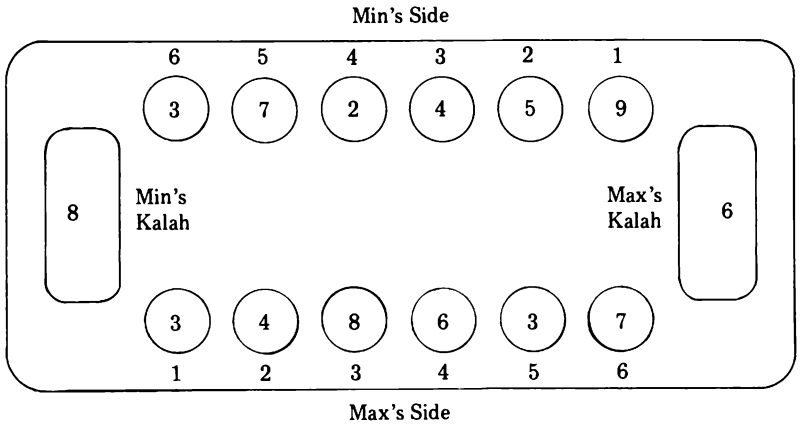


Figure 14-4(a). Before a Capture by Max. Min's pit 4 is the target of Max's pit 3.

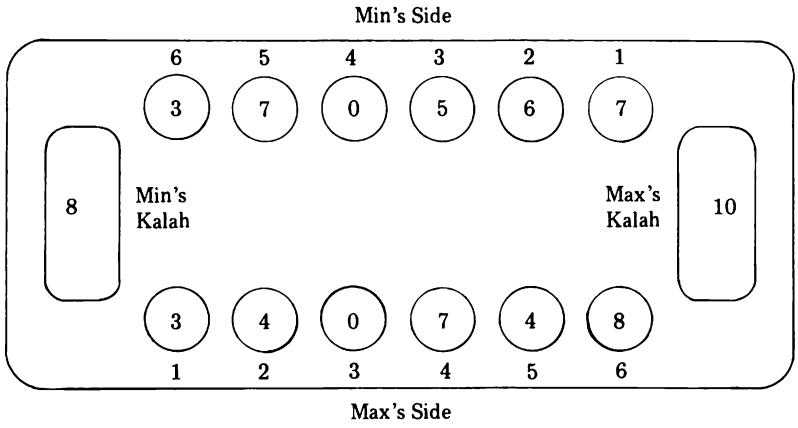


Figure 14-4(b). After Max Captures Min's Pit 4. Max's kalah grew one stone from sowing and three from the capture.

Max should calculate each of Min's responses to Max's moves. Say that one of these responses leads to a win for Min. Then Max would be foolish to make a move that gives Min a chance at an outright win (although sometimes Max might not be able to avoid it). In this case, Max would know what moves not to make. But to find out what move to make, Max will need to build another level of replies to Min's responses to Max's original moves. If Max can always find a winning reply to some set of responses by Min, then Max should select the original move that led to the response that leads to the winning reply (remember the house that Jack built?). If all this is unclear, try building the moves, responses, and replies for the positions of Figure 14-6.

Yet looking ahead two levels still might not be enough. Indeed, although a Kalah game must end,[1] it is difficult to predict how far ahead one might need to look for the end. Each level of lookahead added costs about six times as much time and space as the current level. Something must be done to halt this growth.

[1] Once in a Kalah, a stone can never leave it. Also, there are no cyclic move sequences, since every move must either put at least one stone into a Kalah or move at least one stone nearer a Kalah. By implementing the "glitch" mentioned in the next paragraph, every game must end with each stone in one or the other Kalah.

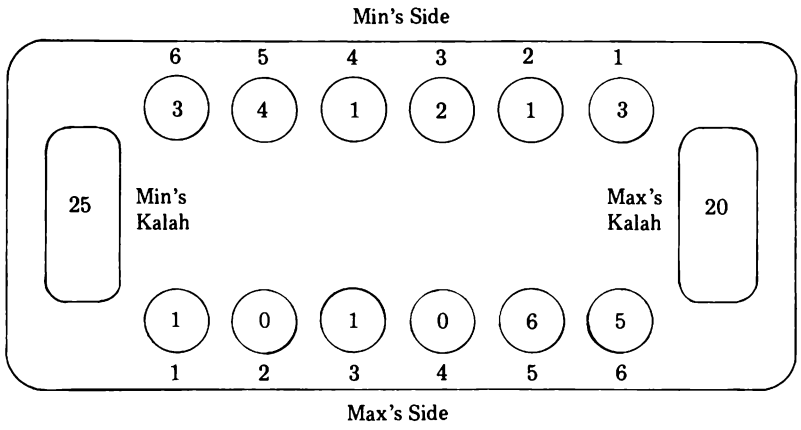


Figure 14-5(a). A Multiple Capture by Max. Min's pits 2, 3, and 4 invite capture in a single play.

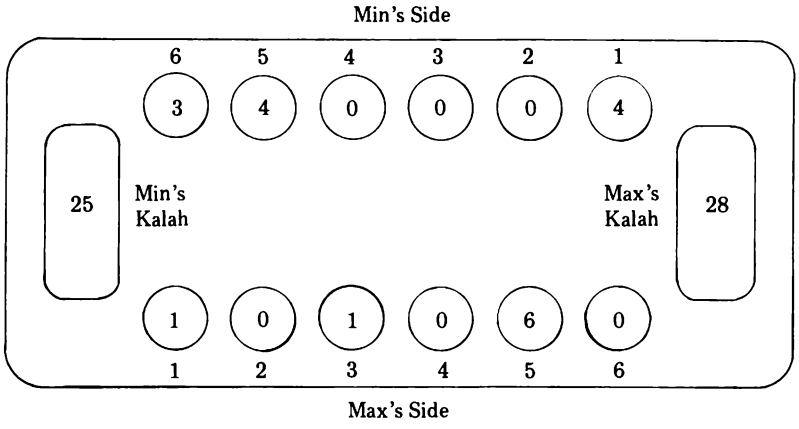


Figure 14-5(b). Min's Pits Are Nearly Cleaned Out by Max. Notice that Max could have captured from pit 5 instead of pit 6.

The answer is a static evaluation function that can estimate the value of a position without tree construction. For Kalah, we use the Kalah score formed by subtracting the number of stones in Min's Kalah from the number in Max's Kalah. If we "glitch" the rules a little bit to say that all the stones are immediately moved into the winner's Kalah, the Kalah score will always be positive when Max has the edge and will be as large as possible when Max has won. So now Max can select the one move out of six that will *maximize* (hence Max's name) the static evaluation function. If two moves are equally good, Max can choose randomly between them.

So now the question of a strategy for Max is answered. Or is it? If maximizing the Kalah score were all the strategy in the game, there would not be much to it. There must be traps that Min can lay for Max, and the way to avoid the traps is to look ahead. The static evaluation can be used to score positions deep in the tree that are not sure wins or losses.

Assume that Max wants to look ahead d levels and let the original position be at level zero. Generate all the six potential moves to level one. From each level one position, generate all the level two positions by applying Min's moves. Keep on until a full tree of positions has been generated down to level d. Occasionally six moves may not be available because one or more pits on a side is empty. Also, a branch may terminate because one player makes a move that ends the game. Notice that all moves at even levels are by Max and at odd levels are by Min.

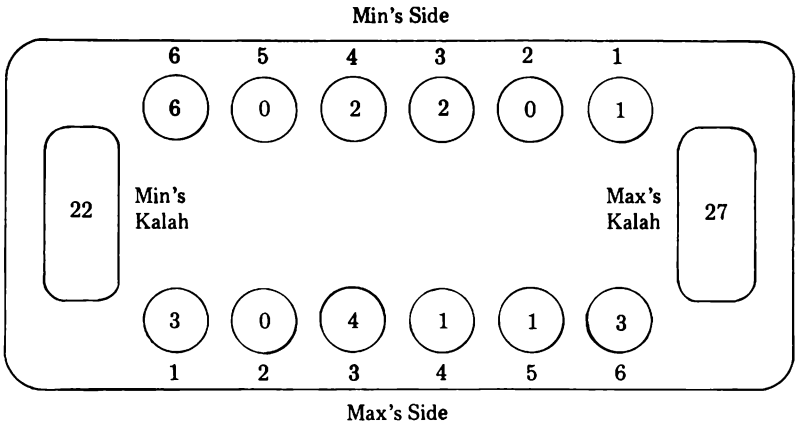


Figure 14-6(a). A Position with Max to Move. Max moves from pit 1.

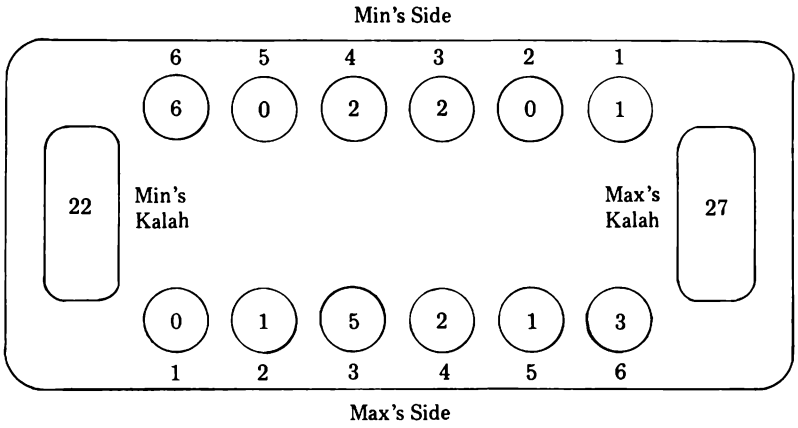


Figure 14-6(b). The Result of a Move by Max. Min will respond by moving from pit 6.

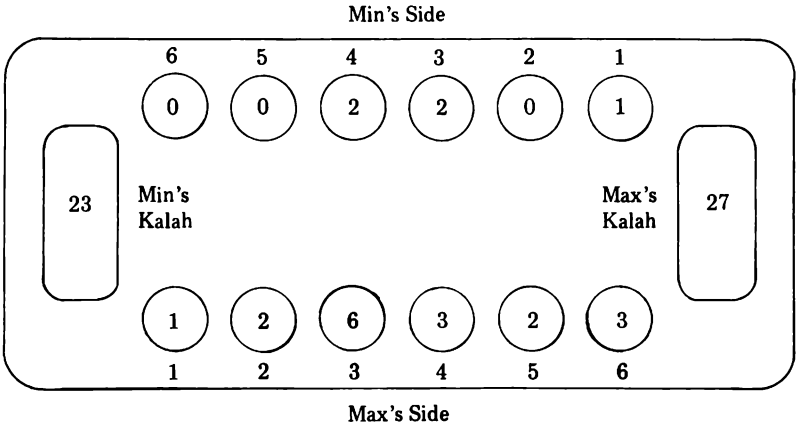


Figure 14-6(c). The Result of a Response by Min. Max will reply by moving from pit 2.

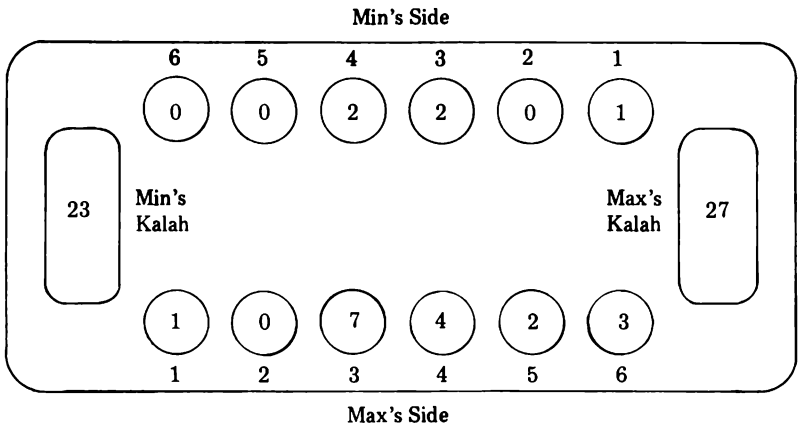


Figure 14-6(d). The Result of a Reply by Max. This is just one of about 6^3 such sequences.

Now backup a score from level d to level zero by performing the following actions at each level from d to zero. Apply the static evaluation function to each leaf at the level. Doing so gives the leaf's score. Form a nonleaf's score by maximizing over the node's immediate children's scores at even levels and minimizing at odd levels. This procedure corresponds to Max always trying to increase the lead and Min always trying to decrease (or, of course, to make it more negative). When a score has been backed all the way up to level zero, choose any of the six initial moves that achieve that score. Note that normally all leaves will be at level d. Also, the tree can be generated by going down all the branches first—that is, depth-first rather than breadth-first as described here. Figure 14-7 shows part of an example game tree. Only one branch is worked out completely down to the leaves. The backed-up values are correct from the information shown, and Max should choose the move from pit 1.

What we have described is the basic minimaxing procedure for playing two-person games. As can be seen, to look ahead d levels in Kalah requires construction of about

$$\sum_{i=1}^{d} 6^i \cong 6^{d+1}$$

positions. Because this function grows so fast, anything that saves effort is desirable. The alpha-beta minimaxing procedure can make it possible to look as much as twice as far ahead for the same amount of work.

The idea is a generalization of this example. Say that at some interior node A of the tree it is Max's move and Max has already, by depth-first search, built a complete tree B for the move from pit 1 and C for the move from pit 2. Suppose further that node B has a backed-up value of 1 and that C has a backed-up value 2. Then node A can be assigned a provisional backed-up value (PBV) of 2. No matter what happens, Max need not accept a value less than 2 for any move at node A. Now assume that Max is beginning to expand the move from pit 3 into node D. Node D is a move for Min. As soon as node D gets a PBV of 2 or less, there need not be any further expansion of the tree below D. The reason is that Min is certainly not going to choose a move with a value greater than 2 if a value 2 or less is available. But Max will not be interested in node D because there is already a chance at a value of 2. So we can stop expanding D. Figure 14-8 shows the tree.

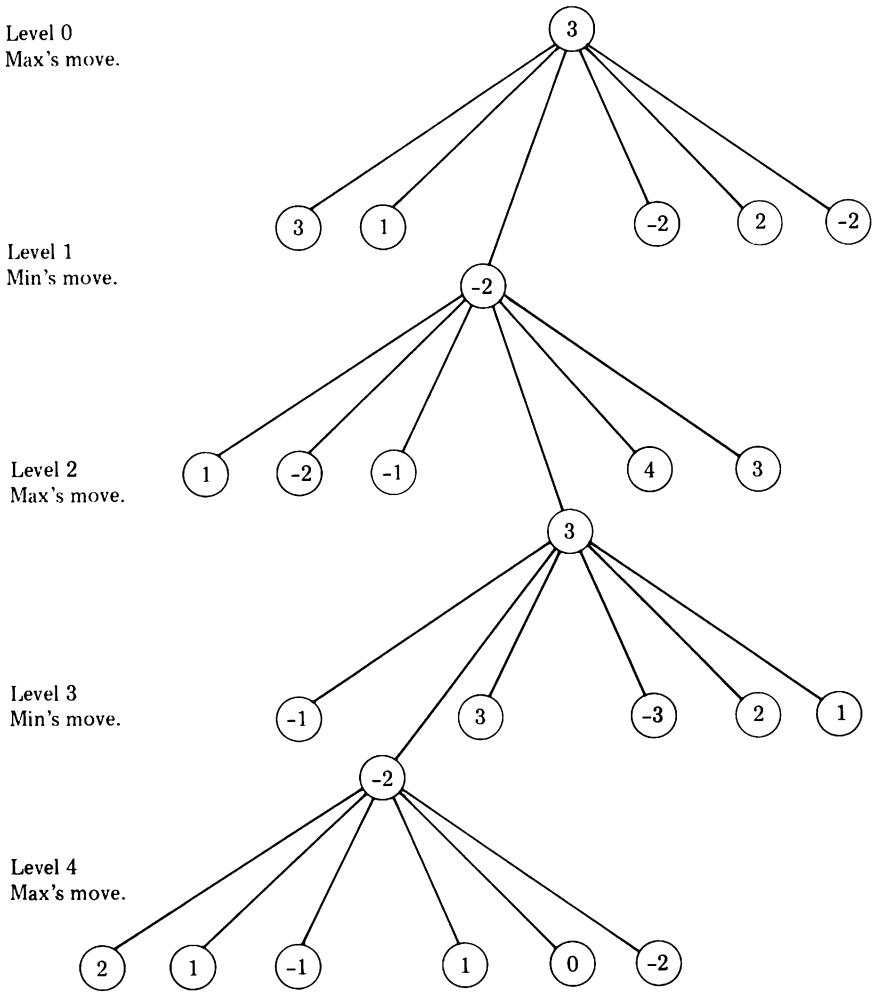


Figure 14-7. A Possible Game Tree. Only one complete path to the tree bottom is shown. Values in circles are backed (hypothetically) from below.

ALPHA-BETA MINIMAXING

To run an alpha-beta minimaxing procedure, begin a depth-first search of the game tree. Every node has attached to it a PBV and a final value (FV). The PBV of a leaf, as well as its FV, is always the static evaluation value. Now the PBV of an interior node is the maximum of the FV's of its successors for a Max node and the minimum for a Min node. Every time a PBV changes, we check to see if the expansion at the node should be stopped. (The PBV is initially minus infinity at internal Max nodes and plus infinity at internal Min nodes.) Cutoff occurs at a Max node any time the node's PBV rises as large as the PBV of any of the node's Min ancestors. Similarly, cutoff occurs at a Min node when the PBV falls as small as the PBV at any one of the node's Max ancestors. When a node is cut off, its PBV is converted to its FV. You should convince yourself that alpha-beta minimaxing always selects the same move as ordinary minimaxing.

Statement of the Theme Write a Kalah-playing program that uses alpha-beta minimaxing. Arrange the program so that it will play either against itself or against a human at a console. The lookahead depth d should be variable, as should be the initial number k of stones per pit and the

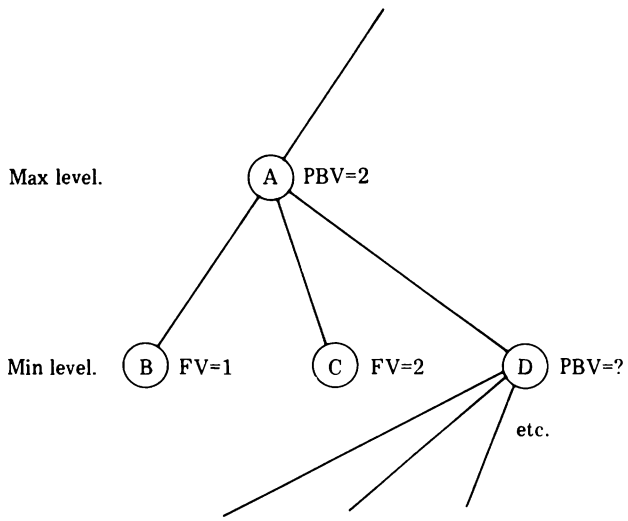


Figure 14-8. The Partial Alpha-Beta Tree Described in the Text. As soon as D's PBV sinks below 3, no more expansion need be done on D and its descendants.

player with the first move. Output of positions and input of moves should be as pleasing as possible. At request, the program should print out the move trees. Just to keep play interesting, the program should choose randomly between equally good moves (as usual, this feature should be disabled during debugging).

Performance Practice Although the description was long-winded, the Kalah program itself is actually quite simple. The main difficulty is building a data structure to represent the game trees and then making sure that the trees are created and destroyed in the right order. Note that you will probably have to write your own routines to provide space for the trees and to reclaim the space. Time efficiency will put the ultimate limit on lookahead depth; be careful in the tree generation routines. It would probably be a good idea to make sure that the minimax procedure is relatively separate from the rest of the program so that changes to it do not affect the whole.

Orchestration This is another problem in which the needs for powerful data structures, good control structures, and execution efficiency seem to conflict. PASCAL is a good compromise, particularly if heavy use of its data allocation and deallocation features is avoided. The tree search cries out for recursive procedures, but they are expensive. Instead try to use language features that convert recursion into an iteration over the data structure.

Playing Time One person for 4 weeks.

Variations on the Theme Although the references suggest a variety of modifications to the alpha-beta procedure, we will discuss only two here. The first tries to improve the efficiency of the alpha-beta cutoffs. Alpha-beta minimaxing works because good moves (for either player) stop the consideration of lesser moves. The sooner a good move is found, the more frequently are bad moves cut off. So we should try to expand good moves first. In fixed ordering, the static evaluation function is used to sort the immediate children of a node before any are evaluated. Then the node with best score is expanded first. Since the static evaluation function is a good guide to what lookahead will

eventually find (we hope), this procedure should improve the chance that good moves are handled first. Be sure to take the minimum first at Min nodes.

The other extension involves changing the static evaluation function. Another commonly used score function subtracts all the stones on Min's side—both those in pits and those in Min's Kalah—from those on Max's side. Any simple linear function of the 14 pit values might be used. Tournaments, as described in Chapter 5, can be used to select the best function. Remember, however, that it is probably the depth of the lookahead that determines overall play quality.

REFERENCES

Aleph$_0$. "Computer Recreations." *Software—Practice and Experience*, 1, pp. 297-300, 1971.

This paper describes the external appearance of a Kalah program and gives some history of similar programs. There is a useful bibliography.

Bell, R. C. *Board and Table Games from Many Civilizations*. Oxford University Press, London, 1969.

Bell gives several versions of the Mancala games in Chapter Four. The book is generally interesting for the wide variety of game lore and cultural history that it includes.

Nilsson, Nils J. *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill, New York, NY, 1971.

Nilsson is probably the best introduction to artificial intelligence. The discussion of minimaxing in Chapter 6 is particularly clear. The suggestions for reading are valuable.

Slagle, James R. *Artificial Intelligence: The Heuristic Programming Approach*. McGraw-Hill, New York, NY, 1971.

Slagle is also a good survey of AI. Since he did some experiments on Kalah, he gives more details about the game.

15

Prime Time

or...

SEARCHING FOR PATTERNS AMONG THE PRIMES

Prime numbers fascinate and frustrate everyone who studies them. Their definition is so simple and obvious; it is so easy to find a new one; multiplicative decomposition is such a natural operation. Why, then, do primes resist attempts to order and regulate them so strongly? Do they have no order at all or are we too blind to see it?

There is, of course, some order hidden in the primes. The Sieve of Eratosthenes shakes the primes out of the integers. First, 2 is a prime. Now knock out every higher even integer (which must all be divisible by 2). The next higher surviving integer, 3, must also be a prime. Knock out all its multiples, and 5 survives. Knock out the multiples of 5, and 7 remains. Keep on this way and each integer that falls through the sieve is prime. This orderly if slow procedure will find every prime. Furthermore, as n goes to infinity, we know that the ratio of primes to nonprimes among the first n integers approaches $(\log_e n)/n$. Unfortunately, the limit is only statistical and does not actually help in finding primes.

In fact, all known methods to list the primes are variations of the tedious sieve. Euler invented the formula $x^2 + x + 41$; for values of x from zero to 39, this function turns out a prime number each time. But no polynomial function can turn out an infinite unbroken series of primes, and Euler's function fails for $x = 40$. Other functions have streaks, but none known is perfect. Patterns do not seem to appear as researchers ponder piles of integer functions.

Patterns do appear, however, if the integers are mapped onto the plane (or into space). One way to do the mapping is shown in Figure 15-1, where the integers are wound around the origin in a left-handed spiral. Figure 15-2 shows the integers worked into a triangular pattern in the positive quadrant. If these arrangements are carried far enough, the primes can be seen to lie heavily along some lines (mostly diagonal) and to avoid other lines entirely. Part of this effect can be explained simply. In both arrangements the integers falling along any diagonal are given by some quadratic polynomial. If the polynomial for a particular line happens to be factorable into rational linear terms, then that line will consist of all composite numbers. So the primes must bunch more heavily on nonfactorable lines willy-nilly. Still, some nonfactorable polynomials seem to be very rich in primes,

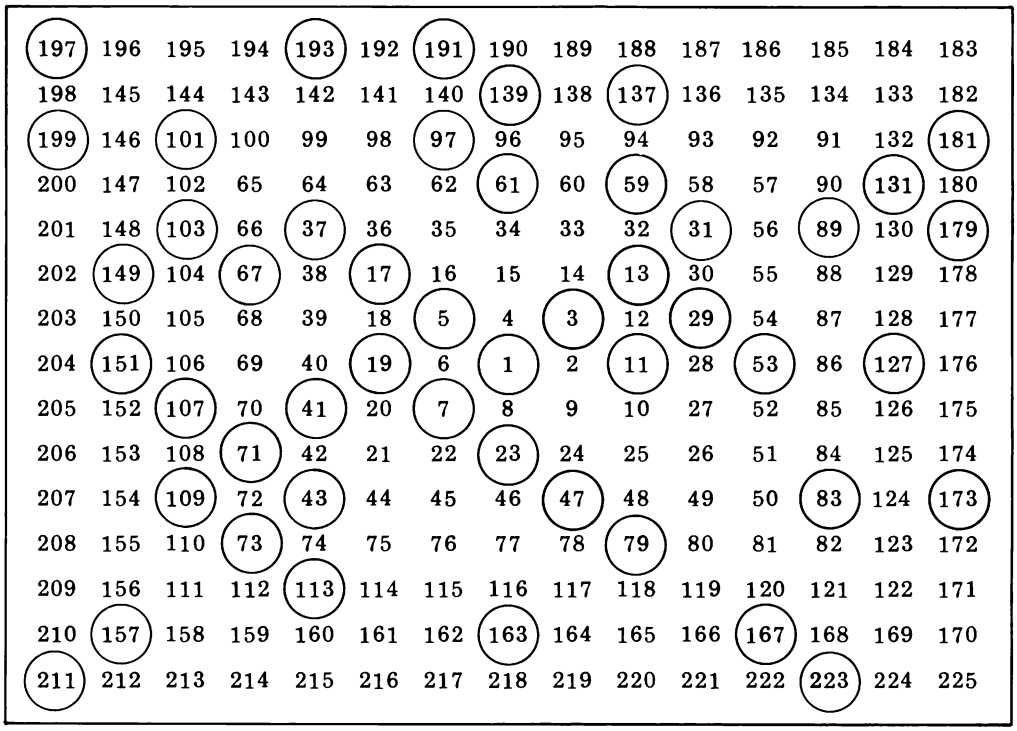


Figure 15-1. Integers Spiraled Counterclockwise

and there is no known reason why they should stay rich while the density of primes among all integers decreases slowly to zero. Stated another way, although factorability of polynomials explains some prime clumping, certain polynomials exist that are richer than simple statistical analysis suggests.

Statement of the Theme Write a program that maps the integers into the plane in some regular pattern and plots where the primes occur. Derive the functions describing the straight lines in your plot and print out those that are exceptionally rich, along with the richness ratio. Make sure that your primality test routines are efficient so that you have time to check the pattern very far out into the integers.

Orchestration This problem is best done in an algebraic language. You must be able to control efficiency of the primality test.

Playing Time One person for 2 weeks.

REFERENCES

Gardner, Martin. "Mathematical Games." *Scientific American*, pp. 120-126, March 1964.

Gauss, Carl Friedrich. *Disquisitiones Arithmeticae.* Yale University Press, New Haven, CT, 1965.

There are hundreds of books on number theory. Strangely, one of the first is still one of the best. It is an inexpensive paperback as well: why not go with the master?

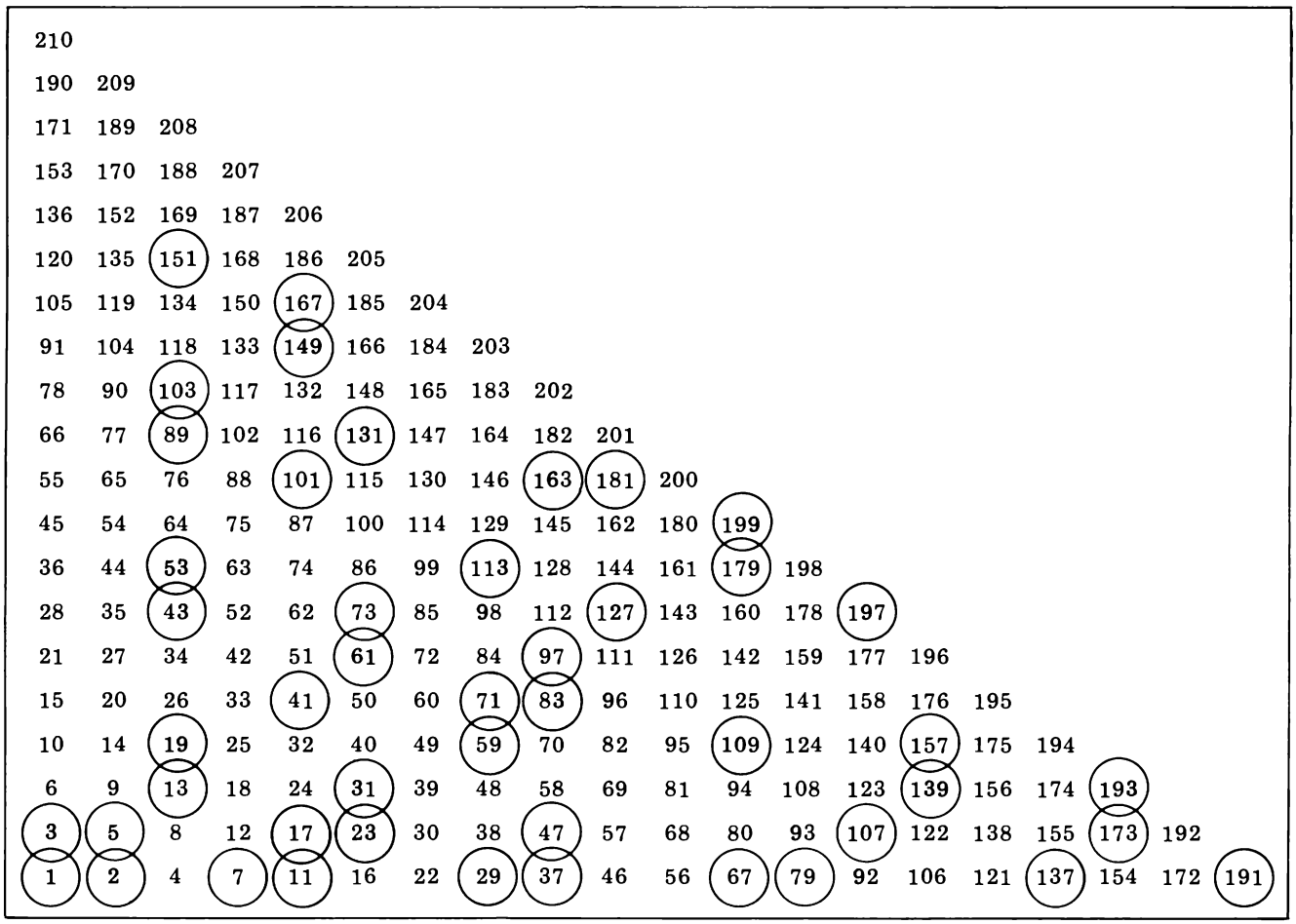


Figure 15-2. Integers in a Triangle

Stein, M. L., S. M. Ulam, and M. B. Wells. "A Visual Display of Some Properties of the Distribution of Primes," *American Mathematical Monthly*, pp. 516-520, May 1964.

Gardner wrote up the results of Stein, Ulam, and Wells in a more popular format, but both papers are easy to read. Not much else has been reported on this topic so it is probably just a fancy of Ulam's. Still, the idea makes pretty pictures, it's a good way to burn extra computer time, and there might be something in it.

16

Gas Pains

or...

A GASOLINE USAGE COMPUTATION

The day of the 30¢ gallon of gas is gone. The day of the 40¢ gallon of gas is gone. The day of the 50¢ gallon of gas is gone. The 60¢ day is here. Soon there may be no gas at all. Consequently, an analysis of personal gasoline expenses is in order.

Many people keep logs of their gasoline purchases. Normally this data might include the date, the mileage indicated on the odometer, the brand of gas, the price per gallon, the number of gallons purchased, and the total cost. The price per gallon, the gallons purchased, and the total cost are all related; this relation will be slightly inexact because of small roundoff errors, but it can still be used to test input validity. Using a computer, you can calculate some other statistics. The derived quantities of interest include the average cost per gallon, the average miles per gallon, the average number of miles traveled per day, the average cost per mile, and the average amount of time that a gallon of gas stays in the tank. It would also be nice to have the same information arranged by brand to see if there are any differences among brands. Table 16-1 is a real gasoline log.

For the purposes of this problem, assume that each log entry refills the tank completely. The first entry establishes a base line for dates and mileages but cannot itself be used as data. Afterward each entry gives mileage and costs for the last tankful, showing the amount of gas replaced and the number of miles traveled. It would also be interesting to print running averages over the latest short time period to see if any short-term changes have occurred.

Statement of the Theme From data in a gasoline usage log, print a variety of control statistics giving the driver knowledge of car costs. Input data for each transaction should include the date, the brand, the odometer reading, the price per gallon, the gallons bought, and the total cost. Output should echo the input and also include the miles on the tank, the miles per gallon, the cost per mile, the cost per gallon, the cost per day, and the days per gallon. All these quantities should be calculated per fill, on average over a short term, and on average since the beginning of the data. In addition, collect the data for the brands and print out the averages for each brand. Do not limit the number of possible brands.

Table 16-1 Some Entries from an Actual Log

Date	*Brand*	*Miles*	*¢/Gal.*	*Gallons*	*Cost*
03/27/74	Texaco	24,370	59.9	13.5	$8.00
04/05/74	Texaco	24,434	59.9	5.5	$3.30
04/11/74	Texaco	24,596	59.9	8.2	$4.88
04/23/74	Mobil	24,862	58.8	12.9	$7.60
05/13/74	Mobil	25,057	61.9	13.9	$8.60
06/11/74	Arco	25,239	62.9	12.5	$7.85
07/12/74	Texaco	25,435	63.3	14.2	$8.90
07/19/74	Chevron	25,713	58.8	12.4	$7.27
07/28/74	Mobil	26,135	60.9	14.1	$8.60
08/07/74	Arco	26,384	60.4	13.1	$8.00
08/08/74	Chevron	26,712	59.9	13.3	$7.90
08/16/74	Arco	26,997	60.9	13.6	$8.30
08/22/74	Mobil	27,068	60.9	4.0	$2.45
08/22/74	Shell	27,362	61.6	11.8	$7.25
08/23/74	Shell	27,606	63.4	10.3	$6.54
08/25/74	Ericson	27,913	60.9	13.6	$8.29
08/26/74	American	28,163	60.9	10.8	$6.55
08/26/74	American	28,487	57.9	14.0	$8.10
08/27/74	DX	28,771	53.9	12.2	$6.60
08/28/74	Conoco	29,114	59.9	14.8	$8.90
08/28/74	Texaco	29,337	58.9	10.2	$6.00
08/28/74	Phillips	29,661	60.9	13.9	$8.35
08/29/74	Chevron	29,912	65.9	10.8	$7.10
08/29/74	Shell	30,147	65.9	10.3	$6.70
08/30/74	Texaco	30,317	60.9	7.6	$4.60
08/31/74	Exxon	30,643	56.9	13.3	$7.60
09/06/74	Shell	30,878	59.9	13.2	$7.90
09/10/74	Shell	31,182	59.9	13.0	$7.80
09/14/74	Exxon	31,467	57.9	13.1	$7.60
09/18/74	Arco	31,711	57.9	10.1	$5.85
09/24/74	Arco	31,984	57.9	12.5	$7.25
09/27/74	Arco	32,225	57.9	9.9	$5.70
10/01/74	Arco	32,455	57.9	9.8	$5.65

Performance Practice This program has no great difficulties. As with several other problems, there is some room for ingenuity in printing dollar amounts. The requirement that the number of brands not be limited implies a requirement that they not be named in advance. As a result, a simple growing table of brands and their associated data will be necessary.

Orchestration Once again COBOL is an obvious candidate; this is exactly the type of task that the language was designed for. If you can find a report-generation language with sufficient power, here is a chance to learn how to use it. Otherwise use any procedural algebraic language.

Playing Time One person for 1 week.

17

Shocking Statistics

or...

HIGHWAY TRAFFIC SIMULATION

One early effect of the energy crisis was the nationwide lowering of the highway speed limit. Most motorists resent this limit when on the long haul. Of course, we now know that the lower limit saves thousands of lives and millions of dollars annually. Most drivers, however, do not realize that in the congested conditions on the urban highways surrounding big cities the lower limit may actually save time. To state the paradox more clearly, if everybody drives more slowly, everybody gets home sooner.

Think back to some time when you were driving down the freeway, clipping right along about 5 miles above the limit even though the highway was busy, and then suddenly everyone slowed to a crawl and you had to stand on the brakes. Next came a quarter, a half, even a full mile of stop-and-go traffic. Finally, the pack broke up and you could come back to speed. But there was no cause to be seen. What happened to the traffic flow?

The reason for the stoppage lies in the theory of fluids. Cars traveling down a highway behave much like particles of a fluid running in a pipe. If the density is high enough and the velocity fast enough, any momentary stoppage of flow will set up a shock wave. The shock wave is an area of very high density; cars (or particles) approaching it slow drastically as they enter the shock area and then speed up as they pass the relatively well-defined shock front into a region of much lesser density. The shock wave will persist for a long time, slowly moving against the flow of traffic and slowly dissipating. We might point out that dissipation occurs because the density in the shock area goes down and can be hastened if drivers will brake gently when they notice a slowdown some distance ahead.

It would be interesting to do experiments on the highway at rush hour, but undoubtedly more than one commuter would object. A more convenient experiment can be done on the computer. Consider a straight section of highway one lane wide and 5 miles long with no passing allowed. Cars enter at one end of the highway, travel down it, and exit without any further effect at the other end. While on the highway, cars try to move at constant speed, although not all at the same speed. To study shock waves, we will randomly introduce slowdowns into the traffic steam.

To run the experiment, we need a generator for cars and one for shock starters. At the beginning of each experiment the highway is empty. Start a car generator that inserts cars onto the highway, choosing a speed and an interval until the next car is started. A car entering the highway may have any speed between 50 and 60 miles an hour, chosen from a uniform random distribution, and the next car will be delayed between 4 and 6 seconds, again uniformly distributed, before it can enter. The nearest that two cars can get together is one car length for each 10 miles per hour that the leading car is traveling; a car length is 10 feet. When the trailing car gets within three times the prohibited distance, it begins to lose one mile per hour per second to match speeds. If the lead car begins to slow drastically, the trailing car waits 0.2 second and then brakes at a rate of 15 mph/sec. The result may cause a tailend collision that will end the experiment.

The actual experiment consists of filling the road with cars, introducing an artificial slowdown, and observing the result. Start with the road empty, inject cars as described above, and continue to do so until cars have been leaving the far end for 2 (simulated) minutes. While still injecting cars, select the car that next passes the 4-mile point and brake it as quickly as possible 0, 10, 20, 30, 40, or 50 miles per hour, let it hold its new speed for 100 yards, and then accelerate it back to its original speed at a rate of 5 mph/sec (cars always try to maintain the speed at which they entered the highway). From the time that the crisis car begins slowing, run the experiment for another 5 minutes and count the cars exiting the highway during this 5-minute period to get the observed experimental value. Cars following the crisis car may also use the 5 mph/sec acceleration when the highway clears in front of them. Run the experiment several times for each crisis slowdown. If there is a tailend accident, all cars behind the accident will automatically stop and be unable to exit. There may be more tailenders behind the crisis.

Statement of the Theme Write a program to do the highway shock-wave experiment. The only input is the number of times to run the experiment for each reduction in speed. Output is basically the average number of cars to exit the highway after each crisis speed reduction. But for debugging and for better understanding of the physical behavior of the system, more output is desirable. In particular, "snapshots" of the road at various times will probably convey considerably more intuitive insight than any set of statistics. If you have a good graphic device, either interactive or film producing, a series of snapshots becomes a movie of the road.

Performance Practice The most difficult problem here is keeping track of all the cars on the road.[1] You might cycle every hundredth to tenth of a simulated second, adjusting the position of each car as appropriate each cycle. As long as the cycle interval is short enough, no significant errors will creep in and the program can be organized nicely as a series of nested loops. But in a lockstep simulation there can be too many cycles. In this case, there will be about 12 minutes of simulated time, about 90 cars on the road at any one time, and, even with a long cycle interval of a tenth of a second, about 7200 cycles or 650,000 individual looks at cars. There is an obvious problem if the program spends much time adjusting any one car. By varying the cycle interval according to traffic conditions, you may ameliorate the difficulty.

The alternative is to adjust the cars' positions only when a critical event occurs. That is, a list is maintained of all events that can be foreseen to happen in the near future, such as a new car entering the highway, a car exiting, a car overtaking another, 2 minutes since the first car left, and time for the crisis car to speed up again. The event list is always kept so that the next event to occur is at the front of the list (the whole list need not be sorted — priority queues and heaps might both have this property). The basic cycle picks the next event off the front of the list, adjusts all car positions on

[1] A subsidiary difficulty is that all the units are in English customary notation. This usage is intentional and your output should be in the same units. It could be worse — we might measure speed in f/f, that is, furlongs per fortnight.

the basis of the new time, notes any events that must be scheduled, enters them on the list, and reorders the list to get the nearest event to the front. The advantage of the critical event simulation is that there may be long stretches—4 or 5 seconds—when everything is rolling along smoothly. The time saved in useless cycles can be applied to the more complicated event list handler.

Orchestration Natural candidates for this problem include the simulation languages such as Simscript and Simula. Of course, if they are not available, an ordinary procedural language is appropriate. No matter which method is chosen, it will probably help to have good data structures to hold the car data and the queue of events.

Playing Time One person for 3 weeks; add a week for making a movie.

Variations on the Theme Strictly speaking, this problem does not study the situation described in the first few paragraphs. Instead of seeing what happens to the shock wave at different average traffic speeds, the experiment looks at different shock intensities. Run the whole thing again with the range of initial speeds between 40 and 50 mph or 60 and 70 mph. Try a normal instead of a uniform distribution for some of the variables. Vary the braking and acceleration functions. In other words, do a study on all the parameters of the situation instead of only the one that we chose.

REFERENCES

Herman, Robert, and Keith Gardels. "Vehicular Traffic Flow." *Scientific American*, pp. 35-43, December 1963.

Herman and Gardels describe several physical experiments on traffic flow and the development of a mathematical theory. Of course, they used the Holland Tunnel in New York City, a resource beyond most of us. If you are interested in tracing down work since 1963 on traffic flow, here is your chance to find out how to use *Science Citation Index* and other bibliographic aids to bring an old article up to date.

18

Readin', 'Ritin', and 'Rithmetic

or...

CONSTRUCTION OF A FORMAT SCANNER

You have probably written at least one program that generated reams of neatly arranged output. Rank upon serried rank of numbers marched off the printer, officered by squads of tidy titles. Nobody ever looked at more than three of the numbers, but it was so easy that it seemed a shame not to print them all. After all, somebody *might* want to know the exact tax rate for worker 1793 at work location 907 during September five years ago.

The reason that you could print such a mass of data without dying from the effort was that something like the FORTRAN format statement helped you by converting all those internal nasty binary numbers to nice-looking character strings. In fact, the same thing happened on input. All the data came on neatly punched cards, and you never even thought about how it was changed so that the CPU could do its little arithmetical tricks. Perhaps you should have thought a little more about your input/output. A data-dependent program can easily spend a quarter to a half of its computation time in I/O service routines, and most of that may be spent interpreting formats and converting data. So that you will never again take data conversion lightly, this problem makes you a library utility programmer.

We have chosen FORTRAN formats to study because they are simple, efficient, and the granddaddy of most other format schemes. Whenever a FORTRAN I/O operation is directed to a human-readable device, a format statement mediates the process. The essential elements of an I/O operation are a variable list, a format, and an I/O stream. Data items are transferred to or from variables in the variable list from or to the I/O stream, depending on whether the operation is input or output. As each item is transferred, enough of the format is interpreted to define the coding of the data item on the I/O stream. The format does not control how much data will be moved, but it does control the details of the movement.

WHAT IS A FORMAT?

A format is a string of characters describing the transformations to be performed on the data. Because it is interpreted each time that it is used, a format could be regarded as a little program. A general format has the form

$$(y\ f^1\ s^1\ f^2\ s^2 \cdots f^n\ s^n\ z)$$

where n may be zero and

y and z are each a possibly null string of slashes.

each f^i is either a single format code or a general format optionally preceded by a natural number.[1]

each s^i is a field separator—that is, commas, slashes, a mixture of the two, or null as appropriate.

Blanks are ignored throughout a format except in one special case noted below, and all numbers are represented as decimal strings.

Now assume that an I/O operation has been invoked. The I/O stream is set at the beginning of the next record.[2] A scan pointer is set to the beginning parenthesis of the format and then scans forward to the first code requiring a variable from the variable list or the right end of the format. This process allows an output statement to write just a data line without transferring any variable data. The format interpreter will have some internal memory (usually implemented as a stack) that will be cleared and that we shall refer to occasionally when we mention that the interpreter "remembers" something. The basic format cycle is simple. The next variable on the list is passed to the interpreter. The scan pointer moves down the format, looking for a format code that will cause data transmission. During the rightward scan codes may be encountered that modify the I/O stream or set parameters in the interpreter; the actions described by these codes are performed without stopping the scan. Some codes are allowed repetition counts, and the associated action is performed until the count is exhausted. The same code may be used for several variables on the list, which means that the interpreter must be able to remember the decremented count from cycle to cycle. If the right parenthesis ending the format is scanned, scanning returns to the nearest unenclosed left parenthesis without a repetition count and, failing the existence of one, to the initial left parenthesis. The three fundamental errors that can occur are running off the end of the input stream, encountering the right end of the format twice in a row without transmitting any data, and not matching the data type of the format code, the variable, and the actual data on the I/O stream (this last part for input only). On output, termination of the I/O operation causes the last partial record to be written.

Now for the format codes themselves. About the only useful grouping that we can make is between self-terminating codes and non-self-terminating codes, which must be followed by a comma, slash, or parenthesis. The interpretor maintains a running scale factor, initially set at zero, which may be modified by scale factor designators. The individual codes are

r(A left parenthesis, optionally preceded by a repetition count r, indicates the beginning of a format group delimited by a following right parenthesis (parentheses within a format must be balanced). The entire group is repeated as indicated by the repetition count. A missing count is assumed to be one.

, The comma terminates codes that need separation from the following code. They have no other effect and redundant commas may be used.

/ The slash serves to terminate non-self-terminating codes and also terminates the

[1] Remember that a natural number is an integer greater than zero.

[2] The I/O stream is divided into records, but these records may be of various lengths. Any given physical device may set limits on record length. The I/O stream is assumed to be set at the end of an imaginary zeroth record before the first I/O operation on the stream. For output, records are created as necessary.

current record on the I/O stream. If the last code performed before the operation ends is a slash, there is no record termination because of the end of the operation. Multiple slashes cause records to be skipped on input or empty records to be created on output.

nX On input, n characters are skipped in the I/O stream; on output, n blanks are inserted into the stream. This code is self-terminating and does not transmit data.

nHh$_1$h$_2$· · ·h$_n$ On input, the next n characters from the I/O stream replace the n characters h$_1$h$_2$· · ·h$_n$ in the format. On output, the n characters h$_1$h$_2$· · ·h$_n$ are moved to the I/O stream. Blanks may be included in the h$_i$; this is the only case when blanks are significant in a format. This code is self-terminating and does not transmit data to or from a variable.

rAw Let g be the number of characters that can be stored in the variable used by this format cycle. On input, if w $\geqslant$ g, the rightmost g of the next w characters on the I/O stream are moved to the variable; otherwise the w input characters will appear on the left of the variable, padded with g–w blanks on the right. On output, if w $\geqslant$ g, then w–g blanks, followed by the g characters of the variable, will be placed in the I/O stream; otherwise the leftmost w characters of the variable are moved to the stream. The repetition count r is optional and this code is non-self-terminating.

rLw On input, the next field of w characters must be blanks, followed by a T or F, followed by any arbitrary characters, providing a value of *true* or *false*, respectively. On output, w–1 blanks, followed by a T or an F, are placed into the I/O stream. The repetition count r is optional and the code is non-self-terminating.

rIw On input, a string consisting of leading blanks, an optional sign, and intermixed blanks and digits is converted to an internal integer. The field is w characters long, and blanks after the sign are treated as zeros. On output, the field is w characters long and consists of blanks followed by a minus sign if necessary and by the digits of the converted integer right-adjusted. The repetition count r is optional and the code is non-self-terminating.

sprFw.d On input, floating point conversion always reads from a field of w characters. If the input data consist of all digits and blanks, or if the only sign character appears to the left of character w–d+1 (counting from 1), the input value is the real that is created by placing a decimal point between positions w–d and w–d+1 of the field. If the input string has a decimal point, it overrides the implied point. If the input string has the form of an integer or real followed by a second signed integer or the character "E" followed by an optionally signed integer, this second integer is taken to be an exponent and the real value is to be multiplied by ten raised to that exponent. When the "E" form of the exponent appears alone, the real number part is assumed to have value one. If no exponent appears in the input string, the number read in is also multiplied by ten to the current scale factor before it is given as value to the input variable. On output, a floating point number is written as x$_1$· · ·x$_r$.y$_1$· · ·y$_d$ with a leading minus sign if necessary and the value rounded to d fractional places. There will always be a decimal point in the output field; so w $\geqslant$ d+1 for all output F codes. Once again, the data is right-justified in its output field. Both the scale factor designator sP and the repetition count r are optional. The new setting s (which may be any signed integer) continues until another scale factor designator is encountered. The code is non-self-terminating.

sPrEw.d On input, the E format operates exactly the same as the F code. The basic form of an output field is $0.y_1 \cdots y_d E z_1 \cdots z_m$, where there may be a minus sign preceding the initial zero or following the E as necessary and where m is large enough so that the maximum exponent and a minus can be accommodated even if they are not necessary. If the current setting of the scale factor is q, the real part of the basic form is multiplied by 10^q and the exponent decremented by q. For q > 0, there will be q digits to the left

of the decimal and max(d-q+1,0) to the right; for $q \leqslant 0$, there will be a single zero to the left and d+q digits to the right. As with the F code, both the scale factor designator sP and the repetition count r are optional and the code is non-self-terminating.

sPrGw.d Input and interpretation of sP and r are the same as for the F code. The output under the G code takes the form of either an F code or an E code, depending on the magnitude of the output value. If M is the output value and $10^{k-1} \leqslant M < 10^k$ for $0 \leqslant k \leqslant d$, then output uses the code F(w-4).(d-k), 4X; otherwise the code Ew.d is used. Note that the scale factor setting has no effect if the F mode is chosen. This code is non-self-terminating.

Statement of the Theme Write a format package for your computer. Generally such a package has a number of entry points available to the ultimate user (a compiler-generated object program as a rule) even though there will also be a number of internal routines that the user must be prevented from accessing. The user entries should include initialization with parameters specifying input or output, the I/O unit, and the format, entries presenting each type of variable (real, integer, logical, or any of them used for alphameric data), and a termination entry. The internal form for data can either be that of the local host computer or the form described for the EC-1 computer in Chapter 25. Test the package sufficiently to show that rounding rules and end cases are working correctly and that errors are reported accurately.

Performance Practice Achieving a consistent understanding of the behavior of real numbers on the host machine is the most difficult part of this problem. Alphameric, integer, and logical data can all be converted rather easily, and format scanning and buffer housekeeping require only fairly simple techniques. You will probably find, however, that implementation of *just* the right rounding rule will require considerable thought and possibly some experimentation. Be sure to test with values just slightly above and below powers of ten, just below 10^{-d}, and so on. And do not be led into the construction of proliferating special cases to handle flaws in earlier work; retrench and try a different approach instead. One of our saddest programming defeats was a format package that grew like Topsy to over 3000 lines of assembly code. How embarrassing to have it replaced by a cleaner, more powerful routine of less than 1000 lines written by someone else! And how happy we were to get rid of that monstrosity forever!

Orchestration This is one of the problems for which we can recommend assembly language. Format packages need good efficiency, and they are programs in which execution time is rather widely spread instead of being concentrated heavily in a few tight loops (the pattern for most programs is for 10% of the code to account for about 90% of the execution time). Also, higher-level languages hide the specific data manipulations necessary in a format package. If you have an implementation language like BLISS or PL/360 (or possibly XPL), it is the most likely candidate, since it will have the good machine control of assembly language without assembly language's bad features.

Playing Time One person for 5 weeks.

Variations on the Theme There are numerous possibilities for extension of formats. New codes can be added. For example,

'x· · ·x' Exactly the same effect as nHx· · ·x. Single internal apostrophes are represented by doubled apostrophes.

Bw, Ow, Zw Input and output in binary, octal, and hexadecimal, respectively. Internal values are regarded as right-adjusted bit patterns when using these codes.

Tn Move immediately to column n of the current record. This movement may cause rereading or rewriting of data on the I/O stream.

Alternatively, the very precise field-width requirements may be weakened. Thus E.d on output would imply that the format package should choose an appropriate value for w, and I alone on input would imply that the next integer should be terminated by a blank, comma, or end of record rather than by a field width. Almost every FORTRAN system has some similar extensions that you might add.

REFERENCES

Anonymous. *USA Standard FORTRAN.* United States of America Standards Institute, New York, 1966.

The format codes that we describe are slightly different from those in the Standard. We feel that the Standard does not reflect industry practice in this area, although we would be happy to see an implementation of Standard specifications instead of those above (the work is about equal). Reading the Standard is an experience in itself, one that the dedicated FORTRAN programmer should undergo. It makes one wonder exactly what language all those compilers translate, for it certainly is not FORTRAN.

19

Patience is a Virtue

or...

SOLITAIRE STATISTICS COLLECTION

Every programmer blocks at one time or another. For no apparent reason, the current program simply will not let itself be written. Each new attempt quickly becomes a trash pile of spoiled coding sheets. The solution is to leave the problem alone for awhile. If the boss complains, explain that you are relaxing your mind in the interest of greater productivity. Then go to a movie. Or play handball until you drop. Find a complaisant member of the opposite sex and discuss *The Critique of Pure Reason.* Lose some money at the track. Or get out your deck of cards and prepare to waste 3 hours trying to win your favorite solitaire (in England, prepare to lose your patience).

Solitaires come in two forms. The first has a set of rules for laying out the cards and additional rules for playing the cards; working out the solitaire is a mechanical ritual and the player is an overlooking automaton. Such games will lead you to a deeper understanding of the emotional state of a computer while it executes one of your programs, but they lack creative tension. So the second form of solitaire allows some decision making on the part of the player. Rather than watching Nature's patterns form, the player vies against Nature, represented by a shuffled deck of cards. These games usually have some artificial winning conditions, but there is little information on what results best play might achieve. With a computer's help we can find real standards against which the player can compare scores. Instead of playing solitaire, programming it may clear the block.

RULES FOR ONE SOLITAIRE GAME

Begin by thoroughly shuffling one ordinary deck of cards. Now deal the cards into a tableau as follows. There is a row of seven piles with, respectively, zero, one, two, . . . , six cards face down and the top card face up from left to right in the middle of the tableau. These packets account for 28 of the cards. Then the last 24 cards are laid out in descending overlapped columns of four on the six rightmost packets. All the cards in a column are face up, and they are overlapped so that the card at the base of a column is on top of the card next up the column, which is, in turn, on top of the next, and so on up to the cards on top of the packets that started the columns. The overlapping should be done so that the rank (or denomination) and suit of each face-up card are clearly visible.

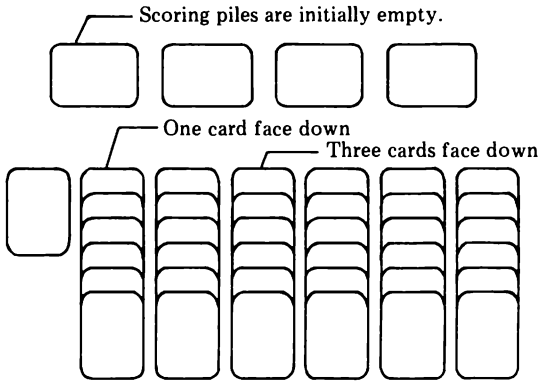


Figure 19-1. The Solitaire Tableau. All cards are face up except those in the packets below columns two through seven.

Finally, space should be reserved at the top of the tableau for four scoring piles, one for each suit. Figure 19-1 is a picture of a complete initial tableau.

A play consists of lifting any face-up card and those overlapping it (that is, those lower down the column) and placing the partial column overlapping the base card of a different column. This movement can take place only if the top card is in the same suit and one rank lower than the card newly covered (in this game, aces rank low—that is, as ones—and kings rank high). Figure 19-2 shows a sample move. If a play exposes the top face-down card of a packet, finish the play by turning over the exposed card. A play may also completely empty a column; then any exposed king, along with its lower overlapping cards, may be moved into the hole on any subsequent play. When an ace is the bottom card of a column, it is moved to the top of the tableau to start a scoring pile for its suit. After the scoring pile for a suit is begun, new cards of the suit may be scored as they appear at the bottom of some column, so long as the cards are scored in strictly ascending rank order. Notice that once a card is in scoring position at the base of a column, there is no reason to avoid scoring it, since it will eventually be scored anyway and until then it only blocks plays on the column.

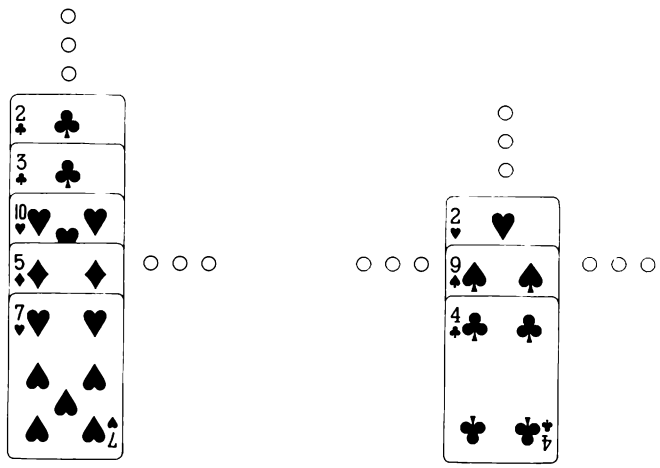


Figure 19-2. A Possible Play Moving One Column to Another. The cards from the 3 of clubs down are moved to overlap the 4 of clubs. Other columns are not shown.

Play continues until no further plays are possible and no cards remain in position to score. The game score is the total number of cards in the scoring piles. In learning this game, we were told that the game was regularly played in Las Vegas (also known as Lost Wages). A deck of cards could be purchased at $1 per card (plus $3 tax) from a casino and the casino would pay $5 for each card scored. Thus the player who scored 11 cards (or $55 worth) would break even, and each additional card scored represented player profit. It seems unlikely that casinos actually did offer these terms, but we suspect that they would have made an outstanding profit had they done so. In fact, how many cards can be expected in the scoring piles? Exactly how obscene would the casinos' profits have been?

ANALYSIS OF THE SOLITAIRE

Each possible original layout gives rise to exactly one optimal result, although several sequences of play may achieve it. Presumably there is some very complicated probabilistic function that calculates the expected value of this optimal result. But even if that function were written out, it would undoubtedly have so many terms that calculation of its values would be extremely onerous. Instead, why not play a large number of games and calculate the interesting statistics from the sample results thus obtained? This idea of a simulation finding a value theoretically directly calculable has appeared in other chapters precisely because, as here, changing the computer into a model of an interesting real process is so valuable. What are the necessary steps for solitaire?

First, there must be routines to deal the cards, to check for the existence of plays in a position, to make a move, to turn over a card from the top of a packet, to score a card—in short, to play the solitaire legally. Using these routines, the result of any given sequence of plays can be checked. On them, we superimpose a search strategy to look for the optimal result. Deal the cards into an initial tableau. From each position as the game progresses the search strategy performs the following steps.

> Note how many potential plays there are in the position. There are at most seven at any time.
>
> If there are no available plays, this sequence is over and its score can be recorded. Reset the position by popping the top position off the position stack and go back to the top of the loop. If the position stack is empty, the search is over.
>
> If there is exactly one available play, make it and return to the first search step.
>
> If there is more than one possible play, order them (the ordering method is irrelevant). Record the position, the ordered list of moves, and the fact that the first move has been made on the position stack. Make the first move and return to the first step. Notice that when a position is unstacked, there is an implicit assumption in step one that determining available moves will always require looking first for a partially completed move list.

The search strategy performs a depth-first search of all possible move sequences, storing positions waiting for investigation on a stack. By making all possible sequences of decisions, the search strategy ensures that an (it may not be unique) optimal sequence is found.

One of the frustrating aspects of this solitaire is the number of times that it is laid out with no plays available at all. Laying out the tableau is merely a complicated way to shuffle. In spite of the good chance of early termination, we still must expect the position tree to grow very large. But the tree actually is a graph because positions may easily repeat after different move sequences. When a position has been investigated once, there is no need to search it again. The value of a game does not depend on the order of the moves or on the particular sequence leading to the optimal result. If each position is saved as it is processed, later positions can be compared against earlier ones

to save duplicate processing. Only the cards, not the possible moves, need be saved. Of course, there is the problem of looking up the saved positions.

Statement of the Theme Write a program to find the mean and standard deviation of the optimal score of the solitaire game described. Make sure that enough cases are sampled to provide good statistical accuracy. If you can, also calculate the mean number of plays and the mean number of decision points on the way to the optimal result. The only input should be the number of games to be played. Output could be simply the statistics required, but probably a great amount of other data is available. In particular, details of the storage allocation for positions may prove revealing.

Performance Practice The organization of the cards within a position and the storage of past positions will be the crucial determinants of efficiency. While a position is active or in the stack, the status of all the packets, of the columns descending from them, and of the scoring piles must be known. Stacked positions need to maintain a list of plays yet to be studied. To speed up searches for legal plays, a vector should give the status of all cards—that is, invisible, already scored, visible in the middle of a column (which column?), or visible at the bottom of a column (again, which column?). Or perhaps you can think of another data structure that will allow quick decisions about possible plays. In any case, when a position is saved some information may be thrown away as already having been investigated, thus saving space.

Two specific techniques will be necessary. First, how does a computer shuffle a deck of cards? Here is a procedure suggested by Knuth. Let **rand52** be a random number function that returns integers uniformly distributed between 1 and 52. Put the cards in an array **card** of 52 elements; it does not matter how the cards are arranged initially. Now for each **i** between 1 and 52, exchange **card[i]** and **card[rand52]**, calling **rand52** anew each time. One shuffle will be enough, using this technique.

Second, how is an old position to be found? This is a classic search problem with a growing data base. The obvious solution seems to be a hash table that uses the whole position as a search key. Since comparison of two entire positions for equality is likely to be expensive, this is a problem for which a virtual hash code seems appropriate. The storage space for old positions may overflow, and you should be prepared to clean it out from time to time. The best way seems to be to keep a count of the number of times that each position is referenced and to throw out those used least often. Alternatively, or perhaps as a double check, keep a list of all old positions, and each time that a position is referenced, move it to the head of the list. When the time comes to destroy some positions, those at the end of the list are candidates, since they have been referenced the longest time ago. The method of old position disposal will influence your choice of search strategy and vice versa. Notice that destroying an old position does not alter the correctness of the analysis, but it may slow it down.

Orchestration This is a problem with a need for easily manipulable data structures of a moderate complexity. Efficiency dictates that control of allocation and deallocation should not be given to the system; so SNOBOL is probably out. ALGOL W, PASCAL, PL/I, LISP, even COBOL, are all candidates. Control structure is not such an issue here. You will be convinced of the value of programmer-defined data structures if you try this problem once in a language like those mentioned above and again in a language like FORTRAN or XPL, in which complicated structures must be built from parallel arrays.

Playing Time One person for 3 weeks.

Variations on the Theme The most obvious extension is to apply this analysis technique to other types of solitaire. As long as there is no shuffling during play, the idea should carry over without difficulty. There must be hundreds of solitaires, and to our knowledge none has had any serious analysis. This problem could also be used to study the effect of varying the storage space and destruction criterion for old positions on the total number of positions studied during analysis. In other words, instead of using search techniques to help you learn about solitaire, use solitaire as a source of data to study search techniques.

REFERENCES

Gibson, Walter B. *How to Play Winning Solitaire*, Frederick Fell, New York, NY, 1964.

This is the only book we have ever seen on this method of play at solitaire.

Knuth, D. E. *The Art of Computer Programming, Volume 3/Sorting and Searching.* Addison-Wesley, Reading, MA, 1973.

Knuth's books keep appearing as references. This time Chapter 6 of Volume 3 will tell you all that you want to know about searching—in particular, searching by hashed techniques. Of course, perusal of the whole chapter might suggest some even better method.

20

Poly Wants a Cracker

or...

A SYMBOLIC ALGEBRA PACKAGE

A major difficulty with most programming languages is that the programmer must break all his equations into pieces for calculation. If a derivative is needed, the programmer must write out the original function, drag out a calculus book, apply the rules therein, and then write down the resultant derivative. At least for polynomials, many operations can be done symbolically if the polynomials are represented in a reasonable way. Some programs might vanish completely if polynomials could be directly manipulated in the computer.

The objects to be manipulated are the rational polynomials. We can define these polynomials recursively.

> Let c be any constant drawn from the real numbers. Then c is a rational polynomial.
>
> Let x be any variable. Then x is a rational polynomial.
>
> Let p and q be any rational polynomials. The p+q, p-q, -p, pq, p/q, and (p) are all rational polynomials. Whenever polynomials are divided, they are simplified so that at most one division sign remains, using rules familiar to high school algebra students.
>
> Let p be any rational polynomial and c any integer constant. Then p^c is a rational polynomial. When c is negative, form the polynomial $1/p^{|c|}$ and simplify the division as above.
>
> Only those objects described by a finite number of applications of the preceding rules are rational polynomials.

Beyond the descriptions, we need to describe what a polynomial might look like on input and output and how to call for operations.

On input, polynomials will be similar to expressions in standard programming languages. A constant can be any string of decimal digits with a decimal point; if the decimal point is missing, the constant is automatically an integer. Constants do not need signs, except as exponents, because of the polynomial formation rules. A variable looks like an identifier and may be any string of upper- and

lowercase alphabetic characters. Because of the limitations of computer character sets, multiplication will be represented by * and exponentiation by ↑. Thus the polynomial

$$2xy + (x^2+y^2)^3$$

could be written 2*X*Y + (X↑2+Y↑2)↑3.

Some other names, particularly functions, will also be identifiers built from alphabetics.

To go with the polynomials, there must be some manipulatory commands so that a user can get answers that are not available from a conventional programming language. For this reason, we will need to name polynomials with identifiers. The most fundamental operation is

Set f to p; This command causes the polynomial name f to be given the value polynomial p. It is a symbolic operation and does not cause evaluation of p. Once an identifier f has been used as a polynomial name, it may not appear in any later operation as a variable; you should plan to keep a table of names, values, and usages during interpretation. The polynomial p might be a polynomial name; in this case, the value currently assigned to p is given to f. All commands will be terminated by semicolons. Examples of this command are

```
Set P to z*x↑2+3.5;
Set fpt to P;
```

Most of the other commands perform some operation on their operands and then leave the results as value for some polynomial name.

Set f to the sum of p and q; Form the algebraic sum of p and q and store the resultant value with name f. In all these commands the input can be free form and lines can flow across record boundaries; only the semicolon is important for ending commands. Operands might be polynomial names, in which case the values assigned to the names are used in the operations.

Set f to the difference p minus q; Form the algebraic difference of p minus q and store the resultant value with name f.

Set f to the product of p and q; Form the algebraic product of p and q and store the resultant value with name f.

Set f to the quotient of p divided by q; Form the algebraic quotient of p divided by q and store the resultant value with name f. This command does not require that the division algorithm for polynomials be employed, since a rational polynomial may include one symbolic division sign. Extra division signs may be eliminated by using high school algebra.

Set f to the c power of p; The polynomial p is raised to the c power and the resultant value is stored with name f. The power c must be an integer or a polynomial name for a constant; if c is negative, the value is $1/p^{|c|}$.

Set f to p with q substituted for x; For each instance of variable x in polynomial p, substitute polynomial q and store the resultant value with name f. Notice that the substitution may reintroduce variable x into f but that such reintroduction does not imply recursive substitution.

Set f to the derivative of p with respect to x; Calculate the derivative dp/dx and store the resultant value in f. Of course, x must be a variable or a polynomial name whose value is a single variable.

Print p; Print the polynomial p in a neat format.

End; End the command string.

The print command suggests another difficulty that must be faced by all algebraic manipulation programs. During calculation polynomials are apt to become quite complicated. However, humans expect to read them in a fairly simple form. Rational polynomials are normally written as simple fractions with both numerator and denominator sums of terms involving only multiplication and exponentiation. Within each term all the constants are multiplied together and the result used as a leading coefficient, the variables are ordered (alphabetically is common), and exponents are modified so that each variable occurs only once. If the leading constant turns out negative, the term is subtracted from, rather than added to, the previous term. If the coefficient turns out to have absolute value zero or one, the term or the coefficient should be dropped as appropriate. Similarly, an exponent of zero should cause the associated variable to be dropped. When an exponent is negative, the term itself is actually a fraction and the denominator should be eliminated by using standard algebraic rules for the summation of proper fractions. Finally, like terms (that is, those with exactly the same pattern of exponents and variables) should be combined with appropriate modifications of the coefficients.

All these simplifications might be done by maintaining the polynomials in some canonical internal form. In this case, the form would be chosen so that all polynomials are always ready to print; after each operation the result might be reorganized into standard format. Alternatively, the print operation can reformat an internal polynomial only when necessary, but this process may require an arbitrarily large amount of work for a print. No matter which method is chosen, algebraic simplification suggests that a distinction should be kept between integer and real constants so that the vagaries of computer arithmetic will not preclude recognition of values of zero and one. Notice that it is customary to suppress exponents of one. Figure 20-1 shows a short program with its output.

Statement of the Theme Write a general polynomial manipulation program with the capabilities outlined above. The input should be a free form list of commands and the output a list of neatly formatted polynomials. Variables and constants, as well as command words, should not be broken across records, but commands and polynomials may well be. The definition of "neat" output is perhaps a little vague, but here is a chance to demonstrate your capability to provide users with what they cannot describe themselves. Be prepared to prove that the polynomial manipulation routines are turning out the correct answers. An important feature of a symbolic manipulator is its ability to do integer arithmetic correctly; make sure that yours has this feature.

```
Set f to (x↑2+y↑2)↑2 + 3*x*y;
Set g to (x+y)↑3 - 4;
Set h to the product of f and g;
Set aaa to f with 2 substituted for y;
Set bbb to the quotient of aaa and h;
Print g;
Print bbb;
End;
```

The output is

```
x↑3 + 3*x↑2*y + 3*x*y↑2 + y↑3 - 4

(x↑4 + 8*x↑2 + 6*x + 16)/(x↑7 + 3*x↑6*y + 5*x↑5*y↑2 + 7*x↑4*y↑3
- 4*x↑4*y - 4*x↑4 + 7*x↑3*y↑4 + 9*x↑3*y↑2 + 5*x↑2*y↑5 + 9*x↑2*y↑3
-8*x↑2*y↑2 + 3*x*y↑6 + 3*x*y↑4 - 12*x*y + y↑7 - 4*y↑4)
```

Figure 20-1. A Sample Program and Its Output

Performance Practice Reading the commands and polynomials will require some simple compiler techniques, particularly a lexical analyzer to recognize the symbols and a syntactic analyzer to construct internal representations. References for other chapters will provide the knowledge needed. During execution you will need to maintain a growing table of names and values, and once again the technique is easy. The most difficult feature of the implementation is the choice of representation for the polynomials themselves. Surely they will be in some sort of tree or list structure, but what sort exactly?

One form would be a standard arithmetic tree with variables and constants at the leaves and operators at the internal nodes. Such a form would be particularly appropriate for substitution and algebraic operations, but it might be messy for printing. An alternate would be a tree with numerator and denominator as topmost branches, terms at the next level, factors at the next. Such a tree might be easy to print and hard to manipulate. Whichever is chosen, remember to copy structures when making substitutions; otherwise a later change to the inserted polynomial will also change the polynomial into which it was inserted.

Orchestration This is another problem that calls for lists or trees and recursive procedures to process them. LISP was invented for such problems, and many other list-processing languages would be equally appropriate. SNOBOL is perhaps slightly weaker in data manipulation internally, but its superior input analysis and output preparation also make it a strong candidate. In fact, any language like PASCAL or PL/I with some string manipulation capability, defined data structures, and recursive procedures should be adequate.

Playing Time One person for 3 weeks.

Variations on the Theme A number of algebraic manipulation systems are in common use. Typically, they grow from a base of functions like those described above. The growth may come in three areas: addition of more data types, addition of new operations, and addition of heuristic procedures that attempt calculations that may not have a well-formed result. New data types and new operations are related. For instance, we might add the trigonometric, exponential, and logarithmic functions to our repertoire of rational polynomials. If so, the exponentiation operation must be changed to allow the use of any operand as exponent, and a logarithm-generation operation specifying base and argument will be needed. Notice that when new data types and operations are added, there must be a check to guarantee that the space of all functions that can be generated under all possible sequences of operations is closed in the sense that there is no generated function that could not, in principle, be written in a Set statement.

Several important mathematical operations have no guaranteed method by which symbolic results may be calculated. Most important among them is integration. Although every rational polynomial has an indefinite integral, the simple example of $1/x$ [whose indefinite integral is $\log_e(x)$] shows that we do not have to look far for a function that breaks out of the closed space of rational polynomials. If, as suggested above, logarithms and exponentials are added to the available functions, the larger function space exacerbates the problem. Even the use of definite integrals will not solve it, since the operation may not result in a constant if the integrand contains variables other than that integrated over or if the limits of integration are not constants. Symbolic integrators were among the first programs written to display "intelligent" behavior; if you double or triple the time spent on this problem, you may be able to build a rudimentary integrator.

The addition of extra base functions will introduce another problem. A standard print format does not exist for the more complicated functions that can now be constructed. Furthermore, the

application of simplification laws is much more a matter of judgment. Because far more algebraic laws are available—trigonometric identities, laws relating exponentials and logarithms, laws about constants—the program could spend much of its time simplifying internal expressions. Simplification to aid human understanding of results is an area of considerable subtlety and importance; a successful implementation is a credit to the programmer's judgment.

REFERENCES

Moses, Joel. "Algebraic Simplification: A Guide for the Perplexed," *CACM*, 14, 8, pp. 527-537, 1971.

Moses, Joel. "Symbolic Integration: The Stormy Decade," *CACM*, 14, 8, pp. 548-560, 1971.

This entire issue of *CACM* is devoted to symbolic algebra and its uses. The two Moses papers are good surveys, but the others will prove interesting as well. Bibliographies here should provide a fine trail for searching out any topics in this whole area.

Perverse Inverse

or...

ERRORS USING FLOATING POINT

High school algebra often teaches techniques that took centuries for the very greatest mathematicians to discover. Among these techniques are solution methods for sets of linear equations and, implicitly, methods for inverting square matrices. Once the budding algebraist learns the algorithms, there seems little question that they will always work; the doubter is convinced by following a few examples through the machinery. What a shock when the same young mathematician writes a program using the simple and guaranteed algorithm and the program fails completely. How could a matrix inverter invented by Gauss, prince of mathematicians, fail?

First, let us review the basic ideas. A matrix is a square array of real numbers, $n \geqslant 1$ elements on a side. The product C of matrix A multiplied on the right by matrix B, written C = AB, is given by

$$C_{ij} = \sum_{k=1}^{n} A_{ik}B_{kj}, \quad 1 \leqslant i \leqslant n, \quad 1 \leqslant j \leqslant n$$

where it is understood that A, B, and C are n by n matrices. Multiplication is noncommutative, since it is possible to find matrices such that $AB \neq BA$. The inverse of matrix A is the matrix A^{-1} such that

$$AA^{-1} = A^{-1}A = I$$

where I is the identity matrix defined by $I_{ii} = I$ and $I_{ij} = 0$ for $i \neq j$. Most matrices have an inverse, but some do not. Unfortunately, the easiest way to find these singular matrices is to try and calculate the inverse and fail.

How is the inverse calculated? The following algorithm is due to Gauss.

> First, set X to I. During the process A will gradually be turned into I and the initial value of I in X will gradually turn into the correct inverse A^{-1}.
>
> For each column of A, working from column 1 on the left to column n on the right, do the steps below. At each stage let the column currently being processed be j.

Let $M = \max_{j \leq i \leq n} |A_{ij}|$. M is the element in column j below row j-1, which has the largest absolute value. If M is zero, A is singular and there is no need to go on. Otherwise exchange row j and the row in which M occurs in both A and X. Finally, divide every element of the new row j in both A and X by the new A_{jj}.

Now for every row i, $i \neq j$, do all the subtractions

$$A_{ik} = A_{ik} - A_{ij}A_{jk}, \quad j \leq k \leq n$$

and

$$X_{ik} = X_{ik} - A_{ij}X_{jk}, \quad 1 \leq k \leq n.$$

The effect of all this element-by-element subtraction is to subtract A_{ij} times row j from row i in both A and X. After this step is done for each j, all elements below and above A_{jj} will be zero and A_{jj} itself will be one. Also, in the A matrix there need not be any subtractions left of column j because all the members of row j to the left of A_{jj} are zero.

Any algebraist will be happy to prove that this algorithm always works perfectly and that when it stops, if A is not singular, then $X = A^{-1}$. You could hardly ask for an algorithm more suited to a structured implementation. But, just for the fun of it, why not do a little test? The Hilbert matrix of order n H^n is defined by

$$H^n_{ij} = 1/(i+j-1), \quad 1 \leq i \leq n, \quad 1 \leq j \leq n.$$

Calculate the inverse of H^n for n = 1, 2, . . . ,20, 25, 30, 35, 40, 45, 50. As you are undoubtedly aware, the answers will not be *quite* correct because of small errors in computer arithmetic, but they should be very close to the exact inverses. The measure of the error is the left residual matrix $L = (H^n)^{-1}H^n - I$ and the right residual matrix $R = H^n(H^n)^{-1} - I$, both of whom should be all zero but probably will not be.

Of course, if the elements of L and R were all 10^{-20}, say, there would be no problem. For all practical purposes, 10^{-20} is zero if the elements of the original matrix average about 1/50 or larger. But there is a precise way of measuring the size of the residuals L and R. Let the row norm of matrix A be defined by

$$|A|_r = \max_{1 \leq i \leq n} \sum_{j=1}^{n} |A_{ij}|.$$

Add to your program that calculates the Hilbert matrix inverses a routine to print a table of $|L|_r$ and $|R|_r$ for each inverse. After you have checked your program for errors, would you please explain why the residuals are so big? Are you *sure* that the program is right?

Your program is correct; computer arithmetic errors are doing the damage. The Hilbert matrices look innocuous, but they are designed to show the effect of a cumulative error on a long series of related computations. You may think that the trouble comes because your computer does not store enough digits with each real number internally. Most computers offer double-precision arithmetic. If you change to double precision in the algorithm, you may be able to ameliorate the problem, but you certainly will not fix it. This whole etude is a study in the effect that limited precision arithmetic has on algorithms that are guaranteed perfect for mathematician's "real" numbers. Applied mathematicians and numerical analysts in programming laboratories spend much of their time modifying theoretical algorithms to work on real computers.[1]

[1] In fact, searching for the maximum element M of a column in step 3 of the inversion algorithm is one such modification. M is called the pivot element, the operation is pivoting, and actually only a nonzero M is necessary. The maximum element is used merely to hold down arithmetic errors in the computer. In fact, when inverting a Hilbert matrix, the pivot element should always be H^n_{jj}; and if the algorithm pivots a row on an element lower down the column, errors in the calculation are already very large.

Statement of the Theme Implement the matrix inversion algorithm and test it on the Hilbert matrices of the orders mentioned above. Print a table or plot a graph of $|L|_r$ and $|R|_r$ versus the order of n of H^n. If your language offers a choice of precision for real numbers, rerun the inversions by using a greater precision to see if the error table or graph shows any improvement. (A wise programmer will arrange the code so that precision can be changed by modifying a few declarations.) Also keep track of how often rows are actually interchanged during pivoting as a record of how badly the algorithm has deviated from theory.

Performance Practice This is a straightforward algorithm implementation. The only difficulty should be checking that the program actually reflects the theoretical definitions and algorithm. Do not optimize the algorithm in any way; you are studying how bad things can be if the mathematician's advice is followed without meeting the cardinal assumption of precise arithmetic.

Orchestration Any algebraic language will be appropriate. FORTRAN was designed for matrix problems. Compare it to a more modern language by coding this etude in both.

Playing Time One person for 1 week.

Variations on the Theme If this problem is much extended, it becomes the core of a semester course in numerical analysis. But you might find out more about the behavior of the errors if you calculated |L| and |R| by using other norms beside the row norm. The column norm is

$$|A|_c = \max_{1 \leq j \leq n} \sum_{i=1}^{n} |A_{ij}|$$

and the L_1 norm, the L_2 norm, and the L_∞ norm are

$$|A|_1 = \sum_{ij} |A_{ij}|,$$

$$|A|_2 = \text{sqrt}\left(\sum_{ij} A_{ij}^2\right),$$

and

$$|A|_\infty = \max_{i,j} |A_{ij}|$$

respectively. Add these norms to your error analysis tables. Do any of them show any significant difference from the others in the shape or rough magnitude of the error curves?

REFERENCES

Conte, S. D., and Carl deBoor. *Elementary Numerical Analysis*, 2nd ed. McGraw-Hill, New York, NY, 1972.

Stewart, G. W. *Introduction to Matrix Computations.* Academic Press, New York, NY, 1973.

Conte and deBoor wrote an excellent introductory numerical analysis text, used in many schools. Certainly it contains more information about numerical analysis than any *normal* person would like to know. But if you insist on finding out more about this matrix problem, Stewart describes the theory and practice of linear algebra.

Pi Are Square

or...

HIGH PRECISION ARITHMETIC ROUTINES

Mathematics, often regarded as a cold science, has its human stories. One of the saddest is that of William Shanks, who in the nineteenth century set himself the task of calculating π very precisely. Over several years Shanks worked away and in 1873 published π to 707 decimal places, with a correction to several digits published later. Perhaps it is just as well that Shanks died in 1882, because in 1946 the calculation was shown to be wrong, starting at the 528th decimal place. Shanks had actually progressed no further than the previous computations.

The check of Shanks' value was probably done with mechanical assistance, but apparently the first use of a computer to help in the calculation of π had to wait until 1949 and the ENIAC. Even then, the project was monumental. George W. Reitwiesner reports "Since the possibility of official time was too remote for consideration, permission was obtained to execute these projects during two summer holiday week ends when the ENIAC would otherwise stand idle." The actual calculations — not the programming! — took 70 hours and produced slightly more than 2000 digits. The computer had to be attended at all times because its limitations required repeated punching and reading of intermediate results. Those first programmers are as far behind us as Shanks was behind them.

How would one go about calculating π? First, we need an expression that can be evaluated. The series

$$\pi/4 = 1 - 1/3 + 1/5 - 1/7 + 1/9 - \cdots$$

is easy enough to understand but converges terribly slowly. Much better is the series for arctangent

$$\arctan(x) = x - x^3/3 + x^5/5 - x^7/7 + \cdots, |x| \leqslant 1.$$

We combine this with the summation formula for tangent

$$\tan(a+b) = (\tan(a)+\tan(b))/(1-\tan(a)\tan(b))$$

Now choose a and b so that $\tan(a+b) = 1 = \tan(\pi/4)$. [For example, let a = arctan(1/2) and b = arctan(1/3) and remember that $\tan(\arctan(x)) = x$ for $-\pi/2 < x < \pi/2$.] Then

$$\arctan(\tan(a+b)) = a+b = \arctan(1) = \pi/4$$

and we can use the series above to find a and b. The actual sums commonly used are

$$\pi/4 = 4\arctan\left(\frac{1}{5}\right) - \arctan\left(\frac{1}{239}\right),$$

$$\pi/4 = 8\arctan\left(\frac{1}{10}\right) - 4\arctan\left(\frac{1}{515}\right) - \arctan\left(\frac{1}{239}\right),$$

and

$$\pi/4 = 3\arctan\left(\frac{1}{4}\right) + \arctan\left(\frac{1}{20}\right) + \arctan\left(\frac{1}{1985}\right).$$

So now we set off to sum these series by computer, and we know that a simple iterative loop is all that is necessary for a sum—except for one problem. Computers have limited precision, and the whole point of this exercise is to find many, many digits of π, far beyond ordinary precision. The obvious first answer is to simulate hand arithmetic in the calculation. Store one decimal digit per entry in very long integer arrays, and it is easy to see how addition, subtraction, and multiplication routines might be written. Hand division is a touch harder but still possible. The difficulty is the time that these routines take. There is seldom reason to notice, but hand methods take time proportional to n^2 to multiply or divide n digit numbers. In contemplating operations on numbers that are thousands of digits long, such costs become prohibitive. Fortunately, better algorithms exist.

HOW TO MULTIPLY QUICKLY

The Toom-Cook algorithm for fast multiplication, described by Knuth, depends on four basic ideas.[1] First, let us assume that we know how to do some operation on inputs of size n in time T(n). If we can split the operation up into r parts, each of which takes fewer than T(n)/r steps, then we can improve the total time by the split, always assuming that the extra housekeeping does not eat up the savings. In addition, if the r parts each consist of reapplying the algorithm to inputs of length n/r *and* the split can be worked on each of the parts, then we can continue splitting right down to inputs so short that their output calculation is trivial and costs only some small constant time. This strategy of divide and conquer usually gains a speedup in the original algorithm's time of at least a division of a logarithm; for example, in multiplication we can go from n^2 by the classical method to $n^{1+7/\sqrt{4\log_2 n}}$ time, which is a considerable speedup for large n (remember that there are leading proportionality constants for both cost functions).

The other three ideas all concern numbers and the manipulation of polynomials. First, notice that if number U is n bits long and has the bit representation, where n is a multiple of (r+1),

$$u_{n-1}u_{n-2}\cdots u_2u_1u_0$$

then it can also be written

$$U_r 2^{rn/(r+1)} + U_{r-1}2^{(r-1)n/(r+1)} + \cdots + U_1 n/(r+1) + U_0$$

where each U_i is a block of n/(r+1) bits from the original representation of U.

[1] We have resisted the temptation to remark that if you want to know how to multiply quickly, you should ask a rabbit.

In fact, $U = \mathscr{U}(2^{n/(r+1)})$, where the polynomial $\mathscr{U}(x)$ is

$$U_r x^r + U_{r-1} x^{r-1} + \cdots + U_1 x + U_0.$$

Second, we observe that if U and V are n bit integers represented in this polynomial format, then their product W is

$$W = UV = \mathscr{U}(2^{n/(r+1)})\mathscr{V}(2^{n/(r+1)}) = \mathscr{W}(2^{n/(r+1)})$$

and if we could only find the coefficients of $\mathscr{W}(x)$, it would be relatively easy to calculate W from $\mathscr{W}$ by using only shifts, adds, and n/r bit multiplications. Third, and fortunately, $\mathscr{W}(x)$ is a polynomial of degree 2r, and it can be interpolated from its values at 0, 1, 2, $\cdots$, 2r-1, 2r. These values are simply $\mathscr{U}(0)\mathscr{V}(0)$, $\mathscr{U}(1)\mathscr{V}(1)$, $\cdots$, $\mathscr{U}(2r)\mathscr{V}(2r)$. Moreover, all this polynomial evaluation and interpolation can be done by using n/r bit multiplications. It seems that the stage may be set for some divide and conquer.

Because the Toom-Cook algorithm is complicated, we shall not explain it all here; you can see Knuth for that. But some basic ideas and notation are necessary. There must also be a representation for longer numbers, and we shall write [p, u] to mean the number u with p bits. Probably [p, u] will have some sort of list or string representation internally. Besides the main algorithm, we will need routines to add and subtract long numbers (just use the standard hand method, working from right to left), to multiply a long number by a small number, to divide a long number by a small number, to shift a long number by adding zeros at the right, and to break a long number [p, u] into the shorter long numbers [p/(r+1), u_r], [p/(r+1), u_{r-1}], $\cdots$, [p/(r+1), u_0] as described above. In addition to the routines that directly manipulate numbers, the algorithm uses four stacks for intermediate storage of partial results and several temporary variables: so several stack manipulation routines, as well as routines to acquire and release storage for long numbers, are needed. Housekeeping may be quite a chore.

THE TOOM-COOK ALGORITHM FOR FAST MULTIPLICATION

The input is two n-bit long positive numbers [n, u] and [n, v]; the output is their product [2n, uv]. There will be four stacks U, V, W, and C, which hold long numbers during the calculation, and a fifth stack, which will hold control codes of operations temporarily in abeyance (there are only three such codes, and small integers could be used for them). The arrays q and r of integers are indexed from 0 to 10; storage must be provided for these two arrays and a few other temporary variables mentioned in the algorithm.

1. (Start up the algorithm.) Set all the stacks empty. Set K to 1, set q_0 and q_1 to 16, set r_0 and r_1 to 4, set Q to 4, and set R to 2.

2. (Build size tables.) While $K < 10$ and $q_{k-1}+q_k \leqslant n$, do the following calculations.

Set K to K+1; set Q to Q+R; if $(R+1)^2 \leqslant Q$, set R to R+1; set q_k to 2^Q; and set r_K to 2^R.

If the loop terminates because K = 10, stop with an error message that n bits is too many and the q and r arrays have overflowed. Otherwise set k to K. Push $[q_K+q_{K-1}, v]$ and then $[q_K+q_{K-1}, u]$ onto stack C (this step will probably require padding [n, u] and [n, v] on the left with zeros). Push control code *stop* onto the control stack.

3. (Main outer loop.) While the control stack is nonempty, do steps 4 through 18. If the control stack *is* empty when returning to this step, terminate with an error message; the control stack must have at least one item in it now.

4. (Inner loop to break down u and v.) While $k > 1$, do steps 5 through 8.

5. (Set parameters for breakdown). Set k to k-1, set s to q_k, set t to r_k, and set p to $q_{k-1}+q_k$.

6. (Breakup top of stack C.) Regard the long number $[q_k+q_{k+1}, u]$ on top of stack C as really t+1 numbers, each s bits long. Break $[q_k+q_{k+1}, u]$ up, working from left to right, into the long numbers $[s, U_t], [s, U_{t-1}], \cdots, [s, U_1], [s, U_0]$. These t+1 numbers are the coefficients of a t-degree polynomial that is to be evaluated at the points 0, 1, $\cdots$, 2t-1, 2t, using Horner's rule. Now calculate $[p, X_i]$, for i = 0, 1, $\cdots$, 2t-1, 2t by evaluating

$$(\cdots([s, U_t]i+[s, U_{t-1}])i+\cdots+[s, U_1])i+[s, U_0]$$

and immediately push $[p, X_i]$ into the U stack. The multiplications can be done by using the multiply long by short routine, and no intermediate or final value will require more than p bits. Pop $[q_k+q_{k-1}, u]$ off stack C.

7. (Continue breakup.) Do exactly the same series of operations as step 6 on the number [m, v] now on top of stack C to form $[p, Y_0], \cdots, [p, Y_{2t}]$ and push them onto stack V in the order that they are formed. Do not forget to pop C.

8. (Refill stack C.) For 2t times, alternately pop stacks V and U and push the values popped onto stack C. The effect is to interweave the values calculated in steps 6 and 7 and store them back into stack C. After this weave the top section of stack C, reading *upward*, should be

$$[p, Y_{2t}], [p, X_{2t}], \cdots, [p, Y_0], [p, X_0],$$

with this last value on top. Now push one *interpolate* code and 2t *save* codes onto the control stack and return to step 4.

9. (Prepare to interpolate.) Set k to 0. Pop stack C twice into ordinary variables u and v. Both u and v will be 32 bits long. Using a multiply routine other than this one, calculate [64, w] = [64, uv]. The multiplication can be done by hardware or in a subroutine, as you see fit.

10. (Interpolate if necessary.) Pop the control stack into variable A. If A is equal to *interpolate*, do steps 11 through 16; otherwise go on to step 17.

11. (Setup interpolation.) Push [m, w] onto stack W (this may be the value from step 9 or from step 16). Set s to q_k, set t to r_k, and set p to $q_{k-1}+q_k$. Now the top section of stack W, reading *upward*, is to be regarded as

$$[2p, Z_0], [2p, Z_1], \cdots, [2p, Z_{2t-1}], [2p, Z_{2t}],$$

with this last value on top.

12. (Outer loop dividing Zs.) For i = 1,2, $\cdots$, 2t, do step 13.

13. (Inner loop dividing Zs.) For j = 2t, 2t-1, $\cdots$ i+1, i,

$$\text{set } [2p, Z_j] \text{ to } ([2p, Z_j] - [2p, Z_{j-1}])/i.$$

The difference will always be positive and the division will always be exact—that is, with no remainder.

14. (Outer loop multiplying Zs.) For i = 2t-1, 2t-2, $\cdots$, 2, 1, do step 15.

15. (Inner loop multiplying Zs.) For j = i, i+1, $\cdots$, 2t-2, 2t-1,

set $[2p, Z_j]$ to $[2p, Z_j] - i[2p, Z_{j+1}]$.

Each difference will be positive and all results will fit in 2p bits.

16. (Form new w and loop again.) Set the polynomial

$$(\cdots([2p, Z_{2t}]2^s+[2p, Z_{2t-1}])2^s+\cdots[2p, Z_1])2^s+[2p, Z_0]$$

into $[2(q_k+q_{k+1}), w]$. This step can be done by using only shifts and long-number addition. Notice that this is the same variable [m, w] as is used in step 9. Pop $[2p, Z_{2t}], \cdots, [2p, Z_0]$ from the W stack. Set k to k+1 and return to step 10.

17. (Check termination.) If A has value *stop*, the result of the algorithm is [m, w], just calculated in step 9 or 16. Exit if so.

18. (Save a value.) The value of A must be *save* (if not, terminate with an error). Set k to k+1 and push $[q_k+q_{k-1}, w]$ onto stack W. This is the same w just calculated in step 9 or 16. Now return to step 3.

NOTES ON THE TOOM-COOK ALGORITHM

We provide little motivation or explanation for this algorithm; some trust is required. Actually, the reason for the lack is that the explanation is extremely long and mathematical and we simply do not have the space. The presentation relies heavily on Knuth; if you want to know more, you should follow up Knuth. However, we do have some notes that may help your understanding.

1. (The structure of the algorithm.) The major difference between our version and Knuth's is the structure of the loops. Figure 22-1 gives a high-level view of the Toom-Cook algorithm.

2. (Table of sizes.) The arrays calculated in step 2 have the values shown in Table 22-1, where the values in the column headed n_k are the largest number of bits that the algorithm will handle when K = k. Obviously, the limit of 10 on K is not a very serious one. It could be raised if desired.

3. (Stack depths in the first loop.) In steps 5 through 8, the maximum depth of the U and V stacks is $2(r_{K-1}+1)$. The C stack may grow to a depth of $\Sigma_{i=1}^{k-1}(r_i+1)$.

4. (Stack depths in the second loop.) The W stack may achieve a total depth of

Table 22-1 A Table of q, r, and n

k	q_k	r_k	n_k
0	16	4	
1	16	4	32
2	64	4	80
3	256	4	320
4	1024	8	1280
5	8192	8	9216
6	65536	16	73728
7	1048576	16	1114112
8	16777216	16	17825792
9	268435456	32	285212672
10	8589934592	32	8858370038

```
begin
    long integer [n, u], [n, v];
    integer K, Q, R, q[0:10], r[0:10];
    long integer stack C, U, V, W;
    control stack code;
    integer k, p, s, t, i, j;
    long integer X, Y, Z, w;
    control A;
    Step 1;
    Step 2.
    while code not empty do
        while k > 1 do
            Step 5; Step 6; Step 7; Step 8;
        end;
        Step 9;
        pop code into A; while A = interpolate do
            Step 11;
            for i = 1 to 2t do Step 13;
            for i = 2t-1 downto 1 do Step 15;
            Step 16;
        end;
        if A = stop then return [m, w];
        if A = save then Step 18; else abort;
    end; * of main loop. *
end; * of Toom-Cook algorithm. *
```

Figure 22-1. A Control Sketch of the Toom-Cook Algorithm

$\sum_{i=1}^{k-1} 2r_j$. The control stack could reach the depth $\sum_{i=1}^{k-1} 2r_j+1$. The top section of stack W is used as an array in steps 14, 15, and 16. This array will have at most $2r_{k-1}+2$ entries in it.

5. (Input sizes.) For any number of bits n in the range $n_{i-1}+1 \leqslant n \leqslant n_i$, the Toom-Cook algorithm takes the same calculation time. That is, the cost of the algorithm is "lumpy" with respect to the size of the input. Thus it makes sense when doing long calculations to pick a number of bits near the high end of one of the ranges for n. Remember also that it takes about three and a third bits to represent one decimal digit.

6. (How to multiply two 32-bit numbers.) Step 9 requires the multiplication of two 32-bit numbers to form a 64-bit product where both factors are always positive. Many computers have the hardware to form such a product, but the result is not available to higher-level languages; other computers do not even have the hardware, of course. So a subroutine must be written to do this multiplication, and since it is the basis of the algorithm's cost, the subroutine should be efficient. Probably breaking the numbers up into pieces and simulating hand multiplication will be good enough. However, if we want to form the product uv, we write u as $u_1 2^{16}+u_0$ and v as $v_1 2^{16}+v_0$ and the product is

$$(2^{32}+2^{16})u_i v_1+2^{16}(u_1-u_0)(v_0-v_1)+(2^{16}+1)u_0 v_0.$$

This can be done entirely with 16-bit subtracts and multiplies and some shifts and adds. Notice that one multiply has been saved.

WHAT ABOUT DIVISION?

When the series are calculated, there are some high-precision divisions to go with the multiplications. Fortunately, division can be done almost as fast as multiplication by making use of the multiplication algorithm. The technique is to guess a reciprocal to the divisor, correct it so that the reciprocal

is accurate, and then multiply by the dividend to find the quotient. The refinement of the reciprocal is by Newton's method.

The input is [m, u] and [n, v], where we assume that $u \geqslant v$ (although this assumption is not essential) and that the nth bit of v is a 1 (that is, that v has no leading zeros). The greater the size difference between u and v, the more accurate the quotient; the difference can be accentuated by multiplying u by a power of 2. Notice that the division algorithm will repeatedly call the multiply algorithm. The first few of these multiplications could be taken care of by ordinary short multiplies. Also, all the multiplies and divides by powers of 2 are actually left and right shifts.

1. (Choose the reciprocal size.) Choose the smallest j such that $s^j \geqslant \max(m, 2n)$. Set k to 2^{j-1}.

2. (Normalize v.) Set [k, v] to $2^{k-n}[n, v]$. This step shifts v left to occupy k bits with the leftmost bit a 1. Set [2, a] to [2, 2].

3. (Compute successive approximations to 1/v.) For i = 1 to j-1 by steps of 1, do Step 4.

4. (Compute a 2^i-bit approximation.) Set $[2^{i+1}, d]$ to

$$2^{3 \cdot 2^i}[2^{i-1}+1, a] - [2^{i-1}+1, a]^2([k, v]/2^{k-2^i})$$

The division in the parentheses (really a right shift) should be done before the multiply; the idea is to throw away bits of v not needed in this approximation so as to speed up the multiply. Although it seems as if there might be more than 2^{i+1} bits in the result d, there never will be. Now set $[2^i+1, a]$ to $[2^{i+1}, d]/2^{2^i-1}$.

5. (Improve the final estimate.) Set [3k, d] to

$$2^{2k}[k+1, a] - [k+1, a]^2 \cdot [k, v]$$

Now set [k+1, a] to

$$([3k, d]+2^{2k-2})/2^{2k-1}$$

6. (Final division.) Output

$$([k+1, a] \cdot [m, u]+2^{k+n-2})/2^{k+n-1}.$$

HOW TO USE THE ALGORITHMS

The actual computation of π requires evaluation of one of the equations listed early in the etude, using the arctangent series given. In fact, for safety, two of the formulas should be used and the final results compared bit for bit. The value of π is given by the common prefix of the two results.

Yet there still remains the problem that the algorithms given here work only for integers; how are the obviously fractional values of the series to be formed? Assume that we want to calculate π to, say, a thousand bits of accuracy. Then what we actually calculate is $2^{1000}\pi$ by multiplying all the numerators by 2^{1000}. This procedure will also serve the purpose of making dividends much larger than divisors (as suggested above) and of providing a stop to the calculation when the quotients become zero.

Now let us choose a series (not necessarily the best) for evaluation, say

$$\pi = 16\arctan\left(\frac{1}{5}\right) - 4\arctan\left(\frac{1}{239}\right).$$

Actually, we will evaluate $2^{1000}\pi$, and so we want to evaluate $2^{1000} \cdot 16\arctan(1/5)$. The first term of this series is $2^{1000} \cdot 16/5$; we will call it a_1 (notice that it is added into the sum.) Now to form term a_{i+1} from term a_i, divide a_i by $5 \cdot 5 \cdot (2i-1)$. If a_i was added into the sum, subtract a_{i+1}; if a_i was subtracted, add a_{i+1}. Calculate the terms of $2^{1000} \cdot 4\arctan(1/239)$ in parallel and stop as soon as any quotient in either series is zero. The result will be about a thousand bits of π. Of course, output will require conversion to decimal.

Statement of the Theme Build routines for the multiply and divide algorithms described above and for all the support services that they require. Use these routines to calculate π to high precision by using one of the series described. Be careful not to intertwine the arithmetic routines too closely with the π calculation; a library of high-precision routines may be useful for other problems. It should be possible to compute to higher precision by providing more storage for results without modifying the code. Output should include statistics on the use of each routine, on the number of times each step is executed in the two central algorithms, and on the use of storage. Such collection will cost very little in the overall problem.

Performance Practice This is a long and difficult etude, not least because the two central algorithms must be taken with a certain amount of trust. As is quite common with real problems, however, the central issue is actually the selection of a data structure rather than the construction of code. How are long integers to be represented? The notation [m, u] suggests a structure. Each long integer should be a pair of a *length* and a *value*. The *length* part is easy to implement, but the *value* is obviously of variable length and will be hard to store directly. So we make the *value* be a pointer into a very long vector of bits, and each pair is now of fixed size. However, our vector is unlikely to be so long that we can afford to use each section of it only once. Storage reclamation routines to recover bits no longer in use will thus be necessary. In fact, what we have just described is a conventional string allocation scheme.

To summarize, we need the following service routines besides the multiply and divide algorithms.

Supply Storage. Given as input a *length*, this routine returns a pointer into the vector of bits that can be used as a *value*. Starting at bit *value*, there are *length* bits that will not be used for anything else.

Return Storage. The pair *length* and *value* are inputs to this routine, and the associated storage in the bit vector is returned for reuse. This routine should be called whenever an item changes length.

Reclaim Storage. This routine must run through the storage in use and try to combine the unused sections of the bit vector into long sections. Normally this routine will be called because a request to supply storage cannot find a long enough string of consecutive bits. Because short problems may be done without this facility, it should be coded last. There are a number of possible ways to keep track of unused storage.

Shift. The input to this routine is a long integer and a shift amount; the output should be a long integer shifted left or right the appropriate amount. This operation corresponds to multiplication or division by a power of 2.

Add. The input to this routine is a pair of long integers, and the output should be their long integer sum, one bit longer than the longer input. Such additions can be done just as they are by hand, working from right to left.

Subtract. This routine is parallel to the addition routine and returns the difference of two long integers.

Zero Suppression. The input to this routine is a long integer, and the output is a shorter long integer with all the leading zeros suppressed but with the same value. If the input happens to be exactly zero, the output should be [1, 0].

Short Multiply. The inputs are two long integers of exactly 32 bits, and the output should be their 64-bit product. The multiplication may be done in any convenient way.

Long/Short Multiply. The inputs are a long integer and an ordinary integer of value 64 or less, and the output should be their long integer product. This operation can be done from right to left, as it might be by hand.

Long/Short Division. The inputs are a long integer and an ordinary integer of value 64 or less, and the output should be the long quotient of the long integer divided by the short integer. This operation can be done from left to right, much as it might be done by hand.

Convert. The input to this routine is a long integer, and the output is the value of the integer written in decimal on some output device. Some more elaborate specifications for this routine might be developed as output needs grow more complicated.

Orchestration PASCAL comes immediately to mind as an implementation language because of the good data structures and control facilities. But PASCAL lacks the ability to convert easily between internal bit representations and programmer's bit representations. Lower-level languages like BLISS and XPL provide closer access to the computer at some loss in expressiveness and safety. PL/I combines higher-level protectiveness with access to representations at the machine level, but the cost is usually execution time and, for this etude, time spent trying to understand some of PL/I's more esoteric features. An implementation in TRAC looks interesting because the string storage problem is automatically resolved.

Playing Time One person for 5 weeks or two people for 3 weeks.

Variations on the Theme Once high-precision arithmetic routines are available, many interesting problems arise. One is the exact evaluation of e. The series for e is the particularly simple

$$e = \sum_{i=0}^{\infty} 1/i!$$

where 0! = 1. Any calculus student can think of many more series and constants.

REFERENCES

Aho, A. V., J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974. Section 8.2, pp. 279-286.

Our multiplication algorithm is drawn from Knuth and the division algorithm from Aho, Hopcroft, and Ullman; we have revised both for our needs. Both books provide considerable background and analysis of the algorithms, including cost estimates. There are also alternate algorithms for multiplication based on the Fast Fourier transform.

Brent, R. P. "A FORTRAN Multiple-Precision Arithmetic Package," Department of Computer Science, Carnegie-Mellon University, May 1976.

Brent discusses a package of subroutines for high-precision arithmetic written in portable machine-indepen-

dent FORTRAN. The bibliography will lead you to other work in this area. The package does not use the Toom-Cook algorithm and Brent explains why.

Brent, R. P. "Fast Multiple-precision Evaluation of Elementary Functions," Stanford University, Technical Report STAN-CS-75-515, August 1975.

Thomas describes the calculus necessary for these calculations and similar ones; the descriptions are simple and classical. Reitwiesner and Shanks and Wrench are two of the papers in the sequence of π calculations. Both provide some historical review and both use the approach suggested in Thomas. Brent develops some entirely new techniques for the calculation of sin, cos, log, arctan, and so on, all based on the elliptic integrals. These algorithms run much faster than the series that we describe. Brent is still a technical report but will probably be published in a journal by the time that this book is published.

Knuth, D. E. *The Art of Programming/Seminumerical Algorithms.* Addison-Wesley, Reading, MA, 1969. Section 4.3.3, pp. 258-280.

Reitwiesner, George W. "An ENIAC Determination of π and e to More than 2000 Decimal Places," *Mathematical Tables and Aids to Computation*, 4, pp. 11-15, 1950.

Shanks, D., and J. W. Wrench. "Calculation of π to 100,000 Decimals," *Mathematics of Computation*, 16, pp. 76-99, 1962.

Thomas, G. B., Jr. *Calculus and Analytic Geometry*, 3rd ed. Addison-Wesley, Reading, MA, 1960. Section 16.3-3, pp. 809-812.

23

Mastermind

or...

OPTIMAL STRATEGIES FOR A GUESSING GAME

Games have themes and motifs just like music. Often the best games, new and old, succeed because they artistically recombine a few of the age-old principles of game design. As in music, a new face of an old idea can be more appealing than a mishmash of half-baked *idees nouveaux*. In the mid-1970s the game *Mastermind* rose to popularity in England and it bids fair to become a classic. You and your computer will enjoy playing.

Mastermind's rules are extremely simple. One player, the setter, writes down a secret combination of any four digits in the range 1 to 6 (repeats are allowed) called the code. The second player, the analyst, tries to uncover the code by a judicious sequence of guesses called probes. A probe, like a code, is any pattern of four digits in the range 1 to 6. The analyst gives the probe to the setter, and the setter must reply by saying how many digits of the probe match the code in both position and number and how many other digits match but are in the wrong place. For example, a probe of 1123 versus a code of 4221 will receive the answer that one digit matches in the right place and one other matches but is in the wrong place. The round goes on until the analyst uncovers the code by giving a probe that exactly matches the code. The players exchange roles and play another round. The winner is the analyst who discovers the other's code in fewer probes. Although luck plays a part, the player who consistently draws correct conclusions from inductive data should have the better record. As a matter of practical policy, you should try to draw negative inferences about what the code could not possibly be from the answers to your probes; psychological tests show that most people find this very difficult. Table 23-1 shows one complete round.

A program that takes the role of setter would be easy to write, and there is a certain fun in sharpening one's wits against puzzles set by the machine. But it would be more interesting if the computer would play the role of analyst as well so that complete games with a winner can be scored. Bob Cooley of Lawrence Livermore Laboratory and D. E. Knuth have developed similar strategies to make the computer a compleat *Mastermind* player. For both strategies, the idea of a solution pool is central. The initial solution pool P_0 consists of all possible codes (and thus has 6^4 members); after the ith probe G_i, the pool P_i consists of all those members of pool P_{i-1} which have not been elim-

Table 23-1 An Example *Mastermind* Game

The code is 4651.		
Probe 1 is 2345.	0 exact matches.	2 matched numbers.
Probe 2 is 4516.	1 exact match.	3 matched numbers.
Probe 3 is 5461.	1 exact match.	3 matched numbers.
Probe 4 is 4165.	1 exact match.	3 matched numbers.
Probe 5 is 4615.	2 exact matches.	2 matched numbers.
Probe 6 is 4651.	4 exact matches.	Game over.

inated by response R_i. Or, in other words, pool P_i is the set of all combinations that *might* still be the code, and the goal of the analyst is to reduce some pool to a single element.

Strategy one by Cooley is slightly simpler. Let probe G_1 be any pattern with one repeated digit—for example, 4311, 6552, 1335—chosen at random. Make the probe and from response R_1 form pool P_1. Now probe G_{i+1} is formed from pool P_i, $i \geqslant 1$, by comparing each combination C of P_i against probe G_i in turn. The combination C that is *least* like G_i is chosen, where likeness is measured first by the number of exact digit matches and for equal numbers of exact matches, by the number of matching but misplaced digits. Thus of the three combinations 2641, 2356, and 1345, 1345 is most like 2345 and 2641 is least like. If there is a tie for dissimilarity, one of the candidates may be chosen at random. As soon as a response of four exact matches is made, the round is over and, of course, from a pool of one element the next probe should always be that element. Experiments show that the size of the pools shrinks by about a factor of 4 after each probe and that no more than 6 probes are ever needed.

The second strategy is due to Donald Knuth and he claims that it is optimal in the sense that it minimizes the largest number of guesses needed to find the code; no code takes more than five probes. The principle of the algorithm depends on the observation that we would like pool P_i to be as small as possible. Thus we choose probe G_i to minimize $|P_i|$ over all possible responses R_i. Any combination C is a candidate for probe G_i. Test each possible combination C against old pool P_{i-1} and let $S_{C,<0,0>}$ be the number of members of P_{i-1} that would give a response of zero exact matches and zero color only matches, let $S_{C,<0,1>}$ be the number of members that would give a response of zero and one, and so on through $S_{C,<4,0>}$ for an exact hit of four exact matches. Now let

$$S_C = \max_{<i,j>} S_{C,<i,j>}$$

and choose for probe G_i the combination C that minimizes S_C (if there is more than one such C, choose one that is a member of P_{i-1} if possible; otherwise choose randomly). You may have noticed that you could use this algorithm to analyze *Mastermind* in advance so that no testing of combinations during the game is necessary. Knuth does such an analysis to show that xxyy, for $x \neq y$, is an optimal first probe using this strategy. A test for your program is whether it comes up with xxyy as a starting probe.

Statement of the Theme Write a program that will play complete games of *Mastermind.* Implement an analysis strategy so the computer can guess as well as set codes. Along with the play routines, your program might keep performance records on players. The local *Mastermind* ace might want to travel to England for the next championships. As with all game programs that may interact with relatively unsophisticated users, input should be simple, and output should be both clear and visually appealing.

Performance Practice The only serious problem in this etude is efficiency in programming the analysis algorithms—efficiency in both space and time. In particular, the second strategy requires a long inner loop. Notice that the combinations are integers written in base 6 (using digits 1 through 6 instead of 0 through 5). The language that you choose will probably have an influence on the representation used, but try to build an efficient inner loop to decode combinations.

Orchestration Almost any procedural language with reasonable data structures should do. This program is largely an exercise in well-structured code.

Playing Time One person for 2 weeks.

Variations on the Theme The most obvious extension is to modify the number of digits from which the code can be formed or the number of positions in the code. A super version of *Mastermind* allows 5 position codes drawn from a vocabulary of 1 to 8. If either parameter is made too large, processing time may become exorbitant, but neither algorithm has any feature that depends on the numbers 6 and 4 crucially. The program could easily read the size of the vocabulary and the length of the code as input and modify its analysis routines accordingly.

REFERENCES

Aleph$_0$ "Computer Recreations," *Software—Practice and Experience*, 1, pp. 201-204, 1971.

Anonymous. *Mastermind.* Invicta Plastics, Ltd. Oadby, Leicester, England.

The original game. It bears a strong resemblance to some traditional games and its simplicity has swept England.

Knuth, D. E. "The Computer as Master Mind." Unpublished, 1976.

Knuth claims his strategy can be shown to be optimal in the way we suggest above by an exhaustive case analysis. But is it still optimal if the vocabulary size and code length are changed? And what strategy is optimal if the goal is to reduce the expected number of probes rather than the maximum number?

Tanenbaum, Andrew S. "Computer Recreations: A Heuristic for Playing Jotto," *Software—Practice and Experience*, 3, pp. 397-399, 1973.

Both articles discuss games similar to *Mastermind.* In each case, actual programs are described and some computer strategies suggested. A tournament among heuristics might be exciting.

Wells, David. "Mastermind." *Games and Puzzles*, 23, pp. 10-11, March/April 1974.

Games and Puzzles is an outstanding English magazine devoted to games, puzzles, and intellectual diversions of all sorts. It is not mathematical in tone; rather it devotes itself to historical, thematic, aesthetic, and strategic analysis of any pastime (well, almost any) that takes up no more than a tabletop. New and old games are continually reviewed. More than a few deep algorithmic problems are suggested by the puzzles. All in all, an excellent buy for the confirmed time-waster.

24

A Code of Dishonor

or...

MATHEMATICAL CRYPTANALYSIS

Imagine the following situation. Because of your truly exceptional technical knowledge and programming skill, you have risen to become the project leader of a large group building a superb and still secret Easy compiler for the EC-1 computer (see Chapters 27 and 25). On your way out of the office at about one in the morning (project leaders must set a good example), you notice a piece of scrap paper stuck in the door (reproduced in Figure 24-1). At first you think it is a core dump and are about to throw it away. Then you look again and see that the characters are arranged in groups of five—a very strange grouping for the EC-1. What can this be?

Back in your office you sit down to consider the problem. The paper is linen with a faint odor of musk about it; the writing is feminine with a Gallic flair. Now that you think about it, the new programmer Miss Hari is perhaps a touch exotic. She speaks with a French accent, she wears black cocktail dresses with a string of black pearls to emphasize her neckline, she fills a room with a musky sensuality when she enters, and she claims that her last job was with a McDonald's regional

```
AZEXQ KOAIL CMKPA TBTZU DZKRK KZTZX BYZUO BCCCV ARVTZ MLYHM TROPE
OAZKI CHZTX BQEYG MMZTO QRDXX TPHMA EETOU OSIZT MFYHB EHGVU KJROP
ERWSE LXJOL JJXGU QLAHY DSZHO QYXKJ YDLEK GOATR CDKRM JYOPH OMMZT
NNQNJ QCOVO KYTOZ RGLXE OMYTT TRYSN QNVBY KPPTL UKMRP YJZJG VQZZE
XQPOM JCEPF ICJVC LEGWJ YOPHO MMZTE RJHVQ COVZP TBHPU QAPSK BFTMS
DCMYZ APSRS SDKVU ROART VVPLF GYCCC VARVP RAOQO BVSJX JORSE IZUDY
VUKLB PMEGO DNQNJ ERGUG YLROP EQNJX TPHMA EWCFN GLPSH PQZZX OQXJZ
ZTBCW RFTAB PCADK ATRSU DLICO CQOYI RYEYD VJSON SIEVS JLAFR ASOOK
YMTIK YXAOL JCDGV EZBYI ZVOYI LLZYM LVSJD SHLIG AXCZP TBAPR BTACC
JYHZN ZYSYT ZBYZP BHBVC QCCRY HUNER BTTZR OARMJ YCMYN KAUYM AQNJC
      ZTGYA XPIPE OKVBC SCTZE UKMRP ZBCKD SMBJS JEZGY PKYSK
```

Figure 24-1. Mysterious Message Found in Computer Center. The accidental creation of English words like TROPE *probably does not signify anything. But notice the repetitions of* ROPE, MMZ, *other short groups, and especially* CCCVARV.

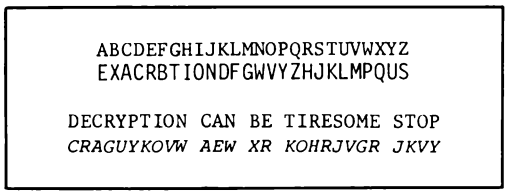


Figure 24-2. Simple Substitution with a Mixed Alphabet. Notice that period is represented by STOP.

processing center in Keokuk. There's something fishy here—wait—could it be?—Miss Hari is a spy for the famous French computer firm Ee Bay Em. And this message?—code giving away secrets of your innovative new compiler. You decide that it must be translated before Miss Hari can be confronted, but there is the problem of decrypting the message. Maybe the computer can help.

FUNDAMENTALS OF A CIPHER

The computer can certainly help or else the National Security Agency has wasted a lot of tax money on equipment. First, we need to learn something about secret messages. The data found is probably a substitution cipher in which letters of the original message are replaced by other letters, following some encryption rule. The message to be enciphered is the plaintext and the result is the ciphertext. The job at hand is to recover the plaintext and the encryption rule (although the latter is necessary only if additional messages using the same cipher may occur). We shall assume that the plaintext is in English. Separation of ciphertext into five character groups presumably hides the ordinary word structure of English, a valuable clue to decryption.[1]

The simplest general class of substitution ciphers uses a mixed alphabet—for example, a permutation of the ordinary alphabet—to build an encryption rule. Figure 24-2 shows a complete plaintext alphabet, a mixed alphabet, and an encryption of a short message where each plaintext letter is replaced by the corresponding letter from the mixed alphabet.[2] As anyone who solves Sunday newspaper puzzles knows, such simple monoalphabetic substitutions are absurdly easy to crack; a ciphered message of 30 to 40 characters is often long enough for decryption. Nevertheless, a little more ingenuity will make the system considerably more secure.

Figure 24-3 shows a Vigenère square built from the mixed alphabet of Figure 24-2. The plaintext alphabet is written across the top and down the left edge of the square. In the first row of the square lies the mixed alphabet. In the second row lies the mixed alphabet rotated left one character; notice that the original first character has migrated all the way to the right end. The square turns the single mixed alphabet into 26 separate mixed alphabets, each named by the plaintext letter to its left. Now Figure 24-4 shows how the keyword LISP is used to encipher a sentence by using the square. The keyword is written repeatedly under the plaintext, and each letter of the plaintext is enciphered by using the alphabet named by the key letter standing under the plain letter. This

[1] Cryptology has some words that are commonly misused by laymen. A cipher disguises a message by shuffling or replacing individual letters; a code replaces words or phrases rather than single letters. Persons privy to the cipher or code encipher or encode their messages and the recipients decipher or decode the messages. Persons trying to learn the secrets decrypt the messages; the difference between the verbs should suggest the difference between knowing a secret and attempting to ferret it out. Someone who uses secret writing is a cryptographer, and someone who tries to read another's secrets is a cryptanalyst. The entire study constitutes the field of cryptology.

[2] In the following discussion plaintext will be written in capitals as PLAINTEXT, ciphertext will be written in italics as *ciphertext*, and any mixed alphabets or keywords (to be defined later) will have a wavy underline written as keyword.

```
   ABCDEFGHIJKLMNOPQRSTUVWXYZ

A  EXACRBTIONDFGWVYZHJKLMPQUS
B  XACRBTIONDFGWVYZHJKLMPQUSE
C  ACRBTIONDFGWVYZHJKLMPQUSEX
D  CRBTIONDFGWVYZHJKLMPQUSEXA
E  RBTIONDFGWVYZHJKLMPQUSEXAC
F  BTIONDFGWVYZHJKLMPQUSEXACR
G  TIONDFGWVYZHJKLMPQUSEXACRB
H  IONDFGWVYZHJKLMPQUSEXACRBT
I  ONDFGWVYZHJKLMPQUSEXACRBTI
J  NDFGWVYZHJKLMPQUSEXACRBTIO
K  DFGWVYZHJKLMPQUSEXACRBTION
L  FGWVYZHJKLMPQUSEXACRBTIOND
M  GWVYZHJKLMPQUSEXACRBTIONDF
N  WVYZHJKLMPQUSEXACRBTIONDFG
O  VYZHJKLMPQUSEXACRBTIONDFGW
P  YZHJKLMPQUSEXACRBTIONDFGWV
Q  ZHJKLMPQUSEXACRBTIONDFGWVY
R  HJKLMPQUSEXACRBTIONDFGWVYZ
S  JKLMPQUSEXACRBTIONDFGWVYZH
T  KLMPQUSEXACRBTIONDFGWVYZHJ
U  LMPQUSEXACRBTIONDFGWVYZHJK
V  MPQUSEXACRBTIONDFGWVYZHJKL
W  PQUSEXACRBTIONDFGWVYZHJKLM
X  QUSEXACRBTIONDFGWVYZHJKLMP
Y  USEXACRBTIONDFGWVYZHJKLMPQ
Z  SEXACRBTIONDFGWVYZHJKLMPQU
```

Figure 24-3. A Vigenère Square Built on the Mixed Alphabet of Figure 24-2.

scheme seems to defeat simple frequency counting as a solution technique because the same plaintext letter will be enciphered different ways, depending on the key letter it falls above. Also, by deciding in advance on a list of keywords and some pattern for changing them, sender and receiver can improve their security because no two messages need have the same keyword, thereby further defying frequency analysis. But all is not black for the cryptographer.

HOW TO BREAK A CRYPTOGRAM

We shall assume that Miss Hari's cryptogram was written by using a Vigenère square, if only because Vigenère was also French. If the assumption is wrong, the solution methods will indicate it. Now if the message were a simple cipher, we could solve it by counting the frequency of each ciphertext letter, dividing each frequency by the length of the message, and comparing the resulting probability with that for plaintext English given in Figure 24-5. For a message as long as this one, the probability distributions will be almost identical when written in decreasing probability order, and

```
DECRYPTION CAN BE TIRESOME STOP
LISPLISPLI SPL IS PLISPLIS PLIS
VGLTNQFQSM LYU NP OKSPISLP IRPI
                or
VCLTN QFQSM LYUNP QKSPI SLPIR PI
```

Figure 24-4. An Encryption Using the Vigenère Square. Notice the repetition of PI at a distance of four. Notice also that the second repetition at a distance of three is spurious. Language statistics show up even in short examples.

E	.12016	M	.02783
T	.09546	P	.02482
A	.07666	F	.02275
O	.07440	G	.02027
I	.07371	B	.01837
N	.06755	Y	.01523
R	.06745	W	.01224
S	.06208	X	.00880
H	.04191	V	.00859
L	.04089	K	.00486
D	.03788	Q	.00343
C	.03664	Z	.00306
U	.03374	J	.00125

Figure 24-5. Probabilities of Letters in English Text. The probabilities were calculated from letter frequencies in this book. There was no attempt to correct for the few characters used in printing control items or the repeated headlines. Pictures and captions were not counted.

so each plaintext letter will have its ciphertext alias revealed. But the Vigenère square defeats such a simple attack. We must determine not only the mixed alphabet but also the keyword; since each works to disguise the other, it is hard to see where to begin.

The correct starting point is to find the length of the keyword. In the example of Figure 24-4, notice that the first, fifth, ninth, . . . , plaintext letters are all enciphered by using ciphertext alphabet A. If we look at only ciphertext letters in every fourth position, we should get a frequency distribution similar to English because these positions were ciphered by using only one mixed alphabet and are thus a simple substitution. Similarly, every fourth ciphertext letter starting in position two, three, or four should also give an Englishlike distribution. In fact, it is possible to measure exactly how Englishlike a frequency distribution is. Form the sum

$$IC = \sum_{i=1}^{26} \frac{f_i(f_i-1)}{N(N-1)}$$

where f_i is the frequency of the ith letter and N is the total number of letters seen. If all the letters in the sample were enciphered by one alphabet, this index of coincidence should have a value above .055 and probably below .075 (the theoretical value is .066).

So our algorithm for guessing the length of the keyword is as follows.

Step 1. For an i between 1 and 20, assume that the length of the keyword is i and do steps 2 through 4. We make 20 the upper limit for convenience only; a longer keyword is certainly possible.

Step 2. For each j between 1 and i, do step 3. These two steps will build i different ICs.

Step 3. Build a frequency distribution by using the letters in positions j, i+j, 2i+j, · · · —in every ith position starting in position j. Calculate IC_j, using the frequency distribution and the formula given above. Be sure to use only the number of letters in the sample and not the length of the whole message as a value for N.

Step 4. If all the IC_1, IC_2, · · · , IC_i are greater than .055, then i is a probable multiple of the length of the keyword. If only one IC_j is less than .055, then i is a possible multiple of the keyword length.

There is a second check on the length of the keyword. Consider two spots in the ciphertext where the same two cipher letters are repeated in exactly the same order, for instance, *RO* in positions 52 and 108 in Figure 24-1. Such a repetition could happen in two different ways: there could be any two letters of plaintext in each spot that were accidentally enciphered by *different* parts of the keyword into the same ciphertext or there could be a repetition in the plaintext that happened to fall over a repetition in the keyword and thus was enciphered twice the same way. In the second case, the distance between the beginnings of the two repetitions must be a multiple of the keyword length. Unfortunately, we have no way of knowing if this repetition happened in the first or second way, and, in fact, chance repetitions of ciphertext pairs are quite probable. But if a sequence of three or more ciphertext letters is repeated, the probability that the repetition is an accident and not an artifact of the repeating key is quite low (practically zero if the repetitive sequence has length four or more). So another way to guess the keyword length is to find all sequence repetitions of length three or more in the ciphertext and measure the distances between the repetitions. Any number that divides 90% or more of such distances is an excellent candidate for the keyword length. Between this and the IC test, the length should be fairly obvious.

Assume that we have discovered a keyword length of k. Then we can break the original ciphertext into k groups $G_1, G_2, \cdots, G_k$, where group G_i begins at position i, for $1 \leqslant i \leqslant k$, and consists of every kth letter thereafter. Each of these k groups has been enciphered by only one Vigenère alphabet and is thus a monoalphabetic substitution. It remains to discover the plain equivalent of each cipher letter for each group. But here we have some help. If we knew the cipher alphabet for any one group, we would also know that we could find the cipher alphabet for any other group by rotating the known alphabet some distance. On the other hand, it would be easier to recover the plaintext equivalents if the frequency distributions for the various groups could be combined into one glorious distribution because the more data used to build a distribution, the more secure the statistical conclusions drawn from it. To do the combination, we will need to know the relative rotations between the alphabets used for each group.

The relative rotations are discovered by a variation of the index of coincidence. For each group G_i, build a frequency distribution and arrange it in *alphabetical* order of ciphertext letters. Table 24-1 shows the distributions for the message of Figure 24-1, assuming that k = 7. Now if $f_{i,\alpha}$ is the frequency of letter α in alphabet i, we define

$$R_{i,j,r} = \sum_{\beta=1}^{26} f_{i,\beta} f_{j,\beta+r}$$

where we assume that if β+r is greater than 26, we rotate back around to the front of the alphabet. The larger $R_{i,j,r}$ is, the greater the chance that the alphabet for group j is r places *down* the Vigenère square from the alphabet for group i. After calculating all the values of $R_{i,j,r}$ (it is not necessary ever to have $j \leqslant i$ for reasons of symmetry), pick out the i and j that give the largest value of $R_{i,j,r}$. Probably group j is shifted r places with respect to group i.

Now form a new supergroup G_{ij} from groups G_i and G_j by setting the frequency $f_{ij,\alpha}$ to $f_{i,\alpha}+f_{j,\alpha+r}$. Throw G_i and G_j out of consideration, replace them with G_{ij}, and repeat the process described in these last two paragraphs. After k-1 repetitions, the relative shifts of all the k alphabets are known. Also, the composite frequency distribution has been found. To find the plaintext equivalents of the ciphertext letters, reorder the ciphertext letters by frequency. The ciphertext letters should now be in the same order as the plaintext English letters of Figure 24-5. Reconstruction of the Vigenère square is easy, the message can now be deciphered, and the keyword can be recovered by trying all 26 patterns of letters of length k that have the spacings dictated by the alphabet rotations. It is possible that some of the low-frequency letters are misplaced, but visual inspection should correct that situation. You should reconstruct the keyword and mixed alphabet because it is common for

Table 24-1 Frequency Distributions for Figure 24-1 if k = 7

G_1		G_2		G_3		G_4		G_5		G_6		G_7	
A	11	A	7	A	0	A	3	A	1	A	8	A	1
B	4	B	5	B	10	B	0	B	1	B	2	B	2
C	1	C	2	C	7	C	14	C	0	C	3	C	0
D	1	D	3	D	1	D	1	D	5	D	0	D	6
E	1	E	3	E	9	E	2	E	6	E	5	E	1
F	0	F	0	F	0	F	1	F	1	F	6	F	3
G	0	G	0	G	6	G	0	G	10	G	2	G	7
H	4	H	1	H	2	H	3	H	0	H	6	H	5
I	1	I	6	I	0	I	4	I	0	I	0	I	2
J	0	J	17	J	1	J	5	J	1	J	4	J	4
K	6	K	1	K	3	K	7	K	3	K	6	K	0
L	9	L	2	L	3	L	2	L	6	L	0	L	0
M	10	M	3	M	0	M	1	M	8	M	0	M	5
N	5	N	1	N	0	N	1	N	2	N	3	N	1
O	1	O	0	O	1	O	12	O	8	O	0	O	17
P	2	P	6	P	1	P	2	P	13	P	3	P	5
Q	0	Q	0	Q	4	Q	3	Q	3	Q	4	Q	8
R	2	R	0	R	9	R	5	R	0	R	9	R	8
S	1	S	4	S	9	S	0	S	5	S	6	S	0
T	9	T	9	T	2	T	10	T	1	T	0	T	5
U	0	U	0	U	6	U	0	U	0	U	8	U	1
V	3	V	11	V	2	V	0	V	2	V	6	V	0
W	2	W	0	W	1	W	0	W	2	W	0	W	0
X	0	X	7	X	2	X	5	X	6	X	0	X	0
Y	10	Y	2	Y	5	Y	10	Y	1	Y	10	Y	4
Z	10	Z	3	Z	9	Z	2	Z	8	Z	2	Z	7

The value of $R_{1,2,0}$ is 354 and the value of $R_{3,6,12}$ is 315. To calculate $R_{3,8,12}$, the frequencies for A through N of group G_3 were multiplied by those of L through Z of group G_6 and those of O through Z from group G_3 by those of A through K of group G_6.

both to have some psychological connection with the subject of the message; recovery provides additional assurance that the solution is correct. By the way, what did Miss Hari say?

Statement of the Theme Write a program that takes as input a cipher message assumed to written by a Vigenère scheme and that prints a decryption of the message. The program should also print the Vigenère square and the keyword that it finds during the solution process. Under control of some special input, intermediate results, such as all the possible keyword lengths, the individual alphabet frequency distributions, and the values of IC, should be printed for inspection. These results will be useful during debugging and to the cryptographer when a computer-proposed solution is not quite correct. Neatness in output presentation is important so that the cryptographer's intuition will not be damaged by noxious computer artifacts.

Performance Practice The algorithms here are certainly easy to understand and implement, but they have the odd property that they do not give absolute answers. For instance, the length of the keyword is only "probable," and the decision to choose one candidate length over another must be made on the weight of the evidence. Similarly, the algorithmic determination of the plain equivalents of low-frequency ciphertext letters must be reviewed to see if legal English words appear from the ciphered message. By adding more statistical knowledge to the program, a better basis for the algorithmic decisions would be available, but the decisions would still require human validation. In addition to coding the algorithms, you will have to supply an implementation of the idea that enough evidence has been collected to justify the program in reaching a conclusion. One way, and

a good one, to provide this judgment function is to write the program on a conversational system that will allow the program and the user to discuss the merits of each decision before it is reached. Generally such a "discussion" consists of the program presenting the facts that support a possible decision and the user either ratifying or vetoing the decision before computation proceeds.

Although the algorithms are imprecise, and imprecision usually makes programmers uncomfortable, this program is easy to test. The first part of the project probably should be an encryption program that will take as input a piece of English text and that, selecting a mixed alphabet and keyword by some random means, outputs a Vigenère square, and prints the encrypted text in the standard five-character format. Blanks and punctuation should be stripped from the input automatically. As an option, the mixed alphabet and keyword should be possible inputs so that particular features of the decrypter can be tested repeatedly. Remember that a test message should be at least 30 or 40 times as long as the keyword for good statistical behavior of the algorithms.

Orchestration This problem is made for a language like SNOBOL, which combines character manipulation with some simple arithmetic capability. More algebraic languages like PL/I, PASCAL, and XPL, which allow reasonable manipulation of characters, will also be good candidates. Whatever language is chosen, try to avoid representing characters by integers; do not let the requirements of computer representation dictate an obscure solution.

Playing Time One person for 2 weeks.

REFERENCES

Gaines, Helen Fouché. *Cryptanalysis.* Dover, New York, NY, 1956.

This is the elementary book that most amateur cryptanalysts probably see first. The Dover edition is an inexpensive paperback, and the book provides detailed solution methods for reasonably complicated ciphers. It was originally written quite sometime ago; so none of the mathematical methods of Sinkov are discussed, but classical techniques are well described. There are some helpful tables.

Gardner, Martin, "Mathematical Games." *Scientific American*, August, 1977, pp. 120-124.

Gardner reports a newly discovered, practically unbreakable cipher. This cipher method uses properties of very large prime numbers and requires a computer for operations. If you do the project in Chapter 22, you will have the tools for a perfectly secure communications method.

Kahn, David. *The Code Breakers.* Macmillan, New York, NY, 1967.

Kahn wrote *the* definitive work on cryptography. Although some interesting material about World War II has surfaced since 1967, this voluminous book contains all the history and most of the methods that any amateur would want to know. The bibliography is excellent. Be careful of the paperback edition; it has been "condensed," whatever that means.

Sinkov, Abraham. *Elementary Cryptanalysis—A Mathematical Approach.* Random House, New York, NY, 1968.

An extremely simple book on cryptanalysis, it does provide some mathematical foundation. Presumably NSA has much more advanced techniques than those discussed here; naturally, however, they're not telling. All of our discussion was derived from Sinkov's materials.

Projects for Compiler Courses

The next four etudes join together to provide a complete compiler course. Any student who worked through all four would have a good grasp of both practical and theoretical problems of translator construction. If the etudes are done in the order presented, each accepts as input the output of the next in line. For example, the computer simulator and loader can be used to test the compiler. Individual students should probably not tackle the compiler or loader; the TRAC interpreter or the computer simulation might be done by one person. All four projects could be done in three quarters or two semesters with proper guidance.

25

Computer Stimulation

or...

SIMULATION OF A TYPICAL LARGE COMPUTER

If you are reading this, you almost certainly have a computer handy. It probably seems a little silly to write a program that does exactly what your computer can already do (if you own the right computer). But are you sure you know precisely what your computer can do? And will the other users allow you to usurp the machine long enough to explore all its idiosyncracies? Bill McKeeman says that you should never start a big project, such as a compiler or operating system, that depends on the computer structure without first writing a simulator. As usual, the way to learn about a topic is to teach someone else (a computer?!?) about it.

The Educational Computer, Model 1, is not a real computer; but, following a time-honored custom, it steals features from several popular machines. EC-1 is simpler than many hardware computers, but this aspect allows more attention to be paid to the structure. The description may not be as complete and detailed as might be found in a computer manual; such completeness would require more space than we have available. You will have to use what you know of other computers to fill the gaps. Throughout, numeric items will be represented in hexadecimal notation (base 16) because it fits nicely on the machine.

MEMORY AND REGISTERS

Each EC-1 is supplied with 2^{16} 8-bit characters of memory, addressed from 0 to $2^{16}-1$. Each memory position can hold any one of the 256 characters from the ASCII character set reproduced in Figure 25-1. Each block of four contiguous characters beginning with a character whose address is evenly divisible by four is a word. Words participate in a number of operations, and the character boundary immediately to the left of a word is a word boundary.

Computations are done in a set of 16 word-size general-purpose registers numbered from 0 to 15. These registers lie over the first 64 characters of memory, and any reference to a character address in the range 0 to 63 references the corresponding character in the register block instead. Some instructions require a register designator to be treated as a character address, which is done by

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	↯	↯	↯	↯	↯	↯	↯	↯	↯	LF	↯	NL	CR	↯	↯
1	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	FS	↯	↯	↯
2		!	"	#	$	%	&		(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	—
6		a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	≡
8	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯
9	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯
A	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯
B	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯
C	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯
D	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯
E	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯
F	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯	↯

Figure 25-1. The ASCII Character Set. Characters marked ↯ are not used on the EC-1. Character NUL is to be ignored; NL ends a record; LF causes a line feed; CR causes a carriage return; FS causes a page eject.

multiplying the designator by four. We note here that bits in a word, character, or what have you will always be numbered from zero on the left.

Two other registers are available. The Instruction Location Counter (ILC) always points to the next instruction to be executed in normal sequence. The Condition Code Register (CCR) is four bits wide. The CCR is generally set as a side effect of instruction execution and may be tested by branch instructions. The four bits are named, from left to right, the overflow bit, the greater than bit, the less than bit, and the equals bit. When the CCR is set by an instruction, it is first cleared entirely to zero and then affected bits are set to one. An overflow causes only the overflow bit to be set. Testing the CCR does not affect its value.

HARDWARE DATA TYPES

Characters have been mentioned above. They are sometimes regarded as positive 8-bit integers. Words may contain 32-bit two's complement integer values. Bit 0 of a word is the sign position and is zero for positive values and one for negative values (this is a function of two's complement notation). When shorter signed integers, such as the immediate operands discussed below, are combined with words, the shorter value has its sign bit propagated leftward to fill the missing bits.

Real numbers also occupy a word. Bit 0 is the sign bit, bits 1 through 7 constitute the exponent, and bits 8 through 31 the fraction. In a positive real number, the sign bit is zero, the exponent field

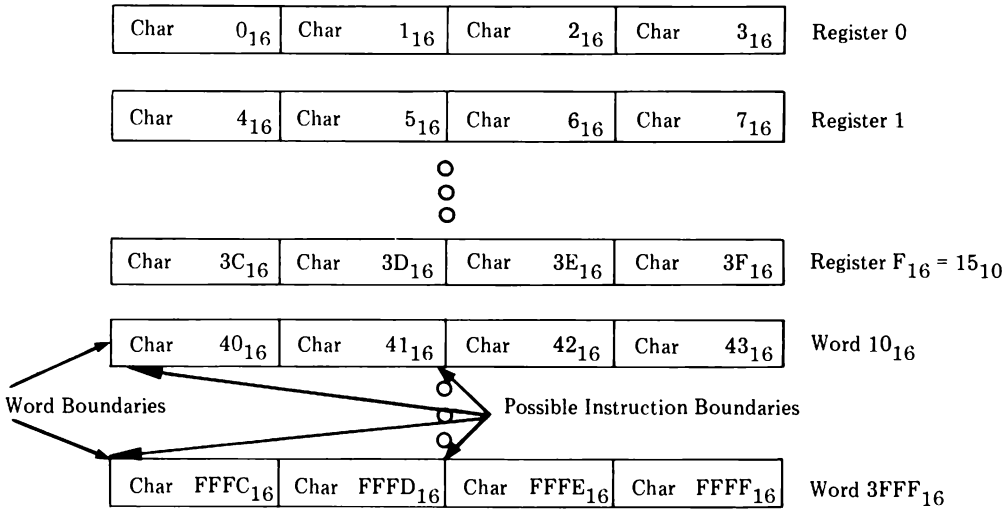


Figure 25-2. Layout of Main Memory. Notice that registers are addressable as memory.

contains an excess 40_{16} exponent of 16, and the fraction contains a 24-bit normalized hexadecimal fraction with an assumed hexadecimal point on its left.[1] Normalization of the hexadecimal fraction requires that at least the leftmost hexadecimal digit be nonzero if any are. If the fraction becomes zero, the entire number is set to zero. Any final result of a real arithmetic operation that cannot be expressed because of limits on exponents causes a real format exception. Negative real numbers are the two's complements of the corresponding positive values. Special short real values are used in real immediate instructions and have their rightmost three fraction digits dropped.

INSTRUCTION FORMATS

Instructions occur in short two-character format and long four-character format. All instructions must begin on even-character boundaries; failure of the ILC to contain an even address at the beginning of an instruction execution cycle causes an illegal instruction address exception. The first character of every instruction contains the indirect bit in bit 0 and the operation code (the opcode) in bits 1 through 7. Not all opcodes are meaningful and not all instructions make use of the indirect bit. An illegal opcode causes an unimplemented instruction exception. In most instructions, bits 8 through 11 designate either a general register or a 4-bit literal value used as a mask, and bits 12 through 15 designate a second general register.

Basically, there are four kinds of instructions: register-to-register (the two-character instructions), register-and-storage, immediate, and character. Each class has its own characteristic interpretation and addressing algorithm detailed here.

1. Register-to-register. In all register-to-register instructions, bits 12 through 15 designate a register used as one operand of the instruction. If the indirect bit is on, the operand is located at the address given by bits 16 through 31 of the register designated by bits 12 through 15 of the instruction. The value in bits 8 through 11 may designate either a register or a mask. Instructions CCS and MCS do not make use of the indirect bit.

2. Register-and-Storage. Register-and-storage instructions usually use bits 8 through 11 to designate a register or form a 4-bit mask to be used as one operand. The rest of the instruction is used to form an effective address with this algorithm:

[1] Excess 40_{16} notation means that the true exponent is found by subtracting 40_{16} from the recorded exponent.

If the indirect bit is 0 and the index register designator (bits 12 through 15) is 0, the effective address is given by the address field (bits 16 through 31) of the instruction.

If the indirect bit is zero and the index register designator is nonzero, the address field is extended to the left with zeros and added (two's complement, of course) to the value in the index register. Bits 16 through 31 of the result form the affected address. The value in the index register is not changed.

If the indirect bit is nonzero and the index register designator is zero, the address field names a two-character indirect field in memory. The contents of the indirect field form the effective address. If the indirect field does not begin on an even-character boundary, an indirect address exception occurs.

If both the indirect bit and the index register designator are nonzero, the indirect field is added to the index register value and the rightmost 16 bits of the sum form the effective address. An indirect address exception may occur.

3. Immediate. All immediate instructions use bits 8 through 11 to designate a target register and bits 12 through 31 to hold an immediate operand. The immediate operand may be a 20-bit two's complement integer, a 20-bit logical vector, or a short-format real number. The indirect bit is ignored by immediate instructions.

4. Character. Character instructions operate the same way as register-and-storage instructions.

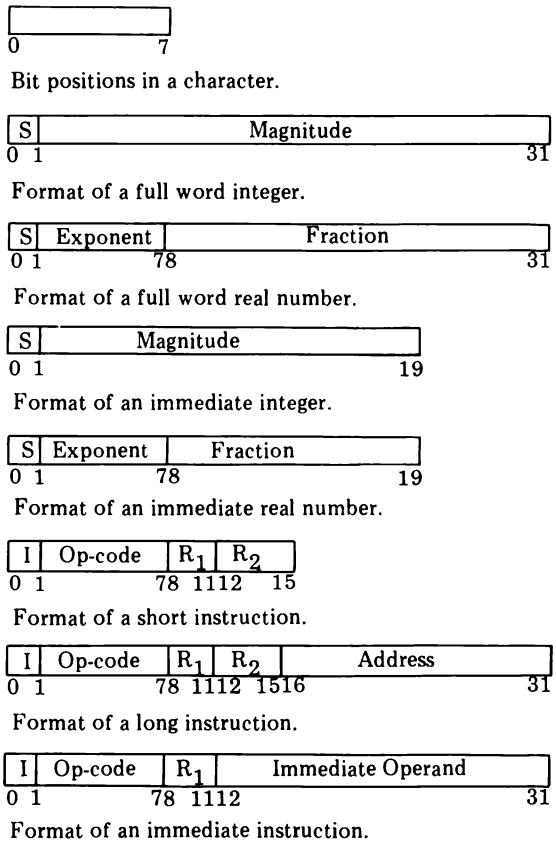


Figure 25-3. Format for Hardware Data Items

INSTRUCTION DESCRIPTIONS

In this section we describe each of the instructions. The first line of a description is a short summary, giving the instruction name, its format (RR, RS, IM, CH), its opcode in hexadecimal, its assembly language format,[2] and the possible condition code bits affected. A word description follows the summary. The code for the CCR effect is O, L, G, E, and None (indicating that the CCR is unaffected).

Load Register RR 00_{16} LR,R1 R2 GLE
The register R1 is loaded with the word at the effective address. The load value is compared with zero and the G, L, or E bit of the CCR set as appropriate.[3] If the effective address does not fall on a word boundary, a word-addressing exception occurs.

Load RS 20_{16} L,R1 A,R2 GLE
This instruction operates in the same way as the Load Register instruction except that the effective address is calculated by using the register-and-storage addressing algorithm.

Load Immediate IM 40_{16} LI,R1 I GLE
This instruction operates like the Load Register instruction except that the loaded value is the immediate operand I with its sign bit extended left 12 bits. No exceptions can occur.

Load Character CH 60_{16} LC,R1 A,R2 GE
Register R1 is cleared to zero, and the character at the effective address is loaded into bits 24 through 31. The loaded value is compared to zero and either the G or E bit of the CCR set.

Load Negative Register RR 01_{16} LNR,R1 R2 OGLE
The register R1 is loaded with the two's complement of the word at the effective address. The loaded result is compared to zero to set the CCR. If overflow occurs, only the O bit of the CCR is set. A word-addressing exception may occur.

Load Negative RS 21_{16} LN,R1 A,R2 OGLE
This instruction operates in the same way as Load Negative Register except that the effective address is calculated by the register-and-storage addressing algorithm.

Load Negative Immediate IM 41_{16} LNI,R1 I GLE
The value loaded into register R1 is the 32-bit two's complement of the 20-bit two's complement value I. Overflow cannot occur. The CCR is set by comparing the loaded value with zero.

Load Negative Character CH 61_{16} LNC,R1 A,R2 LE
The character at the effective address is extended leftward 24 bits with zeros and the resulting word complemented and loaded into register R1. Overflow cannot occur. The loaded value is compared with zero to set the CCR.

Store Register RR 02_{16} STR,R1 R2 GLE
The value in R1 is stored in the word at the effective address. The stored value is compared to zero to set the CCR. A word-addressing exception may occur.

Store RS 22_{16} ST,R1 A,R2 GLE
This instruction operates in the same way as the Store Register instruction with the effective address calculated by the register-and-storage addressing algorithm.

[2] In assembly language, the indirect bit is set by writing an asterisk before the address field, as in

LN,R1 *A,R2

[3] In a comparison to set the CCR, the final result is assumed to hold if the first operand mentioned is on the left of the relation and the second on the right. That is, if the result is less than, it means that the first operand is less than the second.

Store Character CH 62_{16} STC,R1 A,R2 GE
Bits 24 through 31 are stored in the character at the effective address. The stored value is compared to zero to set the CCR.

Swap Register RR 03_{16} SWAPR,R1 R2 GLE
The word in register R1 is exchanged with the word at the effective address. The CCR is set by comparing the value moved into register R1 with zero. A word-addressing exception may occur.

Swap RS 23_{16} SWAP,R1 A,R2 GLE
This instruction operates in the same way as the Swap Register instruction with the effective address calculated by the register-and-storage algorithm.

Swap Character CH 63_{16} SWAPC,R1 A,R2 GE
Bits 24 through 31 are exchanged with the character at the effective address. The CCR is set by comparing the character loaded into the register with zero. Bits 0 through 23 of register R1 are not affected.

And Register RR 04_{16} ANDR,R1 R2 GLE
The logical and of the word in R1 and the word at the effective address is formed and loaded into register R1. Bit G of the CCR is set if the final value in R1 is all ones, bit L is set if the result if mixed zeros and ones, and bit E is set if the result is all zeros. A word-addressing exception may occur.

And RS 24_{16} AND,R1 A,R2 GLE
This instruction operates like the And Register except that the register-and-storage addressing algorithm is used to calculate the effective address.

And Immediate IM 44_{16} ANDI,R1 I LE
The logical and of the word in register R1 and the 20-bit immediate value I extended on the left with 12 zero bits is stored in R1. The CCR is set in the same way as the And Register instruction.

And Character CH 64_{16} ANDC,R1 A,R2 GLE
The character at the effective address is anded with bits 24 through 31 of register R1 and the result is replaced in bits 24 through 31 of R1. Bits 0 through 23 of R1 are not affected. The CCR is set in the same way as the And Register instruction.

Or Register RR 05_{16} ORR,R1 R2 GLE
This instruction operates in the same way as the And Register with logical or replacing logical and.

Or RS 25_{16} OR,R1 A,R2 GLE
This instruction operates in the same way as And with logical or replacing logical and.

Or Immediate IM 45_{16} ORI,R1 I GLE
This instruction operates in the same way as And Immediate with logical and replaced by logical or.

Or Character CH 65_{16} ORC,R1 A,R2 GLE
This instruction operates in the same way as And Character with logical and replaced by logical or.

Exclusive Or Register RR 06_{16} XORR,R1 R2 GLE
This instruction operates in the same way as And Register with logical and replaced by logical exclusive or.

Exclusive Or RS 26_{16} XOR,R1 A,R2 GLE
This instruction operates in the same way as And with logical and replaced by logical exclusive or.

Exclusive Or Immediate IM 46_{16} XORI,R1 I GLE
This instruction operates in the same way as And Immediate with logical and replaced by logical exclusive or.

Exclusive Or Character CH 66_{16} XORC,R1 A,R2 GLE
This instruction operates in the same as And Character with logical and replaced by logical exclusive or.

Not Register RR 07_{16} NOTR,R1 R2 GLE
This instruction operates in the same way as And Register with logical and replaced by logical complement of the second operand, the original value in register R1 being ignored.

Not RS 27_{16} NOT,R1 A,R2 GLE
This instruction operates in the same way as And with logical and replaced by logical complement of the second operand, the original value in register R1 being ignored.

Not Immediate IM 47_{16} NOTI,R1 I GLE
This instruction operates in the same way as And Immediate with logical and replaced by logical complement of the extended immediate value, the original value in register R1 being ignored.

Not Character CH 67_{16} NOTC,R1 A,R2 GLE
This instruction operates in the same way as And Character with logical and replaced by logical complement of the second operand, the original value of bits 24 through 31 of register R1 being ignored.

Branch Conditions Set Register RR 08_{16} BCSR,M1 R2 None
If the logical and of the contents of the CCR and the 4-bit logical mask M1 is nonzero, the contents of the ILC are replaced by the effective address.

Branch Conditions Set RS 28_{16} BCS,M1 A,R2 None
This instruction operates in the same way as Branch Conditions Set Register with the effective address calculated by the register-and-storage addressing algorithm.

Branch Conditions Reset Register RR 09_{16} BCRR,M1 R2 None
If the logical and of the contents of the CCR and the 4-bit logical mask M1 is zero, the contents of the ILC are replaced by the effective address.

Branch Condition Reset RS 29_{16} BCR,M1 A,R2 None
This instruction operates in the same way as Branch Conditions Reset Register with the effective address calculated by the register-and-storage addressing algorithm.

Branch and Link Register RR $0A_{16}$ BALR,R1 R2 None
The current contents of the ILC are loaded into register R1 and the effective address is loaded into the ILC. If the indirect bit is not on, the effective address is register designator R2 multiplied by 4.

Branch and Link RS $2A_{16}$ BAL,R1 A,R2 None
The current contents of the ILC are stored in register R1 and the ILC is loaded with the effective address of the instruction.

Save Condition Register RR $0B_{16}$ SACR,M1 R2 None
If the logical and of the CCR and the 4-bit mask field M1 is nonzero, a word of all one bits is stored in the effective address; otherwise a word of all zeros is stored. A word-addressing exception may occur.

Save Condition RS $2B_{16}$ SAC,M1 A,R2 None
This instruction operates in the same way as the Store Conditions Register instruction with the effective address calculated by the register-and-storage addressing algorithm.

Save Condition Character CH $6B_{16}$ SACC,M1 A,R2 None
If the logical and of the CCR and the 4 bit mask field M1 is nonzero, a character of all one bits is stored at the effective address; otherwise a character of all zero bits is stored.

Compare Register RR $0C_{16}$ CR,R1 R2 GLE
The results of an algebraic comparison between the contents of register R1 and the word at the effective address are used to set the G, L, or E bits of the CCR as appropriate. A word-addressing exception may occur.

Compare RS $2C_{16}$ C,R1 A,R2 GLE
This instruction operates the same way as Compare Register except that the effective address is calculated by the register-and-storage addressing algorithm.

Compare Immediate IM $4C_{16}$ CI,R1, I GLE
The 32-bit value in register R1 is compared algebraically with the 32-bit value constructed by propagating the immediate operand's sign bit leftward 12 bits, and the result is used to set the G, L, or E bit of the CCR as appropriate.

Compare Character CH $6C_{16}$ CC,R1 A,R2 GLE
Bits 24 through 31 of register R1 are compared as an 8-bit positive integer with the character at the effective address, and the result is used to set the G, L, or E bit of the CCR as appropriate.

Compare Character String RR $0E_{16}$ CCS,M1 R2 GLE
Register designator R2 names a register pair R2 and (R2+1) mod 16 (the second register will be called R2+1 throughout). The pair R2 and R2+1 should contain a string descriptor doubleword, with a character address A1 in bits 16 through 31 of register R2, a length L in bits 0 through 15 of register R2+1, and a character address A2 in bits 16 through 31 of register R2+1. To begin execution, A1, A2, and L are moved to internal registers, the CCR is set to zero, and the E bit of the CCR is set to one. A loop is started.

First, if L is zero, bits 0 through 15 of both registers are set to zero, bits 16 to 31 of R2 are set to the internal value of A1, bits 16 through 31 of R2+1 are set to the internal value of A2, and the instruction terminates.

Second, the character of A1 is compared as an 8-bit integer to the character at A2 and the result used to set the appropriate bits of the CCR.

Third, if the E bit of the CCR is not one, bits 0 through 15 of register R2 are set to zero, bits 16 through 31 of R2 to the internal value of A1, bits 0 through 15 of R2+1 to the internal value of L, bits 16 through 31 of R2+1 to the internal value of A2, and the instruction terminates.

Finally, L is decremented by 1, A1 is incremented by the mask M1 interpreted as a 4-bit two's complement integer, and A2 is incremented by 1, and the loop returns to the first step.

Move Character String RR $0F_{16}$ MCS,M1 R2 None
Registers R2 and (R2+1) mod 16 contain a string descriptor doubleword as described in Compare Character String. The L, A1, and A2 fields are loaded into internal registers. A loop is begun.

First, if L is zero, bits 0 through 15 of registers R2 and R2+1 are set to zero, bits 16 through 31 of R2 to A1, bits 16 through 31 of R2+1 to A2, and the instruction terminates.

Second, the character at location A1 is stored at character location A2.

Third, L is decremented by 1 and A2 is incremented by 1.

Finally, A1 is incremented by the mask M1 interpreted as a 4-bit two's complement integer and the loop returns to its first step.

Supervisor Call RS $2E_{16}$ SVC,R1 A,R2 None
Program execution is interrupted and a call made to a controlling supervisor program.

Execute RS $2F_{16}$ EX,R1 A,R2 None
The instruction at the effective address is executed. The effects of the subject instruction become the effects of the Execute instruction. If the effective address is not even, an execute address exception occurs. Execute instructions may be nested to any depth. Note that the ILC is changed only if explicitly modified by the subject instruction.

Load Address RS $4E_{16}$ LA,R1 A,R2 None
Register R1 is loaded with the instruction's effective address.

Load Multiple RS $6E_{16}$ LM,R1 A,R2 None
Registers R1 through R2 are loaded from consecutive words in memory, beginning at the effective address (the effective address is calculated by assuming that the index register designator is zero). If R2 is less than R1, registers R1 through 15 and 0 through R2 are loaded. A word-addressing exception may occur.

Store Multiple RS $6F_{16}$ STM,R1 A,R2 None
Registers R1 through R2 are stored into consecutive words of memory, beginning at the effective address (the effective address is calculated by assuming the index register designator is zero). If R2 is less than R1, registers R1 through 15 and 0 through R2 are stored. A word-addressing exception may occur.

Add Register RR 10_{16} AR,R1 R2 OGLE
The word in R1 is added to the word at the effective address and the result is placed in R1. The sum is compared to zero to set the CCR. If overflow occurs, only the O bit of the CCR is set. A word-addressing exception may occur.

Add RS 30_{16} A,R1 A,R2 OGLE
This instruction operates in the same way as Add Register with the effective address calculated by the register-and-storage addressing algorithm.

Add Immediate IM 50_{16} AI,R1 I OGLE
The 20-bit two's complement immediate operand I is added to the value in register R1 and the sum stored in R1. The sum is compared to zero to set the CCR. If overflow occurs, only the O bit of the CCR is set.

Add Character CH 70_{16} AC,R1 A,R2 OGLE
The character at the effective address is extended 24 bits to the left with zeros and added to the value in register R1 with the result loaded into R1. The sum is compared to zero to set the CCR. If overflow occurs, only the O bit of the CCR is set.

Subtract Register RR 11_{16} SR,R1 R2 OGLE
The word at the effective address (the subtrahend) is subtracted from the value in register R1 (the minuend) and the difference is stored in R1. The difference is compared to zero to set the CCR. If overflow occurs, only the O bit of the CCR is set. A word-addressing exception may occur.

Subtract RS 31_{16} S,R1 A,R2 OGLE
This instruction operates the same way as Subtract Register with the effective address calculated by the register-and-storage addressing algorithm.

Subtract Immediate IM 51_{16} SI,R1 I OGLE
The 20-bit two's complement integer immediate operand I (the subtrahend) is subtracted from the value in register R1 (the minuend) and the result stored in register R1. The dif-

ference is compared to zero to set the CCR. If overflow occurs, only the O bit of the CCR is set.

Subtract Character CH 71_{16} SC,R1 A,R2 OGLE
The character at the effective address (the subtrahend), treated as a positive integer by extension 24 bits leftward with zeros, is subtracted from the value in register R1 (the minuend) and the result stored in R1. The difference is compared to zero to set the CCR. If overflow occurs, only the O bit of the CCR is set.

Reverse Subtract Register RR 12_{16} RSR,R1 R2 OGLE
This instruction operates the same way as the Subtract Register instruction except that the roles of the minuend and the subtrahend are reversed.[4]

Reverse Subtract RS 32_{16} RS,R1 A,R2 OGLE
This instruction operates the same way as Subtract except that the roles of the minuend and the subtrahend are reversed.

Reverse Subtract Immediate IM 52_{16} RSI,R1 I OGLE
This instruction operates the same way as Subtract Immediate except that the roles of the minuend and the subtrahend are reversed.

Reverse Subtract Character CH 72_{16} RSC,R1 A,R2 OGLE
This instruction operates the same way as the Subtract Character with the roles of the minuend and the subtrahend reversed.

Multiply Register RR 13_{16} MR,R1 R2 OGLE
The value in register R1 and the word at the effective address are multiplied and the low-order 32 bits of the product are stored in register R1. The result in register R1 is compared to zero to set the CCR. If overflow occurs, only the O bit of the CCR is set. A word-addressing exception may occur.

Multiply RS 33_{16} M,R1 A,R2 OGLE
This instruction operates the same way as Multiply Register except that the effective address is calculated by the register-and-storage addressing algorithm.

Multiply Immediate IM 53_{16} MI,R1 I OGLE
The low 32 bits of the product of the value in register R1 and the 20-bit immediate value I are stored in register R1. The product in register R1 is compared to zero to set the CCR. If overflow occurs, only the O bit of the CCR is set.

Multiply Character CH 73_{16} MC,R1 A,R2 OGLE
The low 32 bits of the product of the value in register R1 and the positive 8-bit integer in the character at the effective address are stored in register R1. The value in register R1 is compared to zero to set the CCR. If overflow occurs, only the O bit of the CCR is set.

Divide Register RR 14_{16} DR,R1 R2 OGLE
The value in register R1 (the dividend) is divided by the word at the effective address (the divisor) and the quotient is stored in register R1. The quotient is selected so that the remainder is nonnegative. The quotient is compared to zero to set the CCR. If overflow occurs, only the O bit of the CCR is set. A word-addressing exception may occur. If the divisor is zero, the zero divisor exception occurs and register R1 is unchanged.

Divide RS 34_{16} D,R1 A,R2 OGLE
This instruction operates the same way as Divide Register except that the effective address is calculated with the register-and-storage addressing algorithm.

Divide Immediate IM 54_{16} DI,R1 I OGLE
The value in register R1 (the dividend) is divided by the 20-bit two's complement integer

[4] In all the reversed instructions, although the roles of the two operand *values* are interchanged, the result is still stored in the same place.

immediate value I (the divisor) and the quotient is stored in register R1. The quotient is selected so that the remainder is nonnegative. The quotient is compared to zero to set the CCR. If overflow occurs, only the O bit of the CCR is set. If the divisor is zero, the zero divisor exception occurs and the register R1 is unchanged.

Divide Character CH 74_{16} DC,R1 A,R2 GLE
The value in register R1 (the dividend) is divided by the positive 8-bit integer at the effective address (the divisor) and the quotient is stored in register R1. The quotient is selected so that the remainder is nonnegative. The quotient is compared to zero to set the CCR. If the divisor is zero, the zero divisor exception occurs and register R1 is unchanged. Overflow is not possible.

Reverse Divide Register RR 15_{16} RDR,R1 R2 OGLE
This instruction operates the same way as Divide Register except that the roles of the dividend and divisor are reversed.

Reverse Divide RS 35_{16} RD,R1 A,R2 OGLE
This instruction operates the same way as Divide except that the roles of the dividend and the divisor are reversed.

Reverse Divide Immediate IM 55_{16} RDI,R1 I GLE
This instruction operates the same way as Divide Immediate except that the roles of the dividend and the divisor are reversed. Overflow is not possible.

Reverse Divide Character CH 75_{16} RDC,R1 A,R2 GLE
This instruction operates the same way as Divide Character except that the roles of the dividend and the divisor are reversed.

Remainder Register RR 16_{16} REMR,R1 R2 GE
The value in register R1 (the dividend) is divided by the word at the effective address (the divisor) and the nonnegative remainder is stored in register R1. The remainder is compared to zero to set the CCR. A word-addressing exception may occur. If the divisor is zero, the zero divisor exception occurs and register R1 is unchanged.

Remainder RS 36_{16} REM,R1 A,R2 GE
This instruction operates the same way as Remainder Register except that the effective address is calculated by the register-and-storage addressing algorithm.

Remainder Immediate IM 56_{16} REMI,R1 I GE
The value in register R1 (the dividend) is divided by the 20-bit two's complement value I (the divisor) and the nonnegative remainder is stored in register R1. The remainder is compared to zero to set the CCR. If the divisor is zero, the zero divisor exception occurs and register R1 is unchanged.

Remainder Character CH 76_{16} REMC,R1 A,R2 GE
The value in register R1 (the dividend) is divided by the 8-bit positive integer (the divisor) at the effective address and the nonnegative remainder is stored in register R1. The remainder is compared to zero to set the CCR. If the divisor is zero, the zero divisor exception occurs and register R1 is unchanged.

Reverse Remainder Register RR 07_{16} RREMR,R1 R2 GE
This instruction operates the same way as Remainder Register except that the roles of dividend and divisor are reversed.

Reverse Remainder RS 37_{16} RREM,R1 A,R2 GE
This instruction operates the same way as Remainder except that the roles of dividend and divisor are reversed.

Reverse Remainder Immediate IM 57_{16} RREMI,R1 I GE
This instruction is the same as Remainder Immediate except that the roles of dividend and divisor are reversed.

Reverse Remainder Character CH 77_{16} RREMC,R1 A,R2 GE
This instruction is the same as Remainder Character except that the roles of dividend and divisor are reversed.

Real Add Register RR 18_{16} FAR,R1 R2 GLE
The value in register R1 is added to the real number at the effective address and the sum is stored in register R1. The sum is compared to zero to set the CCR. Both word-addressing and real-format exceptions may occur.[5]

Real Add RS 38_{16} FA,R1 A,R2 GLE
This instruction is the same as Real Add Register except that the effective address is calculated by the register-and-storage addressing algorithm.

Real Add Immediate IM 58_{16} FAI,R1 I GLE
The sum of the value in register R1 and the real short format immediate operand I is stored in register R1. A real format exception may occur.

Real Subtract Register RR 19_{16} FSR,R1 R2 GLE
The real number at the effective address (the subtrahend) is subtracted from the value in the register R1 (the minuend) and the difference is stored in register R1. The difference is compared to zero to set the CCR. Both word-addressing and real format exceptions may occur.

Real Subtract RS 39_{16} FS,R1 A,R2 GLE
This instruction is the same as Real Subtract Register except that the effective address is calculated by the register-and-storage addressing algorithm.

Real Subtract Immediate IM 59_{16} FSI,RI I GLE
The short format real immediate operand I (the subtrahend) is subtracted from the value in register R1 (the minuend) and the difference is stored in register R1. The difference is compared to zero to set the CCR. A real-format exception can occur.

Reverse Real Subtract Register RR $1A_{16}$ RFSR,R1 R2 GLE
This instruction is the same as Real Subtract Register with the roles of the minuend and subtrahend reversed.

Reverse Real Subtract RS $3A_{16}$ RFS,R1 A,R2 GLE
This instruction is the same as Real Subtract with the roles of the minuend and subtrahend reversed.

Reverse Real Subtract Immediate IM $5A_{16}$ RFSI,R1 I GLE
This instruction is the same as Real Subtract Immediate with the roles of the minuend and the subtrahend reversed.

Real Multiply Register RR $1B_{16}$ FMR,R1, R2 GLE
The value in register R1 and the real number at the effective address are multiplied and the product is stored in register R1. The product is compared to zero to set the CCR. Both word-addressing and real format exceptions may occur.

Real Multiply RS $3B_{16}$ FM,R1 A,R2 GLE
This instruction is the same as Real Multiply Register except that the effective address is calculated by the register-and-storage addressing routine.

Real Multiply Immediate IM $5B_{16}$ FMI,R1 I GLE
The value in register R1 is multiplied by the real short format immediate value I and the product is stored in register R1. The product is compared to zero to set the CCR. A real format exception may occur.

[5] The mnemonics for the real arithmetic instructions are prefixed with the letter "F" because the historical name for real-number implementations is "floating point." This name also gives rise to the the FLOATR, FLOAT, and FLOATI mnemonic opcodes.

Real Divide Register RR $1C_{16}$ FDR,R1 R2 GLE
The value in register R1 (the dividend) is divided by the real number at the effective address (the divisor) and the quotient is stored in register R1. The quotient is compared with zero to set the CCR. Word-addressing, real format, and zero divisor exceptions may occur.

Real Divide RS $3C_{16}$ FD,R1 A,R2 GLE
This instruction is the same as Real Divide Register except that the effective address is calculated by the register-and-storage addressing algorithm.

Real Divide Immediate IM $5C_{16}$ FD1,R1 I GLE
The value in register R1 (the dividend) is divided by the real short format immediate value I (the divisor) and the result stored in register R1. The quotient is compared to zero to set the CCR. Both real format and zero divisor exceptions may occur.

Reverse Real Divide Register RR $1D_{16}$ RFDR,R1 R2 GLE
This instruction is the same as Real Divide Register with the roles of dividend and divisor reversed.

Reverse Real Divide RS $3D_{16}$ RFD,R1 A,R2 GLE
This instruction is the same as Real Divide with the roles of dividend and divisor reversed.

Reverse Real Divide Immediate IM $5D_{16}$ RFDI,R1 I GLE
This instruction is the same as Real Divide Immediate with the roles of dividend and divisor reversed.

Convert To Real Register RR $1E_{16}$ FLOATR,R1 R2 GLE
The 32-bit two's complement integer at the effective address is converted to a real number and stored in register R1. The real result is compared to zero to set the CCR. A word-addressing exception may occur.

Convert To Real RS $3E_{16}$ FLOAT,R1 A,R2 GLE
This instruction is the same as Convert To Real Register except that the effective address is calculated by the register-and-storage addressing algorithm.

Convert To Real Immediate IM $5E_{16}$ FLOATI,R1 I GLE
The 20-bit two's complement integer immediate operand I is converted to real format and stored in register R1. The result is compared to zero to set the CCR.

Convert To Integer Register RR $1F_{16}$ FIXR,R1 R2 OGLE
The integer portion of the real number at the effective address is converted to a 32-bit two's complement integer and stored in register R1. If overflow occurs, the result is zero and the O bit of the CCR is set. The result is compared to zero to set the other bits of the CCR. A word-addressing exception may occur.[6]

Convert To Integer RS $3F_{16}$ FIX,R1 A,R2 OGLE
This instruction is the same as Convert To Integer Register except that the effective address is calculated by the register-and-storage addressing algorithm.

Convert to Integer Immediate IM $5F_{16}$ FIXI,R1 I OGLE
The real short format immediate operand I is converted to a 32-bit two's complement integer and the result stored in register R1. If overflow occurs, the result is zero and the O bit of the CCR is set. The result is compared to zero to set the other CCR bits.

Real Floor RS 78_{16} FLOOR,R1 A,R2 GLE
The real format integer not greater algebraically than the real number at the effective address is stored in register R1. The result is compared to zero to set the CCR. A word-addressing exception can occur.

[6]These instructions are named FIXR, FIX, and FIXI because integer implementations have been called "fixed point" historically.

Real Ceiling RS 79_{16} CEIL,R1 A,R2 GLE
The real format integer not smaller algebraically than the real number at the effective address is stored in register R1. The result is compared to zero to set the CCR. A word-addressing exception may occur.

Minimum RS $7A_{16}$ MIN,R1 A,R2 LE
The values in register R1 and in the word at the effective address are compared and the minimum stored in register R1. The CCR is set by comparing the original register R1 value with the final one. A word-addressing exception may occur.

Maximum RS $7B_{16}$ MAX,R1 A,R2 GE
This instruction is the same as Minimum except that the maximum replaces the minimum.

Shift Logical RS $7C_{16}$ SHIFTL,R1 A,R2 OGLE
The effective address is treated as a 16-bit two's complement integer called the shift count. The value in register R1 is shifted leftward by the amount of the shift count if positive and rightward if negative, the shift distance measured in bits. Bits shifted off either end of the register are lost. If a 1 bit is lost, the O bit of the CCR is set. The result is compared to zero to set the other CCR bits.[7]

Shift Circular RS $7D_{16}$ SHIFTC,R1 A,R2 GLE
This instruction works the same way as Shift Logical except that bits shifted off one end of the register fill vacated positions on the other. Overflow is not possible.

Shift Arithmetic RS $7E_{16}$ SHIFTA,R1 A,R2 OGLE
This instruction works like Shift Logical on left shifts and propagates bit 0 rightward on right shifts. Overflow occurs only on left shifts when a bit shifted into the sign bit differs from one shifted out.

Shift Real RS $7F_{16}$ SHIFTR,R1 A,R2 GLE
The effective address is interpreted as a 16-bit two's complement shift count. The fraction part of the absolute value of the real number in register R1 is shifted left or right in 4-bit units logically, vacated 4-bit positions being filled with hexadecimal zeros. If the resulting fraction is zero, so is the result. Otherwise the shift count is subtracted from the exponent and the resulting value stored with the original sign in register R1. Overflow cannot occur, but a real format exception may. The result is compared to zero to set the CCR.

EXCEPTIONS AND SUPERVISOR CALLS

The input/output structure on modern computers is at least as complicated as the CPU. To avoid doubling the size of the problem, we assume that a supervisor monitors the progress of every user program. The supervisor can be invoked directly by the Supervisor Call instruction and indirectly by an exceptional occurrence. The Supervisor Call instruction uses its various fields to code the function desired and to supply parameters. The following constitute a bare minimum of functions with the R1 register designator selecting the function.

R1 = 0 Exit the running program and clean up after it.

R1 = 1 Read an integer from the input stream and store it at the effective address of the SVC (the address must name a word).

R1 = 2 Read a real number and store it at the effective address.

R1 = 3 Read a character and store it at the effective address.

[7] A shift count with absolute value greater than 32 causes the same effect as some count with absolute value 32 or less. The smaller count can replace the larger when any shift instruction is executed.

Table 25-1 A Summary of Operation Codes

	0	1	2	3	4	5	6	7
0	LR	AR	L	A	LI	AI	LC	AC
1	LNR	SR	LN	S	LNI	SI	LNC	SC
2	STR	RSR	ST	RS	-	RSI	STC	RSC
3	SWAPR	MR	SWAP	M	-	MI	SWAPC	MC
4	ANDR	DR	AND	D	ANDI	DI	ANDC	DC
5	ORR	RDR	OR	RD	ORI	RDI	ORC	RDC
6	XORR	REMR	XOR	REM	XORI	REMI	XORC	REMC
7	NOTR	RREMR	NOT	RREM	NOTI	RREMI	NOTC	RREMC
8	BCSR	FAR	BCS	FA	-	FAI	-	FLOOR
9	BCRR	FSR	BCR	FS	-	FSI	-	CEIL
A	BALR	RFSR	BAL	RFS	-	RFSI	-	MIN
B	SACR	FMR	SAC	FM	-	FMI	SACC	MAX
C	CR	FDR	C	FD	CI	FDI	CC	SHIFTL
D	-	RFDR	-	RFD	-	RFDI	-	SHIFTC
E	CCS	FLOATR	-	FLOAT	LA	FLOATI	LM	SHIFTA
F	MCS	FIXR	EX	FIX	-	FIXI	STM	SHIFTR

Row titles give low 4 bits of operation code; column titles give the high 3 bits.

R1 = 4 Cause the input stream to space ahead to a new record.

R1 = 5 Write the word at the effective address as an integer on the output stream.

R1 = 6 Write the word at the effective address on the output stream as a real number.

R1 = 7 Write the character at the effective address to the output stream.

R1 = 8 Write an end of record on the output stream.

R1 = 9 and R2 = 0 End tracing instruction execution.

R1 = 9 and R2 = 1 Begin tracing instruction execution. Print a running record of each instruction executed.

R1 = A The effective address of the SVC must be a word address. The low halfword gives the low address and the high halfword the high address of a section of memory to dump. The dump should display memory between the limits in both hexadecimal and character-string format. You may find it useful to display instruction mnemonics also. The dump routine should notice and not print duplicated lines.

R1 = F This supervisor call will never be assigned for system use and can be used for any purpose by the simulator.

It is assumed that integers and real numbers on the input/output streams are terminated by blanks.

Exceptions occur when errors arise during the course of instruction execution. The program is interrupted and the supervisor notified of the cause of the exception and the location of the offending instruction. A summary of exceptions follows.

Illegal Instruction Address. At the start of an instruction execution cycle, the ILC does not contain an even value.

Unimplemented Instruction. There is no operation defined for this operation code.

Indirect Address. The indirect address is not even.

Word Addressing. The address of a purported word operand to an instruction is not divisible by four.

Real Format. The result of some real-valued operation cannot be expressed within the format for normalized real numbers.

Execute Address. The effective address of an Execute instruction is not even.

Zero Divisor. The divisor in a division or remainder operation is zero.

Wraparound Instruction. A four-character instruction begins at FFFE.

The response of the supervisor to an exception is left to the implementor but should include a report to the user of the occurrence.

ABSOLUTE LOAD FILES

The absolute load file describes the contents of EC-1 memory prior to execution. Normally such files are produced by the EC Loader from relocatable load language, and in a practical system the files would be in some binary format to save space; for this problem we describe a format that can be keypunched to aid your debugging. The records of the physical file are each 80 *external* characters long, and the legal characters are the digits, the letters *A*, *B*, *C*, *D*, *E*, *F*, *N*, and blank. Most of the time these external characters will be strung together to form hexadecimal numbers. Notice that it takes *two* external characters to form one two-digit hexadecimal number that, in turn, specifies the data necessary to fill one EC-1 *internal* character position.[8]

Each record except the last has a standard format. Character 1 is a checksum of all the other hexadecimal digits formed by adding and ignoring the carryout. Characters 2 through 4 are a hexadecimal sequence number and the first record is sequenced *000*; out-of-sequence records should be flagged as nonfatal errors. After this prefix, the rest of the record consists of count-address-data triples. The count field is one digit long and tells how many character positions in memory are to be filled by the following data. The address field is four digits long and gives the hexadecimal start address for the data in EC-1 memory. Finally, the data field contains two digits for each memory character to be filled, and each digit pair is read as a hexadecimal integer specifying eight bits of data to enter memory. There may be several such triples on one record, but no triple may cross a record boundary. The first blank occurring in a count field terminates the useful data on a record, and the rest of the record may be used for comments if desired. The last record has the characters *END* in characters 1 through 3 and a four-digit hexadecimal program start address in characters 4 through 7. As an example, the record

E10241A2301020304207FF1BEC

has a checksum of *E*, is sequenced *102*, and puts the (rather meaningless) four characters *01020304* of data at *1A23* and the two characters *1BEC* at *07FF*. Notice that it takes *eight* hexadecimal digits to specify *four* characters of internal memory.

Statement of the Theme Write a simulator for the EC-1 computer. Input for the simulator should be an absolute load file and the input stream for the simulated program. The basic output should be the output stream from the program. In addition to the simulator, write at least two programs for the EC-1 to test the simulator's correctness. Of course, you will have to hand-assemble these programs into absolute load file format.

Besides the basic output, your simulator should be able to trace and dump the simulated program.

[8] This multiple use of the word "character" will crop up again in the discussion of the EC Loader. Try to keep clear the distinction between character positions in memory and characters on input and output files. An internal character can always hold enough data to represent one external character, but an external character may not code enough data to fill one internal character position.

The trace should show the instruction broken down, the effective address calculation, and the operands and results in both their natural and hexadecimal formats. A dump should include the memory printed (perhaps under option control) in hexadecimal, instruction mnemonics, integer, real, and character formats. Repetitious groups of data should be printed only once with a note of the repetition. Control of the trace and dump may go directly to the simulator, perhaps *via* a console, or may be driven by supervisor calls during execution.

Performance Practice Computer simulators are basically simple if they follow the ordinary instruction location-decoding-execution cycle. There is an obvious 128-way branch once the operation code has been found. These 128 separate instruction implementation routines can make use of a variety of common subroutines to pursue their individual tasks, the subroutines including effective address calculation, CCR setting, exception checking, result storage, and the like. A clear layout will help in the demonstration of correctness. However, efficiency in the instruction cycle is also quite important; otherwise time costs can be prohibitive. These two needs must be balanced. Other portions of the simulator need not be quite as efficient, for they will presumably be executed much less often.

Several groups who have done this project in the past report that construction of a simple assembler to generate test cases is time well spent. If the EC Loader is also one of your projects, careful construction of the assembler will allow you to use it to test the loader as well.

Orchestration Here again is a program where clarity and efficiency trade off. Consider using a higher-level language in which crucial routines can be replaced by assembly language after debugging. Because conventional subroutine linkages can be expensive, this is a chance to experiment with languages that allow small patches of assembly language to be inserted in-line. Some FORTRANs, some ALGOLs, and XPL, among others, have this feature.

Playing Time Six weeks for one person; 3 weeks for two or three people. If more than one person participates, each is responsible for one EC-1 program.

Variations on the Theme The most obvious extension is to keep a profile of the execution of the simulated program. How often is each instruction executed? How often is each internal subroutine used? What is the pattern of memory references? Of register references? And so on *ad libitum.* This information can be extremely difficult to find on a real computer but is usually a powerful aid in understanding a complex happening.

REFERENCES

Bell, C. Gordon, and Allen Newell. *Computer Structures: Readings and Examples.* McGraw-Hill, New York, NY, 1971.

Bell and Newell discuss general principles of machine architecture and illustrate with about 30 examples. Considerable space is given to the development of a notation for machine description. Many examples are drawn from the (heavily edited) papers of the original machine designers.

IBM Corporation. *IBM System/360 Principles of Operation.* IBM System Reference Library, GA22-6821-8, November 1970.

Xerox Data Systems. *Xerox Sigma 7 Computer Reference Manual.* Order #90 09 501, 1971.

The EC-1 is quite similar to both the Sigma 7 and the 360. It may be a help to compare these machines with the EC-1. Other editions of these manuals will do as well as the ones mentioned.

26

EC Loader

or...

A LINKING LOADER

Loaders are the stepchildren of most systems. Users do not like them because they are simply another unnecessary delay on the way to The Answer. Compiler writers do not like them because they are parts of the system (what do you mean they're not?). System programmers do not like them because they are just another utility. Perhaps only the business manager likes the loader because so much time (and money) is spent executing it. But you will find writing a loader for the Educational Computer, Model 1, a rewarding problem.

EC Loader is a fairly standard relocating loader designed for use with the Easy language (Chapter 27) and the EC-1 computer (Chapter 25). EC Loader concentrates on interpreting relocatable load language, handling a fairly elaborate symbol table, and detecting load-time errors. Easy language programs may include separately compiled program segments, and EC Loader supports type checking among segments and library searching to build complete programs. Other relocatable load language features make forward branch generation simple so that compilers can be one pass instead of two. Many commercial loaders provide even more elaborate services, but such services usually require input/output complexity that is inappropriate for an etude. You will probably not appreciate EC Loader completely (either its good or bad points) unless you build a compiler or read up on loaders in the references.

GENERAL PLAN OF THE LOADER

Inputs to EC Loader are a program file and a library file, each of which may consist of several modules. If it is easy on your system, either program or library might consist of more than one file. Every module in the program must be loaded first, and modules in the library file are loaded only if they satisfy a primary reference. By the end of the load exactly one start address must have been defined or there is a fatal load error. Output is an absolute load file as described in Chapter 25. EC Loader maintains both an absolute load counter (ALC) and a relocatable load counter (RLC). The ALC starts at address 40_{16} at the beginning of the load. Each time that a new module begins, the ALC is set to beyond the module high-water mark to the next work boundary and the RLC is

reset to zero.[1] After the last module is loaded, the ALC is moved up to a word boundary beyond the module high-water mark for one final time and used to define the value of the absolute external symbol *HighestLocation.*

The relocatable load language consists of a series of load commands, each eight bits long. Commands are coded with a variable-length encoding so that some parameters can be buried in the commands themselves. Other parameters may be variable length, and expressions used in command can be arbitrarily complex. All names must be declared before use; once declared, they are referenced by number rather than name to save space in the load module. Since modules are to be generated by the one-pass Easy compiler, they do not have sizes declared; EC loader must maintain a high-water marker so that the size of a module can be found. External names defined within the current module must be declared before any other load specification. After each module is finished, local symbols must be eliminated from the symbol table, global symbols retained, and update work done on the load map. Special symbols used only on the map can be defined for diagnostic purposes. EC Loader can also check types for external procedures and their arguments.

PHYSICAL RECORD LAYOUT

Records of a load module are variable-length strings of external characters; these characters are either hexadecimal digits that group together to form integer values or characters drawn from the EC-1 ASCII set that represent themselves in names.[2] The first six characters of a record always concern the physical structure of the record and are always in the same format. Character 1 is *1* on the last record of a module and *0* on all other records. Characters 2 through 4 contain a three-digit hexadecimal sequence number beginning with *000* on the first record of a module. Out-of-sequence records are a nonfatal error. The module sequence number wraps around to *000* when there are 1000 records in the module. Characters 5 and 6 give the length of the record in hexadecimal; this length includes the first six characters and thus falls between *07* and *FF.* The useful data on the record lies in a second section following the fixed prefix and consists of a series of logical load items. Logical load items may be broken freely over physical-record boundaries.

A logical load item begins with two hexadecimal digits giving the load operation and continues with more parameters as necessary. Parameters that occur as numbers are always represented by a string of hexadecimal digits as long as necessary to supply the number. Names begin with a two-hexadecimal-digit-count field and then continue with the actual characters of the name, as many as are required by the count. Thus *692C04Fool* is a Declare External Primary (operation *69*) with a short symbol number *2C* and the four-character name *Fool.* If a parameter is an expression, it will include its own operations and may be arbitrarily long. Notice that it takes two external hexadecimal digits to fill one internal memory character with data.

LOAD ITEM DESCRIPTIONS

Each separate load item is defined by an 8-bit operation code that appears in bit format as the second entry on the title line of the descriptive paragraph. Parameters follow the operation code if they exist. The names of the parameters give a general indication of their use. It is not always possible to say how long a parameter will be. Variable-length parameters will always have lengths

[1] In general, if A is the address of the beginning of the current load module, the relation

$$ALC - A = RLC$$

will always hold. That is, the ALC and the RLC always move in tandem.

[2] To understand the distinction between internal and external characters, reread the section on the absolute load file in Chapter 25.

or explicit stopping operations attached to them. Some parameters will be buried in the operation code.

DATA LOAD ITEMS

Load Absolute *0000cccc* Data

The count field *cccc* has one added to it to give the number of internal characters of data to be loaded. The rest of the item consists of the numeric data to be loaded without relocation at the current ALC. The ALC and the RLC are advanced past the loaded data.

Load Word with Relocation *00010000* Data

The load counters are advanced to an even-character boundary if they are not on one, the 8-digit load data item is loaded into the next four characters, its low two characters are relocated by the start of the module, and the load counters are advanced.

Load Expression *000110ww* Expression

The width parameter *ww* has one added to it and that many character positions are loaded from the argument expression. The expression is calculated in 32 bits and the low portion is used as necessary. Load counters are advanced past the loaded data.

Load Relative to Symbol *0001010l* Symbol Data

The location counters are advanced to the next even boundary if they are not on one. The eight digits of data are loaded. The low two characters of the data in memory are relocated by adding the value of the symbol whose number is the first argument. The symbol number occupies two digits if *l* is *0* and four digits if *l* is *1*. Relocation must be by a relocatable symbol, but the symbol value need not be defined yet.

LOAD TIME EXPRESSIONS

Expressions to be evaluated at load time are calculated in a 32-bit-wide accumulator that is always tagged with a type (either absolute or relative). The string of load items defining a load expression may be arbitrarily long and always ends with an Expression End operator. Only symbols whose values are already defined may be used in an expression and only the type combinations in Table 26-1 are legal. The last combination is legal only if both items are from the current module. The accumulator is usually initialized with an absolute zero.

Arithmetic Operation *0010lsot* Operand

If the *o* bit is zero, the operation is an add; otherwise it is a subtract. If the *s* bit is zero, the operand is a constant and the *t* bit determines if the constant is absolute (*t=0*) or relative (*t=1*); otherwise the operand is a symbol number. The *l* bit determines if the operand takes two (*l=0*) or four (*l=1*) digits. The called-for operation is performed and the accumulator type and value set as appropriate.

Table 26-1 Legal Accumulator Combinations

Accumulator Type	*Operator*	*Operand Type*	*Result Type*
Absolute	+	Absolute	Absolute
Absolute	+	Relative	Relative
Relative	+	Absolute	Relative
Absolute	-	Absolute	Absolute
Relative	-	Absolute	Relative
Relative	-	Relative	Absolute

Set Accumulator to RLC *00110000* None
The current value of the accumulator is ignored and the accumulator is set to the current value of the RLC as a relative value.

End Expression *00110001* None
The current expression ends and the value in the accumulator is its value.

DEFINED SYMBOLS

Symbols generally must be declared before use, and declaration associates an internal loader symbol number with the name. Symbol numbers can be used only once within any one module. Definitions may occur throughout the module. Symbol names may be any length from 0 to 255 characters. A symbol name consists of a two-digit length field followed by the characters in the name. At the end of the module all but external and map symbols must be removed from the loader's symbol table.

Define External Symbol *010lt000* Number Expression
The external symbol whose number is given by the first argument is given the value of the second argument. The symbol number is two digits long if *l* is *0* and four digits long otherwise. The symbol is absolute if bit *t* is *0* and is relative otherwise. The types of the defining expression, of the symbol already declared in the table, and given by parameter *t* must all be the same. Any references to the symbol should be filled in at the time of definition.

Define Map Symbol *0100t001* Name Expression
The argument expression is evaluated and assigned to the argument name for later output on the map. The type of the symbol is absolute if *t=0* and relative otherwise.

Define Forward Reference *010lt01h* Number Expression
The forward reference whose number is given by the first argument is defined by the argument expression. The length of the reference number is two digits if *l=0* and four digits otherwise. The symbol is absolute if *t=0* and relative otherwise. If *h=0*, then the forward reference is held in the symbol table; otherwise the reference is deleted after definition.

Declare External Reference *011lt00p* Number Name
The symbol with the argument name is declared with the symbol number given by argument number as a reference to an external symbol in another module. If *l=0*, the symbol number is two digits long; otherwise it is four. The type is absolute if *t=0* and relative otherwise. If *p=0*, the reference is primary and must be searched for and satisfied; otherwise the reference is secondary and need be satisfied only if the symbol is defined for another reason.

Declare Forward Reference *011lt010* Number
A forward reference with symbol number given by the argument is declared. If *t=0*, the number length is two digits; otherwise it is four. The symbol is absolute if *t=0* and relative otherwise.

Declare External Name *011lt011* Number Name
The symbol with name given by the second argument and number given by the first argument is declared as an external symbol that will be defined in this module. External declarations must be the first items in the module. The symbol number is two digits long if *l=0* and four otherwise. The type is absolute if *t=0* and relative otherwise.

Define Procedure Types *011l0100* Number Count $Type_1 \cdots Type_{Count}$
The symbol whose number is the first argument is given a chain of types whose length is given by the two-character second argument. The symbol number is two digits long if *l=0* and four otherwise. Each type is one character long. The symbol must have been de-

cleared as an external name to be defined in this section. This command and the next can be used by the Easy compiler to check that external procedures have the correct number and types of arguments. The compiler must establish a coding of possible argument types into integers that holds over all compilations.

Check Procedure Types *0111l0101* Number Count $Type_1 \cdots Type_{Count}$
The arguments to this function are the same as those to Define Procedure Types except that the symbol must be an external reference from the current module. The types mentioned here are compared with those of the referenced symbol, and if the match is not exact, a fatal load error is announced.

MISCELLANEOUS OPERATIONS

Set Start Address *10010000* Expression
The argument expression, which must be relative, is used to set the start address for the load. This operation must occur exactly once in a load. There is a fatal load error otherwise. The start address should be noted on the map by using some made-up name.

Set to Even *10100000*
Move both location counters to the next even address if they are not at one.

Set to Word *10110000*
Move both location counters to the next higher word address if they are not at one.

Set Location Counters *11000000* Expression
Set both location counters to the value of the expression, which must be relative and greater than or equal to zero. Remember that ALC-start = RLC.

Pad *11010000*
Ignore this operation. This operation may occur in expressions.

End Module *11111111*
End the current module. If this command does not occur on a physical record with a *1* in character 1, report a fatal load error.

AN EXAMPLE PROGRAM

To clarify some of these ideas, we present a sample program in EC-1 assembly language and its load module as generated by a hypothetical assembler. The program itself is unimportant, but many loader features are demonstrated. The assignment of name numbers to names would probably be more consistent if done by a real assembler. Load code is shown *without* its physical-record controls and boundaries and *with* comments tying it to the assembly language.

```
*
*    This program tests the Pythagorean relation on the values stored at
*    X, Y, and Z.  The external procedure Square is used to calculate
*    the square of a value and is entered one word past its head.  The
*    symbol Good goes on the map only.
*
          DEF     Pythagoras
          REF     Square
          MAP     Good
*
```

```
*
Add          FAR,2    1          Add register 1 to register 2 and
             BCRR,0   *15        return via register 15.
Pythagoras   L,1      X          Square X.
             BAL,15   Square+4   Into register 1.
             LR,2     1          Save it in register 2.
             L,1      Y          Do the same for Y
             BAL,15   Square+4   and add it in
             BAL,15   Add        to the sum.
             L,1      Z          Now square Z, subtract
             BAL,15   Square+4   the running sum, and test
             FSR,1    2          for zero.
             BCS,1    Good
             SVC,7    False      Print 'F' and quit.
             SVC,0    0
Good         SVC,7    True       Print 'T' and exit via
             BCRR,0   *14        register 14.
True         DSC      'T'        Define a character constant.
False        DSC      'F'
X            DSF      3.         Define a real constant.
Y            DSF      4.
Z            DSF      5.
             END      Pythagoras
```

Now the load language. All numbers are hexadecimal.

```
6B 01          0BPythagoras    The external name Pythagoras must be declared
                               first.
68 02          06Square        Routine Square is a primary reference which must
                               be loaded.
03 1821890F                    The FAR and the BCRR can be loaded without any
                               relocation since they are absolute instructions.
48 01                          The value of Pythagoras is defined by the
                               following expression.
30
31
7A 0201        01X             The symbol X is a forward reference and for
                               variety's sake we give it a four digit symbol
                               number.
15 0201        20100000        This L operation needs to be relocated by X when
                               X is defined.
14 02          2AF00004        Similarly, the BAL needs relocation by Square.
                               Note that the offset need not be zero.
01 0021                        The LR is two characters of absolute data.
7A 0202                        Symbol Y is handled like X.
15 0202        20100000        So is the second load.
14 02          0AF00004        Another BAL to Square.
```

```
10 0AF00000                    The BAL to Add can be done with a normal
                               relocation and it is just happenstance that the
                               offset is zero.
7A 0203                        Declare Z.
15 0203          20100000      Load the L operation.
14 02            0AF00004      The final call to Square+4.
01 1912                        FSR can be loaded absolutely.
7A 0204                        Declare the label Good.
15 0204          28100000      Load the branch on zero.
7A 0205                        Form a declaration for False.
15 0205          2E700000      A SVC to write False.
03 2E000000                    The exit SVC is absolute.
49 04Good                      Remember that Good is a map symbol.
30                             The expression is just the relocation counter.
31
5A 0204                        Forward reference 0204 is also defined as Good.
30
31
7A 0206                        True is the last forward reference necessary.
15 0206          2E700000      The SVC to print True.
01 890E                        The final branch is absolute.
5A 0206                        The definitions for these last forward
                               references should be all just loads of the RLC.
30
31
00 54                          And the data can be loaded absolutely.
5A 0205
30
31
00 46
B0                             Each of the floating constants requires a word
                               boundary.
5A 0201
30
31
03 41300000                    The hexadecimal for floating 3.
B0
5A 0202
30
31
03 41400000
B0
5A 0203
30
31
03 41500000
90                             The program start address is Pythagoras.
25 01                          Its symbol number is 01.
31
FF                             End the module.
```

Statement of the Theme Build EC Loader. The input should be a series of relocatable object files and the output both an absolute load file and a load map. Be sure to flag load errors. The map should appear sorted both by name and by address. Symbols on the map should include external symbols and references and defined map symbols. Indicate also the extent of each module.

Performance Practice Most loaders spend their time in two ways: doing input/output and manipulating the symbol table. Because there is a great deal of data to read and write, there is a lower limit on input/output time. In any case, the emphasis here is not on techniques to improve input/output, and a higher-level language probably will not help much. But the symbol table operations can be improved by careful thought. Arrange your loader so that new symbol table routines can be switched in and out easily. Write a simple table handler, debug the whole loader, and then try to improve symbol manipulation.

Past experience indicates that you should begin work on your test programs early. Turning assembly language into load modules is harder than it looks. You may want to use the assembler for EC-1 that was suggested as a debugging tool in Chapter 25. Try to exercise every operation at least once.

Orchestration Use a high-level procedural language. There is some bit picking, but it is not overwhelming in terms of language choice. Remember that variable-length records must be read.

Playing Time Six weeks for one person; 3 weeks for two or three people. Each person participating must write one relocatable-object test deck.

REFERENCES

Barron, D. W. *Assemblers and Loaders.* Macdonald, London, 1969.

Presser, Leon, and John R. White. "Linkers and Loaders." *Comput. Surveys*, 4, 3, pp. 149-167, 1972.

Barron is a simple introduction to assembly and loading. The loader described is similar to ours, and implementation details are given. Presser and White describe the system used on the IBM 360. The 360 does not have the need for relocation that other systems do; the concentration is on linking.

27

Easy Does It

or...

A COMPILER FOR AN ALGEBRAIC LANGUAGE

A compiler is always a large program. To write one from scratch, even in a pedagogical environment, is a major undertaking. Although Easy is designed to reduce the pain while providing as much enlightenment as possible, this still is the hardest problem in the book. Do not tackle it unless you (and some helpful friends) have plenty of time and energy.

THE EASY LANGUAGE

Easy is a general-purpose, procedural, algebraic programming language. Its roots lie in ALGOL, ALGOL 68, and PASCAL. Like them, it is designed to be compiled, loaded, and executed on a reasonably conventional computer (the EC-1 described in Chapter 25 is a good example). The syntax is described by a context-free grammar suitable for parsing by LR(1) techniques. The semantics are similar to the languages described above, and we will let an informal description suffice, trusting to the reader's skill to fill any gaps. In the text below, logically connected portions of the grammar are described with the associated semantics.

COMPILATIONS

```
⟨compilation⟩ ::= ⟨program segment⟩
                | ⟨compilation⟩ ⟨program segment⟩
⟨program segment⟩ ::= ⟨main program⟩
                    | ⟨external procedure⟩
```

A ⟨compilation⟩ is a string of self-contained ⟨program segment⟩ s; each segment is either a ⟨main program⟩ or an ⟨external procedure⟩. All the segments of a ⟨compilation⟩ will be associated together by the loader, but it is not necessary that all segments needed to complete a load be compiled together. A load must contain exactly one ⟨main program⟩.

PROGRAMS

⟨main program⟩ ::= ⟨program head⟩ ⟨program body⟩ ⟨program end⟩
⟨program head⟩ ::= PROGRAM ⟨identifier⟩ :
⟨program body⟩ ::= ⟨segment body⟩
⟨program end⟩ ::= END PROGRAM ⟨identifier⟩ ;

Each ⟨main program⟩ is named and the closing name must match the opening. The ⟨program body⟩ is, in common with most of the other grouping statements, a ⟨segment body⟩ and may contain all of the normal features of a program. We might as well note here that reserved words, identifiers, and constants must not be broken over record boundaries and must be separated one from another by blanks, operators, comments, or record ends.

EXTERNAL PROCEDURES

⟨external procedure⟩ ::= ⟨external subprogram⟩
| ⟨external function⟩
⟨external subprogram⟩ ::= ⟨external subprogram head⟩ :
⟨external subprogram body⟩
⟨external subprogram end⟩
⟨external function⟩ ::= ⟨external function head⟩ :
⟨external function body⟩
⟨external function end⟩
⟨external subprogram head⟩ ::= EXTERNAL PROCEDURE
⟨external procedure name⟩
⟨external function head⟩ ::= EXTERNAL FUNCTION
⟨external procedure name⟩
⟨external type⟩
⟨external procedure name⟩ ::= ⟨identifier⟩
| ⟨identifier⟩ ⟨external parameter list⟩
⟨external parameter list⟩ ::= ⟨external parameter head⟩)
⟨external parameter head⟩ ::= (⟨external parameter⟩
| ⟨external parameter head⟩ ,
⟨external parameter⟩
⟨external parameter⟩ ::= ⟨identifier⟩ ⟨external type⟩
| ⟨identifier⟩ ⟨external type⟩ NAME
⟨external type⟩ ::= ⟨basic type⟩
⟨external subprogram body⟩ ::= ⟨segment body⟩
⟨external function body⟩ ::= ⟨segment body⟩
⟨external subprogram end⟩ ::= END EXTERNAL PROCEDURE ⟨identifier⟩ ;
⟨external function end⟩ ::= END EXTERNAL FUNCTION ⟨identifier⟩ ;

The ⟨external procedure⟩'s offer the ability to compile separate modules and link them together at load time. Only one ⟨external procedure⟩ of a given name can occur in any one load. The ⟨external parameter⟩'s and the return values of ⟨external function⟩'s can only be of one of the ⟨basic type⟩'s. The ⟨formal parameter⟩'s are call by value unless tagged by the reserved word NAME, in which case they are call by name. Each ⟨external subprogram⟩ has an implicit ⟨return statment⟩ before the ⟨external subprogram end⟩, but ⟨external function⟩'s must exit *via* an explicit ⟨return statement⟩ with a returned value; thus it is a semantic error, sometimes detectable during compilation, to exit an ⟨external function⟩ through its end. There is no connection between the

variables within an ⟨external procedure⟩ and any variables in a calling procedure except through the argument passage mechanism. However, when segments are linked together (probably by a relocating loader like that of Chapter 26), the linking mechanism will check that parameter, argument, function, and return types match.

Several of the important differences between Easy and current languages show up in the description of ⟨external procedure⟩'s. Any commercial language would provide a method for sharing declarations of all sorts between ⟨program segment⟩'s. Such sharing adds considerable complexity without commensurate educational rewards and the same holds for more general argument types for ⟨external procedure⟩'s. On the other hand, commercial languages often do not allow call by name on grounds of expense, whereas we feel that the student who masters the call by name parameter passage mechanism will be untroubled by any other. Easy is wordier than similar languages because recent research shows that well-distributed redundancies are a big help in correcting syntax errors. Similarly, the Easy programmer must explicitly distinguish between subroutine and function declarations. Although the compiler could deduce the difference, the language forces the programmer to state clearly the intention of the coding.[1] Finally, Easy allows the compiler and loader to check all type matching, following the dictum that the run-time system should do as little as possible.

SEGMENTS

⟨segment body⟩ ::= ⟨type definition part⟩ ⟨variable declaration part⟩
⟨procedure definition part⟩
⟨executable statement part⟩
⟨type definition part⟩ ::=
| ⟨type definition part⟩ ⟨type definition⟩
⟨variable declaration part⟩ ::=
| ⟨variable declaration part⟩
⟨variable declaration⟩
⟨procedure definition part⟩ ::=
| ⟨procedure definition part⟩
⟨procedure definition⟩
⟨executable statement part⟩ ::= ⟨executable statement⟩
| ⟨executable statement part⟩
⟨executable statement⟩

A ⟨segment body⟩ consists of at least one ⟨executable statement⟩ optionally preceded, in order, by ⟨type definition⟩'s, ⟨variable declaration⟩'s, and ⟨procedure definition⟩'s. The scope of any name is the entire remaining body of the segment and may be used in the following definitions and declarations. No name may be declared or defined more than once in a ⟨segment body⟩, and as in ALGOL, a name may be redefined or redeclared in an inner ⟨segment body⟩.

TYPES

⟨type definition⟩ ::= TYPE ⟨identifier⟩ IS ⟨type⟩ ;
⟨type⟩ ::= ⟨basic type⟩

[1] One of the chief difficulties in proving programs correct is extracting what the program is intended to do from what the programmer wrote as imperative commands. Since the notion of correctness depends on matching intent to performance, any help in recovering intent is valuable. The Easy distinction between functions and subroutines is a small step in this direction. Also note that good programming practice suggests that no procedure should be used both as function and as subroutine and that Easy simply enforces the rule.

```
        | ⟨arrayed type⟩
        | ⟨structured type⟩
        | ⟨type identifier⟩
⟨basic type⟩ ::= INTEGER
               | REAL
               | BOOLEAN
               | STRING
⟨arrayed type⟩ ::= ARRAY ⟨bounds⟩ OF ⟨type⟩
⟨bounds⟩ ::= [ ⟨bounds expression⟩ ]
           | [ ⟨bounds expression⟩ : ⟨bounds expression⟩ ]
⟨bounds expression⟩ ::= ⟨expression⟩
⟨structured type⟩ ::= STRUCTURE ⟨field list⟩ END STRUCTURE
⟨field list⟩ ::= ⟨field⟩
               | ⟨field list⟩ , ⟨field⟩
⟨field⟩ ::= FIELD ⟨identifier⟩ IS ⟨type⟩
⟨type identifier⟩ ::= ⟨identifier⟩
```

A ⟨type definition⟩ abbreviates a ⟨type⟩ with a single ⟨identifier⟩ and the abbreviation can be used in the future where a ⟨type⟩ could be. The types include the built-in ⟨basic type⟩'s, array building ⟨arrayed type⟩'s, structure building ⟨structured type⟩'s, and abbreviated ⟨type identifier⟩'s. The INTEGER and REAL ⟨basic type⟩'s may use the hardware integer and real types and follow all the normal rules. The BOOLEAN ⟨basic type⟩ consists only of the two constants TRUE and FALSE. Items typed STRING are arbitrarily long strings of characters where "arbitrary" may be implementation dependent but should always be at least several thousand.

Arrays are single dimensional but may be of arbitrary ⟨type⟩ so that arrays of arrays of arrays of · · · may be declared. If no explicit lower bound for an array is given, the lower bound is one. The ⟨bound expression⟩'s may be arbitrarily complicated as long as they are reducible to integers. They may only contain variables declared in surrounding ⟨segment body⟩'s (not in the current ⟨segment body⟩) or in the formal parameters of a surrounding procedure. The upper bound of a pair must be no less than the lower bound. The compiler should check where possible, but, in general, this will require a run-time check. Different instances of the same ⟨arrayed type⟩ are not regarded as the same ⟨type⟩ for the purposes of compile-time type checking. However, an ⟨arrayed type⟩ may be named with a ⟨type identifier⟩ to allow such type reuse.

A ⟨structured type⟩ is similar to a record in PASCAL. The field ⟨identifier⟩'s are used as selectors for items of the field ⟨type⟩'s. Because of the recursive definition, structures may have substructures. A particular ⟨identifier⟩ can name only one ⟨field⟩ in a ⟨structured type⟩ but can be reused as a variable name or the name of a field in another (even subordinate) ⟨structured type⟩.

DECLARATIONS

```
⟨variable declaration⟩ ::= DECLARE ⟨declared names⟩ ⟨type⟩ ;
⟨declared names⟩ ::= ⟨identifier⟩
                   | ⟨declared names list⟩ )
⟨declared names list⟩ ::= ( ⟨identifier⟩
                        | ⟨declared names list⟩ , ⟨identifier⟩
```

A ⟨variable declaration⟩ gives to the ⟨identifier⟩'s in its ⟨declared names⟩ its ⟨type⟩. The ⟨identifier⟩'s are not initialized. A name (except for a field selector) can have at most one definition or declaration in a ⟨segment body⟩.

INTERNAL PROCEDURES

⟨procedure definition⟩ ::= ⟨subprogram definition⟩
| ⟨function definition⟩
| ⟨external subprogram definition⟩
| ⟨external function definition⟩
⟨subprogram definition⟩ ::= ⟨subprogram head⟩ : ⟨subprogram body⟩
⟨subprogram end⟩
⟨function definition⟩ ::= ⟨function head⟩ : ⟨function body⟩ ⟨function end⟩
⟨external subprogram definition⟩ ::= ⟨external subprogram head⟩ ;
⟨external function definition⟩ ::= ⟨external function head⟩ ;
⟨subprogram head⟩ ::= PROCEDURE ⟨procedure name⟩
⟨function head⟩ ::= FUNCTION ⟨procedure name⟩ ⟨type⟩
⟨subprogram body⟩ ::= ⟨segment body⟩
⟨function body⟩ ::= ⟨segment body⟩
⟨subprogram end⟩ ::= END PROCEDURE ⟨identifier⟩ ;
⟨function end⟩ ::= END FUNCTION ⟨identifier⟩ ;
⟨procedure name⟩ ::= ⟨identifier⟩
| ⟨identifier⟩ ⟨internal parameter list⟩
⟨internal parameter list⟩ ::= ⟨internal parameter head⟩)
⟨internal parameter head⟩ ::= (⟨internal parameter⟩
| ⟨internal parameter head⟩ , ⟨internal parameter⟩
⟨internal parameter⟩ ::= ⟨identifier⟩ ⟨type⟩
| ⟨identifier⟩ ⟨type⟩ NAME

There may be only one procedure of a given name defined immediately in any one ⟨segment body⟩. An ⟨external subprogram⟩ or ⟨external function⟩ definition supplies only the heading because an ⟨external procedure⟩ in the same or a different ⟨compilation⟩ will supply the body. The local definition and the eventually supplied procedure must match exactly in procedure name, order, type, and mode of formal parameters, and this correspondence will be checked by the loader. Remember that parameters to an ⟨external procedure⟩ must be of a ⟨basic type⟩.

The definition of local procedures is similar. The ⟨internal parameter⟩'s may be of any ⟨type⟩, as may be the return value of a function. A subprogram has an implicit ⟨return statement⟩ before its end, but a function must be exited by an explicit ⟨return statement⟩ with a value. The parameters are normally call by value but are call by name if marked with NAME. The procedures themselves are like ALGOL procedures and are fully recursive. A ⟨procedure name⟩ may not be used before it is declared.

EXECUTABLE STATEMENTS

⟨executable statement⟩ ::= ⟨assignment statement⟩
| ⟨call statement⟩
| ⟨return statement⟩
| ⟨exit statement⟩
| ⟨conditional statement⟩
| ⟨compound statement⟩
| ⟨iteration statement⟩
| ⟨selection statement⟩
| ⟨repeat statement⟩
| ⟨repent statement⟩

```
                | ⟨input statement⟩
                | ⟨output statement⟩
                | ⟨null statement⟩
```

Because the ALGOL 68 forms of the conditional and iteration statements are used, there is no need to have several separate syntactic classes of statements. Statements are all terminated with semicolons.

ASSIGNMENTS

```
⟨assignment statement⟩ ::= SET ⟨target list⟩ ⟨expression⟩ ;
⟨target list⟩ ::= ⟨target⟩
                | ⟨target list⟩ ⟨target⟩
⟨target⟩ ::= ⟨variable⟩ ⟨replace⟩
⟨replace⟩ ::= :=
```

In an ⟨assignment statement⟩, the ⟨type⟩ of all the ⟨target⟩'s and of the assigned ⟨expression⟩ must be the same. The ⟨target⟩'s are evaluated from left to right to find the storage locations and only then is the expression evaluated to calculate the stored value. Structures and arrays may be assigned if ⟨type⟩'s are identical. The use of the keyword SET is an example of Easy's wordiness. This particular redundancy aids correction when *other* keywords are misspelled (a common user error).

PROCEDURE CALLS

```
⟨call statement⟩ ::= CALL ⟨procedure reference⟩ ;
⟨procedure reference⟩ ::= ⟨procedure identifier⟩
                        | ⟨procedure identifier⟩ ⟨actual argument list⟩
⟨procedure identifier⟩ ::= ⟨identifier⟩
⟨actual argument list⟩ ::= ⟨actual argument head⟩ )
⟨actual argument head⟩ ::= ( ⟨expression⟩
                         | ⟨actual argument head⟩ , ⟨expression⟩
```

Only defined procedures that include the ⟨call statement⟩ in the range of their names may be called. The actual arguments must correspond exactly in number, order, and type with the procedure's formal parameters. After the ⟨return statement⟩ enclosed in the called procedure is executed, control passes to the statement following the call. The keyword CALL is used for the same reason as SET in the ⟨assigned statement⟩.

RETURNS

```
⟨return statement⟩ ::= RETURN ;
                     | RETURN ⟨expression⟩ ;
```

A ⟨return statement⟩ may occur only in a procedure and causes return of control to the calling statement. There is an implicit ⟨return statement⟩ at the end of subprograms. Subprogram returns must be without value and function returns must be with a value of the same type as the function.

EXITS

⟨exit statement⟩ ::= EXIT ;

This statement causes a tidy exit from the entire program and a return to the supervisor. It must be the last statement executed —*not written* — in a ⟨program⟩.

CONDITIONALS

⟨conditional statement⟩ ::= ⟨simple conditional statement⟩
| ⟨label⟩ ⟨simple conditional statement⟩
⟨simple conditional statement⟩ ::= ⟨conditional clause⟩ ⟨true branch⟩ FI ;
| ⟨conditional clause⟩ ⟨true branch⟩
⟨false branch⟩ FI ;
⟨conditional clause⟩ ::= IF ⟨expression⟩
⟨true branch⟩ ::= THEN ⟨conditional body⟩
⟨false branch⟩ ::= ⟨else⟩ ⟨conditional body⟩
⟨else⟩ ::= ELSE
⟨conditional body⟩ ::= ⟨segment body⟩

A ⟨conditional statement⟩ selects and executes its ⟨true branch⟩ or its ⟨false branch⟩, depending on whether the ⟨expression⟩, which must be Boolean, is true or false. Each branch is a ⟨segment body⟩ and may contain all needful definitions, declarations, and statements without any further bracketing. Control passes to the statement following the conditional after execution of the selected branch.

COMPOUNDS

⟨compound statement⟩ ::= ⟨simple compound⟩
| ⟨label⟩ ⟨simple compound⟩
⟨simple compound⟩ ::= ⟨compound head⟩ ⟨compound body⟩ ⟨compound end⟩
⟨compound head⟩ ::= BEGIN
⟨compound body⟩ ::= ⟨segment body⟩
⟨compound end⟩ ::= END ;
| END ⟨identifier⟩ ;

There is little need for a ⟨compound statement⟩ in Easy because of the rest of syntax. However, it is useful with REPEAT and REPENT statements. Declarations and definitions begin (optionally) a compound. If a trailing ⟨identifier⟩ is included, there must be a ⟨label⟩ and the ⟨identifier⟩ must match the ⟨label⟩.

ITERATIONS

⟨iteration statement⟩ ::= ⟨simple iteration statement⟩
| ⟨label⟩ ⟨simple iteration statement⟩
⟨simple iteration statement⟩ ::= ⟨iteration head⟩ ⟨iteration body⟩
⟨iteration end⟩
⟨iteration head⟩ ::= ⟨for⟩ ⟨iteration target⟩ ⟨control⟩ DO
⟨iteration body⟩ ::= ⟨segment body⟩

```
⟨iteration end⟩ ::= END FOR ;
                  | END FOR ⟨identifier⟩ ;
⟨for⟩ ::= FOR
⟨iteration target⟩ ::= ⟨variable⟩ ⟨replace⟩
⟨control⟩ ::= ⟨step control⟩
            | ⟨step control⟩ ⟨while control⟩
⟨step control⟩ ::= ⟨initial value⟩ ⟨step⟩
                 | ⟨initial value⟩ ⟨limit⟩
                 | ⟨initial value⟩ ⟨step⟩ ⟨limit⟩
⟨initial value⟩ ::= ⟨expression⟩
⟨step⟩ ::= BY ⟨expression⟩
⟨limit⟩ ::= TO ⟨expression⟩
⟨while control⟩ ::= WHILE ⟨expression⟩
```

The easiest way to explain the effect of the ⟨iteration statement⟩ is to write a small piece of "meta-Easy" that will replace the ⟨iteration statement⟩. This "definition" as given in Figure 27–1 should be applied to the ⟨iteration statement⟩ to find its effect. The "definition" *does* imply recalculation of the ⟨target⟩, ⟨limit⟩, and ⟨step⟩ at each iteration. The predicate "exists" is a meta-Easy way to ask about the exact options used in a particular ⟨iteration statement⟩. If the closing ⟨identifier⟩ is used, it must match the (necessarily existent) ⟨label⟩.

Pedagogy overwhelms practice once again in the definition of Easy ⟨iteration statement⟩'s. The dynamic redefinition of control values is inherited from ALGOL; most other languages avoid it because of the cost. But if you can implement dynamic definition, you will have learned more than enough to build simpler static iterations. The stepping iteration and the while iteration are combined in one statement to make Easy a smaller language; a practical language might separate them. Be careful to reevaluate the ⟨iteration target⟩ on each cycle; if it is a formal parameter or an array element, reevaluation may select a different specific variable during each iteration.

SELECTION

```
⟨selection statement⟩ ::= ⟨simple selection⟩
                        | ⟨label⟩ ⟨simple selection⟩
⟨simple selection⟩ ::= ⟨selection head⟩ ⟨selection body⟩ ⟨selection end⟩
⟨selection head⟩ ::= SELECT ⟨expression⟩ OF
⟨selection body⟩ ::= ⟨case list⟩
                   | ⟨case list⟩ ⟨escape case⟩
⟨selection end⟩ ::= END SELECT ;
                  | END SELECT ⟨identifier⟩ ;
⟨case list⟩ ::= ⟨case⟩
              | ⟨case list⟩ ⟨case⟩
⟨case⟩ ::= ⟨case head⟩ ⟨case body⟩
⟨case head⟩ ::= CASE ⟨selector⟩ :
⟨selector⟩ ::= ⟨selector head⟩ )
⟨selector head⟩ ::= ( ⟨expression⟩
                  | ⟨selector head⟩ , ⟨expression⟩
⟨escape case⟩ ::= ⟨escape head⟩ ⟨case body⟩
⟨escape head⟩ ::= OTHERWISE :
⟨case body⟩ ::= ⟨segment body⟩
```

```
      SET ⟨iteration target⟩ := ⟨initial value⟩;
top:  IF ⟨while control⟩ exists
         THEN SET stoploop := NOT ⟨while control⟩;
         ELSE SET stoploop := FALSE; FI;
      IF stoploop THEN GOTO end; FI;
      IF ⟨limit⟩ exists & (⟨iteration target⟩ > ⟨limit⟩)
         THEN GOTO end; FI;
      ⟨iteration body⟩
      IF ⟨step⟩ exists
         THEN SET stepvalue := ⟨step⟩;
         ELSE SET stepvalue := 1; FI;
      SET ⟨iteration target⟩ := ⟨iteration target⟩ + stepvalue;
      GOTO top;
end:  · · ·
```

Figure 27-1. Meta-Easy to Define the ⟨iteration statement⟩ to Find Its Effect. The "definition" does imply recalculation of the ⟨target⟩.

A ⟨selection statement⟩ operates as follows.

The ⟨expression⟩ in the ⟨selection head⟩ is evaluated.

The ⟨case⟩'s in the ⟨case list⟩ are processed from first to last.

For each ⟨case⟩, the ⟨expression⟩'s in the ⟨selector⟩ are evaluated one by one from left to right. As each ⟨expression⟩ is evaluated, its value is compared with the value of the original ⟨expression⟩ in the ⟨selection head⟩. If the two are equal, the corresponding ⟨case body⟩ is executed and control then passes out of the ⟨selection statement⟩ to the next ⟨statement⟩ in sequence without any further activity.

If no ⟨case⟩ is selected and if there is an ⟨escape case⟩, then the ⟨case body⟩ of the ⟨escape case⟩ is executed and control passes out of the ⟨selection statement⟩. Otherwise the ⟨selection statement⟩ has no effect beyond side effects of the various expression evaluations.

The types of all ⟨expression⟩'s used to select a ⟨case⟩ must be the same. If an ⟨identifier⟩ is used in the ⟨selection end⟩, it must match the (necessarily existent) ⟨label⟩ on the ⟨selection statement⟩.

REPEAT AND REPENT

⟨repeat statement⟩ ::= REPEAT ⟨identifier⟩ ;
⟨repent statement⟩ ::= REPENT ⟨identifier⟩ ;

A ⟨repeat statement⟩ causes a transfer of control back to the beginning of the *enclosing* statement body labeled with ⟨identifier⟩. All intervening surrounding segment bodies and the statements of which they are a part are terminated as if they had been exited normally from the bottom, and all associated storage is destroyed. The ⟨label⟩ transferred to must be in the same procedure as the ⟨repeat statement⟩. If there is no such surrounding labeled statement, the ⟨repeat statement⟩ is semantically in error. The ⟨repent statement⟩ has the same semantics with the exception that control passes to the point immediately following the surrounding labeled statement rather than to the head of that statement. Notice that a ⟨repeat statement⟩ causes reexecution of the statement to whose head control was transferred.

INPUT AND OUTPUT

```
⟨input statement⟩ ::= INPUT ⟨input list⟩ ;
⟨input list⟩ ::= ⟨variable⟩
                | ⟨input list⟩ , ⟨variable⟩
⟨output statement⟩ ::= OUTPUT ⟨output list⟩ ;
⟨output list⟩ ::= ⟨expression⟩
                | ⟨output list⟩ , ⟨expression⟩
```

The ⟨input statement⟩ causes transmission of data items from the input stream to the ⟨variable⟩'s in the ⟨input list⟩. Input items may be of only the basic types, and they must match in type the corresponding variable. An input item has the same appearance as a constant of the same type. Items on the input stream must be separated by blanks or new record characters.

The ⟨output statement⟩ similarly causes transmission of its ⟨output list⟩ ⟨expression⟩'s to the output stream. The ⟨expression⟩'s must be of the basic types, and the exact format on the output stream is up to the implementor as long as the values can be read back in again. Each ⟨output statement⟩ writes a new record character on completion.

NULLS AND LABELS

```
⟨null statement⟩ ::= ;
⟨label⟩ ::= ⟨identifier⟩ :
```

The ⟨null statement⟩ causes no action. A ⟨label⟩ is an ⟨identifier⟩, is declared by use, and may not be declared or defined except as a field selector in the same ⟨segment body⟩.

EXPRESSIONS

```
⟨expression⟩ ::= ⟨expression one⟩
                | ⟨expression⟩ | ⟨expression one⟩
                | ⟨expression⟩ XOR ⟨expression one⟩
⟨expression one⟩ ::= ⟨expression two⟩
                    | ⟨expression one⟩ & ⟨expression two⟩
⟨expression two⟩ ::= ⟨expression three⟩
                    | NOT ⟨expression three⟩
⟨expression three⟩ ::= ⟨expression four⟩
                      | ⟨expression three⟩ ⟨relation⟩ ⟨expression four⟩
⟨expression four⟩ ::= ⟨expression five⟩
                     | ⟨expression four⟩ || ⟨expression five⟩
⟨expression five⟩ ::= ⟨expression six⟩
                     | ⟨expression five⟩ ⟨adding operator⟩ ⟨expression six⟩
                     | ⟨adding operator⟩ ⟨expression six⟩
⟨expression six⟩ ::= ⟨expression seven⟩
                    | ⟨expression six⟩ ⟨multiplying operator⟩
                          ⟨expression seven⟩
⟨expression seven⟩ ::= FLOOR ( ⟨expression⟩ )
                      | LENGTH ( ⟨expression⟩ )
                      | SUBSTR ( ⟨expression⟩ , ⟨expression⟩ ,
                              ⟨expression⟩ )
```

```
                    | CHARACTER ( ⟨expression⟩ )
                    | NUMBER ( ⟨expression⟩ )
                    | FLOAT ( ⟨expression⟩ )
                    | FIX ( ⟨expression⟩ )
                    | ⟨expression eight⟩
⟨expression eight⟩ ::= ⟨variable⟩
                    | ⟨constant⟩
                    | ⟨function reference⟩
                    | ( ⟨expression⟩ )
```

Expressions operate in a fairly standard way. The operators |, XOR (exclusive or), &, and NOT all must have Boolean operands. The equality and inequality ⟨relation⟩'s may hold between any two items of the same type. Strings may be compared to strings with any of the ⟨relation⟩'s. Two strings are equal if and only if they are exactly the same, and string A is less than string B if a prefix of A is equal to a prefix of B and there are no more characters in A or the next character of A is less than the next character of B in the collating sequence. Any two integers or reals may be compared by using any ⟨relation⟩; integers are implicitly converted to real if compared to reals. The result of any comparison is always of Boolean type.

Only integers and reals may be combined by using ⟨adding operator⟩'s and ⟨multiplying operator⟩'s. If an integer is combined with a real, the integer is converted to real prior to the operation. Real numbers may not be used in the MOD operation; in a divide or MOD operation involving only integers the quotient is always chosen so that the remainder is nonnegative. The operands of the catenation operator || are normally strings and the value is a string; reals, integers, and Booleans are converted to their output string form before the operation.

The FLOOR function takes a real as argument and returns as value the real both integral and not more than the argument. The LENGTH function takes a string as argument and returns its length as an integer. The first argument of the SUBSTR function is a string, and the value is a substring whose first character (counting from zero) is named by the second integer argument and whose length is given by the third integer argument. The CHARACTER function takes as argument an integer and returns a single character string whose character is indexed by the argument in the collating sequence. The NUMBER function returns as integer value the index in the collating sequence of the first character of the argument string. The FLOAT function converts its integer argument to a real value, and the FIX function converts its real argument to an integer value. SUBSTR, FIX, and CHARACTER may cause run-time errors.

VARIABLES

```
⟨variable⟩ ::= ⟨identifier⟩
             | ⟨variable⟩ . ⟨identifier⟩
             | ⟨variable⟩ [ ⟨expression⟩ ]
```

A ⟨variable⟩ is a simple ⟨identifier⟩, a ⟨variable⟩ with a field selector, or a ⟨variable⟩ with an array subscript. Of course, all ⟨variable⟩'s must be declared. An ⟨identifier⟩ is a terminal syntactic item. It must begin with an upper- or lowercase alphabetic and may continue with an arbitrary number of alphabetics and decimal digits. Reserved words may not be used as ⟨identifier⟩'s, and both reserved words and ⟨identifier⟩'s must be separated from other nonoperator lexical items by at least one blank, comment, or new line character. No lexical item may be broken across a record boundary.

CONSTANTS

```
⟨constant⟩ ::= ⟨integer constant⟩
             | ⟨real constant⟩
             | ⟨boolean constant⟩
             | ⟨string constant⟩
⟨boolean constant⟩ ::= TRUE
                     | FALSE
```

An ⟨integer constant⟩ is an unbroken string of decimal digits and must be separated from other non-operator lexical items by at least one blank, comment, or new line character. A ⟨real constant⟩ is an unbroken string of decimal digits, followed immediately by a period, followed optionally by another decimal string. Otherwise ⟨real constant⟩'s follow the same rules as ⟨integer constant⟩'s. A ⟨string constant⟩ is a double quote ", followed by an arbitrary string of characters not including a double quote, and terminated by a double quote. Double quotes may be included in ⟨string constant⟩'s by adding pairs; for example, """" is the string of exactly one double quote. Otherwise ⟨string constant⟩'s operate like ⟨identifier⟩'s. In particular, new line characters may not appear in strings.

FUNCTION CALLS

```
⟨function reference⟩ ::= ⟨function identifier⟩ ( )
                       | ⟨function identifier⟩ ⟨actual argument list⟩
⟨function identifier⟩ ::= ⟨identifier⟩
```

A ⟨function identifier⟩ is an ordinary ⟨identifier⟩ that occurs in some function definition. Functions with no arguments are called with the first form of the ⟨function reference⟩.

LEXICAL ITEMS

```
⟨relation⟩ ::= ⟨
             | ⟩
             | =
             | ⟨ =
             | ⟩ =
             | ⟨ ⟩
⟨adding operator⟩ ::= +
                    | -
⟨multiplying operator⟩ ::= *
                         | /
                         | MOD
```

Operators also include :, ;, (,), „, [,], &, |, ||, and :=, and do not include XOR, NOT, and MOD for separation purposes. Comments begin with /*, continue with any string not including */, end with */, and may appear wherever a separator blank may appear. Comments act as separators.

AN EXAMPLE PROGRAM

The following program illustrates some Easy features. It would be extremely difficult to pack every possible usage into a few lines, but PROGRAM Eratosthenes should give some flavor of the language. It might also bė used as a test program for compilers, since its output is so obvious and yet it does a nontrivial computation. Perhaps the only real trickiness (stolen from ALGOL) is the use of the loop controls to do all the calculations of interest inside FUNCTION integersqrt. Here is the program.

```
PROGRAM Eratosthenes:

   /* This example EASY program reads an input integer topnum and then
      uses the Sieve of Eratosthenes (see Chapter 15) to build
      a table of all the primes between 1 and topnum and to print that
      table.
   */

   DECLARE topnum INTEGER;

   FUNCTION abs(x REAL) REAL:

      /* This function returns the real absolute value of its real
         argument.
      */

      IF x < 0 THEN RETURN -x; ELSE RETURN x; FI;

   END FUNCTION abs;

   FUNCTION integersqrt(a INTEGER) INTEGER:

      /* This function takes an integer as argument and returns as value
         the floor of the square root of the argument.
         The FOR loop which calculates the square root is simply
         Newton's approximation.  The last FOR loop makes sure
         that the integer value calculated is really the floor of
         the square root of a.  Notice the rather tricky use
         of the iteration and the null subject statements.
      */

      SELECT TRUE OF
         CASE (a < 0): OUTPUT "a < 0 in FUNCTION integersqrt."; EXIT;
         CASE (a = 0): RETURN 0;
         CASE (a > 0):
            DECLARE (x, ra) REAL;
            DECLARE epsilon REAL;
            DECLARE sqrt INTEGER;
            SET ra := FLOAT(a);
            SET epsilon := 0.0000001*ra;
```

```
        FOR x := ra/2. BY (ra/x-x)/2. WHILE abs(ra-x*x) > epsilon
              DO ; END FOR;
        FOR sqrt := FIX(x)-1 BY 1 WHILE (sqrt+1)*(sqrt+1) <= a
              DO ; END FOR;
        RETURN sqrt;
   END SELECT;

END FUNCTION integersqrt;

INPUT topnum;
IF topnum > 0 THEN
      DECLARE sieve ARRAY[1:topnum] OF BOOLEAN;
      DECLARE (i, limit, count) INTEGER;
      FOR i := 1 TO topnum DO SET sieve[i] := TRUE; END FOR;
      SET limit := integersqrt(topnum)+1; /* Avoid repeating square root */
      FOR i := 2 TO limit DO
         IF sieve[i] THEN
            DECLARE j INTEGER;
            FOR j := 2*i BY i TO topnum DO SET sieve[j] := FALSE; END FOR;
         FI;
      END FOR;
      SET count := 0;
      FOR i := 1 TO topnum DO
         IF sieve[i] THEN
            SET count := count + 1;
            OUTPUT "Prime[" || count || "] = " || i;
         FI;
      END FOR;
   ELSE
      OUTPUT "Input value " || topnum || " non-positive.";
   FI;
   EXIT;

END PROGRAM Eratosthenes;
```

THE EXECUTION ENVIRONMENT

Easy will require a fairly elaborate run-time support system. Because of the recursive procedures, an activation stack will be needed. Because of the strings, a heap is necessary. Because of the input/output statements, some contact with the supervisor will be required. For big implementations, these functions would probably be found in a library of run-time routines. Such routines, however, entail more work than is necessary for this project. You may want to provide these services through supervisor calls to a monitor routine that can be written in your favorite language.

Statement of the Theme Write a compiler for Easy that will generate relocatable object code for the EC-1 computer or some other computer (see Chapter 25). Test your compiler by writing some Easy programs, compiling them, loading them with the EC Loader (see Chapter 26), and running them on the EC-1. Be sure that the compiler accepts all syntactically and lexically correct

programs and detects all incorrect ones (semantic errors may terminate compilation of otherwise valid programs). Error handlers need not correct errors or continue compilation, but they should be precise about the causes and locations of errors. Listing output can be primitive, since there will be other details to worry about. Document carefully the order in which you did the work and what troubles you had.

Performance Practice This is the largest project in the book. Completion will require you to take advantage of all the help that you can get. Generally projects are designed so that you build the whole program from the ground up, but in this case you should use all available compiler construction tools. In particular, syntactic analysis routines driven by grammar descriptions are now quite common and can save considerable time. You should program in a higher-level language (remember that efficiency is not a goal), and one with built-in strings can speed the program development considerably.

Specific compiler construction techniques are described in the books listed in the bibliography. Although we will not try to duplicate that material here, we can suggest an order of implementation: first the lexical analyzer, then the syntactic analyzer, and, finally, the semantic synthesizer and code-generation routines. Doing the problem this way will allow you to test as you go instead of hoping that, when you are all done, everything will work right.

It is quite probable that you will not complete this project. So you should consider the order in which you are going to implement the various language features. If you choose the right ordering, the finished product will be a subset of the complete Easy language. Begin with the implementation of single ⟨program segment⟩'s. Within them, try to implement ⟨expression⟩'s, the basic statement, internal procedures, and declarations of variables with ⟨basic type⟩'s. The ⟨iteration statement⟩ will probably be the hardest to generate code for and should be done last. At an early stage you should write code-generator routines to output relocatable object code so that these routines can be used easily later.

The run-time environment for Easy will certainly include a stack discipline for allocating storage to procedures. When the other statements are well underway, begin to work on structured data types. The run-time allocation for strings will need a heap that will probably share space with the stack. These run-time allocation mechanisms will require you to change your supervisor on the EC-1, and early versions should not include any garbage collection to reclaim returned heap space. Parallel to this effort, you can be working on compilation of separate segments. Remember to insert run-time debugging code into the object modules right from the beginning.

Orchestration Do the compiler in some higher-level procedural language. XPL was designed for compiler construction. SNOBOL is not allowed, since compilers are much too easy to implement in SNOBOL and you will not learn enough.

Playing Time Two, three, or four people for 10 weeks. Each participant is independently responsible for one Easy program for test purposes.

REFERENCES

Gries, David. *Compiler Construction for Digital Computers.* Wiley, New York, NY, 1971.

Gries is the outstanding book on compiler implementation. All the techniques are here for simple compilers and pointers are available for the hard ones. There is not enough emphasis on table-driven LR(k) parsing

techniques, but that situation is easily remedied by the other references. If you know which language you want to compile, Gries will show you how to do it.

McKeeman, W. M., J. J. Horning, and D. B. Wortman. *A Compiler Generator.* Prentice-Hall, Englewood Cliffs, NJ, 1970.

This book describes the XPL language and its use in compiler construction. The XPL compiler, written in XPL, serves as an example of the technique. Early printings use the SMSP table-driven parsing technique, which is now obsolete, but later printings have a good discussion of LR(k) parsing. Unfortunately, the published listing of the compiler does not have a modification for LR(k), and the SLR(k) table generator of DeRemer is not included. This book is also the manual for XPL.

Nicholls, John E. *The Structure and Design of Programming Languages.* Addison-Wesley, Reading, MA, 1975.

Pratt, Terrence W. *Programming Languages Design and Implementation.* Prentice Hall, Englewood Cliffs, NJ, 1975.

Nicholls and Pratt are both books to be read before a language is designed or a compiler is implemented. Instead of discussing particular compiler techniques, they study the effects of language structures on the programmer, the run-time system, the eventual program user, and the compiler implementor. There is considerable weighing of alternate solutions to common problems in programming language design.

28

Off the Beaten TRAC

or...

BUILDING A TRAC INTERPRETER

There are few programming languages that a beginner can implement singlehandedly in just a few weeks, but TRAC[1] is one of them. Calvin Mooers wanted to design a programming language that would be simple, elegant, powerful, and interactive. He managed to make from the old idea of the macro a language that met these goals, that could also be useful in batch mode, and that was extremely easy to implement (the first processor took a weekend to write). In fact, this book was edited and typeset in manuscript using a dialect of TRAC.

THE TRAC LANGUAGE

Consider yourself seated in front of an interactive console through which the TRAC processor is running. You run your programs by typing them, over line boundaries if necessary, and finishing each one with a special terminating metacharacter (initially the apostrophe '). As soon as the processor sees the metacharacter, it interprets your program and returns the result by typing it on the console. You may now type another program, thereby restarting the cycle. The programs themselves may be arbitrary character strings, but certain special substrings invoke built-in TRAC functions. Functions can be used for fairly standard arithmetic and character manipulation and can also store and recall results of other functions so that vast pyramids of values can be built.

A function invocation has either of the forms #(· · ·) or ##(· · ·), where the interior can itself be any arbitrary string. The body of the function is separated by commas into arguments (no commas means one argument) that are evaluated from left to right in the same way as the program. The first argument is assumed to be the name of a built-in function, and the value is computed by supplying the arguments to the function. If the function was the single sharp #(· · ·) form, the value is rescanned; otherwise the value is passed over without rescanning. The form (· · ·), where the interior string is parenthesis-balanced, protects the interior string from evaluation. The processor

[1] TRAC is the registered trademark of the Rockford Research Institute, Cambridge, MA.

simply strips the outer parentheses and passes by the interior. If *ml* is the multiply function and *ad* the addition function, then the input string

```
((3+4))*9 = #(ml,#(ad,3,4),9)'
```

will evaluate to

```
(3+4)*9 = 63
```

Note that an extra pair of parentheses is needed to prevent the internal pair around *3+4* from being stripped.

The actual operation of the processor is controlled by a precise procedure called the TRAC algorithm. Interior to the processor are three structures: the active string, the neutral string, and the scan pointer. A convenient visualization is to think of the neutral string on the left, the scan pointer in the middle, and the active string on the right. The scan pointer always points at the left of the active string, looking at the first character. Normally characters move from active to neutral string, where they are accumulated until there are enough to form all the arguments of some function. Then the function is evaluated and the result put back in the active or neutral string according to the function type.

THE TRAC ALGORITHM

The algorithm consists of 10 numbered steps. When the processor is executed, it begins by going to step 1. Throughout the algorithm there are mentions of marking the neutral string in various ways. Think of these markers as flags attached to the affected characters (of course, the markers will probably be implemented by pointers). In any real implementation there is some special "break" key to interrupt the endless cycle dictated by the algorithm.

1. Clear the processor by emptying the neutral string, deleting the contents of the active string, if any, filling the active string with the string #(*ps*,#(*rs*)), and setting the scan pointer to the first character of the active string.[2] Go on to the next step.

2. Examine the character under the scan pointer. If there is none—that is, if the active string is the null string—return to step 1.

3. If the character under the scan pointer is a tabulate, a line feed, a record end, or a carriage return, delete it, advance the scan pointer, and return to step 2.

4. If the character under the scan pointer is a left parenthesis, delete it and scan forward until the *matching* right parenthesis is found. After all of the intervening characters have been moved without change to the neutral string, the right parenthesis deleted, and the scan pointer moved to the character following the right parenthesis, return to step 2. If the matching right parenthesis cannot be found, go back to step 1.

5. If the character under the scan pointer is a comma, delete it, mark the rightmost character of the neutral string as the end of one argument and the next character as the beginning of a new argument, advance the scan pointer, and return to step 2.

6. If the character under the scan pointer is a sharp sign and the next succeeding character is a left parenthesis, an active function is beginning. Delete the sharp sign and the left parenthesis, advance the scan pointer beyond them, mark the rightmost character of

[2] Mooers now uses the idling string #(ps, (*CR*-*LF*))#(ps,#(rs)), where *CR* is the carriage return and *LF* is the linefeed.

the neutral string as the beginning of both an argument and an active function, and return to step 2.

7. If the character under the scan pointer is a sharp sign and the next two succeeding characters are another sharp sign and a left parenthesis, a neutral function is beginning. Delete the triple ## (, advance the scan pointer beyond them, mark the rightmost character of the neutral string as the beginning of both an argument and a neutral function, and return to step 2.

8. If the character under the scan pointer is a sharp sign that did not meet the conditions of step 6 or 7, move it to the right end of the neutral string, advance the scan pointer, and return to step 2.

9. If the character under the scan pointer is a right parenthesis, a function is ending. Delete the right parenthesis, advance the scan pointer, and mark the rightmost character of the neutral string as the end of an argument and the end of a function. Now the neutral string from the rightmost begin function marker to the just inserted end function marker constitutes a TRAC function invocation. (If there is no begin function marker in the neutral string, return to step 1.) Take the function arguments (all those marked since the beginning of the function) and all the function and argument markers out of the neutral string. (Because of the way the markers are placed, all begin markers are actually on the character immediately preceding the item marked.) The first argument is assumed to be the name of a TRAC built-in function. Evaluate the function with the given arguments: extra arguments are ignored and missing ones are automatically supplied as the null string. The function value is catenated to the right of the neutral string if the function was marked as neutral and to the left of the active string if marked active; in the latter case, the scan pointer is reset to the leftmost character of the new active string. If the first argument is not the name of any built-in function, simply provide a null string as function value. Return to step 2.

10. If the character under the scan pointer did not meet any of the conditions of steps 3 through 9, attach it to the right of the neutral string, delete it from the active string, advance the scan pointer, and return to step 2.

THE TRAC FUNCTIONS

The TRAC functions are listed here in their active forms, but each can be called in neutral mode as well. The value of a function is always a string; any function, particularly those whose most important activity is a side effect, might return the null string. In addition to the structures already mentioned, the processor can store strings in an area called forms storage. Each form has three parts: a form name, which may be any string whatsoever; a form body, which may also be any string; and a form pointer, which initially points just in front of the first character of the form body. The form pointer always points just before the body, just after it, or between two characters; that is, it always points into a *gap* between characters. Form bodies may include ordinal segment markers intermixed with their characters. Each such marker has some positive integer associated with it, and these integers need not be distinct. The function descriptions have been slightly sanitized from those originally given by Mooers.

#(*rs*) "Read String" (One argument) The value of this function is the input character stream up to but not including the next metacharacter. This string does include any carriage returns, line feeds, record ends, or tabulates that the system would normally pass to a program. The metacharacter is always discarded and, in a record-oriented system, so is everything following the metacharacter on the same record.

#(*rc*) "Read Character" (One argument) The next input character is returned as value regardless of what it may be. Any character, including the metacharacter, may be read this way.

#(*ps,X*) "Print String" (Two arguments) The second argument *X* is printed on the output device. The function value is the null string.

#(*cm,X*) "Change Metacharacter" (Two arguments) This null-valued function changes the metacharacter to the first character of the string *X*. If *X* is null, no change is made. The metacharacter is initially the apostrophe.

#(*ds,N,B*) "Define String" (Three arguments) This null-valued function creates a form with name *N* and body *B* and with its form pointer just before the first character of *B*. If there is a form with name *N* already, its previous body and form pointer are lost.

#(*ss,N,P1,P2,* · · ·) "Segment String" (At least three arguments) This null-valued function creates ordinal segment markers in the form *N*. The nonnull arguments *P1*, *P2*, . . . , are processed in turn from left to right (null arguments are ignored). Argument *Pi* is processed in the following way. The body of form *N* is scanned from left to right for the first substring exactly equal to *Pi*. The matching substring must not contain any already-existing segment markers. If it does not, the substring is taken out of the form body and an ordinal segment marker numbered *i* replaces it. The matching process begins again at the character following the marker. The form pointer is replaced at the left end of the form when the segmentation is finished. A form may be segmented more than once.

#(*cl,N,A1,A2,* · · ·) "Call String" (Two or more arguments) The value of this function is the body of the form *N* with its segment markers filled in. All those segment markers numbered 1 are filled with argument *A1*, those numbered 2 with *A2*, and so on. As many arguments to *cl* are needed as the highest-numbered segment marker in form *N*. Remember that excess arguments are ignored and missing ones are supplied with the null string.

#(*cs,N,Z*) "Call Segment" (Three arguments) The value of this function is the substring of the form *N* from the current location of the form pointer to the next segment marker to the right (for the purposes of this function, the end of the body is counted as a marker). The marker is not part of the value, and the form pointer is left just before the character immediately to the right of the marker. If the form pointer is already at the right end of the body, the function value is argument *Z* returned in active mode regardless of the function mode.

#(*cc,N,Z*) "Call Character" (Three arguments) The value of this function is the character immediately following the form pointer in the form *N*. The form pointer is advanced just beyond the selected character. Segment markers are always ignored by the form pointer, since they are not characters. If the form pointer is already at the right end of the string, the function value is argument *Z* returned in active mode regardless of the mode of the function call.

#(*cn,N,D,Z*) "Call *N* Characters" (Four arguments) The value of this function is a substring of form *N*. Starting at the form pointer and reading right or left, depending on whether *D* is positive or negative, the value is |*D*| characters of the form body in the chosen direction.[3] The characters of the value are in the same order as they were in the body; that is, the string is not reversed if *D* is negative. Segment markers are, of course, ignored. The form pointer is moved to point between the selected substring and the first unread character in the appropriate direction. (If *D* is zero, the value is null and the pointer does not move.) If the form pointer should move off either end of the form, the function value is argument *Z* returned in active mode regardless of the mode of the function call.

#(*in,N,X,Z*) "Initial Match" (Four arguments) The form *N* is searched rightward from the form pointer for a substring containing no segment markers and exactly matching argument *X*.

[3] The interpretation of a string as a number will be discussed later.

If such a match is found, the value of the function is the substring of the form from the original pointer location to the character immediately preceding the match (segment markers are dropped from the value) and the form pointer is moved just before the character immediately following the matching substring. If no match is found, the argument *Z* is returned as value in active mode regardless of the mode of the function call, and the form pointer is stationary.

#(cr,N) "Call Restore" (Two arguments) This null-valued function returns the form pointer of form *N* to its initial position just before the first character of the form.

#(dd,N1,N2, · · ·) "Delete Definition" (Two or more arguments) This null-valued function deletes the forms named *N1*, *N2*, . . ., from forms storage.

#(da) "Delete All" (One argument) This null-valued function deletes all the forms from forms storage.

TRAC performs arithmetic on strings of decimal characters. The arithmetic value of a string is given by the longest suffix of the string that can be described exactly as all decimal digits preceded by at most one plus or minus sign. Thus the value of *3* is three; of *a-4* is negative four; of *++++200* is two hundred; and of the null string and of *abc* the null string. The null string acts as zero in arithmetic operations. Arithmetic is arbitrary precision—there is no catering to the limits of any underlying hardware. The result of an arithmetic operation is itself at least such a decimal string with no leading zeros or plus signs for positive results and with zero represented as *0*.

#(ad,A,B) "Add" (Three arguments) The value of this function is the sum of the arithmetic values of arguments *A* and *B* with the nonnumeric prefix of *A* prefixed to the result. The prefix of *B* is lost.

#(su,A,B) "Subtract" (Three arguments) The value of this function is the result of subtracting the arithmetic value of argument *B* from that of argument *A*. The nonnumeric prefix of *A* is prefixed to the resultant decimal string, and the prefix of *B* is lost.

#(ml,A,B) "Multiply" (Three arguments) The value of this function is the result of multiplying the arithmetic values of arguments *A* and *B* and prefixing that result with the nonnumeric prefix of *A*. The prefix of *B* is lost.

#(dv,A,B,Z) "Divide" (Four Arguments) The value of this function is the numeric value of argument *A* divided by the numeric value of argument *B*, and the result is prefixed with the nonnumeric prefix of *A*. The prefix of *B* is lost. The division operation is done in integer mode, and only the integral portion of the quotient is retained. The value of the remainder is always nonnegative. If the value of *B* is zero, the function value is the argument *Z* in active mode regardless of the mode of the function call.

TRAC Boolean values operate in the same way as arithmetic values. The Boolean value of a string is the longest suffix of the string that consists entirely of zeros and ones—in other words, a binary string. Thus the Boolean value of *abc0100* is *0100*; of *12345678 90* is *0*; of *43210* is *10*; and of *abc* is null by convention.

#(bu,A,B) "Boolean Union" (Three arguments) The value of this function is the bitwise Boolean union of the Boolean values of the arguments *A* and *B*. If the two Boolean values are not the same length, the shorter is extended to the *left* with zeros to equalize the lengths. Any non-Boolean prefixes of the arguments are lost.

#(bi,A,B) "Boolean Intersection" (Three arguments) The value of this function is the Boolean intersection taken bitwise of the Boolean values of its arguments *A* and *B*. If the arguments are of unequal length, the longer is truncated from the left to equalize the lengths. Any non-Boolean prefixes of the arguments are lost.

#(*bc*,*A*) "Boolean Complement" (Two arguments) The value of this function is the bitwise Boolean complement of the Boolean value of its argument *A* and is the same length as that value. The non-Boolean prefix of the argument is lost.

#(*bs*,*S*,*A*) "Boolean Shift" (Three arguments) The value of this function is the Boolean shift of the Boolean value of argument *A*. The number of positions to shift is given by the arithmetic value of argument *S*. If the value is positive, the shift is to the left; if the value is negative, the shift is to the right. The function value is the same length as the Boolean value of *A*, and vacated positions are filled with zeros. The non-Boolean prefix of *A* is lost.

#(*br*,*S*,*A*) "Boolean Rotate" (Three arguments) The value of this function is the Boolean end-around rotation of the Boolean value of the argument *A*. The number of positions to rotate is given by the arithmetic value of argument *S* and is to the left if that value is positive and to the right if negative. Rotation does not change the length of the Boolean value. The non-Boolean prefix of *A* is lost.

#(*eq*,*A*,*B*,*T*,*F*) "Equality" (Five arguments) The value of this function is the argument *T* if the argument *A* is exactly equal, as a string, to the argument *B*, and is the argument *F* otherwise. Notice that *T* and *F* may be any strings whatever.

#(*gr*,*A*,*B*,*T*,*F*) "Greater" (Five arguments) The value of this string is the argument *T* if the arithmetic value of argument *A* is greater than the arithmetic value of argument *B*, and is argument *F* otherwise.

#(*sb*,*A*,*F1*,*F2*, · · ·) "Store Block" (Three or more arguments) This function stores the forms named by arguments *F1*, *F2*, . . . on some external storage medium. When all the forms have been stored, they are erased from forms storage and a new form with name *A* is created whose body is the "address" in external storage of the stored block of forms. If a form named *A* exists already, its old value is lost. The "address" of the block must be a string, and the forms on external storage must be accessible through any form having that string as body. The function value is null.

#(*fb*,*A*) "Fetch Block" (Two arguments) This null-valued function retrieves from external storage the block of forms whose "address" is the body of the form named *A*. The forms are brought back to forms storage, and if some of the forms are already in forms storage, their values are overwritten. The external block of forms remains accessible.

#(*eb*,*A*) "Erase Block" (Two arguments) This null-valued function releases the external storage holding the block of forms stored at the "address" given by the body of the form *A* so that the block is no longer accessible and deletes the form *A*.

#(*ln*,*S*) "List Names" (Two arguments) The value of this function is a list of all the form names currently resident in forms storage. The form names are separated by the string *S*.

#(*pf*,*N*) "Print Form" (Two arguments) This null-valued function prints the body of form *N* with the form pointer and segment markers shown. Mooers suggests printing the form pointer as ⟨↑⟩ and a segment marker as ⟨*i*⟩. Thus, the value of *pf* might be something like *aB*⟨*2*⟩*cdE*⟨*1*⟩*f*⟨↑⟩⟨*1*⟩*hiJ*, showing one instance of marker 2 and two instances of marker 1.

#(*tn*) "Trace On" (One argument) This null-valued function causes the processor to begin tracing the evaluation of functions. Each time a function is ready for evaluation, all its arguments are printed on the output device. If the processor is running on an interactive system, a pause is inserted after each argument list is printed. If the user types exactly a null line, the processor continues by evaluating the function; any other input causes a transfer to step 1 of the TRAC algorithm.

#(*tf*) "Trace Off" (One Argument) This null-valued function turns off the trace feature; if trace was not on, the function has no effect.

EXAMPLES

The following examples should give some flavor of the TRAC language. However, be wary of assuming that all the features of some function are illustrated by a particular example.

```
#(ds,AA,Cat)'
#(ds,BB,(#(cl,AA)))'
#(ps,(#(cl,bb)))'
#(ps,##(cl,BB))'
#(ps,#(cl,BB))'
```

The first line of this series of programs (notice that each line ends with an apostrophe, the metacharacter) causes a form with name *AA* and body *Cat* to be stored and a null line to be printed. The second line similarly creates a form with body *#(cl,AA)* and name *BB* and prints a null line. Now comes the interesting part. The *ps* function on the third line prints *#(cl,BB)* because the inner function is protected by parentheses, that on the fourth prints *#(cl,AA)* because the inner function is neutral, and that on the fifth line prints *Cat.*

```
#(dd,#(ln,(,)))'
```

This program will do exactly the same thing as an invocation of *#(da).* The arguments to *dd* are form names, and *ln* generates a list of all form names neatly separated by commas.

```
#(ds,Factorial,(#(eq,X,1,1,(#(ml,X,#(cl,Factorial,#(su,X,1)))))))'
#(ss,Factorial,X)'
#(cl,Factorial,5)'
```

These three lines define a factorial function in the standard recursive way, segment the string on the argument *X*, and then calculate the value of 5! Each of the sets of protective parentheses is necessary, the outer pair to protect the *eq* from evaluation during form creation and the inner to avoid doing the multiply if the test is true. Try deleting either pair or changing the *eq* to a neutral function.

```
#(cl,Factorial,5
#(ds,Factorial,(
#(eq,X,1,
1,
(#(ml,X,#(cl,Factorial,#(su,X,1)))))))
#(ss,Factorial,X))'
```

This example has the same effect as the last one; a factorial function is defined and 5! is calculated. But it takes advantage of the fact that the arguments of a function are evaluated before the function itself to bury the definition and segmentation of *Factorial* inside the call of *Factorial.* Notice that no comma is required after the argument *5* in the call of *Factorial* because *ds* always returns the null string as value.

Statement of the Theme Write a TRAC processor for your local system. The processor must implement the TRAC algorithm and built-in functions as described here. If your system has any sort of permanent file storage, use it for the external forms storage blocks. Blocks stored during one run of the processor should be accessible during later runs. It would be wise to include many internal debugging aids in your processor.

Performance Practice Each system has its own conventions about character sets and end-of-line devices. In the algorithm there is a step that deletes unprotected carriage returns, line feeds, and tabulation characters from scanned material. This feature allows input material to be typed without

worrying that line breaks occasioned by line-length limits of physical devices will be incorporated into program material willy-nilly. But it also implies that such characters must be explicitly represented in the input stream so that programs that wish to format their output may do so. For example, if *CR-LF* stands for the carriage return-line feed sequence, the input *OneCr-LFTwo* will print *OneTwo* on one line, but the input *One(Cr-LF)Two* will print *One* on one line and *Two* on the next. Make sure that your processor can handle such situations correctly.

A rather cavalier attitude was taken during the algorithm description toward the marking and recovery of arguments and functions in the neutral string. In fact, it is important to maintain an exact record of functions and arguments, as well as the order in which they occur. The easiest way to do so is with a stack. The stack is entirely internal to the processor and is reset to empty every time the algorithm goes through step 1. Each time a new function beginning is marked by the algorithm, a new function block is stacked. A function block has a fixed base section that includes, at a minimum, the stack location of the next lower function block, the mode of the function, the location in the neutral string of the first character of the function, the number of arguments begun and completed for the function, and the start location in the neutral string of the beginning of the argument currently being scanned. Above the fixed section there is an entry for each finished argument with the beginning and end of the argument. Some of this information may be implicit in other items. The most common implementation errors involve mishandling null strings and losing track of information about functions low in the stack while manipulating ones higher up.

Similarly, greater care is necessary with forms storage than was indicated in the preceding discussion. Since both name and body are strings, it makes sense to store them together in some sort of string storage. Also needed is storage for the form pointer and the segment markers, plus a way to look up forms. The easiest way to handle this situation is to allocate a large forms space and store the forms as a linked list. Each form will have a fixed head section that will store a pointer to the next form, the form pointer, the starting location and length of the form name, and the start and length of the body. The name and body can be stored immediately following the body, and a segment marker can be some character pair surrounding an integer (making sure that there is some other way to represent the marker pair as legitimate characters). Form lookup can be done by running down the list of forms and form deletion by unchaining the form and moving the surrounding forms to close up the hole. Alternative techniques for forms storage exist, each with its own advantages and disadvantages.

There is a difficulty with storage allocation for the processor. Each of the neutral string, active string, argument stack, and forms storage might need more storage than can be predicted in advance. If fixed allocations are made to these areas, any one of them might run out of space while the others still had enough to spare. A standard technique to solve this problem is available. Consider first the neutral and active strings. Because the neutral string occurs naturally to the left of the active string, allocate one area to both of them with the neutral string growing from the left end to the right and the active string from the right leftward. In this way, there can be no storage overflow until both the neutral and active string combined use up all the storage allocated to the pair. The argument stack and forms storage can be paired in the same way, and one of the interpair gaps can be used to hold temporarily strings that are being copied from one place to another.

A further refinement may be added. Allocate the storage for the run-time structures in one big area in the following order from left to right: the neutral string growing rightward; a free-space gap; the active string growing leftward; without any gap the argument stack growing rightward; a free-space gap; and forms storage growing leftward. Now if the active-neutral strings pair run out of space while the argument stack and forms storage have some left, move the active string and argument stack rigidly to the right some distance, thus donating space from the right free gap to the left. Obviously, the same trick will work going the other way so that the processor will not be out of space until there is really no space to be had.

Throughout the discussion of storage allocation it has been assumed that we could obtain a large, amorphous chunk of memory that could store characters, pointers, integers, logical flags, and so on at will. In at least some languages, such as PASCAL and ALGOL, doing so is difficult if not impossible. Of course, in assembly language all memory is amorphous, but the design freedom is usually paid for in mistakes in referencing memory. This tradeoff between convenience and security is common and will have to be faced when you choose the language for your implementation.

Your processor should be instrumented so that it observes, records, and reports such items as the number of times each algorithm step is executed, the number of times each built-in function is called, the maximum length of each internal storage area, the number of times storage is repacked because of overflow, and the number of times each internal routine is called. This instrumentation can serve three purposes. First, it can serve as a debugging aid. If there is some a priori relationship among counts that is not holding or if some counts are suspiciously out of line, a probable error has been spotted. Second, the counts can guide improvement in those routines that are costing the most time and tuning of those parameters that control space. Finally, suitably annotated and explained, the counts can guide a TRAC user in correct and efficient use of the processor. Instrumentation counts should probably be presented to the user at the end of each run.

Orchestration Here is another problem with which to study the influence of language on programming. If you choose a higher-level language like PASCAL with its many built-in safeguards, you will find most of the processor absurdly easy to code, but you will probably pay for this ease in efficiency and trouble in arranging storage allocation. Alternatively, if you choose assembly language, you may well find that your program is efficient in both space and time but that it is long-winded and hard to debug and that you will have to build many routines that other languages would provide for you. An intermediate-level language—XPL, BLISS, or FORTRAN—can combine the advantages (and probably disadvantages) of the other two approaches. Or perhaps some parts of the program should be written in one language and the rest in another. In any case, your documentation should discuss the reasons for your language choice and your reflections in hindsight.

Playing Time This problem should take one person 7 weeks, two people 4 weeks, or three people 3 weeks.

Variations on the Theme One obvious extension is to allow a function whose first argument is a form name to be a call on that form—that is, *#(XYZ, · · · , · · · · · ·)* is converted into *#(cl,XYZ, · · · , · · · · · ·)*. If, by convention, undefined forms are assumed to have null strings for bodies, this situation automatically takes care of the case of the null value for undefined functions. And if the forms are searched before the built-in functions, the user can then overlay a built-in function with a customized one.

The other obvious way to extend TRAC is to allow more built-in functions. We will suggest two sets of them. The first will add some amenities to the string- and character-handling functions. The second set will broaden the input/output capabilities.

STRING AND CHARACTER HANDLING

#(qm) "Query Metacharacter" (One argument) The value of this function is the one character string consisting of the current metacharacter.

#(sl,A) "String Length" (Two arguments) The value of this function is the length of the string *A* expressed as a decimal string. The length of the null string is zero.

#(cd,C) "Character to Decimal" (Two arguments) The value of this function is a decimal string

giving the position in the local implementation's character set of the first character of the argument *C*. If *C* is null, so is the value.

#(dc,D) "Decimal to Character" (Two arguments) The value of this function is the single character string whose index in the local implementation's character set is given by the arithmetic value of argument *D*. If argument *D* does not index any character, the function value is null.

#(sr,N) "Segment Range" (Two arguments) The value of this function is a decimal string giving the ordinal number of the highest-numbered segment marker of the form named by argument *N*. If form *N* does not exist or if it has no segment gaps, the function value is zero.

#(cr,R1,R2,V) "Change Radix" (Four arguments) The value of this function is calculated by finding the arithmetic value of argument *V* in radix *R1* and rewriting it in radix *R2*. The possible digits for the potential radices are, in ascending order, *0*, *1*, . . . , *9*, *A*, *B*, . . . , *Z*. Thus binary uses *0* and *1*, decimal *0* through *9*, and hexadecimal *0* through *F*. The arithmetic value of a string in a given radix is the longest suffix of that string that optionally begins with + or - and otherwise consists entirely of legal digits from the radix. This rule is the same as the one for TRAC decimal values. The arguments *R1* and *R2* name radices by giving the largest legal digit in the radix (for example, decimal is *9*, binary *1*, and hexadecimal *F*). If either *R1* or *R2* is not a single character in the range *1* through *Z*, the function value is null.

INPUT/OUTPUT

#(hl) "Halt" (One argument) This null-valued function causes an immediate exit from the TRAC processor.

#(ai,F,Z) "Assign Input" (Three arguments) This null-valued function assigns the input device to the file named by argument *F* and sets the input file pointer to point to the first line of the file. If the file cannot be assigned, the function value is the argument *Z* in active mode regardless of the function call mode. Each succeeding call to *rs* causes the device to read up to the next metacharacter and to advance the input file pointer. If argument *F* is null, control is returned to the default input file (the console in an interactive system).

#(ao,F) "Assign Output" (Two arguments) This null-valued function assigns the output device to the file named *F* and, if such a file does not exist, creates it. If argument *F* is null, it returns the output device to the default device (the console in interactive systems).

#(sp,N) "Set Pointer" (Two arguments) This null-valued function sets the input file pointer on the record that follows *N-1* metacharacters. If this does not describe a record in the file or if the file is the console, the function has no effect.

#(rp) "Read Pointer" (One argument) This function returns as value a decimal string giving the current value of the input file pointer. If the input device is attached to the console, the value is null.

#(rs,Z) "Read String" (Two arguments) This function modifies the earlier *rs* function so that if no input is available from the current input file, the value is the argument *Z* in active mode regardless of the mode of the function call.

REFERENCES

Brown, P. J. *Macro Processing and Techniques for Portable Software.* Wiley, New York, NY, 1975.

Mooers, Calvin N. "Computer Software and Copyright," *Computing Surveys*, 7, 1, pp. 45-72, 1975.

Although not strictly about TRAC, this article's concerns were engendered, at least in part, by fears for TRAC's purity. Mooers states, if not the law, at least a clear position on the issues of protection for programs.

Mooers, Calvin N. "How Some Fundamental Problems are Treated in the Design of the TRAC Language." In *Symbol Manipulation Languages and Techniques*, edited by D. G. Bobrow. North-Holland Publishing Co., Amsterdam, pp. 178-190, 1968.

In this paper Mooers discusses some of the design decisions for TRAC. There is a comparison of TRAC with some other languages with similar purposes, notably LISP. After you have learned some of the fundamentals of TRAC, this paper may give you a bit of philosophy to leaven the knowledge.

—— "TRAC, A Procedure-Describing Language for the Reactive Typewriter." *CACM*, 9, 3, pp. 215-219, 1966.

Nelson, Theodor H. *Computer Lib* or *Dream Machines.* Hugo's Book Service, Chicago, IL, 1974.

A *Whole Earth Catalog* of computer lore. This is a fascinating miscellany. It even has two titles, depending on whether you start reading from the front or the back. Nelson thinks TRAC is the wave of the future and provides a nice primer.

Strachey, C. "A General Purpose Macrogenerator." *Comput. J.*, 8, 3, pp. 225-241, 1966.

Mooers' paper describes the TRAC language in its reference form. The problem description is largely a paraphrase with some obvious improvements made to the built-in functions. Strachey's paper describes a very similar language that substitutes a more powerful definition facility for the built-in functions and contains a listing of a model processor. The two papers were written independently and are a striking example of historical imperative.

Wegner, Peter. *Programming Languages, Information Structures, and Machine Organization.* McGraw-Hill, New York, NY, 1968.

Both Brown and Wegner discuss TRAC and set it in the context of other macroprocessors and computer science generally.

Solutions

The solutions presented here are complete programs, but they do not attempt to show everything that a student must supply. We discuss the craft of programming as applied to these specific problems, particularly some fine practical points that are often ignored in textbooks. Students would provide more external documentation, more output, and more demonstration of correctness. Also notice that our choice of languages does not necessarily constitute endorsement.

29

Map Coloring Made Easy

or...

A COMPLETE PROBLEM SOLUTION

In this chapter we present a complete program for the map-coloring problem of Chapter 3. Any reader who has tried will appreciate how difficult it is to write up a program without making the algorithms seem easy. Once a solution to a puzzle is known, the reader's natural reaction is "I could have thought of that" and not "Oh, how clever." And, of course, programs are a type of puzzle solution. What is always missing from a writeup is the sense of exploration, the history of false starts and backtracks, the stupid mistakes that hid the obvious path. And if we try to supply all the development here, you will be alternately bored to tears and hysterical about our ignorance. So this will be a relatively straight walk from problem to solution with only a few glances at the side paths that we might have taken.[1]

The problem as stated is to color an arbitrary finite map or undirected graph with a minimum number of colors such that no two adjacent regions or nodes are the same color. Although it is not necessary to decide exact input and output formats or internal data representation now, we should figure out what properties of a graph will be needed by any program. Surely we must know how many nodes the graph has, we must be able to name each node in some orderly way, we must be able to color a node and then query that color later, we must be able to decide if two nodes are adjacent, and we must be able to generate a large number of different colors. The easiest way to provide all these facilities is to assume that the nodes are named with the integers 1, 2, . . . , up to n, where there are n nodes, and that the colors are similarly named with the positive integers (certainly a large number of colors will be available this way). It will be wise to postpone discussing exactly how adjacency should be tested.

Coloring might be approached in two ways: we could assume that each node already has a distinct color and try to eliminate some colors, or we could assume that no node is colored and try to add as few colors as possible. Either way we encounter a bad theoretical result (or perhaps lack of a result): nobody knows how to color a map without, in the worst case, enumerating all possible colorings with the minimal number of colors. Most experts believe that there is no faster way to

[1] To reinforce the analogy between writing and programming, we will admit that this chapter had more than a few false starts itself.

color maps than by searching all possibilities. In different words, no matter how clever your map-coloring program is and no matter how fast it is *on average*, there will be worst-case maps of n nodes requiring k colors that will take your program about n^k units of time to color. It is possible that someone may become lucky and find an algorithm without this horrendously expensive worst-case behavior, but theoreticians think it improbable. So we should try a simple, fast program rather than a supercomplicated one that is unlikely to do any better.[2]

The proposed solution relies on the observation that if a subgraph cannot be colored with k colors, then surely the whole graph cannot be colored with k colors. Each step of the algorithm tries to add one more node to the already colored subgraph. If that addition is not successful, the currently colored subgraph is recolored every possible way by using the same colors. If the new node still cannot be added, the number of colors is increased by one because a subgraph has been found that cannot be successfully colored. In the algorithm description we will assume that there is a vector **color** that contains the color currently assigned to each node. A logical function **connect** tells if node i is connected to node j.

ALGORITHM FOR GRAPH COLORING

1. (Initialization.) Assume that the graph has **totalnodes** nodes, that **topcolor** contains the number of the highest color yet assigned, and that zero means no color has yet been assigned to a node. For each node **n**, set **color[n]** to zero (that is, set all nodes uncolored). Set **topcolor** to zero and **currentnode** to one.

2. (Main loop.) While **currentnode** is less than or equal to **totalnodes**, do steps 3 through 7. This step runs the loop until all the nodes have been colored. Each time a loop iteration begins, all the nodes from 1 to **currentnode-1** have a legal coloring with **topcolor** colors.

3. (Prepare one node.) Increment **color[currentnode]** by one. Set the Boolean variable **loopflag** to the value of the relation **color[currentnode]** $\leqslant$ **topcolor.** Now the node under consideration has a nonzero color, and it is necessary to test the compatibility of **currentnode** with its neighbors. Notice that a node being added for the first time will always have color zero. Adding one to zero will give the node a legitimate color. While **loopflag** is true, do steps 4 and 5.

4. (Test colors of adjacent nodes.) Set **loopflag** to *false* and set **i** to one. While **i** is less than **currentnode,** do step 5.

5. (Check each neighbor's color.) If node **i** is connected to node **currentnode** [that is, if **connect(i, current)** is true] and if **color[i]** is the same as **color[currentnode]**, then **currentnode** is colored illegally. In this case, set **i** to **currentnode** to terminate the loop begun in step 4, increment **color[currentnode]** by one to try the next color in sequence, and set **loopflag** to the value of **color[currentnode]** $\leqslant$ **topcolor.** Otherwise simply increment **i** by one to test the next neighbor. Notice that the assignment to **loopflag** overrides the assignment in step 4, but that it may still leave **loopflag** with a *false* value. Also, if the loop begun in step 4 terminates normally (that is, without the forced assignment of **currentnode** to **i**), then the loop begun in step 3 will also be exited. It is important that

[2] This program illustrates the importance of theoretical computer science to practicing programmers. Many other common combinatorial problems, most notably the traveling salesman problem, share this extremely expensive worst-case behavior with map coloring. But quite often a small change in a problem makes some very efficient solution possible. A working programmer need not know all the problems and solutions, but he must recognize the hard problems and go to the literature or an expert for answers. Incidentally, if an *optimal, minimal, maximal,* or *exact* solution is not necessary, then heuristics that provide good approximate solutions quickly are worth trying and exist for many problems.

only nodes with numbers strictly less than **currentnode** be used in the legality check because higher nodes have not yet been colored.

6. (Go ahead or backtrack?) If **currentnode** has been colored correctly, we want to move on to the next node; otherwise we must back up. So if **color[currentnode]** > **topcolor**, set **color[currentnode]** to zero and decrement **currentnode** by one; otherwise increment **currentnode** by one. Notice that the color of a node with number higher than **currentnode** is always zero, and when we back up to an already colored node, we continue advancing its color from where we last left off.

7. (Add a color if necessary.) If **currentnode** is zero, increment **topcolor** by one and set **currentnode** to one. If we have backed all the way down, the number of colors must be increased.

It is easy to see that this algorithm must terminate. At the very least it will stop when every node has a separate color. It should also be reasonably clear that all the nodes from one to **currentnode-1** are colored correctly each time that the main loop starts. Slightly less obvious is the fact that **topcolor** will be increased only if there is no way to color some initial subgraph with the current value of **topcolor**. Experiments on some small graphs should help, however. Notice that the algorithm works correctly for the graph of zero nodes, for graphs with completely disconnected nodes, and for graphs with all nodes connected.

AN ACTUAL IMPLEMENTATION

We use FORTRAN for our implementation. Certainly FORTRAN is a very poor language with weak data definition and control structure facilities. However, we choose it precisely to show that a well-structured, clear program can be written even in an awkward language. Ideally, every program could be written in one of the newer, more expressive languages; in practice, most programmers will need to work in archaic languages from time to time and must not succumb to the temptation to program badly because of bad tools.

The logical function **connect** can be represented in FORTRAN as a two-dimensional logical array in which the entry indexed by I and J is *true* if and only if node i is connected to node j. The first item in the input will be a record with the number of nodes in the graph to be colored. From then on sets of records will be read, each set describing the connections for one node. The first record of a set will contain a node number and the number of neighbors that the node has; the remaining records will be a listing of the neighbors in any order. Input will end with a node number of zero. Nodes may be described in any order, and the description of a single node may be broken into several sections. When node i is connected to node j, node j will be connected automatically to node i. The only errors will be node numbers out of range and attempts to connect a node to itself. Now we reproduce the actual program.

```
1 C
2 C         A PROGRAM TO COLOR GRAPHS.
3 C
4 C
5 C         AUTHOR -- CHARLES WETHERELL, 12 JULY 1976.
6 C         LAST DATE MODIFIED -- 14 JULY 1976.
7 C
8 C         THIS PROGRAM COLORS ARBITRARY FINITE UNDIRECTED GRAPHS
```

```
C        WITH A MINUMUM NUMBER OF COLORS.  THE METHOD USED IS
C        STANDARD BACKTRACKING WITH NO HEURISTICS.  THE GRAPH IS
C        REPRESENTED BY A LOGICAL ADJACENCY MATRIX.  BECAUSE OF THE
C        POLYNOMIAL COMPLETENESS OF GRAPH COLORING, SOME WORST CASES
C        MAY TAKE TIME EXPONENTIAL IN THE NUMBER OF INPUT NODES.
C        DISCUSSION OF THE PROBLEM AND ITS SOLUTION CAN BE FOUND IN
C
C           WETHERELL, C.  ETUDES FOR PROGRAMMERS.  PRENTICE-HALL,
C              ENGLEWOOD CLIFFS, NJ.  1978.
C
C        WHICH ALSO CONTAINS A LARGER BIBLIOGRAPHY.
C
C
C        THE ALGORITHM IS IMPLEMENTED AS ONE LARGE PROGRAM AND ALL
C        THE VARIABLES LISTED IN THIS GLOSSARY ARE GLOBAL TO THE
C        WHOLE PROGRAM.  THE FIRST SECTION IS VARIABLES USED WITH THE
C        GRAPH AND IN THE COLORING ALGORITHM.
C
C        MAXNOD -- THE MAXIMUM NUMBER OF NODES ALLOWED IN A GRAPH.
C                  FORTRAN WILL NOT ALLOW US TO USE THIS AS AN ARRAY
C                  BOUND, BUT IT CAN BE USED TO CHECK THE LEGITIMACY
C                  OF INPUT DATA.
C        TOTNOD -- THE NUMBER OF NODES IN THE GRAPH TO BE COLORED.
C        CONNCT -- THE LOGICAL NODE ADJACENCY MATRIX.
C        COLOR  -- A VECTOR INDEXED BY NODES GIVING THE COLOR OF EACH
C                  NODE.
C        TOPCLR -- THE NUMBER OF COLORS IN USE AT ANY TIME.
C        CURNOD -- THE NODE CURRENTLY NEEDING A COLOR.
C        LOPFLG -- A LOGICAL FLAG USED TO CONTROL AN INNER LOOP OF
C                  THE ALGORITHM.
C        NODCNT -- A LOOP COUNTER FOR THE OUTER LOOP.
C        CLRCNT -- A LOOP COUNTER FOR THE MIDDLE LOOP.
C        TSTCNT -- A LOOP COUNTER FOR THE INNER LOOP.
C
C        THESE VARIABLES ARE USED TO READ THE INPUT AND TO PRINT THE
C        INPUT ECHO AND THE OUTPUT.
C
C        MASNOD -- THE NODE ABOUT TO HAVE ITS CONNECTIONS READ.
C        NUMNBR -- THE NUMBER OF NEIGHBORS MASNOD HAS.
C        NGHBOR -- A VECTOR OF NEIGHBORS FOR MASNOD.
C        LINE   -- A HOLD AREA FOR A ROW OF INTEGERS TO BE PRINTED.
C        TEXT   -- A HOLD AREA FOR A ROW OF CHARACTERS TO BE PRINTED.
C        HYPHEN -- THE CHARACTER -.
C        BLANK  -- THE CHARACTER BLANK.
C        STAR   -- THE CHARACTER *.
C
C        GENERAL USE VARIABLES.
C
```

```
C         RDUNIT -- THE UNIT NUMBER FOR THE INPUT FILE.
C         WRUNIT -- THE UNIT NUMBER FOR THE OUTPUT FILE.
C         I,J,K  -- GENERAL USE INDEX VARIABLES.
C
C
C         FORTRAN REQUIRES DATA STATEMENTS AFTER DECLARATIONS.
C         THE COMPILER USED FOR THIS PROGRAM REQUIRES CHARACTER
C         STRING CONSTANTS TO BE SET OFF WITH APOSTROPHES INSTEAD OF
C         THE NORMAL HOLLERITH NOTATION.
C
      INTEGER MAXNOD
      INTEGER TOTNOD
      LOGICAL CONNCT(70, 70)
      INTEGER COLOR(70)
      INTEGER TOPCLR
      INTEGER CURNOD
      LOGICAL LOPFLG
      INTEGER NODCNT, CLRCNT, TSTCNT
C
      INTEGER MASNOD
      INTEGER NUMNBR
      INTEGER NGHBOR(70)
      INTEGER LINE(70)
      INTEGER TEXT(70)
      INTEGER HYPHEN, BLANK, STAR
C
      INTEGER RDUNIT, WRUNIT
      INTEGER I, J, K
C
C
      DATA MAXNOD/70/
      DATA HYPHEN/'-'/, BLANK/' '/, STAR/'*'/
      DATA RDUNIT/1/, WRUNIT/2/
C
C
C         THE FORTRAN IN WHICH THIS PROGRAM WAS WRITTEN REQUIRES
C         DYNAMIC ATTACHMENT OF DATA FILES TO LOGICAL I/O UNITS USING
C         THE SYSTEM SUBROUTINE ASSIGN.  AFTER THIS IS DONE, ALL I/O IS
C         LIKE STANDARD FORTRAN.
C
      CALL ASSIGN('GRAPHDATA ', RDUNIT)
      CALL ASSIGN('COLORGRAPH ', WRUNIT)
C
C
C         NOW THE ADJACENCY MATRIX IS CLEARED TO ALL FALSE AND THE
C         ACTUAL NUMBER OF NODES IS READ.  IF THE NUMBER OF NODES IS
C         ILLEGAL, THE PROGRAM STOPS IMMEDIATELY.
C
```

```
105         DO 20 I = 1, MAXNOD
106            DO 10 J = 1, MAXNOD
107               CONNCT(I, J) = .FALSE.
108 10         CONTINUE
109 20      CONTINUE
110         READ (RDUNIT, 1000) TOTNOD
111         IF (TOTNOD .GE. 0 .AND. TOTNOD .LE. MAXNOD) GOTO 30
112            WRITE (WRUNIT, 2000) TOTNOD
113            STOP 1
114 30      CONTINUE
115         WRITE (WRUNIT, 2010) TOTNOD
116 C
117 C
118 C          EACH NEW SET OF CONNECTION DATA HAS A HEADER RECORD WITH A
119 C          NODE NUMBER AND THE NUMBER OF NEIGHBORS.  THE HEADER IS
120 C          FOLLOWED BY RECORDS CONTAINING THE NEIGHBORS.  INPUT STOPS
121 C          WHEN A NODE NUMBER IS ZERO.  ANY ERROR EXCEPT CONNECTING A
122 C          NODE TO ITSELF IS FATAL BECAUSE THE NUMBER OF NEIGHBORS MAY
123 C          BE CONFUSED AND CAUSE A HEADER TO BE MISSED.
124 C          PLEASE NOTE THE INSERTION JUST BEFORE LABEL 65 TO TEST
125 C          FOR CONNECTING NODES OUT OF RANGE.  THIS ERROR IS NOT FATAL.
126 C
127 40      CONTINUE
128         READ (RDUNIT, 1010) MASNOD, NUMNBR
129         IF (MASNOD .EQ. 0) GOTO 90
130         IF (MASNOD .GE. 1 .AND. MASNOD .LE. TOTNOD) GOTO 50
131            WRITE (WRUNIT, 2020) MASNOD
132            STOP 2
133 50      CONTINUE
134         IF (NUMNBR .GE. 1 .AND. NUMNBR .LE. TOTNOD) GOTO 60
135            WRITE (WRUNIT, 2030) NUMNBR
136            STOP 3
137 60      CONTINUE
138         READ (RDUNIT, 1020) (NGHBOR(I), I = 1, NUMNBR)
139         DO 80 I = 1, NUMNBR
140            J = NGHBOR(I)
141            IF (J .GE. 1 .OR. J .LE. TOTNOD) GOTO 65
142               WRITE (WRUNIT, 2035) J, MASNOD
143               GOTO 80
144 65         CONTINUE
145            IF (MASNOD .NE. J) GOTO 70
146               WRITE (WRUNIT, 2040) MASNOD
147               GOTO 80
148 70         CONTINUE
149            CONNCT(MASNOD, J) = .TRUE.
150            CONNCT(J, MASNOD) = .TRUE.
151 80      CONTINUE
152         GOTO 40
```

```
153 C
154 C
155 C        THE ECHO OUTPUT IS A PICTURE OF THE ADJACENCY MATRIX.  SINCE
156 C        FORTRAN PERFORCE PRINTS T OR F FOR LOGICAL VALUES AND SINCE
157 C        MATRICES WITH SOLID CHARACTERS ARE HARD TO READ, WE TURN
158 C        EACH ROW OF THE MATRIX INTO A ROW OF *'S FOR TRUE AND BLANKS
159 C        FOR FALSE.  THERE IS ALSO SOME MANIPULATION DONE HERE TO GET
160 C        NICE HEADINGS.  NONE OF THE PRINTOUT WILL BE VERY PRETTY IF
161 C        THERE ARE MORE NODES THAN CAN FIT ACROSS A PAGE.  ADVANCED
162 C        TECHNIQUES FOR PRINTING LARGE MATRICES COULD BE USED HERE.
163 C
164 90     CONTINUE
165        DO 100 I = 1, TOTNOD
166           LINE(I) = MOD(I, 10)
167 100    CONTINUE
168        WRITE (WRUNIT, 2050) (LINE(I), I = 1, TOTNOD)
169        DO 110 I = 1, TOTNOD
170           TEXT(I) = HYPHEN
171 110    CONTINUE
172        WRITE (WRUNIT, 2060) (TEXT(I), I = 1, TOTNOD)
173        DO 130 I = 1, TOTNOD
174           DO 120 J = 1, TOTNOD
175              TEXT(J) = BLANK
176              IF (CONNCT(I,J)) TEXT(J) = STAR
177 120       CONTINUE
178           WRITE (WRUNIT, 2070) I, (TEXT(J), J = 1, TOTNOD)
179 130    CONTINUE
180 C
181 C
182 C        NOW THAT THE DATA HAS BEEN READ AND ECHOED, WE CAN PROCEED
183 C        WITH THE ALGORITHM.  THE FIRST STEP IS INITIALIZATION OF ALL
184 C        VARIABLES INCLUDING LOOP COUNTERS.
185 C
186        DO 140 I = 1, TOTNOD
187           COLOR(I) = 0
188 140    CONTINUE
189        TOPCLR = 0
190        CURNOD = 1
191        NODCNT = 0
192        CLRCNT = 0
193        TSTCNT = 0
194 C
195 C
196 C        WE WILL NOT COMMENT THE MAIN LOOP EXTENSIVELY SINCE IT IS AN
197 C        EXACT TRANSLITERATION OF THE ALGORITHM IN THE PROBLEM
198 C        DESCRIPTION.  ALSO, SUCH COMMENTARY BREAKS THE FLOW OF THE
199 C        CODING AND LOOKS TERRIBLE WRITTEN ALL IN CAPITALS.  NOTICE
200 C        THAT SEVERAL LOGICAL TESTS ARE INVERTED BECAUSE OF THE WAY
```

```
C        FORTRAN IF STATEMENTS WORK AND THAT INDENTATION OF
C        CONDITIONALS IS A RATHER DIFFICULT AFFAIR.  THE FOLLOWING
C        TABLE OF STEP NUMBERS AND LABELS MAY HELP TO COORDINATE
C        CODE WITH ALGORITHM.
C
C           STEP 2   LABEL 150
C           STEP 3   LABEL 160
C           STEP 4   LABEL 180
C           STEP 5   LABEL 200
C           STEP 6   LABEL 230
C           STEP 7   LABEL 250
C
  150   CONTINUE
        IF (CURNOD .GT. TOTNOD) GOTO 260
  160      CONTINUE
           NODCNT = NODCNT + 1
           COLOR(CURNOD) = COLOR(CURNOD) + 1
           LOPFLG = COLOR(CURNOD) .LE. TOPCLR
  170      CONTINUE
           IF (.NOT. LOPFLG) GOTO 230
  180         CONTINUE
              CLRCNT = CLRCNT + 1
              LOPFLG = .FALSE.
              I = 1
  190         CONTINUE
              IF (I .GE. CURNOD) GOTO 220
  200            CONTINUE
                 TSTCNT = TSTCNT + 1
                 IF (CONNCT(CURNOD, I) .AND.
     1                  COLOR(CURNOD) .EQ. COLOR(I)) GOTO 210
                    I = I + 1
                    GOTO 190
  210            CONTINUE
                    I = CURNOD
                    COLOR(CURNOD) = COLOR(CURNOD) + 1
                    LOPFLG = COLOR(CURNOD) .LE. TOPCLR
                    GOTO 190
  220         CONTINUE
              GOTO 170
  230      CONTINUE
           IF (COLOR(CURNOD) .LE. TOPCLR) GOTO 240
              COLOR(CURNOD) = 0
              CURNOD = CURNOD - 1
              GOTO 250
  240      CONTINUE
              CURNOD = CURNOD + 1
  250      CONTINUE
           IF (CURNOD .GE. 1) GOTO 150
```

```
249              TOPCLR = TOPCLR + 1
250              CURNOD = 1
251              GOTO 150
252 C
253 C
254 C         NOW THE GRAPH IS COLORED AND ALL THAT REMAINS IS TO PRINT
255 C         THE RESULTS.  WE PRINT FOR EACH COLOR A LIST OF NODES WHICH
256 C         ARE PAINTED THAT COLOR.  THERE IS ONCE AGAIN A LITTLE
257 C         MANIPULATION TO BUILD THE OUTPUT LINES.
258 C
259 260    CONTINUE
260        WRITE (WRUNIT, 2080) TOPCLR
261        I = 1
262 270    CONTINUE
263        IF (I .GT. TOPCLR) GOTO 290
264           J = 0
265           DO 280 K = 1, TOTNOD
266              IF (COLOR(K) .NE. I) GOTO 280
267                 J = J + 1
268                 LINE(J) = K
269 280       CONTINUE
270           WRITE (WRUNIT, 2090) I, (LINE(K), K = 1, J)
271           I = I + 1
272           GOTO 270
273 290    CONTINUE
274        WRITE (WRUNIT, 2100) NODCNT, CLRCNT, TSTCNT
275        STOP
276 C
277 C
278 C         INPUT FORMATS.
279 C
280 1000   FORMAT(I4)
281 1010   FORMAT(2I4)
282 1020   FORMAT(20I4)
283 C
284 C
285 C         OUTPUT FORMATS.
286 C
287 2000   FORMAT(I5, 32H IS NOT A LEGAL NUMBER OF NODES.)
288 2010   FORMAT(24H THE NUMBER OF NODES IS  I4)
289 2020   FORMAT(I5, 27H IS AN ILLEGAL NODE NUMBER.)
290 2030   FORMAT(I5, 30H IS AN ILLEGAL NEIGHBOR COUNT.)
291 2035   FORMAT(I5, 25H IS ILLEGAL CONNECTED TO  I4)
292 2040   FORMAT(6H NODE  I4, 27H MAY NOT BE SELF-CONNECTED.)
293 2050   FORMAT(/18H ADJACENCY MATRIX: //
294       1       6X, 126I1)
295 2060   FORMAT(6X, 126A1)
296 2070   FORMAT(I5, 1H| 126A1)
```

```
297 2080  FORMAT(//32H THE NUMBER OF COLORS NEEDED IS  I3//)
298 2090  FORMAT(7H COLOR  I3, 8H NODES  (10I5))
299 2100  FORMAT(24H THE OUTER LOOP COUNT =  I8/
300       1       24H THE MIDDLE LOOP COUNT = I8/
301       2       24H THE INNER LOOP COUNT =  I8)
302        END
```

The sample data are taken from the map of Chapter 3. The states are numbered from top to bottom in columns from left to right. On the right of each header card is the name of the associated state; the FORMAT statement is unconcerned about anything beyond the eighth character. The data are split into two columns to save printing space here but otherwise appears exactly as they would go into the program.

```
49
 1   2       WASHINGTON
 2   5
 2   4       OREGON
 1   3   4    5
 3   3       CALIFORNIA
 2   4   7
 4   5       NEVADA
 2   3   5    6    7
 5   6       IDAHO
 1   2   4    6    8    9
 6   5       UTAH
 4   5   7    9   10
 7   4       ARIZONA
 3   4   6   11
 8   4       MONTANA
 5   9  12   13
 9   6       WYOMING
 5   6   8   10   13   14
10   6       COLORADO
 6   9  11   14   15   16
11   4       NEW MEXICO
 7  10  16   17
12   3       NORTH DAKOTA
 8  13  18
13   6       SOUTH DAKOTA
 8   9  12   14   18   19
14   6       NEBRASKA
 9  10  13   15   19   20
15   4       KANSAS
10  14  16   20
16   6       OKLAHOMA
24  28  29   30
26   4       MISSISSIPPI
21  22  27   31
27   4       ALABAMA
26  31  32   33
28   3       MICHIGAN
23  25  29
29   5       OHIO
25  28  30   34   35
30   6       KENTUCKY
20  25  29   31   35   36
31   8       TENNESSEE
20  21  26   27   30   32   36   37
32   5       GEORGIA
27  31  33   37   38
33   2       FLORIDA
27  32
34   6       PENNSYLVANIA
29  35  39   40   41   42
35   5       WEST VIRGINIA
29  30  34   36   42
36   6       VIRGINIA
30  31  35   37   42   43
37   4       NORTH CAROLINA
31  32  36   38
38   2       SOUTH CAROLINA
32  37
39   5       NEW YORK
34  40  44   45   46
40   3       NEW JERSEY
34  39  41
41   3       DELAWARE
```

```
10  11  15  17  20  21
17   4      TEXAS
11  16  21  22
18   4      MINNESOTA
12  13  19  23
19   6      IOWA
13  14  18  20  23  24
20   8      MISSOURI
14  15  16  19  21  24  30  31
21   6      ARKANSAS
16  17  20  22  26  31
22   3      LOUISIANA
17  21  26
23   4      WISCONSIN
18  19  24  28
24   4      ILLINOIS
19  20  23  25
25   4      INDIANA
```

```
34  40  42
42   5      MARYLAND
34  35  36  41  43
43   2      WASHINGTON DC
36  42
44   3      VERMONT
39  45  48
45   5      MASSACHUSETTS
39  44  46  48  49
46   3      CONNECTICUT
39  45  49
47   1      MAINE
48
48   3      NEW HAMPSHIRE
44  45  47
49   2      RHODE ISLAND
45  46
 0
```

The program output looks best when printed on long output paper and hung on the wall; the aspect ratio of the printing here destroys some of the symmetry. Notice how printing the solution horizontally saves considerable space without losing any information. We were originally tempted to print node numbers and colors vertically, which would have been slightly easier but not nearly as tidy. The actual output is shown on the page at right.

NOTES ON THE PROGRAM

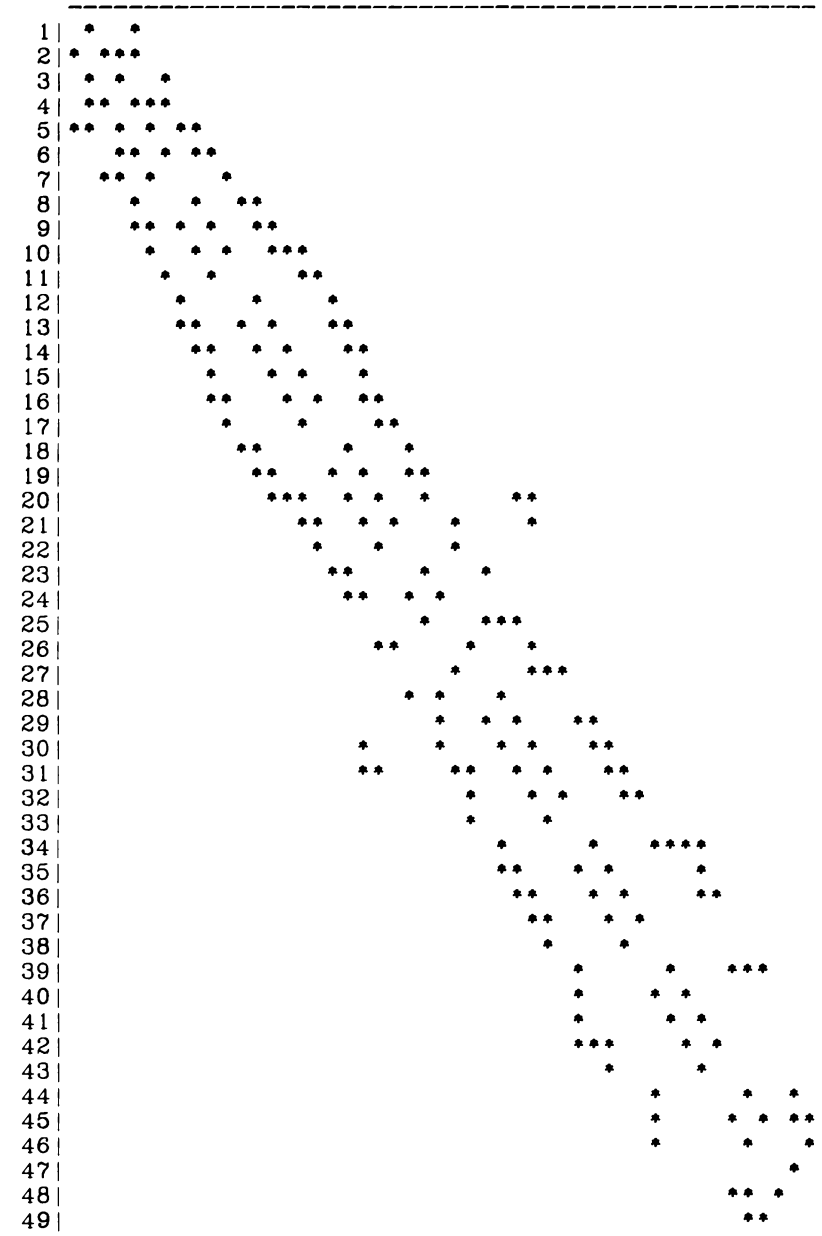
Lines 1-19. The head of a program serves the same purpose as the abstract of a technical paper. No matter how big or small the program, roughly the same information as is given here should be supplied somewhere very near the top. In particular, the program *must* point to any parallel external documents.

Lines 22-26. If this program were larger, there would be more material on algorithmic questions before the data definitions. As it is, the all-important glossary precedes the program text. Since modules should never be more than several pages long, we prefer the narrative glossary in which variables are introduced in a combined order of function, importance, and appearance. Obviously, the amount of explanation in the glossary depends on the other comments, but no variable should be left out. We try to make use of six-character names for mnemonic purposes; it is a hard fight against one of FORTRAN's silliest rules.

Lines 62-89. All variables are declared even though FORTRAN does not always require declarations—better be safe than sorry and better to ensure that the program reader knows our intentions exactly. The comments on lines 62-65 explain why we are diverging from standards: the compiler made us do it; and if the program is run elsewhere, the appropriate change is pinpointed. The I/O uses variable unit numbers so that if the program is moved to another system with different unit-assignment conventions, only one change will be necessary to make the I/O work.

```
THE NUMBER OF NODES IS   49

ADJACENCY MATRIX:
```



```
THE NUMBER OF COLORS NEEDED IS   4

COLOR   1 NODES       1    3    6    8   11   14   18   21   24   27
  28   30   34   37   43   44   46   47
COLOR   2 NODES       2    7    9   12   15   17   19   25   26   32
  35   39   41   48   49
COLOR   3 NODES       4   10   13   20   22   23   29   33   36   38
  40   45
COLOR   4 NODES       5   16   31   42
THE OUTER LOOP COUNT =       157
THE MIDDLE LOOP COUNT =      254
THE INNER LOOP COUNT =      2661
```

Lines 92-98. Throughout the program one paragraph of English comment is followed by one paragraph of FORTRAN coding. Be careful not to comment so often that the flow of the coding is obscured. Of course, there is no justification for too few comments. This program has 145 comment lines and 157 statement lines. By the way, every comment was written before the attached code.

Lines 105-109. A label should never appear on anything but a CONTINUE or FORMAT statement. FORTRAN requires many labels, and debugging often causes them to move; do not invite disaster by labeling substantive statements that may be moved inadvertently. Similarly, end only one DO-loop at a time.

Lines 110-115. Checking of input is simultaneously one of the dullest and most important program duties. Never trust anybody's data, even your own. Notice the inversion of the sense of the logical test in the IF statement. Such inversions are common in FORTRAN. It is hard to find a consistent scheme of indentation for conditional statements because of the lack of an *else*, but the several examples here provide some models.

Lines 139-151. Label 65 shows the value of labels that run upward in order, increasing by a large, regular step size. The input check just before label 65 was added because of an error encountered while preparing data. With ordered labels there was no difficulty inserting the added code. Notice also the use of variable J, required because standard FORTRAN will not allow subscripted variables to appear as subscripts. Only by following all the rules, even the distasteful ones, do we guarantee transportability and correctness. Most compilers would not require the temporary use of J, but it is best to be prepared for those that might.

Lines 155-162. Comments must explain *why* the code does what it does; generally the coding itself is sufficient indication of *how* things are done.

Lines 196-211. It would be redundant to repeat in the program all the algorithm development from the outside materials; the bibliographic citation will guide the program reader. But there had better be good outside materials, and it is usually a good idea to write them before the program is started.

Lines 238-239. This free-floating GOTO may seem a little odd. It exists so that the structure of the coding exactly parallels the structure of the algorithm. Efficiency nuts may complain that line 226 could simply have GOTO 220 replaced with GOTO 170, saving two lines of code and a microsecond or two during each loop. True enough, but at what cost in understanding? How many computer microseconds does it take to pay for 5 minutes of programmer time?

Lines 277-301. FORMATs are collected together at the bottom of the program to avoid cluttering the main flow. Each class of I/O, each separate file, unit, or usage, has its own sequence of labels. These labels are kept separate from one another and from ordinary program labels. Notice that Hollerith data are transmitted by using old-fashioned nH formats because standard FORTRAN requires the silly counts even though most compilers do not.

FINAL COMMENTS

So concludes our discussion of map coloring. We might note that original algorithm development took about 5 hours, writing the program took about 8, keying it in to a time-sharing system took about 3 hours, and testing/debugging took about 2 hours. Much of the testing time was spent verifying the correspondence of input and output. Two typing errors and two misplaced labels were discovered either by the compiler or by ridiculous output on the first run. The extra input check

was discovered during test data preparation. There was one logic flaw in the algorithm, and it was uncovered by the first substantial set of data.

This problem also illustrates the importance of documentation for future guidance. The algorithm described here was actually developed about 6 months before the chapter was written. A formal proof of correctness was also made, and the class of programmers that saw the algorithm and proof agreed unanimously that both were exactly right. But before this chapter could be written, burglars stole the only copy of the algorithm and proof (although we have no idea what they wanted with them). So we had to spend about 5 hours recovering the original reasoning and still made a logical error. May the burglars' computer have perpetual parity errors.

REFERENCES

Appel, K., and W. Haken. "Every Planar Map is Four Colorable." *Bulletin of the American Mathematical Society*, 82, 5, pp. 711-712, September 1976.

Knuth, D. E. "Estimating the Efficiency of Backtrack Programs." *Mathematics of Computation*, 29, 129, pp. 121-136, 1976.

Although the map-coloring algorithm has a worst-case computation time exponential in the size of the input graph, its *average* execution is typically quite short. However, analytic derivation of the average would probably be beyond our capabilities. Knuth describes a method for estimating the speed of a backtrack program. The estimation may be done by hand after running some test cases, and Knuth illustrates the method with examples.

Steen, Lynn Arthur. "Solution of the Four Color Problems." *Mathematics Magazine*, 49, 4, pp. 219-272, September 1976.

One more Ph.D. gone glimmering. Appel and Haken have announced a proof that the Four Color Conjecture is true. At the time of this writing, the proof is circulating in manuscript, and several eminent combinatoric specialists think that it is probably valid. By the time this book is published, the paper should be in the mathematics journals. Steen describes the proof method and its mathematical importance. A computer was used extensively to develop the proof and to check the 1936 cases in the final version.

30

Compressed Solutions

or...

A PROGRAM FOR TEXT COMPACTION

Chapter 11 presents an interesting algorithm for the compression of textual data. If you were a programmer working in a library or for a newspaper, say, you might expect your supervisor to plunk Chapter 11 down on your desk and ask you to have a shot at adding the algorithm to your local system. At first the project is appealing because it seems probable that the algorithm might provide considerable storage savings. But then doubts arise.

> How complicated are the algorithms necessary to do the dictionary matching, how complex the data structures? How much time will they take?
>
> The algorithm depends on parameter settings. How should the parameters be chosen?
>
> How sensitive is the algorithm's compression rate to changes in the text to be compressed?
>
> The algorithm trades processing time for storage space. What is an appropriate exchange rate?
>
> Assuming that the first attempt at a program is clear and correct but not very efficient, how can the program be revised to improve efficiency? Can the original design include the possibility of revision without a major rewrite?
>
> Finally, how much programmer time is it reasonable to spend on this project before deciding that it has been done well enough?

These questions should also occur to your supervisor, along with the problem of whether you were the right programmer for the task. You cannot, however, allow yourself to be paralyzed by doubt. Attack by ignoring (although not forgetting) some difficulties while concentrating on those that seem tractable. In our solution we will attempt a design that begins with simple data structures and algorithms; and after the program runs correctly, we will try to improve it by adding sophistication. Parameters may well affect behavior, and we will try to make them easily changeable; but in our first version we will simply guess at possible values. In order to study the effect of changing the program, its input, or its parameters, we will build the program without relying on any larger system so that its cost will not be hidden by outside expenses. In practice, though, text compression would

always be part of some larger problem involving large quantities of text. Fatigue and boredom will set limits to our work, not diminishing returns.

HOW TO SHARPEN A PENCIL

Every author has methods for preparing to write. Some sober up; a few drink; some wake at 4 AM; some rise at noon; some need silence; some need gaiety; some make do with a sharp pencil and a legal pad. Programmers are authors, too, and need to find comfortable methods and settings. Before we begin solving the compression problem, we would like to suggest some ways to make the *process* of programming more pleasant and fruitful. These methods work for us; at least try them before going back to your own ways.

Programs must be readable, and a primary component of readability is consistency. Consistency gives rise to recognizable style, since repeated functions will be given similar encodings. To improve your programming consistency, try to ensure that the physical environment is the same each time that you program. Almost all our programs—and most of this book—have been written in a decrepit old green platform rocker fitted with an improvised writing board of plastic sheeting and plywood. The chair is in the living room, where there is easy access to classical music on the stereo and to iced tea in the refrigerator. It's true that coding cannot go on when guests are present; yet home is so much quieter than any office that we do a great deal more work. You may prefer some other environment; search until you find one that is really comfortable and then figure out how to reproduce the comfort whenever you code.

Many programmers work by sketching a few lines of code on the back of an old envelope and then rushing off to enter the fragment into a terminal to give it a whirl. The rest of their time is spent patching pieces on here and there, and the final result looks as if Dr. Frankenstein had rushed in to do emergency surgery on Humpty Dumpty. In fact, another important part of readability is discipline; one way to achieve discipline is to use coding sheets. The free spirits mentioned above often sneer at programmers who work by filling in little boxes, but coding sheets have a strong stabilizing effect on style and make it much easier to develop and use a consistent indentation scheme. Code written on sheets can also be keypunched. The drudgery relief is pleasant enough, but, more important, keypunchers *read* and verify coding. Thus many clerical errors never enter the machine-readable source, and one more person looks at the program, which is never a bad thing.

Now for an iconoclasm: try writing your programs in ink. Most programming instructors will tell you that you should never be afraid to admit a mistake (true enough)—that the eraser may be your most powerful tool. Unfortunately, the eraser may be too powerful. If eliminating mistakes becomes easy, you may not think carefully enough before committing one. When you write in ink, the cost of errors is higher, especially since keypunchers probably will not accept a smudged, overwritten sheet. Bad errors will force you to recopy a whole page. With luck you will slow down your writing and think more deeply about each line; time invested during writing will pay dividends during testing and debugging. Even if you have a text editor that encourages or enforces a good programming style, you should try these last two suggestions. Coding must be thought about carefully, and our experience is that the electric fields around computer terminals tend to short-circuit thought.

Instead of using a flowchart, try writing comments first. Each time that you begin a new routine, write a long comment describing the routine's purpose and construction as lucidly as possible. Treat writing the comment as a major programming task. Once finished, set it to one side as reference and *translate* it into the programming language that you are using. Unless the language is very prolix—for example, assembly language—

the translation should not be much longer than the comment and may be shorter. By writing the comment on separate sheets (of coding paper, presumably), the comment may be treated as specification and will not need to be rewritten merely because the code that translates it turns out to be wrong. Of course, if you revise your view of the routine, you will probably need to rewrite both comment and code.

Do not turn any of your program into machine-readable form until you have written all the code. Once written, however, immediately run the compiler and eliminate all clerical errors, all misspelled identifiers, misplaced semicolons, and such. You are shooting for a clean listing, preferably with a complete cross-reference table. This "clean" listing is not error free, of course; it's merely free of those particularly simple-minded errors that a compiler can detect. Now sit down with the program and go over it as if someone else were trying to sell it to you. Check every statement; trace every loop; look for variables that are unused or misused; try to simplify the control flow; even correct the punctuation in the comments. The reason for doing so on the listing rather than the coding sheets is that many errors stand out more clearly when against a background of neatly printed program. This is also the time to adjust the program layout to be as visually attractive as possible. If you have so many changes marked on the listing that you cannot see the program clearly, make the changes, print a new listing, and start over. Paper is cheap compared to the cost of a missed bug.

Obviously, these suggestions are not appropriate for all programs and all situations. Many programming systems provide batteries of tools to aid in the program-development process. To ignore these tools would be foolish and wasteful. They are particularly useful for building one-shot programs and for keeping track of large projects. Yet we cannot warn too strongly against letting the computer come between you and your solution. The computer's siren song "Let me do your thinking" has lured many a programmer to defeat.

PROBLEM ANALYSIS

Because the two major algorithms (dictionary construction and text encoding) have already been designed, we must consider how to surround them with supporting code. First, notice that both algorithms walk down the input text from left to right and are interested only in matching the next few characters of the input against the dictionary. This means that the same input routine can be used for both construction and encoding and that we do not have to worry about the details of input access as long as the input routine always provides characters to match or an end-of-file indication. Second, both algorithms must look up strings in the dictionary, but neither is sensitive to the lookup method. So once again the algorithms may share a common service routine, and there is no need to specify the details yet. Third, the encoding algorithm requires at least one character not otherwise used in the input text as a coding control character. Instead of selecting an arbitrary character before the text is read, we can have the input routine keep track of all the characters seen in the input and use any not seen for encoding purposes. The dictionary building procedure, coded in XPL, is shown on the right-hand page.[1]

Some commentary is in order. The procedure is written in XPL, a language sufficiently similar to both PASCAL and PL/I that it is readily understandable (an illustration of the fact that reading a specific language is usually quite easy). XPL's virtues for this problem include strings as a built-in data type and good control structures; its drawbacks include single-dimensional vectors with fixed bounds as the only structured data type. The unusual features appearing here are the string con-

[1] The line numbers on these procedures are those from the complete program as printed on pages 179–189.

```
298 BUILD.DICTIONARY: PROCEDURE;
299
300     /* DICTIONARY CONSTRUCTION CONTINUES UNTIL THE INPUT ROUTINE
301        FAILS TO RETURN ANY DATA.  THE TEST FOR A NULL STRING IS
302        SIMPLE IF WE CHECK THE LENGTH AGAINST ZERO.  THE
303        DICTIONARY.SEARCH ROUTINE RETURNS -1 IF NO MATCH IS FOUND AND
304        THEN THE FIRST CHARACTER OF THE INPUT IS FORCED AS THE MATCH
305        AND INTO THE DICTIONARY.  NOTICE THAT THE ACTUAL STRING
306        MATCHED IS PICKED UP FROM THE DICTIONARY ENTRY.  COALESCENCE
307        TAKES PLACE AS NECESSARY, THE MATCH IS REMEMBERED, AND THE
308        INPUT PREPARED FOR ANOTHER CYCLE.
309     */
310
311     DECLARE  (MATCH, LAST.MATCH) CHARACTER,
312              (COUNT, LAST.COUNT) FIXED,
313              INDEX FIXED,              /* THE ENTRY LOCATION     */
314              THRESHOLD FIXED;          /* COALESCENCE THRESHOLD */
315
316     LAST.MATCH = '';
317     LAST.COUNT = 0;
318
319     DO WHILE TRUE;
320        CALL FILL.INPUT.BUFFER;
321        IF LENGTH(INPUT.BUFFER) = 0 THEN RETURN;
322        INDEX = SEARCH.DICTIONARY(INPUT.BUFFER);
323        IF INDEX = -1
324           THEN INDEX = BUILD.ENTRY(SUBSTR(INPUT.BUFFER,0,1));
325        MATCH = DICTIONARY.STRING(INDEX);
326        COUNT, DICTIONARY.COUNT(INDEX) = DICTIONARY.COUNT(INDEX) + 1;
327        THRESHOLD = COALESCENCE.THRESHOLD;
328        IF COUNT >= THRESHOLD & LAST.COUNT >= THRESHOLD THEN
329           DICTIONARY.COUNT(BUILD.ENTRY(LAST.MATCH||MATCH))=FIRST.COUNT;
330        LAST.MATCH = MATCH;
331        LAST.COUNT = COUNT;
332        INPUT.BUFFER = SUBSTR(INPUT.BUFFER, LENGTH(MATCH));
333     END;
334
335 END BUILD.DICTIONARY;
```

catenation operator || and the SUBSTR function used to select a substring of an existing string.[2] The input routine **FILL.INPUT.BUFFER** fills the input buffer if it is empty and returns the null

[2] If V is a string variable or expression, then SUBSTR(V, S, L) is the substring of V that begins at character S (counting the first character of the string as the zero position) and running for L characters. If argument L is omitted, the whole suffix of V starting at position S is returned. LENGTH returns as value the number of characters in its string argument. Line 332 of BUILD.DICTIONARY uses SUBSTR and LENGTH together to truncate the matched string from the head of INPUT.BUFFER.

string if the input file is exhausted. BUILD.DICTIONARY returns when there is no more input. Notice that testing for a length of zero is the same as testing if a string is null and is preferred here because LENGTH is an efficient operation in XPL. Now let's see the input procedure.

```
132 FILL.INPUT.BUFFER: PROCEDURE;
133
134     /* IF INPUT.BUFFER IS EMPTY, THEN THIS ROUTINE TRIES TO READ A
135        LINE FROM THE SOURCE.FILE.  THE LINE READ GOES INTO
136        INPUT.BUFFER WITH A NULL LINE THE SIGNAL FOR END OF FILE.  IF
137        FLAG PRINT.SOURCE IS ON, THEN THE INPUT IS ECHOED.  IF FLAG
138        CHECK.CHARACTERS IS ON, THE INPUT IS SCANNED FOR CHARACTER
139        USAGE.
140     */
141
142     DECLARE  I FIXED;
143
144     IF LENGTH(INPUT.BUFFER) > 0 THEN RETURN;
145     INPUT.BUFFER = INPUT(SOURCE.FILE);
146     IF PRINT.SOURCE THEN PRINT INPUT.BUFFER;
147     IF CHECK.CHARACTERS THEN
148        DO I = 0 TO LENGTH(INPUT.BUFFER)-1;
149           CHARACTER.USED(BYTE(INPUT.BUFFER,I)) = TRUE;
150        END;
151
152 END FILL.INPUT.BUFFER;
```

Input and output use built-in functions and always read and print strings. PRINT is actually a macro disguising the output function (see line 58 of the program). FILL.INPUT.BUFFER echoes the input if desired and also keeps a record of every character seen. The function BYTE, when used in an expression, converts a character selected from a string into an integer so that the character can be used arithmetically; here the characters are used to index Boolean vector CHARACTER.USED, a record of all the characters seen. BYTE is also used in BUILD.ENCODING.TABLE to turn integers back into characters; thus BYTE serves the same functions as both ORD and CHR in PASCAL.

Our first attempt at a data structure for the dictionary will be a simple, unordered table to be searched linearly. Such a structure will be trivial to debug but will probably be painfully inefficient; once we have everything working, we can try to speed up the searches. Each dictionary entry will have four fields: the actual string; the entry's frequency during dictionary construction; the code assigned to the entry; and a usage count for the entry during compression. These fields are held in four parallel vectors declared on lines 66-73 of the main program (now we begin to feel XPL's weakness in data structures). The first legitimate entry is always at index 0 and the last at index DICTIONARY.TOP; the maximum size of the dictionary is given by macro DICTIONARY.SIZE. Search requires only a pass completely through every entry in the dictionary; new entries can be added to the end of the table. Entry deletion squeezes low-frequency entries out by copying high-frequency entries over them; you should convince yourself that no data are lost in the loop of lines 261-270. Here is the whole program with the dictionary manipulation routines in lines 195-296. Notice how the parameters affecting compression have been moved out into their own subroutines on lines 154-193 so that they can be found and changed easily. We choose convenience over efficiency here; in a working version, the parameter routines would be eliminated and the chosen functions copied right where they are to be used.

```
 1 /*   ~$F  LARGE FREE STRING SPACE
 2    A TEXT COMPRESSION PROGRAM.
 3
 4    AUTHOR -- CHARLES WETHERELL  18 AUGUST 1976.
 5    LAST DATE MODIFIED -- 25 AUGUST 1976.
 6
 7    THIS PROGRAM COMPRESSES TEXT FILES USING THE MAYNE-JAMES DICTIONARY
 8    CONSTRUCTION ALGORITHM.  INPUT IS AN ARBITRARY TEXT FILE AND OUTPUT
 9    IS THE COMPRESSED FILE ALONG WITH AN ENCODING DICTIONARY.  SOME
10    STATISTICS ON PROGRAM EFFICIENCY AND COMPRESSION RATE ARE KEPT.
11    TWO PASSES BY THE SOURCE FILE ARE NECESSARY.  ON THE FIRST PASS THE
12    DICTIONARY IS BUILT AND A RECORD OF ALL CHARACTERS SEEN IS KEPT.
13    BETWEEN PASSES THE CHARACTER RECORD IS USED TO ADD ENCODINGS TO THE
14    DICTIONARY ENTRIES.  DURING THE SECOND PASS, LONG HIGH FREQUENCY
15    STRINGS ARE REPLACED BY SHORTER ENCODING STRINGS AS THE COMPRESSED
16    FILE IS WRITTEN.  FURTHER INFORMATION ABOUT THE TECHNIQUE CAN BE
17    FOUND IN CHAPTER 11 OF
18
19       WETHERELL, C.S.  ETUDES FOR PROGRAMMERS.  PRENTICE-HALL,
20          ENGLEWOOD CLIFFS, NJ.  1978.
21
22    IN THIS VERSION OF THE ALGORITHM, ENDS OF INPUT LINES STOP STRING
23    MATCHES DURING DICTIONARY CONSTRUCTION AND TEXT ENCODING; THE
24    CARRIAGE RETURN IS NOT TREATED LIKE A CHARACTER.  ONLY CHARACTERS
25    WHOSE INTERNAL REPRESENTATIONS LIE IN THE RANGE 1 TO 127 WILL BE
26    CONSIDERED FOR ENCODING PAIRS SO THAT THE PAIRS WILL HAVE
27    REASONABLE PRINT REPRESENTATIONS. IN A FULL WORKING IMPLEMENTATION
28    ALL 256 AVAILABLE CHARACTERS WOULD BE USED FOR ENCODING.
29
30    THE DICTIONARY SEARCH, ENTRY, AND CLEANUP ROUTINES ARE WRITTEN
31    SO THAT THEY MAY BE CHANGED QUITE EASILY.
32    THE ALGORITHMS CAN BE MODIFIED BY REPLACING THE BODIES OF PROCEDURES
33    SEARCH.DICTIONARY, CLEAN.DICTIONARY, AND BUILD.ENTRY, ALONG WITH
34    MAKING ANY NECESSARY CHANGES TO PREPARE.THE.PROGRAM.  THE VARIOUS
35    THRESHOLDS PARAMETERS ARE ALL CALCULATED BY FUNCTIONS AND CAN BE
36    BE MODIFED BY CHANGING THE FUNCTION DEFINITIONS.
37    IF THERE IS A DATA STRUCTURE ADDED FOR SEARCHING, MAKE SURE THAT
38    BUILD.ENCODING.TABLE LEAVES THE STRUCTURE IN GOOD SHAPE AFTER CODES
39    ARE ADDED AND ENTRIES OF LENGTH TWO AND LESS ARE DELETED.
40
41    THIS VERSION USES SIMPLE LINEAR SEARCH, A HYPERBOLIC THRESHOLD
42    FOR COALESCENCE, A MEAN THRESHOLD FOR DELETION, AND AN INITIAL
43    COUNT OF ONE FOR COALESCED ENTRIES.
44 */
45
46 /* SOME MACROS TO IMPROVE XPL AS A LANGUAGE.                          */
47
```

```
48 DECLARE  LOGICAL  LITERALLY 'BIT(1)',
49          TRUE     LITERALLY '1',
50          FALSE    LITERALLY '0',
51          NOT      LITERALLY '~',       /* TO IMPROVE PRINTING */
52          COMPUTE.TIME LITERALLY 'COREWORD("1E312")*2'; /* THE CLOCK */
53
54 /* DECLARATIONS FOR I/O UNITS                                        */
55
56 DECLARE  SOURCE.FILE LITERALLY '1',
57          ECHO.FILE   LITERALLY '2',
58          PRINT LITERALLY 'OUTPUT(ECHO.FILE) =';
59
60 /* DECLARATIONS FOR THE INPUT ROUTINE                                */
61
62 DECLARE  INPUT.BUFFER CHARACTER,
63          (PRINT.SOURCE, CHECK.CHARACTERS) LOGICAL,
64          CHARACTER.USED("FF") LOGICAL;  /* I.E. 256 DIFFERENT ENTRIES*/
65
66 /* DECLARATIONS FOR THE DICTIONARY                                   */
67
68 DECLARE   DICTIONARY.SIZE LITERALLY '100',
69           DICTIONARY.STRING(DICTIONARY.SIZE) CHARACTER,
70           DICTIONARY.COUNT(DICTIONARY.SIZE) FIXED,
71           DICTIONARY.CODE(DICTIONARY.SIZE) CHARACTER,
72           DICTIONARY.USAGE(DICTIONARY.SIZE) FIXED,
73           DICTIONARY.TOP FIXED;
74
75 /* CONTROL FOR ENCODING PRINT.                                       */
76
77 DECLARE  PRINT.ENCODING LOGICAL;
78
79 /* DECLARATIONS FOR ENCODING STATISTICS */
80
81 DECLARE  SEARCH.COMPARES FIXED,
82          BUILD.COMPARES FIXED,
83          COMPRESS.COMPARES FIXED;
84
85 DECLARE  TIME.CHECK(10) FIXED;
86
87 DECLARE  (INPUT.LENGTH, OUTPUT.LENGTH) FIXED;
88
89 I.FORMAT: PROCEDURE(NUMBER, WIDTH) CHARACTER;
90
91    /* FUNCTION I.FORMAT CONVERTS ITS ARGUMENT NUMBER INTO A STRING
92       AND THEN PADS THE STRING ON THE LEFT TO BRING THE LENGTH UP
93       TO WIDTH CHARACTERS.  ALL OF THIS IS JUST THE FORTRAN
94       INTEGER FORMAT.
95    */
```

```
 96
 97    DECLARE  NUMBER FIXED,
 98             WIDTH FIXED;
 99
100    DECLARE  STRING CHARACTER,
101             BLANKS CHARACTER INITIAL ('                              ');
102
103    STRING = NUMBER;
104    IF LENGTH(STRING) < WIDTH
105       THEN STRING = SUBSTR(BLANKS,0,WIDTH-LENGTH(STRING)) || STRING;
106    RETURN STRING;
107
108 END I.FORMAT;
109
110 PREPARE.THE.PROGRAM: PROCEDURE;
111
112    /* ONLY SIMPLE CLEARING OF THE DICTIONARY, THE CHARACTER RECORD,
113       THE STATISTICS, AND A FEW SCALARS IS REQUIRED.
114    */
115
116    DECLARE I FIXED;
117
118    DO I = 0 TO DICTIONARY.SIZE;
119       DICTIONARY.STRING(I), DICTIONARY.CODE(I) = '';
120       DICTIONARY.COUNT(I), DICTIONARY.USAGE(I) = 0;
121    END;
122    DICTIONARY.TOP = -1;
123
124    DO I = 0 TO "FF"; CHARACTER.USED(I) = FALSE; END;
125    INPUT.BUFFER = '';
126
127    SEARCH.COMPARES = 0;
128    INPUT.LENGTH, OUTPUT.LENGTH = 0;
129
130 END PREPARE.THE.PROGRAM;
131
132 FILL.INPUT.BUFFER: PROCEDURE;
133
134    /* IF INPUT.BUFFER IS EMPTY, THEN THIS ROUTINE TRIES TO READ A
135       LINE FROM THE SOURCE.FILE.  THE LINE READ GOES INTO
136       INPUT.BUFFER WITH A NULL LINE THE SIGNAL FOR END OF FILE.  IF
137       FLAG PRINT.SOURCE IS ON, THEN THE INPUT IS ECHOED.  IF FLAG
138       CHECK.CHARACTERS IS ON, THE INPUT IS SCANNED FOR CHARACTER
139       USAGE.
140    */
141
142    DECLARE  I FIXED;
143
```

```
144     IF LENGTH(INPUT.BUFFER) > 0 THEN RETURN;
145     INPUT.BUFFER = INPUT(SOURCE.FILE);
146     IF PRINT.SOURCE THEN PRINT INPUT.BUFFER;
147     IF CHECK.CHARACTERS THEN
148        DO I = 0 TO LENGTH(INPUT.BUFFER)-1;
149           CHARACTER.USED(BYTE(INPUT.BUFFER,I)) = TRUE;
150        END;
151
152 END FILL.INPUT.BUFFER;
153
154 COALESCENCE.THRESHOLD: PROCEDURE FIXED;
155
156     /* THIS PROCEDURE CALCULATES THE THRESHOLD FOR COALESCING
157        TWO DICTIONARY ENTRIES INTO ONE.  HERE, THE REQUIREMENT IS
158        THAT THE ENTRIES HAVE FREQUENCIES GREATER THAN THE RECIPROCAL
159        OF THE RATIO OF SPACE REMAINING IN THE DICTIONARY.
160     */
161
162     RETURN DICTIONARY.SIZE/(DICTIONARY.SIZE-DICTIONARY.TOP+1) + 1;
163
164 END COALESCENCE.THRESHOLD;
165
166 DELETION.THRESHOLD: PROCEDURE FIXED;
167
168     /* THIS FUNCTION RETURNS THE THRESHOLD NECESSARY FOR AN ENTRY
169        TO BE RETAINED IN THE DICTIONARY AT CLEANUP TIME.
170        IN THIS VERSION, THE FREQUENCY MUST BE GREATER THAN THE
171        ROUNDED UP MEAN FREQUENCY.
172     */
173
174     DECLARE  SUM FIXED,
175              I FIXED;
176
177     SUM = 0;
178     DO I = 0 TO DICTIONARY.TOP;
179        SUM = SUM + DICTIONARY.COUNT(I);
180     END;
181     RETURN SUM/(DICTIONARY.TOP+1) + 1;
182
183 END DELETION.THRESHOLD;
184
185 FIRST.COUNT: PROCEDURE FIXED;
186
187     /* THIS FUNCTION RETURNS THE COUNT GIVEN A COALESCED ENTRY WHEN
188        IT IS FIRST ENTERED IN THE DICTIONARY.
189     */
190
191     RETURN 1; /* CURRENTLY GIVE A COUNT OF 1. */
```

```
192
193 END FIRST.COUNT;
194
195 SEARCH.DICTIONARY: PROCEDURE(TEST.STRING) FIXED;
196
197    /* THIS FUNCTION SEARCHES THE DICTIONARY FOR THE LONGEST MATCH
198       WITH THE HEAD OF THE ARGUMENT TEST.STRING.  IF NO MATCH IS
199       FOUND, THE ROUTINE RETURNS -1 AS VALUE; IF A MATCH IS FOUND,
200       THE INDEX OF THE MATCH IS RETURNED AS VALUE.
201
202       THIS ROUTINE PERFORMS A SIMPLE LINEAR SEARCH OF THE DICTIONARY
203       FROM THE ZEROTH ENTRY TO THE ENTRY DICTIONARY.TOP.  IF AN
204       ENTRY'S LENGTH IS LONGER THAN THE LONGEST CURRENT MATCH AND
205       STILL NO LONGER THAN THE ARGUMENT, THEN THE ENTRY IS MATCHED
206       AGAINST THE ARGUMENT.  EQUALITY WILL CAUSE THE MATCH TO BE
207       UPDATED. NOTICE THAT BY STARTING THE INDEX AT -1, THE RETURN
208       VALUE WILL BE PROPER EVEN IF NO MATCH IS FOUND.
209    */
210
211    DECLARE  TEST.STRING CHARACTER;
212
213    DECLARE  INDEX FIXED,
214             (MATCH.LENGTH, ARG.LENGTH, ENTRY.LENGTH) FIXED,
215             I FIXED;
216
217    INDEX = -1;
218    MATCH.LENGTH = 0;
219    ARG.LENGTH = LENGTH(TEST.STRING);
220
221    DO I = 0 TO DICTIONARY.TOP;
222       ENTRY.LENGTH = LENGTH(DICTIONARY.STRING(I));
223       IF ENTRY.LENGTH > MATCH.LENGTH
224             & ENTRY.LENGTH <= ARG.LENGTH THEN
225          IF DICTIONARY.STRING(I)
226                = SUBSTR(TEST.STRING,0,ENTRY.LENGTH) THEN
227             DO;
228                INDEX = I;
229                MATCH.LENGTH = ENTRY.LENGTH;
230             END;
231    END;
232    SEARCH.COMPARES = SEARCH.COMPARES + DICTIONARY.TOP + 1;
233    RETURN INDEX;
234
235 END SEARCH.DICTIONARY;
236
237 CLEAN.DICTIONARY: PROCEDURE;
238
```

```
239     /* CLEAN.DICTIONARY ELIMINATES AT LEAST ONE LOW FREQUENCY ENTRY
240        FROM THE DICTIONARY AND RESTORES THE SMALLER DICTIONARY TO
241        THE FORMAT IT HAD BEFORE CLEANING.
242        THE WHILE LOOP SURROUNDING THE BODY OF THE PROCEDURE GUARANTEES
243        THAT AT LEAST ONE ENTRY IS DELETED FROM THE DICTIONARY BEFORE
244        RETURN.  IF THE INITIAL THRESHOLD IS NOT HIGH ENOUGH TO DELETE
245        AN ENTRY, THE THRESHOLD IS INCREMENTED UNTIL SOMETHING IS
246        DELETED.
247
248        THE DICTIONARY IS JUST A LINEAR TABLE WITH NO STRUCTURE SO
249        ENTRIES CAN BE DELETED BY PUSHING THE RETAINED ENTRIES TOWARD
250        THE ZERO END OF THE TABLE OVERWRITING THE REMOVED ENTRIES.
251     */
252
253     DECLARE  I FIXED,
254              THRESHOLD FIXED,
255              OLD.TOP FIXED,
256              NEW.TOP FIXED;
257
258     OLD.TOP = DICTIONARY.TOP;
259     THRESHOLD = DELETION.THRESHOLD;
260     DO WHILE OLD.TOP = DICTIONARY.TOP;
261        NEW.TOP = -1;
262        DO I = 0 TO DICTIONARY.TOP;
263           IF DICTIONARY.COUNT(I) >= THRESHOLD THEN
264              DO;
265                 NEW.TOP = NEW.TOP + 1;
266                 DICTIONARY.STRING(NEW.TOP) = DICTIONARY.STRING(I);
267                 DICTIONARY.COUNT(NEW.TOP) = DICTIONARY.COUNT(I);
268              END;
269        END;
270        DICTIONARY.TOP = NEW.TOP;
271        THRESHOLD = THRESHOLD + 1;
272     END;
273
274 END CLEAN.DICTIONARY;
275
276 BUILD.ENTRY: PROCEDURE(ENTRY.STRING) FIXED;
277
278     /* BUILD.ENTRY ADDS ENTRY.STRING TO THE DICTIONARY WITH A COUNT
279        OF ZERO AND RETURNS AS VALUE THE INDEX OF THE NEW ENTRY.
280
281        BECAUSE THE DICTIONARY IS SEARCHED LINEARLY, THE NEW ENTRY
282        CAN SIMPLY BE ADDED AT THE END.  THE ONLY REQUIREMENT IS THAT
283        THE DICTIONARY MAY NEED TO BE CLEANED BEFORE THE NEW ENTRY
284        CAN BE ADDED.
285     */
286
```

```
287     DECLARE  ENTRY.STRING CHARACTER;
288
289     IF DICTIONARY.TOP+2 >= DICTIONARY.SIZE
290        THEN CALL CLEAN.DICTIONARY;
291     DICTIONARY.TOP = DICTIONARY.TOP + 1;
292     DICTIONARY.STRING(DICTIONARY.TOP) = ENTRY.STRING;
293     DICTIONARY.COUNT(DICTIONARY.TOP) = 0;
294     RETURN DICTIONARY.TOP;
295
296 END BUILD.ENTRY;
297
298 BUILD.DICTIONARY: PROCEDURE;
299
300     /* DICTIONARY CONSTRUCTION CONTINUES UNTIL THE INPUT ROUTINE
301        FAILS TO RETURN ANY DATA.  THE TEST FOR A NULL STRING IS
302        SIMPLE IF WE CHECK THE LENGTH AGAINST ZERO.  THE
303        DICTIONARY.SEARCH ROUTINE RETURNS -1 IF NO MATCH IS FOUND AND
304        THEN THE FIRST CHARACTER OF THE INPUT IS FORCED AS THE MATCH
305        AND INTO THE DICTIONARY.  NOTICE THAT THE ACTUAL STRING
306        MATCHED IS PICKED UP FROM THE DICTIONARY ENTRY.  COALESCENCE
307        TAKES PLACE AS NECESSARY, THE MATCH IS REMEMBERED, AND THE
308        INPUT PREPARED FOR ANOTHER CYCLE.
309     */
310
311     DECLARE  (MATCH, LAST.MATCH) CHARACTER,
312              (COUNT, LAST.COUNT) FIXED,
313              INDEX FIXED,              /* THE ENTRY LOCATION     */
314              THRESHOLD FIXED;          /* COALESCENCE THRESHOLD */
315
316     LAST.MATCH = '';
317     LAST.COUNT = 0;
318
319     DO WHILE TRUE;
320        CALL FILL.INPUT.BUFFER;
321        IF LENGTH(INPUT.BUFFER) = 0 THEN RETURN;
322        INDEX = SEARCH.DICTIONARY(INPUT.BUFFER);
323        IF INDEX = -1
324           THEN INDEX = BUILD.ENTRY(SUBSTR(INPUT.BUFFER,0,1));
325        MATCH = DICTIONARY.STRING(INDEX);
326        COUNT, DICTIONARY.COUNT(INDEX) = DICTIONARY.COUNT(INDEX) + 1;
327        THRESHOLD = COALESCENCE.THRESHOLD;
328        IF COUNT >= THRESHOLD & LAST.COUNT >= THRESHOLD THEN
329            DICTIONARY.COUNT(BUILD.ENTRY(LAST.MATCH||MATCH))=FIRST.COUNT;
330        LAST.MATCH = MATCH;
331        LAST.COUNT = COUNT;
332        INPUT.BUFFER = SUBSTR(INPUT.BUFFER, LENGTH(MATCH));
333     END;
334
```

```
END BUILD.DICTIONARY;

BUILD.ENCODING.TABLE: PROCEDURE;

   /* CODE CONSTRUCTION HAS TWO STEPS.  IN THE FIRST, EVERY
      DICTIONARY ENTRY OF LENGTH TWO OR ONE IS THROWN OUT BECAUSE
      THERE IS NO POINT IN REPLACING SUCH STRINGS WITH A TWO
      CHARACTER CODE.  SECOND, CODES ARE ASSIGNED USING CHARACTERS
      UNSEEN IN THE TEXT AS STARTERS.  WHEN SUCH PAIRS RUN OUT,
      NO MORE CODES ARE ASSIGNED EVEN IF THERE ARE MORE ENTRIES IN
      THE DICTIONARY.

      NOTICE THE LINES BELOW WHICH CONSTRUCT THE DICTIONARY CODE.
      THE APPARENTLY SENSELESS CATENATION OF TWO BLANKS BUILDS A
      COMPLETELY NEW STRING INTO WHICH THE CODE CHARACTERS CAN BE
      INSERTED.  THIS IS A BAD GLITCH IN XPL AND YOU PROBABLY WON'T
      UNDERSTAND IT UNLESS YOU PROGRAM IN XPL FOR SOME TIME.
   */

   DECLARE  (I, J) FIXED,
            TOP FIXED;

   TOP = -1;
   DO I = 0 TO DICTIONARY.TOP;
      IF LENGTH(DICTIONARY.STRING(I)) > 2 THEN
         DO;
            TOP = TOP + 1;
            DICTIONARY.STRING(TOP) = DICTIONARY.STRING(I);
            DICTIONARY.COUNT(TOP) = DICTIONARY.COUNT(I);
         END;
   END;
   DICTIONARY.TOP = TOP;

   TOP = -1;
   DO I = 1 TO "7F";    /* LOOP OVER ELIGIBLE START CHARACTERS */
      IF NOT CHARACTER.USED(I) THEN
         DO J = 1 TO "7F";    /* LOOP OVER SECOND CHARACTERS   */
            IF TOP = DICTIONARY.TOP THEN RETURN;
            TOP = TOP + 1;
            DICTIONARY.CODE(TOP) = '  ||  ';
            BYTE(DICTIONARY.CODE(TOP),0) = I;
            BYTE(DICTIONARY.CODE(TOP),1) = J;
         END;
   END;
   DICTIONARY.TOP = TOP;

END BUILD.ENCODING.TABLE;
```

```
383 COMPRESS.TEXT: PROCEDURE;
384
385    /* ENCODING WORKS ALMOST THE SAME WAY AS DICTIONARY CONSTRUCTION.
386       HERE, THOUGH, THE INPUT STREAM IS CONVERTED TO OUTPUT LINES
387       AS THE ENCODINGS ARE FOUND.  THE LOOP RUNS FOR AS LONG AS
388       THERE IS INPUT.
389    */
390
391    DECLARE  OUTPUT.BUFFER CHARACTER,
392             INDEX FIXED;
393
394    INPUT.BUFFER = '';
395    PRINT '';
396    PRINT '*** THE COMPRESSED TEXT ***';
397    PRINT '';
398    CALL FILL.INPUT.BUFFER;
399    DO WHILE LENGTH(INPUT.BUFFER) > 0;
400       INPUT.LENGTH = INPUT.LENGTH + LENGTH(INPUT.BUFFER);
401       OUTPUT.BUFFER = '';
402       DO WHILE LENGTH(INPUT.BUFFER) > 0;
403          INDEX = SEARCH.DICTIONARY(INPUT.BUFFER);
404          IF INDEX > -1 THEN
405             DO;
406                OUTPUT.BUFFER = OUTPUT.BUFFER
407                                || DICTIONARY.CODE(INDEX);
408                DICTIONARY.USAGE(INDEX) = DICTIONARY.USAGE(INDEX) + 1;
409                INPUT.BUFFER = SUBSTR(INPUT.BUFFER,
410                                      LENGTH(DICTIONARY.STRING(INDEX)));
411             END;
412          ELSE
413             DO;
414                OUTPUT.BUFFER = OUTPUT.BUFFER
415                                || SUBSTR(INPUT.BUFFER,0,1);
416                INPUT.BUFFER = SUBSTR(INPUT.BUFFER,1);
417             END;
418       END;
419       OUTPUT.LENGTH = OUTPUT.LENGTH + LENGTH(OUTPUT.BUFFER);
420       IF PRINT.ENCODING THEN PRINT OUTPUT.BUFFER;
421       CALL FILL.INPUT.BUFFER;
422    END;
423
424 END COMPRESS.TEXT;
425
426 PRINT.SUMMARY.STATISTICS: PROCEDURE;
427
```

```
428    /* SUMMARY STATISTICS INCLUDE A SECOND PRINTING OF THE SEARCH
429       STATISTICS, THE DICTIONARY ITSELF, AND THE COMPRESSION RATE.
430       IN A WORKING VERSION, BOTH THE COMPRESSED TEXT AND THE
431       DICTIONARY WOULD HAVE ALSO BEEN PRINTED ON SEPARATE FILES FOR
432       RE-EXPANSION LATER.  NOTICE THE COMPLICATION TO PRINT A RATIO
433       AS A DECIMAL IN A LANGUAGE WITHOUT FLOATING POINT NUMBERS.
434    */
435
436    DECLARE  I FIXED,
437             RATIO CHARACTER;
438
439    PRINT '';
440    PRINT '*** COMPRESSION STATISTICS ***';
441    PRINT '';
442    PRINT 'CODE FREQUENCY   USAGE   STRING';
443    DO I = 0 TO DICTIONARY.TOP;
444       PRINT ' ' || DICTIONARY.CODE(I) ||
445                            || I.FORMAT(DICTIONARY.COUNT(I), 9) ||
446                            || I.FORMAT(DICTIONARY.USAGE(I), 9)
447                            || ' |' || DICTIONARY.STRING(I) || '|';
448    END;
449    PRINT '';
450    PRINT '   CHARACTERS IN INPUT = ' || INPUT.LENGTH;
451    PRINT '   CHARACTERS IN OUTPUT = ' || OUTPUT.LENGTH;
452    RATIO = (1000*OUTPUT.LENGTH)/INPUT.LENGTH + 1000;
453    PRINT '   COMPRESSION RATE = .' || SUBSTR(RATIO,1);
454    PRINT '   COMPARES DURING DICTIONARY CONSTRUCTION = '
455                      || BUILD.COMPARES;
456    PRINT '   COMPARES DURING COMPRESSION = '
457                      || COMPRESS.COMPARES;
458    PRINT '*** TIME CHECKS IN MILLESECONDS ***';
459    PRINT '   TIME TO PREPARE = '
460                      || TIME.CHECK(0) - TIME.CHECK(1);
461    PRINT '   TIME TO BUILD DICTIONARY = '
462                      || TIME.CHECK(1) - TIME.CHECK(2);
463    PRINT '   ENCODING TABLE TIME = '
464                      || TIME.CHECK(2) - TIME.CHECK(3);
465    PRINT '   COMPRESSION TIME = '
466                      || TIME.CHECK(3) - TIME.CHECK(4);
467
468 END PRINT.SUMMARY.STATISTICS;
469
470 /* THE MAIN ROUTINE MUST ASSIGN THE I/O UNITS TO FILES, INITIALIZE
471    NEEDED VARIABLES, CALL THE DICTIONARY CONSTRUCTION ALGORITHM, BUILD
472    THE ENCODING TABLE, AND THEN ENCODE THE OUTPUT.  MOST OF THIS WORK
473    IS DONE IN CALLED PROCEDURES.
474 */
475
```

```
476 TIME.CHECK(0) = COMPUTE.TIME;
477 CALL ASSIGN('TEXT', SOURCE.FILE, "(1) 1000 0110");
478 CALL ASSIGN('COMPRESS', ECHO.FILE, "(1) 0010 1010");
479 PRINT '*** BEGIN THE TEXT COMPRESSION. ***';
480 PRINT '';
481
482 CALL PREPARE.THE.PROGRAM;
483 PRINT.SOURCE = TRUE; CHECK.CHARACTERS = TRUE;
484 TIME.CHECK(1) = COMPUTE.TIME;
485 CALL BUILD.DICTIONARY;
486 BUILD.COMPARES = SEARCH.COMPARES;
487 TIME.CHECK(2) = COMPUTE.TIME;
488 CALL BUILD.ENCODING.TABLE;
489
490 CALL ASSIGN('TEXT', SOURCE.FILE, "(1) 1000 0110"); /* A REWIND */
491 SEARCH.COMPARES = 0;
492 PRINT.SOURCE, CHECK.CHARACTERS = FALSE;
493 PRINT.ENCODING = TRUE;
494 TIME.CHECK(3) = COMPUTE.TIME;
495 CALL COMPRESS.TEXT;
496 COMPRESS.COMPARES = SEARCH.COMPARES;
497
498 TIME.CHECK(4) = COMPUTE.TIME;
499 CALL PRINT.SUMMARY.STATISTICS;
500
501 EOF EOF EOF EOF EOF EOF
```

RESULTS

We ran the program on a short prefix of itself used as data; the results are printed below with line numbers for reference. Lines 67-71 show the encoding dictionary; with such a short text file it is no wonder the dictionary is short. Compression is only 0.973, partly because text lines in the host system are not padded out with those blanks so beloved by most compression routines. Still some interesting compression takes place, as witness line 62. Notice that compression "D F" was not used because another compression ate up the "D" first. Here are the results.

```
 1 *** BEGIN THE TEXT COMPRESSION. ***
 2
 3 /* ~$F  LARGE FREE STRING SPACE
 4     A TEXT COMPRESSION PROGRAM.
 5
 6     AUTHOR -- CHARLES WETHERELL  18 AUGUST 1976.
 7     LAST DATE MODIFIED -- 25 AUGUST 1976.
 8
 9     THIS PROGRAM COMPRESSES TEXT FILES USING THE MAYNE-JAMES DICTIONARY
10     CONSTRUCTION ALGORITHM.  INPUT IS AN ARBITRARY TEXT FILE AND OUTPUT
11     IS THE COMPRESSED FILE ALONG WITH AN ENCODING DICTIONARY.  SOME
12     STATISTICS ON PROGRAM EFFICIENCY AND COMPRESSION RATE ARE KEPT.
13     TWO PASSES BY THE SOURCE FILE ARE NECESSARY.  ON THE FIRST PASS THE
14     DICTIONARY IS BUILT AND A RECORD OF ALL CHARACTERS SEEN IS KEPT.
```

```
    BETWEEN PASSES THE CHARACTER RECORD IS USED TO ADD ENCODINGS TO THE
    DICTIONARY ENTRIES.  DURING THE SECOND PASS, LONG HIGH FREQUENCY
    STRINGS ARE REPLACED BY SHORTER ENCODING STRINGS AS THE COMPRESSED
    FILE IS WRITTEN.  FURTHER INFORMATION ABOUT THE TECHNIQUE CAN BE
    FOUND IN CHAPTER 11 OF

       WETHERELL, C.S.  ETUDES FOR PROGRAMMERS.  PRENTICE-HALL,
          ENGLEWOOD CLIFFS, NJ.  1978.

    IN THIS VERSION OF THE ALGORITHM, ENDS OF INPUT LINES STOP STRING
    MATCHES DURING DICTIONARY CONSTRUCTION AND TEXT ENCODING; THE
    CARRIAGE RETURN IS NOT TREATED LIKE A CHARACTER.  ONLY CHARACTERS
    WHOSE INTERNAL REPRESENTATIONS LIE IN THE RANGE 1 TO 127 WILL BE
    CONSIDERED FOR ENCODING PAIRS SO THAT THE PAIRS WILL HAVE
    REASONABLE PRINT REPRESENTATIONS. IN A FULL WORKING IMPLEMENTATION
    ALL 256 AVAILABLE CHARACTERS WOULD BE USED FOR ENCODING.


*** THE COMPRESSED TEXT ***

/* ~$F  LARGE FREE STRING SPACE
!!A TEXT COMPRESSION PROGRAM.

!!AUTHOR -- CHARLES WETHERELL  18 AUGUST 1976.
!!LAST DATE MODIFI!$-- 25 AUGUST 1976.

!!THIS PROGRAM COMPRESSES TEXT FILES USING THE MAYNE-JAMES DICTIONARY
!!CONSTRUCTION ALGORITHM.  INPUT IS AN ARBITRARY TEXT FILE AND OUTPUT
!!IS THE COMPRESS!$FILE ALONG WITH AN ENCODING DICTIONARY.  SOME
!!STATISTICS ON PROGRAM EFFICIENCY AND COMPRESSION RATE ARE KEPT.
!!TWO PASSES BY THE SOURCE FILE ARE NECESSARY.  ON THE FIRST PASS THE
!!DICTIONARY IS BUILT AND A RECORD OF ALL CHARACTE!.SEEN IS KEPT.
!!BETWEEN PASSES THE CHARACTER RECORD IS US!$TO ADD ENCODINGS TO THE
!!DICTIONARY ENTRIES.  DURING THE SECOND PASS, LONG HIGH FREQUENCY
!!STRINGS ARE REPLAC!$BY SHORTER ENCODING STRINGS AS THE COMPRESSED
!!FILE IS WRITTEN.  FURTHER INFORMATION ABOUT THE TECHNIQUE CAN BE
!!FOUND IN CHAPTER 11 OF

WETHERELL, C.S.  ETUDES FOR PROGRAMMERS.  PRENTICE-HALL,
!!!!!!ENGLEWOOD CLIFFS, NJ.  1978.

!!IN THIS VERSION OF THE ALGORITHM, ENDS OF INPUT LINES STOP STRING
!!MATCHES DURING DICTIONARY CONSTRUCTION AND TEXT ENCODING; THE
!!CARRIAGE RETURN IS NOT TREAT!$LIKE A CHARACTER.  ONLY CHARACTERS
!!WHOSE INTERNAL REPRESENTATIONS LIE IN THE RANGE 1 TO 127 WILL BE
!!CONSIDER!$FOR ENCODING PAI!.SO THAT THE PAI!.WILL HAVE
!!REASONABLE PRINT REPRESENTATIONS. IN A FULL WORKING IMPLEMENTATION
!!ALL 256 AVAILABLE CHARACTE!.WOULD B!#S!$FOR ENCODING.

*** COMPRESSION STATISTICS ***

CODE FREQUENCY   USAGE   STRING
 !!          25      26 |    |
 !.           1       4 |RS  |
 !#           1       1 |E U|
 !$           1       7 |ED  |
 !a           1       0 |D F|

   CHARACTERS IN INPUT = 1424
   CHARACTERS IN OUTPUT = 1386
   COMPRESSION RATE = .973
   COMPARES DURING DICTIONARY CONSTRUCTION = 80740
   COMPARES DURING COMPRESSION = 6740
*** TIME CHECKS IN MILLESECONDS ***
   TIME TO PREPARE = 108
   TIME TO BUILD DICTIONARY = 9492
   ENCODING TABLE TIME = 34
   COMPRESSION TIME = 1372
```

AN ATTEMPTED IMPROVEMENT

As predicted above, linear dictionary search is not too efficient; when the whole program source was compressed, the program ran 127 seconds on a moderately fast computer, terrible for a 500-line file. Notice that each time that the dictionary is searched, the input must be matched against every

dictionary entry to ensure that the longest match is found. If the dictionary entries were kept in order from longest to shortest, the search could stop as soon as a match was found because the match would perforce be the longest possible. The search procedure would not need to change otherwise, and deletion of low-frequency strings would not disturb the longest-to-shortest ordering. But insertion of a new entry requires that all shorter entries be moved one slot farther up the table. To know where to make the insertion, we add the vector LENGTH.VECTOR whose ith component points to the dictionary entry where strings of length i (if there are any) or shorter (if there are none of length i) begin. If there are no strings of length i or shorter, LENGTH.VECTOR(I) has value DICTIONARY.TOP+1. The routines below keep the new data structure correct. To make a new version of the compression program, we insert the following code in the obvious places in the program above. Notice how the comments for the modified routines are written so that the changed code can be inserted easily.

```
/* DECLARATIONS FOR LENGTH POINTER VECTOR */

DECLARE  LENGTH.VECTOR(255) FIXED;

   /* ADDITION TO PREPARE.THE.PROGRAM TO INITIALIZE LENGTHS. */

   DO I = 0 TO 255;
      LENGTH.VECTOR(I) = 0;
   END;

SEARCH.DICTIONARY: PROCEDURE(TEST.STRING) FIXED;

   /* THIS FUNCTION SEARCHES THE DICTIONARY FOR THE LONGEST MATCH
      WITH THE HEAD OF THE ARGUMENT TEST.STRING.  IF NO MATCH IS
      FOUND, THE ROUTINE RETURNS -1 AS VALUE; IF A MATCH IS FOUND,
      THE INDEX OF THE MATCH IS RETURNED AS VALUE.

      THE SEARCH FOR A MATCH AGAINST TEST.STRING BEGINS AT THE
      FIRST ENTRY FOR THE LENGTH OF TEST.STRING AND WORKS DOWN
      THROUGH THE SHORTER STRINGS.  BECAUSE ARGUMENTS ARE CALL BY
      VALUE, TEST.STRING MAY BE MODIFIED AT WILL.  THE LOOP DOWN
      OVER STRING LENGTHS MUST BE A WHILE LOOP BECAUSE THERE ARE NO
      DOWNWARD STEPPING ITERATION LOOPS IN XPL.
   */

   DECLARE  TEST.STRING CHARACTER;

   DECLARE  (L, I) FIXED;

   L = LENGTH(TEST.STRING);
   DO WHILE L > 0;
      TEST.STRING = SUBSTR(TEST.STRING, 0, L);
      DO I = LENGTH.VECTOR(L) TO LENGTH.VECTOR(L-1)-1;
         SEARCH.COMPARES = SEARCH.COMPARES + 1;
         IF DICTIONARY.STRING(I) = TEST.STRING THEN RETURN I;
      END;
```

```
37          L = L - 1;
38       END;
39       RETURN -1;
40
41    END SEARCH.DICTIONARY;
42
43    CLEAN.DICTIONARY: PROCEDURE;
44
45       /* CLEAN.DICTIONARY ELIMINATES AT LEAST ONE LOW FREQUENCY ENTRY
46          FROM THE DICTIONARY AND RESTORES THE SMALLER DICTIONARY TO
47          THE FORMAT IT HAD BEFORE CLEANING.
48          THE WHILE LOOP SURROUNDING THE BODY OF THE PROCEDURE GUARANTEES
49          THAT AT LEAST ONE ENTRY IS DELETED FROM THE DICTIONARY BEFORE
50          RETURN.  IF THE INITIAL THRESHOLD IS NOT HIGH ENOUGH TO DELETE
51          AN ENTRY, THE THRESHOLD IS INCREMENTED UNTIL SOMETHING IS
52          DELETED.
53
54          ONE PASS ELIMINATES ALL ENTRIES WITH LOW FREQUENCY BY SQUEEZING
55          THE DICTIONARY TOWARDS THE ZERO END.  NOTICE THAT THIS
56          SQUEEZE DOES NOT DISORDER ENTRIES BY LENGTH, BUT IT DOES
57          INVALIDATE LENGTH.VECTOR.  LENGTH.VECTOR IS RESET BY A SECOND
58          PASS THROUGH THE SHORTENED DICTIONARY
59       */
60
61       DECLARE  (I, J) FIXED,
62                OLD.LENGTH FIXED,
63                THRESHOLD FIXED,
64                OLD.TOP FIXED,
65                NEW.TOP FIXED;
66
67       OLD.TOP = DICTIONARY.TOP;
68       THRESHOLD = DELETION.THRESHOLD;
69       DO WHILE OLD.TOP = DICTIONARY.TOP;
70          NEW.TOP = -1;
71          DO I = 0 TO DICTIONARY.TOP;
72             IF DICTIONARY.COUNT(I) >= THRESHOLD THEN
73                DO;
74                   NEW.TOP = NEW.TOP + 1;
75                   DICTIONARY.STRING(NEW.TOP) = DICTIONARY.STRING(I);
76                   DICTIONARY.COUNT(NEW.TOP) = DICTIONARY.COUNT(I);
77                END;
78          END;
79          DICTIONARY.TOP = NEW.TOP;
80          OLD.LENGTH = 256;
81          DO I = 0 TO DICTIONARY.TOP;
82             IF LENGTH(DICTIONARY.STRING(I)) < OLD.LENGTH THEN
83                DO;
```

```
84                DO J = LENGTH(DICTIONARY.STRING(I)) TO OLD.LENGTH-1;
85                   LENGTH.VECTOR(J) = I;
86                END;
87                OLD.LENGTH = LENGTH(DICTIONARY.STRING(I));
88             END;
89       END;
90       DO I = 0 TO OLD.LENGTH-1;
91          LENGTH.VECTOR(I) = DICTIONARY.TOP + 1;
92       END;
93       THRESHOLD = THRESHOLD + 1;
94    END;
95
96 END CLEAN.DICTIONARY;
97
98 BUILD.ENTRY: PROCEDURE(ENTRY.STRING) FIXED;
99
100   /* BUILD.ENTRY ADDS ENTRY.STRING TO THE DICTIONARY WITH A COUNT
101      OF ZERO AND RETURNS AS VALUE THE INDEX OF THE NEW ENTRY.
102
103      IF L IS THE LENGTH OF ENTRY.STRING, THEN ALL OF THE ENTRIES
104      BEGINNING WITH THE FIRST OF LENGTH L-1 AND RUNNING DOWN
105      THROUGH LENGTH 1 SHOULD BE MOVED ONE SLOT UP THE TABLE.  THE
106      VACATED HOLE IS JUST RIGHT FOR ENTRY.STRING.  ALL
107      LENGTH.VECTOR VALUES FOR L-1 TO 0 MUST BE INCREMENTED.
108   */
109
110   DECLARE  ENTRY.STRING CHARACTER;
111
112   DECLARE  (L, I) FIXED;
113
114   IF DICTIONARY.TOP+2 >= DICTIONARY.SIZE
115      THEN CALL CLEAN.DICTIONARY;
116   L = LENGTH(ENTRY.STRING);
117   I = DICTIONARY.TOP;
118   DO WHILE I >= LENGTH.VECTOR(L-1);
119      DICTIONARY.STRING(I+1) = DICTIONARY.STRING(I);
120      DICTIONARY.COUNT(I+1) = DICTIONARY.COUNT(I);
121      I = I - 1;
122   END;
123   DICTIONARY.TOP = DICTIONARY.TOP + 1;
124   DICTIONARY.STRING(LENGTH.VECTOR(L-1)) = ENTRY.STRING;
125   DICTIONARY.COUNT(LENGTH.VECTOR(L-1)) = 0;
126   DO I = 0 TO L-1;
127      LENGTH.VECTOR(I) = LENGTH.VECTOR(I) + 1;
128   END;
129   RETURN LENGTH.VECTOR(L-1) - 1;
130
131 END BUILD.ENTRY;
```

```
132
133     /* ADDITION TO BUILD.ENCODING.TABLE WHICH RESETS THE LENGTH VECTOR
134        SO THAT THERE WILL NOT APPEAR TO BE ANY STRINGS OF LENGTHS
135        2, 1, OR 0 WHEN SEARCH.DICTIONARY RUNS DURING COMPRESSION.
136        OF COURSE, ALL SUCH STRINGS HAVE BEEN ELIMINATED FROM THE
137        DICTIONARY DURING THE CONSTRUCTION OF THE TWO CHARACTER CODES.
138     */
139
140     DO I = 0 TO 2;
141        LENGTH.VECTOR(I) = DICTIONARY.TOP + 1;
142     END;
```

We can expect that there should be fewer comparisons using length-first search during dictionary construction and that more time may be spent with the more elaborate entry insertion procedure. Table 30-1 shows that both dictionary construction time and encoding time are worse with the length-first search even though fewer comparisons are made. Notice, however, that in the original program a comparison may require only an inexpensive comparison of string lengths, whereas under length-first search all comparisons require expensive string comparison operations. Something more elaborate will have to be done to improve efficiency. On the other hand, this change illustrates an important debugging principle. If you replace a program structure with a functionally equivalent one, there should be no change in the relationship between input and output. The dictionary organization must not affect the actual compression process; only the parameters of compression should modify dictionary output. Thus we can check our changes to the dictionary routines by making sure that the output using simple linear search and that using length-first search are exactly the same (barring timing statistics). This is also a negative check on the correctness of the other parts of the program; if a bug existed in any other procedure, it might well interact with the dictionary routines. Since the output stays constant when a correct length-first search is added, probably (but not certainly) no bug elsewhere was interacting with the dictionary.

THE CHOICE OF PARAMETERS

Four parameters control dictionary construction: the size of the dictionary, the threshold for entry coalescence, the initial count given a coalesced entry, and the threshold for entry deletion. Our choices may have been a little eccentric. The dictionary size is given by macro DICTIONARY.SIZE

Table 30-1 Comparison of Two Dictionary Algorithms

	File 1_{Lin}	*File* 1_{Len}	*File* 2_{Lin}	*File* 2_{Len}
Input characters	16920	16920	17714	17714
Output characters	8516	8516	16439	16439
Compression rate	.503	.503	.928	.928
Dictionary time	52864 ms.	56432 ms.	92882 ms.	54870 ms.
Encoding time	19082 ms.	55965 ms.	23524 ms.	35374 ms.
Search compares	453049	271803	816480	489126
Encoding compares	122912	116046	151770	130307

File 1 is a typical industrial data file, and File 2 is the source of the linear search program. Compression of the source program is poor because of the compression parameters chosen. The file name subscripts indicate linear or length-first dictionary searching.

Table 30-2 A Small Study of Compression Parameters

	Normal	*COA=5*	*DEL=5*	*CNT=5*	*COA=5 DEL=5*	*COA=5 CNT=5*	*DEL=5 CNT=5*	*COA=5 DEL=5 CNT=5*
Data for File 1								
Input size	16920	16920	16920	16920	16920	16920	16920	16920
Output size	8516	8080	7659	8154	7271	7701	7357	7003
Compression rate	.503	.478	.453	.482	.430	.455	.435	.414
Dictionary entries	16	29	32	25	41	31	41	56
Dictionary size	250	288	371	319	425	475	586	738
Dictionary + output	8756	8368	8030	8473	7696	8176	7943	7741
Overall compression	.517	.495	.475	.501	.455	.483	.469	.458
Dictionary time	56432	52916	63732	54916	59842	56128	114350	92498
Compression time	55968	54984	50496	50016	52536	49536	52224	47716
Data for File 2								
Input size	17714	17714	17714	17714	17714	17714	17714	17714
Output size	16439	17695	14038	16439	14200	16567	14993	13358
Compression rate	.928	.998	.792	.928	.801	.935	.846	.754
Dictionary entries	10	2	20	9	24	7	17	35
Dictionary size	65	11	140	63	170	38	101	214
Dictionary + output	16504	17706	14178	16502	14370	16605	15904	13572
Overall compression	.932	1.000	.800	.932	.811	.937	.898	.766
Dictionary time	92882	65582	69532	64444	68756	59206	227614	199830
Compression time	23524	41310	37774	37662	41814	33888	39940	42416

The notation COA=5 (DEL=5, CNT=5) means that the coalescence threshold (deletion threshold, initial count) has been set to 5 for that column of data. The rows labeled Dictionary Size give the minimum number of characters needed to record the dictionary entries and their encodings. Overall compression is derived by summing the output file size and the dictionary size and dividing by the input file size. Times are in milliseconds.

set here to 100 (remember that XPL vectors start at zero) and the initial count is set to one by function FIRST.COUNT. Entries are coalesced if both have frequencies greater than or equal to

DICTIONARY.SIZE/(DICTIONARY.SIZE-DICTIONARY.TOP+1)+1

that is, if both have frequencies greater than the reciprocal of the space remaining. The idea (which did not work too well) is that it should be harder to enter a coalesced string when the dictionary is nearly full; because of its form, we call it a hyperbolic coalescence threshold. Entries are deleted if their frequency is not at least as large as the mean frequency; this was intended to be an approximation to a median threshold.

To discover how eccentric these choices actually were, we tried varying each one except dictionary size to a fixed value of five. Table 30-2 shows the result for both our files. From the table it appears that both the coalescence and the deletion threshold were ill-advised. It is possible that the hyperbolic coalescence does not let enough combined entries into the table and that the mean deletion threshold is a poor approximation to the median and throws too many entries out. This is only a tiny study and we should look at more data. Yet until the program becomes more efficient, we cannot stand the boredom.

FINAL COMMENTS

This chapter is not quite what an A project would be. The program itself could stand a little more work, particularly on the comments. More than a few are muddy and unhelpful. We printed the program thus half done so that you could see what your half-done programs look like to others.

Go back and read the program again with the question "How could I make this line (comment, procedure) crystal clear?" in your mind. After you have finished being snide about the skills of others, be objective and see if you can use the answers on some of your own programs.

A complete project would also include some demonstration of the correctness of the algorithms, more output, and at least some suggestions on improving the dictionary efficiency. The coding took us about eight hours and debugging about four. Of the four debugging hours, two were spent on the standard clerical errors and in improving the design of output messages. The two other debugging hours were spent adding the '+1' to line 91 of the change file. That one subtle little mistake caused much difficulty and changed the behavior of length-first search completely. The error was finally uncovered when we explained the whole program to another programmer. Following the principle of the unjaundiced eyeball, he did not share our preconceptions about the program and immediately found the bug. We also spent about an hour adding and adjusting the timing variables (timing is not well documented in our local system), and we spent about four hours playing with the parameters.

Index

Notation and Abbreviations

ETUDES
FOR PROGRAMMERS

CHARLES WETHERELL

Etudes for Programmers is a collection of large-scale problems for "learning by doing." Each problem includes a real-world background, discussion of appropriate programming techniques, detailed requirements for correct solution, extensions, and annotated bibliography. These are realistic problems similar to those encountered in actual practice.

Two of the problems are completely solved by the author. The solutions concentrate on good programming techniques, measuring the quality of the program and the output, and possible extensions of the problem. They are models of what solutions to any programming job should be and contain many practical hints about writing good programs.

Among its outstanding features, the book:

- Discusses programming problems of sufficient length to require you to face challenges of "big" as opposed to "toy" programs.
- Provides detailed and careful analysis of the real-world situation surrounding each program problem.
- Provides self-contained problems that may be done without outside help.
- Offers references to sources for programming information and to further reading about problem subjects.
- Includes a complete set of four projects for a programming language course—macro interpreter, compiler, relocating loader, and computer simulator.
- Lets the reader choose interesting problems for himself.
- Shows how "academic" algorithms can be used to solve "real" problems.
- Puts some fun back into programming.

PRENTICE-HALL, INC., Englewood Cliffs, New Jersey 07632