IC STUDY PROBLEMS

Mary Shaw (ed.)

August 31, 1971

Carnegie-Mellon University
Department of Computer Science
Pittsburgh, Pa. 15213

TABLE OF CONTENTS

INTRODUCTION

One of the goals of the immigration course is to present an overview of the field of Computer Science, including introductions to a variety of interesting problem areas. Another is to instill in the entering student an appreciation that Computer Science includes problems which can be studied in depth.

We have chosen a problem-oriented format to help satisfy both of these goals, because:

1. in many cases it is easier to use a concrete example to explain the focus of an area than to give general descriptions and abstract proofs; and

2. one of the best ways to appreciate significant problems is to try to solve some.

Background

In order to collect a group of worthwhile problems that can be solved with a reasonable amount of effort, the Computer Science department sponsored an IC problem competition in the Spring of 1970. All the graduate students in the department were asked to submit problems touching on major aspects of Computer Science together with complete solutions of the problems. To stimulate interest, ten prizes of $100.00 were announced.

The specifications were:

1. It should be possible for students in the IC (not just advanced students) to do each problem within the time limit of two to

three work sessions.

2. The problem should be elegant and have an elegant solution.

3. The problem should touch on or illustrate some central concept of Computer Science.

4. The problem should involve a non-trivial programming effort, which should be an integral part of obtaining the solution (not just "tacked on").

5. The problem, or the associated programming, should provide insight into the programming language used.

6. There should be problems utilizing all types of programming languages -- algebraic languages, list languages, pattern matching languages, etc.

7. Problems should be usable in future ICs as well as the next one.

8. To be useful, the submitted problem should consist of:

   (a) A problem statement;

   (b) A discussion of the conceptual rationale behind the problem, including comments on how to teach the problem in the IC, what sort of preparation is required, etc.;

   (c) A worked solution.

A problem selection committee consisting of Nico Habermann, Bob Lieberman, and Hans Berliner evaluated the problems and recommended

awards. The final list of prizes was based on those recommendations, with an additional problem promoted from a list of alternates for a total of eleven. A prize of $100.00 was awarded the author (or divided between the authors) of each of the prize problems.

Problem Descriptions

The Mutation Problem

Birol Aygun

This problem is derived from problems in genetics involving estimation of the probabilities of mutation processes. It requires finding the most probable path from one string over a small alphabet to another, given certain assumptions and probabilities of elementary string transformations.

The Power of Heuristics

Hans Berliner

This problem shows the power of heuristics as a means of controlling processes. It is cast as a Tic-Tac-Toe tournament in which heuristic processes represent players.

Hamming Codes

Susan Gerhart

Error-detecting and -correcting codes are often used to expedite communication over noisy channels. One of the best-known and most elegant of these methods is due to R. W. Hamming; the problem is to implement that method.

One Man's Program is Another Man's Data

Anita Jones

The problem is to implement a line editor allowing insertions and deletions of lines of text and replacement of strings of symbols at locations determined by context within a line.

Exercise in Symbolic Polynomial Manipulation

Rudy Krutar

A scheme for fast multiplication based on the identity $(Ax + B) * (Cx + D) = AC(x^2 + x) + (A - B)(D - C)x + BD(x + 1)$ can be applied to polynomial multiplication. The exercise is to build a polynomial package using it.

Coroutines

Åmund Lunde

The coroutine concept is a generalization of the subroutine concept, establishing a completely symmetric relationship between two or more routines instead of the caller-callee relationship that obtains in subroutine calls. The problem involves implementing a coroutine scheme and using it to build part of a lexical scanner.

Area of a Region

Leroy Richardson and Richard Young

The problem involves implementing and comparing three very different approaches to finding the area of a contiguous region in a two-dimensional space composed of unit squares.

A Problem in Simple Languages

George Robertson

The problem is to write a program to test whether a given string is a legal sentence in a simple Polish-prefix language.

Turing Machine Simulation

Larry Snyder

This problem asks students to write an interactive Turing machine simulator that can be used to design and debug a Turing machine.

Firing Squad Synchronization Problem

Tim Teitelbaum

The firing squad problem is well known in finite-state machine theory. The student must develop an interactive system for experimenting with such machines and then use that system to solve the firing squad problem.

Trees, Trees, Trees

Tim Teitelbaum

LISP is designed to have data only at the terminal nodes of trees. This problem requires the student to design and manipulate a representation of trees with data at all nodes.

In addition to using many of these prize problems we added five other problems for the 1971 IC:

1. The two questions from the 24-hour take-home qualifiers given in the academic year 1970-1971. The problems are interesting, the time constraints are reasonable, and a certain body of local expertise exists. The questions are presented here

precisely as they were posed. They may serve as a "dry run" for students who have not yet taken the qualifier.

2. The "Busy Beaver" problem. For two-symbol Turing machines with some given number of states, find the machine which will, when started on a blank tape, halt with the longest possible continuous string of 1's on the tape.

3. Analysis of algorithms as performed by Knuth. This is an example of a significant problem that does not involve a computer solution.

4. Simulation of a small computer. This problem was included to teach students something about machine languages and information flow within a computer.

Use of the Problems in the IC

For the purposes of presenting the problems, we have divided the six-week IC into three two-week periods, with four of these problems to be presented in each fortnight according to the following scheme:

* One problem is introduced each day of the first week. The introduction includes:

A description of the domain of Computer Science from which the problem is drawn:

Some issues that lead to the problem;

The specification of the problem;

Some pointers to the implications of the problem;

References;

Some techniques that may be of value (a hint or a language suggestion).

Reasonable questions that occur on the spot will, of course, be answered. Each of the entering students should attend all of these sessions.

* On the day of the second week corresponding to the day of the initial presentation, each problem is discussed further. In this discussion,

Students who are doing well with the problem present what they are doing;

Students who are having trouble ask questions;

The instructor gives more background and context. For example, he might talk about other uses of the same or similar data structures.

These sessions will be of benefit to students who have given some serious thought to the solution of the problems. Hopefully, this set will include all of the entering students.

* When a student has completed the solution of a problem as defined below, he submits it to the instructor who presented the problem. This should be done by the end of the fortnight to prevent the problems from dragging on forever. The instructor comments on the solution to the student and selects the best of the submitted solutions for possible publication.

The schedule for the problems presented in the 1971 IC is:

First fortnight

Language in programming lab: APL

M	Sep 13 and 20	Teitelbaum	Firing Squad Problem
T	Sep 14 and 21	Parnas	November (1970) qualifier
Th	Sep 16 and 23	Snyder	Turing Machine Simulator
F	Sep 17 and 24	Gerhart	Hamming Codes

Second fortnight

Language in programming lab: Algol

M	Sep 27, Oct 4	Berliner	Power of Heuristics
T	Sep 28, Oct 5	Richardson and Young	Area of a Region
Th	Sep 30, Oct 7	Selman	Busy Beaver
F	Oct 1 and 8		Analysis of Algorithms

Third fortnight

Language in programming lab: LISP

M	Oct 11 and 18	Robertson	Problem in Simple Languages
T	Oct 12 and 19	Bell	Simulation of a Small Computer
Th	Oct 14 and 21		Symbolic Polynomial Manipulation
F	Oct 15 and 22		May (1971) qualifier

Wednesdays are left open to provide time for extra discussion if it is needed. Three languages (APL, Algol, and LISP) are introduced in the

programming labs at the rate of one language each fortnight. The schedule has been arranged so that the languages are introduced as they are needed for problems.

Students who already know one or more of these languages will be encouraged to learn another without formal instruction.

A Note to the Students

We expect that you will be able to obtain complete solutions to two of the four problems discussed during each fortnight and to do enough work on the other two to participate in the second week's discussions. A complete solution to a programming problem consists of:

1. A statement of your approach to the problem and the technique used to solve it (this isn't a term paper -- two or three pages should do it unless you really get into the problem);

2. A running program, together with --

3. Sufficient documentation that someone else can understand your code. This might consist of extensive comments in the program, a separate piece of prose, and even, if you are so inclined, a flow chart;

4. Runs with test cases showing that the program runs properly, together with --

5. Some kind of written explanation justifying how the data you have used shows that the program runs. (Again, this isn't a term paper -- use common sense; rigorous proofs of programs are not required.)

Solutions to nonprogramming problems will take a rather different form, but should exhibit about the same level of detail.

Try to complete your problems rather than letting them go on and on or succumbing to the temptation to add just one more feature. ("90%

coded and 70% debugged" is an absorbing state.) We hope to publish the best of the solutions.

Please note that no grades will be given for this work, or for any work in the IC or (at least for Computer Science graduate students) for any course work done in the department. Your energies during the IC should be directed toward learning new things, not rehashing old ones. Since assignments are informal and there are no grades, there is no penalty for doing a less than elegant solution for a new problem instead of a polished job on a familiar one.

Here are a few guidelines for selecting which problems to work on:

1. Try to solve problems in at least two programming languages that you have not used before (BASIC doesn't count!). If you already know two of the three languages presented in the IC, look around for one you don't know.

2. If you have never programmed in machine language, be sure to do the simulation of a small computer.

3. If you have written a compiler or a parser, pick something other than the problem in simple languages.

4. If you have never experimented with finite-state machines or Turing machines, try to do either the firing squad problem or the Turing machine simulator, or both.

5. If you are already an expert programmer in a variety of languages, work either analysis of algorithms or the May, 1971 qualifier. These involve extensive analysis but not programming.

Mary Shaw
August, 1971

THE MUTATION PROBLEM

Birol Aygun

Motivation

This problem originates in a class of genetics problems involving estimations of the probabilities of mutation processes. This highly simplified and solvable version of this problem is also a very interesting exercise in computing and has applications in some areas of artificial intelligence, such as recognition of linear patterns.

The problem is also open-ended in the sense that most solutions will not be practical for very large cases of the problem. Hence ingenuity is required for drastic reductions in the computing time and space required.

1. Given a string M of m characters and a string N of n characters, all chosen from a small alphabet of, say, 4 characters (A,B,C,D).

2. Two kinds of primitive mutation operations: deletion of a single character and insertion of a single character in a string.

3. Fixed independent probabilities P_D and P_I for a single deletion and a single insertion respectively (i.e., $P_D(A) = P_D(B) = P_D(C) = \ldots$ and similiarly for P_I).

Find

1. An algorithm to determine a sequence of mutation operations on the string N (for the normal string) to transform it into the string M (for the mutant string) that has the highest probability of happening under the stated assumptions in 3. above.

2. Clearly, the solution required is not unique, i.e., there may be more than one sequence of mutations that yield the same result with the same maximum probability. Find an algorithm that determines the class of all solutions, each of which has the same maximum probability.

Remarks

1. The solution should be provable, i.e., that it has the maximum probability, and, for part 2, that it has not missed any solutions.

2. Magnitude range: the strings M and N may be up to several million characters in length. Check the practicality of your solution for strings of that size.

Examples and Hints

M = A B B C D D A B C A D C B

N = B B D C A A B C

Example strings above

1. What does independence of deletions and insertions imply in probability computation?

2. Are any sequences found in both M and N? What about the ordering of such sequences?

THE POWER OF HEURISTICS

Hans Berliner

Background

The main purpose of this IC Problem is to show the power of heruistics as a means of controlling processes. There are many processes for which we do not know perfect controlling functions, but by having them controlled by heuristic rules, we are able to obtain a high standard of performance from the process. Examples of this type of activity occur in the areas of Artificial Intelligence and Operating Systems. For instance, in a time sharing system with virtual memory, the problem of which page to kick out of main memory when a page fault occurs is resolved by using heuristic rules. Usually, a rule is tried and evaluated according to how much it improves the performance of the system. This is kept up until the point of diminishing returns is reached. This problem is intended to teach this method by setting up a situation in which heuristics, represented by processes, can compete in the same environment. Then by comparing the success of each of the processes on the same task, we can determine the usefulness of each set of heuristic rules.

The environment in which the problem is set is Tic-Tac-Toe. We let each heuristic process respresent a player in a Tic-Tac-Toe tournament, and then pit processes with different degrees of "intelligence" against one another. It is important to note that:

1. Tic-Tac-Toe can be played perfectly (so as never to lose and to maximize winning chances) by merely resorting to a table-look-up procedure, or to a complete tree search of all possibilities which would, however, be rather time consuming.

2. However, the intent of this exercise is to teach how to build heuristic models and to show that one heuristic procedure can have an overwhelming dominance over another procedure with less "intelligence," even though the first does not play perfectly.

Other things which can be learned from doing this problem are:

1. How a thoughtful problem representation can save programming effort and execution time.

2. How an appropriate experimental design can allow ready comparison of the different effects being studied.

3. How to use a random number generator.

Since the total task requires a significant amount of work in the design and implementation of the program, you may find it desirable to work in teams of 2 to 4. You will use a set of heuristic rules to define a player in a Tic-Tac-Toe tournament. When you have defined several such players, write a program for simulating such players in a tournament. Be sure that each player has an equal chance of starting the game against every other player. Each of the players in your tournament should be at a different skill level. The skill you impart to each of your players should be a function of the move selection routines that each particular player has access to. For instance, the worst player in the tournament could be one that plays at random. Other players may use the strategy of the center square if it is free, be able to defend against simple opponent's threats, or be a compound of several such strategies. By

carefully choosing compound strategies, you can create a player hierarchy where each player is better than the one below him. Before you start, consider that after each move the supervisory program has to check to see if anyone has won. Consider the effect of how the Tic-Tac-Toe board is represented on how easy it is to perform checks such as these. If it has been a long time since you have played Tic-Tac-Toe, you may want to play a few games to re-acquaint yourself with some useful strategies.

Have the program tabulate results. Then write a short critique on the relative skills of the various players in your tournament. Why do you think the results came out the way they did? Can you rank the efficiency of the heuristics you used? Is there a point of diminishing returns?

HAMMING CODES

Susan Gerhart

Error-detecting and -correcting codes are used to provide communication over noisy channels in many applications of computers. One of the best-known and most elegant coding schemes is that originated by R.W. Hamming (see references).

Consider the transmission of n-bit messages. Hamming's method encodes the n-bit message as a n + k - bit binary sequence, where the extra k bits provide for error detection and correction in any of the n + k positions of the sequence. A decoder then maps the n + k - bit sequence into a k - bit message sequence. An error in any one position of the transmitted sequence will be given in the check sequence.

Example: $n = 4$, $k = 3$

Let $m_1\ m_2\ m_3\ m_4$ be the message to be transmitted as the sequence $x_1\ x_2\ x_3\ x_4\ x_5\ x_6\ x_7$. The following equations are used in the encoding:

$$x_1 = m_1 \oplus m_2 \oplus m_4$$
$$x_2 = m_1 \oplus m_3 \oplus m_4$$
$$x_3 = m_1$$
$$x_4 = m_2 \oplus m_3 \oplus m_4$$
$$x_5 = m_2$$
$$x_6 = m_3$$
$$x_7 = m_4$$

$\oplus$ is the exclusive -or or sum modulo 2 operator

Assume the received message is $y_1\ y_2\ y_3\ y_4\ y_5\ y_6\ y_7$.

The decoder computes $k_1\ k_2\ k_3$ where

$$k_1 = y_4 \oplus y_5 \oplus y_6 \oplus y_7$$

$$k_2 = y_2 \oplus y_3 \oplus y_6 \oplus y_7$$

$$k_3 = y_1 \oplus y_3 \oplus y_5 \oplus y_7$$

If one and only one digit is transmitted incorrectly, say $y_j \neq x_j$, then $k_1\ k_2\ k_3$ will give the binary representation of j. $k_1\ k_2\ k_3$ will be 0 if no errors occur. If multiple errors occur, then the correction will take place but give an incorrect result.

Now, to generalize the process, consider a code which requires n information bits per message. An additional k bits are required to point to any of the n + k bits of the encoding which might be in error. The sufficient condition is

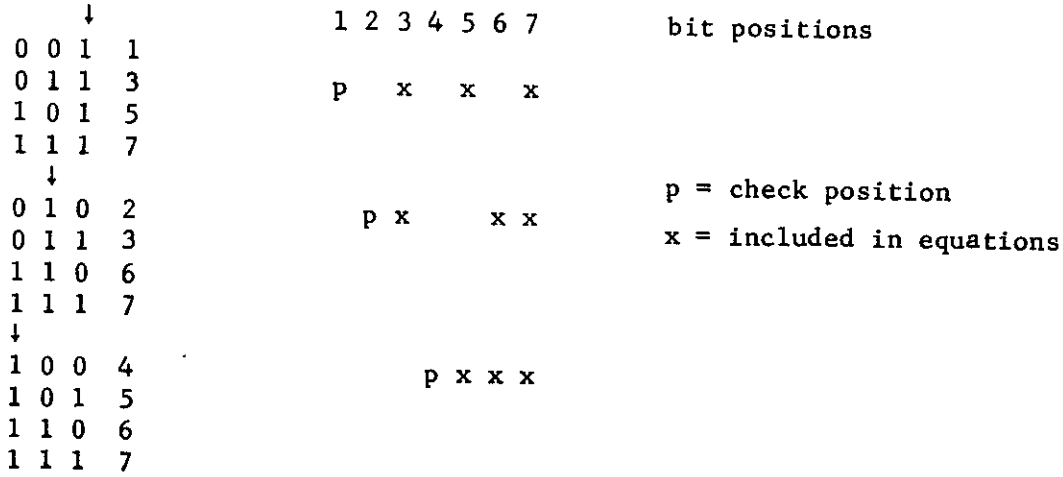
$$2^k \geq n + k + 1$$

A general method for the assignment of equations in the encoding is:

1. Use the positions numbered by powers of 2 for check bits.

2. Assign the bits of the original message in order to the remaining positions.

To see how to form the equations for the check positions, group the binary representations of the positions by occurrence of powers of 2:



Position 1 serves as a parity check for positions 1,3,5,7 of the encoding (and positions 1,2,4 of the message). Similarly, for positions 2 and 4.

Solution requirements

Construct a system of programs which enable the encoding of messages, transmission of messages corrupted in one position, and decoding into the original messages. Of course, transmission of error-free messages should also be possible.

APL is recommended for the solution because it offers operators for manipulation of number systems and arrays. The author's solution used approximately 25 different APL operators in three one-line, loop-free functions, along with a control program for testing. The absence of loops was possible because the APL operators afforded the necessary control flow.

References

[1] R. W. Hamming, "Error Detecting and Error Correcting Codes," Bell System Technical Journal, Vol. XXVI, April, 1950.

[2] Herman Hellerman, Digital Computer System Principles, McGraw-Hill, p. 322.

[3] Ralph A. Amato, "Error Detecting and Correcting Methods," Computer Design, June, 1964.

ONE MAN'S PROGRAM IS ANOTHER MAN'S DATA

Anita Jones

This problem requires implementation of a line editor which will allow insertion and deletion of lines of text, and replacement of symbol strings at locations determined by context.

Implementation of the editor provides a basis for considering the concepts and mechansims germane to string processing [1,2] (in contrast to numerical processing):

1. The text to be edited appears as a sequence of symbols. Only the fact that each symbol is distinguishable is important -- the information encoded in the sequence of symbols could be π computed to a thousand places as well as program, prose, or poetry.

2. A cursor is moved through the text to find the location at which an editing operation is to be performed.

3. A particular location within the text is determined by context -- by the surrounding or preceding characters. (The 'carriage return,' 'line feed' and 'blank' characters become very visible!)

4. A pattern matching mechanism must be employed to search for variable-sized strings of symbols.

A note of realism: The editor to be implemented is a simplified version of the line editor in a conversational PL/I - based system called CPS (Conversational Programming System).

Solution Notes

The editor is most easily implemented in a string processing language like SNOBOL which provides pattern recognition facilities as well as string manipulation. However, any language which permits easy representation of variable length strings could be used (e.g., ALGOL extended with strings).

The Problem

Design and program the algorithm MERGE to enable an editor to accept two input files, TEXT and MODIFY, concurrently and to output an edited file, NEWTEXT.

TEXT is a sequential stream of lines in which two lines are separated by a line feed character. (Refer to such an instance as LF.) With each line of TEXT is associated an implicit line number equal to the number of LF's that precede the line in TEXT. Line j consists of those characters following LF[j] up to and including LF[j+1].

NEWTEXT is of the same format as TEXT. It is created by editing the TEXT file as directed by the contents of MODIFY.

MODIFY consists of a sequential stream of INSERT, DELETE and REPLACE commands.

MERGE processes TEXT line by line with no backtracking although multiple scans of a single TEXT line may be necessary in the case of REPLACE. MERGE may thus be seen as moving a cursor through the stream of TEXT lines, possibly altering a line as the cursor passes over it.

All lines to the output side of the cursor are written on NEWTEXT. Lines to the input side of the cursor comprise the portion of the TEXT file which may still be subject to alteration by MERGE.

MODIFY command formats:

INSERT <num> <delim> insertion string <delim> LF

where <num> is a non negative integer referring to the line in TEXT associated with that number.

<delim> is defined as the first non-blank, non-LF, non-numeric character following <num>. In a single command all instances of <delim> are the same character.

INSERT causes MERGE to:

1. Scan TEXT from current cursor position until cursor has passed over LF [<num>].

2. Insert the delimited insertion string followed by the carriage return and line feed characters into the TEXT string to the output side of the cursor.

DELETE <num> <num1> LF

where <num1> is null or has a value greater than or equal to that of <num>. <num> and <num1> are separated by 1 or more blanks.

DELETE causes MERGE to

1. Scan TEXT from current cursor position until the cursor has passed over LF[<num>].

2. Delete all characters on the input side of the cursor up to and including LF[<num1>+1], if <num1> was specified or up to and including LF[<num>+1], if <num1> is null.

REPLACE <num> <delim>α_1 <delim>β_1 <delim>α_2 <delim>β_2 <delim>....LF

A REPLACE command may be used to edit a single TEXT line. LF may not appear in the REPLACE command except as the terminator of the command.

1. Scan TEXT from current cursor position until cursor has passed over LF[<num>].

2. Let the first non-blank, non-numeric, non-LF character after <num> be the delimiter.

3. Set J=1.

4. Scan the command string for the next two occurrences of <delim> to determine the recognition string, α_J and the replacement string β_J.

5. If LF was encountered before the recognition string and replacement strings were found, this command is completely processed.

6. Scanning the line, replace each occurrence of the current recognition string with the corresponding replacement string.

7. J=J+1.

8. Go to step 4.

NB: The edited line is not yet moved to the output side of the cursor, so that re-editing of the line may occur.

References

[1] D. J. Farber, R. E. Griswold, and I. P. Polonsky, "SNOBOL, A String Manipulation Language," CACM, 11 (Jan, 1964), pp. 21-30.

[2] Madnick, Stuart E., "String Processing Techniques," CACM, 10 (July, 1967), pp. 420-424.

Example: Given the following TEXT file:

```
procedure cal(y,n);
  value y,n;  integer y,n,;
begin
  y := if (y/4)* 4 = y then 1 else 0;
  comment 1900 < y < 2100 causes abort;
  d := n + (if n > (59 + t) then 2 - t else 0);
  m := ((d + 91) - (m* 3066) / 100;
  d := (d + 91) - (m * 3055) / 100;
  m := m - 2;
  if y ≤ 1900  ∨ y ≥ 2100 then begin
                                 m := 0;
                                 d := 0
                               end
end calendar
```

and these commands in the MODIFY file:

```
REPLACE 0 /1/lendor/)/m,d)/or/ar/nm/n,m/
REPLACE  1      #,;#,m,d;#;#,t;#
INSERT 2%comment
   acm algorithm 398--tableless date conversion
   input   y   the year
           n   day of the year
   output  m   month of the year
           d   day of the month;%
REPLACE 3     ay :at :a
DELETE  4
INSERT 4  ! comment  the following statement is unnecessary
    if it is known that 1900 < y < 2100;
  t := if (y/400) * 400 = y ∨ (y/100) * 100 ≠ y then t else 0;!
REPLACE 6@- (m* 3066) / 100@* 100) /  30558@8@@
DELETE 9 12
```

result in the NEWTEXT file:

```
procedure calendar(y,n,m,d);
  value y,n;  integer y,n,m,d,t;
comment
   acm algorithm 398--tableless date conversion
   input   y   the year
           n   day of the year
   output  m   month of the year
           d   day of the month;
begin
  t := if (y/4)* 4 = y then 1 else 0;
  comment  the following statement is unnecessary
    if it is known that 1900 < y < 2100;
  t := if (y/400) * 400 = y ∨ (y/100) * 100 ≠ y then t else 0;
  d := n + (if n > (59 + t) then 2 - t else 0);
  m := ((d + 91) * 100) / 3055;
  d := (d + 91) - (m * 3055) / 100;
  m := m - 2;
end calendar
```

Notes

'REPLACE 0...' uses multiple scans of line 0. The last 2 replacements are possible only after the first has been accomplished.

'INSERT 2...' is inserting the appropriate 'carriage return' and LF characters used in generating the format of the inserted prose, i.e., they are non-printing characters.

There are 2 commands used on line 4.

'INSERT 4...' Note that the 2 'blanks' preceding the word 'comment' serve to space the prose appropriately in the NEWTEXT file.

'REPLACE 6...' replaces '8' by the null string.

Example 2:

Given the following input TEXT file:

```
The time has come the walrus said
  To speak of many things
    Of sailing ships and sealing wax
      Of cabbages and kings
```

and these REPLACE commands in the MODIFY file:

```
REPLACE 2=Of sailing=SLT chips= ships==seal=whirl=
REPLACE 2 *wax*tracks*
REPLACE 3 4c4B4k4r4
```

The output file NEWTEXT contains the following edited lines:

```
The time has come the walrus said
  To speak of many things
    SLT chips and whirling tracks
      Of Babbages and rings
```

POLYNOMIAL MANIPULATION WITH FAST MULTIPLICATION

R. A. Krutar

Background

This problem is elegant in its simplicity, as is the solution. It touches on the following central concepts of Computer Science: representation of data structures, formula manipulation, and trade-offs in time and space. It provides insight into the input language for LISP. Solutions to the problem will use list languages and pattern matching languages. However, the programming effort is definitely nontrivial--the author's solution is a bit tricky, and the path to any solution contains traps.

The Problem

Several programming languages have been designed as aids in performing formula manipulation. Polynominal manipulation, a special case of formula manipulation, particularly lends itself to the building of efficient systems. The following descrption is taken from Knuth[1].

The problem is to implement a polynomial manipulation program which can take advantage of a fast multiplication rule that reduces the number of multiplications required to calculate $(Ax + B)(Cx + D)$ from the four of the obvious approach to the three needed in:

$$\underset{\text{1st}}{ACx^2} + \underset{\text{1st}}{(AC} + \underset{\text{2nd}}{(A - B)(D - C)} + \underset{\text{3rd}}{BD)x} + \underset{\text{3rd}}{BD}$$

The trade-off is increased addition, subtraction, and shifting. Squaring an n-th degree polynomial takes time proportional to:

$$n^{\log 3} \doteq n^{1.57}$$

rather than n^2 as obtained in the obvious method. Empirical tests and a priori estimates of execution time can be made.

Assume we split a polynomial into two parts: those terms with odd exponents and those with even exponents. We may factor x from each of the odd terms and thereby represent the polynomials as Ax + B where A and B have only even terms and as such are polynomials in x-squared, which can similarly be split. We must permit a constant as a polynomial to limit an infinite regression. A polynomial is then a binary tree with constants at all the leaves. We here use a point as an infix operator in a linear representation of these trees.

The first three examples are from Knuth:

$x = 1 \;.\; 0$

$x^2 = 0 \;.\; (1 \;.\; 0)$

$x^3 - 3x^2 + 3x - 1 = (1 \;.\; 3) \;.\; (-3 \;.\; -1)$

$5x^4 - 7x^2 + 3 = 5x^4 - 7x^2 + 3x^0$

$= 0 * x + [5(x^2)^2 - 7\ (x^2)^1 + 3(x^2)^0]$

$= 0 * x + [-7\ (x^2)^1 + [5\ (x^4)^1 + 3(x^4)^0]$

and this is represented as:

$0 \;.\; (-7.\ (5 \;.\; 3))$

$6x^5 - 4x^3 + 2x = [6x^4 - 4x^2 + 2]\ x + 0$

$= [-4\ (x^2)^1 + [6\ (x^2)^2 + 2\ (x^2)^0]]\ x + 0$

and this is represented as:

$(-4 \;.\; (6 \;.\; 2)) \;.\; 0$

This representation is only on paper. It must be encoded in terms of a representation of a programming language. Fortunately, LISP uses the point as an infix operator to represent binary trees. However, the point is eliminated whenever the right branch is a list or tree, e.g.,

```
1 . 0 = (1 . 0)

0 . (1 . 0) = (0 1 . 0)

(1 . 3) . (-3 . -1) = ((1 . 3) -3 . -1)

0 . (-1/2 . (1/24 . 1)) = (0 -.5 0.04166 . 1)

(-1/6 . (1/120 . 1) . 0 = ((-.16666 0.00833 . 1) . 0)
```

The functions needed for multiplication are: simplification (0 . k = k when k is a constant), addition, subtraction, and multiplication by x. An auxiliary function is also useful. Other interesting functions you may wish to write are: differentiation by x, substitution of a constant or polynomial for x, synthetic division, and translation to and from other representations (the reading and printing finctions).

Test data should either show the special capabilities of each function or be so constructed that the correct result is obvious. In the example below the tests of DX (differentiate by X) and SUBS (substitute for X) generate correct values which are clearly related to the exponents of the test data.

```
DX( (1 1 1 1 1 . 1) )

DX ((1 1 1 1 1 . 1))

VALUE = ((((16 . 8) . 4) . 2) . 1)


SUBS( 10 (1 1 1 1 . 1) )

SUBS (10 (1 1 1 1 . 1 ))

VALUE = 100010111
```

Hints

A constant polynomial has no odd terms and one even term. Primitives which select the odd terms or the even terms or combine two polynomials should take this fact into account.

Reference

[1] Knuth, D.E., "How Fast Can We Multiply?" The Art of Computer Programming: Seminumerical Algorithms, Vol. 2, Sec. 4.3.3.

COROUTINES

Amund Lunde

This problem, implementing coroutines in Algol, requires knowledge of the finer points of the language, such as own variables and switches. It also illustrates one of the tasks of the lexical scanner of a compiler: to interpret the intricacies of a hardware representation. A programmer with a fair knowledge of Algol should be able to program this problem in the allotted time. The concept of coroutines is explained below.

The Coroutine Concept

The coroutine concept is a generalization of the subroutine concept, establishing a completely symmetric relationship between the two (or more) routines, instead of the caller-callee relationship of subroutine calls. That is: when one coroutine transfers control to (or activates) another, a "reactivation point" is set in the former immediately after the activation-statement, and the local data are preserved. When control returns to this routine, execution resumes at the reactivation point using the values of the local data that existed the last time control passed out of this routine. The reactivation point is a generalization of the return address in a subroutine call, but is associated with the caller rather than the callee. Hence, control can be transferred into one coroutine from any other coroutine with which it cooperates and not necessarily from the one into which it passed the control last time.

Coroutines are an important tool in programming, especially in systems programming and in simulation. Nevertheless, coroutine sequencing has not found its way into many of the higher level languages currently

in use. Examples of languages with coroutines are Simula and Simula-67 (a simulation language built on Algol and its generalization), and Bliss (a language for systems programming on the PDP-10, developed at CMU).

The purpose of this problem is to investigate how coroutine-sequencing can be achieved to some extent for Algol procedures. The caller-callee relationship remains to some extent, but a reactivation point may be maintained for each procedure, and local data may be preserved.

The Problem

Many languages, like Algol-60, contain symbols which do not exist on a standard keypunch. Hence, "hardware representations" of these symbols are invented that use only the characters used in, say, Fortran. In one language (Simula-67) part of this hardware representation could be:

SYMBOL	NAME	HARDWARE	
:	colon	..	
:=	becomes	..= OR .=	
:-	denotes	..- OR .-	
;	semicolon	.,	
.	dot	.	(between identifiers)
.	point	.	(not between identifiers)

An early part of the compiler has to replace this notation by a unique and uniform internal representation.

Write two coroutines, "USER" and "GETSYM," to analyze the above representation. The outputs from "GETSYM" should be integers uniquely representing the above (and possibly other) symbols. Since we do not

want to write a compiler now, the "USER" may simply encode these as strings (abbreviations of the names of the symbols) and print them more than one to a line (say, 30 to a line if each string is of length 4).

The string of input-characters should be interpreted left to right so that the largest possible legal combination of characters is used before a symbol is output, i.e.,

..- is denotes (not point denotes, colon minus, or point point minus).

....= is colon becomes (not point colon becomes).

A...B is A colon point B (not A point point point B or A point colon B).

Example: Input and Output

. . . = A B . . = C D . .

50 51 99 51 99 50

where: 50 = colon

51 = becomes

99 = others (one 99 for the group)

You could also encode:

50 into COL

51 into BEC

99 into OTH (for more readable output)

Note:

Students who feel they know all about Algol but want to learn Bliss, may program the problem in Bliss, using the standard coroutine facilities.

Reference

[1] Knuth, D.E., Fundamental Algorithms, p. 190 ff., p. 226.

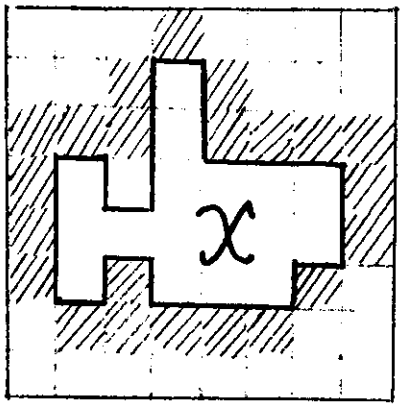
AREA OF A REGION

Leroy C. Richardson
Richard M. Young

Background

A region of two-dimensional space is divided into uniform square cells, each of which is designated as being either "white" or "black." The black cells form a connected mass, so that by stepping horizontally, vertically, or diagonally it is possible to move from any black cell to any other, passing only over black cells. Thus the black cells divide the set of white cells into isolated regions; there are no separate "islands" of black cells.

We are interested in finding the area of a region of horizontally or vertically connected white cells totally enclosed by a boundary of black cells. For example, the area marked X:



Choose a representation, such as a two-dimensional array, in which the basic operations available are to determine whether a cell is black or white, and to move from a cell to any of its four neighbors. Assume that you are given the location of a white cell in the region whose area is required.

1.) Since the area of the white region is defined to be the number of cells in it, the most straightforward way to compute the area is simply to go ahead and count the cells. Write a program to do this; it will have to visit each cell in the region at least once.

Hints

A.) Be careful not to count white cells which do not in fact belong to the region whose area is wanted.

B.) This technique is quite straightforward and there are many ways to write the program. Try to find a program which is elegant and reflects the structure of the task. You may want to write several different versions, to see how different programming languages lend themselves most naturally to iterative or recursive control structures.

2.) For large, sensibly-shaped regions, visiting every cell in the region is inefficient. By making use of some very simple algebraic properties, we can determine the area simply from knowledge of its boundary. There is no need to examine the cells in the interior.

Can you write a program which computes the area by visiting only white cells adjacent to the boundary?

Hints

A.) Using (x,y) coordinates to describe the white region, we can regard the whole area as composed of a number of columns of vertically connected cells. Suppose the y-coordinates of the top and bottom cells in column i are $YTOP_i$ and $YBOT_i$. Then we know that

$$\text{area} = \sum (YTOP_i - YBOT_i + 1)$$

where the summation ranges over all the columns composing the region.

B.) The process of tracing around the boundary of a region is known as "edge-following" and is interesting in itself. To trace clockwise around a region is analogous to walking around the whole of a room while always keeping one's left hand touching the wall.

Try using this analogy if you have difficulty programming the edge-follower. The secret is always to keep turning "as left as you can."

C.) Once again, try writing the code so that it corresponds elegantly and clearly to the structure of the task. If you still have difficulty with the edge-follower, it may actually help to draw an elegant flow-chart first, and then encode it.

3.) One way to approach the task of finding the area is to think of the initially given white cell as a "seed," which is "grown" to cover all the white cells immediately adjacent to it, each of which is then also grown outwards to cover all the white cells adjacent to it, and so on until the whole region is filled up. The area then is the total number of cells grown (including the original seed).

Suppose you have available a programming system which can operate simultaneously (in a single operation) on the whole of an array at once; i.e., in each cycle of computation the whole connected mass of white cells already reached can be expanded outwards by one cell in just one operation or statement in the programming language. Can you devise a simple algorithm that takes advantage of these array operations to find the area of the region?

We suggest using either of two approaches:

3.1) Program the algorithm in APL, which effectively provides simultaneous operations on arrays.

3.2) Assume that you have available a computer capable of working with arbitrarily long bit-strings as words. Assume a reasonable set of operations for the machine: parallel logical operations, shifting, counting the number of 1's in a word, etc.

Can you devise an appropriate representation of two-dimensional regions as bit-strings, and write an area-finding algorithm that takes advantage of the parallelism of such a machine?

Hints

A.) How do you tell when the whole region has been covered? What happens on subsequent cycles?

B.) "Growing" a single cell is equivalent to shifting it one cell up, down, left, and right (if the adjacent cells are also white) and "superimposing" the five cells. Can you generalize this to a whole connected mass of cells?

A PROBLEM IN SIMPLE LANGUAGES

George Robertson

Motivation

Before considering a complex language such as ALGOL, it is convenient to study a very simplified form of language which has only a few simple syntax rules. The results of this study can then be extended to a subset of the Algol language which can in turn form the basis for constructing a translator for Algol-like languages.

A language consists of a set of basic symbols (usually finite) called the alphabet and certain strings of these symbols. Its syntax consists of rules for classifying and transforming these strings into words. By a string we mean a finite sequence of symbols from the alphabet which may be exhibited by writing the symbols in linear order from left to right. We shall denote strings by Greek letters. If α and β are strings, then "$\alpha\beta$" shall denote the string consisting of the symbols of α followed in order by those of β. We can define a function L, called the length, as follows:

D1. If α is a string, then $L(\alpha)$ = number of symbols in α counting repetitions.

In other words, the function L maps strings onto the set of non-negative integers. Two strings will be considered the same if

1. They have the same length, and

2. They have identical symbols in the same positions.

One of the more useful languages for mathematical purposes is leading operator, or prefix, "Polish" notation. The rules of word formation in

this case are very simple. The symbols in the alphabet are classified as letters and connectives, and associated with each connective is a unique positive integer, n, called the degree of the connective. The two rules for word formation are:

W1. A string consisting of a single letter is a word.

W2. If α is a connective of degree n, and β_1, $\beta_2 \ldots \beta_n$ are words then $\alpha\beta_1 \ \beta_2 \ldots \beta_n$ is a word.

The use of a leading connective structure eliminates the necessity of parentheses, either explicit or implied by operator heirarchy.

As an example, let us consider Algol-like simple arithmetic expressions defined by the following syntax:

<letter> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

<adding operator> ::= +|-

<multiplying operator> ::= *|/

<primary> ::= <letter>|(<simple arithmetic expression>)

<term> ::= <primary>|<term> <multiplying operator> <primary>

<simple arithmetic expression> ::= <term>|

<simple arithmetic expression> <adding operator> <term>

Then, the alphabet of the simple "Polish" notation becomes:

1. A,B...,Z as letters

2. +,-,*,/ as connectives of degree 2

Examples of simple arithmetic expressions in both the Algol-like and the Polish notations are:

	Algol-like notation	"Polish" notation
A.	(A + B) * C/D	/ * + A B C D
B.	A * (B + C / D)	* A + B / C D
C.	A * B + C / D	+ * A B / C D
D.	A *(B + C) / D	/ * A + B C D

We are now in a position to define a simple language.

D2. A language $\mathcal{L}$ is <u>simple</u> if its alphabet consists only of letters and connectives, and if W1 and W2 are the rules of word formation in $\mathcal{L}$.

We can define a function ρ, called the rank, which has as its domain all strings in $\mathcal{L}$ and its range will be the set of integers. The definition is as follows:

D3. 1. If σ is a letter, then $\rho(\sigma) = -1$.

2. If σ is a connective of degree n, then $\rho(\sigma) = n-1$.

3. If σ is the null string, then $\rho(\sigma) = 0$.

4. If $\sigma = \sigma_1\sigma_2$ and $L(\sigma_1) = 1$, then $\rho(\sigma) = \rho(\sigma_1) + \rho(\sigma_2)$.

Thus if σ is "$a_1a_2...a_k$", and "a_i" is a letter or connective for each i, then

$$\rho(\sigma) = \rho(a_1) + \rho(a_2) + \ldots + \rho(a_k) = \sum_{i=1}^{k} \rho(a_i).$$

and we see that the rank operation ρ is additive.

A question that we would now like to answer is: If we are given an arbitrary string σ in language $\mathcal{L}$, then can we determine if σ is a word in $\mathcal{L}$ by a purely mechanistic approach? In other words, does an algorithm

exist for determining whether a string σ in $\mathcal{L}$ is a word in $\mathcal{L}$? The answer to the question is in the affirmative and is based on an important theorem due to Rosenbloom. [1]

D4. If σ is a string in $\mathcal{L}$, and $\sigma = \sigma_1\sigma_2$, then σ_1 is a head of σ and σ_2 is a tail of σ.

Rosenbloom's theorem can be stated as follows:

T1. If $\mathcal{L}$ is a simple language, and σ is a string in $\mathcal{L}$, then σ is a word in $\mathcal{L}$ if and only if

1. $\rho(\sigma) = -1$, and

2. If σ_1 is any head of σ, and $\sigma_1 \neq \sigma$, then $\rho(\sigma_1) \geq 0$.

The proof of Rosenbloom's theorem can be found in his book along with some suggested exercises.

The Problem

Write a LISP function called WORD which will determine whether or not a string σ is a legal word in Polish prefix notation. The argument to the function should be a list representing the string σ, and the value of the function should be either T or NIL.

Examples:

WORD ((+ * A B / C D)) should return the value T.

WORD ((A * B + C / D)) should return the value NIL (Rule 2).

WORD ((+ * A / B C D)) returns T.

WORD ((+ * / A - B C)) returns NIL (Rule 1).

Hints

Once you have convinced yourself that Polish prefix notation is a simple language in the sense of definition D2, then the problem reduces to a problem of implementing the algorithm described in Rosenbloom's theorem. You will find that the key to the implementation involves substituting the ranks of symbols in the input list for the symbols themselves. Hence, a table look-up procedure of some kind is needed. A careful examination of the LISP interpreter (both EVAL and APPLY) will reveal that a useful table look-up procedure does exist in LISP.

Reference

[1] Rosenbloom, Paul, The Elements of Mathematical Logic, Dover, 1950, pp. 152-157.

TURING MACHINE SIMULATION PROBLEM

Larry Snyder

Motivation

Even before the invention of modern computers, A. M. Turing [4] described a theoretical model of a computing machine. Although very simple in structure, the Turing machine (under a plausible set of assumptions) has been proven to possess some very remarkable properties. For example, a Turing machine can compute any function that can possibly be computed. There are well defined functions which no Turing machine (and hence no computer) can compute the solution to. Given a Turing machine program for certain functions, there is a Turing machine program for the same function which will run faster [1]. These and other results will be discussed later. Our interest here is to develop a thorough understanding of the workings of this simple machine and to develop a program which may be used later in the Immigration Course when non-computability is studied using the Busy Beaver Problem [2]. In addition there are several programming techniques which this problem is intended to emphasize, namely, the building of a programming model on which experiments are to be run, gaining expertise in some conversational programming language and experience with data structures and storage allocation.

The Problem

Choose a conversational programming language and write a program to simulate a Turing machine. (For those who aren't familiar with Turing machines, a good description is found in Minsky [3], reprinted at the end of this problem description.) The program should be highly interactive and allow you to specify machines and tapes conveniently and to monitor their behavior. Keep in mind that you will be running experiments with your program later during the Immigration Course. Your program should allow:

1.) Specification of the tape alphabet, the Turing machine itself and the initial tape configuration.

2.) Specification of experiment parameters:

A.) Initial state and read head position.

B.) Maximum number of state transitions, and maximum amount of Usage. (This is because many Turing machines never halt and you want to prevent infinite cycling.)

3.) Tracing facilities to allow monitoring of state transitions while the Turing machine is running.

4.) Printing of all relevant information, e.g., the tape, states, read head position, etc.

Sample Problems

It might be helpful to prepare several Turing machines to be used while debugging your simulator. Here are several suggestions:

1.) Addition of two integers represented in:

A.) Unary marks (e.g., the integer i is represented with with i+1 marks). This problem is trivial.

B.) Binary. This is more challenging. Think of various tape formats to simplify the problem.

C.) Decimal. This is quite complex.

2.) Checking for well formed parenthesis sequences, i.e., a machine to accept sequences like ()(()) and reject ((). A solution is in Minsky, but try it yourself before looking.

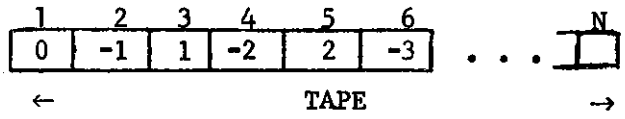
3.) Accepting a unary sequence if it has 2^i marks, for any non-negative i. This one is easy.

4.) A machine which prints its own description in quintuples. This problem is reasonably difficult.

Things to Watch for

One of the important decisions you must make is how to represent the "infinite" tape. Obviously, your representation will be finite, but be sure it is flexible enough. Here are two possible representations (you may think of others):

1.)

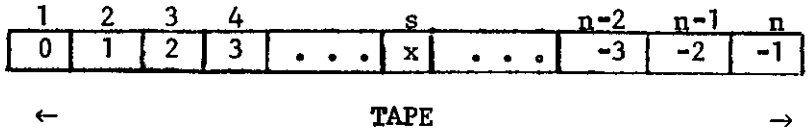


The tape vector is a vector of length n. TAPE[1] is the 0 cell, all odd numbered elements are positive cells, all even numbered elements are negative cells, such that:

$$\text{CELL}[i] = \begin{cases} \text{TAPE}[1] \text{ if } i=0 \\ \text{TAPE}[2i+1] \text{ if } i>0 \\ \text{TAPE}[2i] \text{ if } i<0 \end{cases}$$

This model is easily extended if additional tape is needed.

2.)



The tape is a vector of length n. The non-negative cells begin at TAPE[1] and go to some limit s<n. The negative cells begin at TAPE[n] and are stored backwards to the limit s, such that:

$$\text{CELL}[i] = \begin{cases} \text{TAPE}[i+1] & \text{for } i \geq 0 \\ \text{TAPE}[n+1] & \text{for } i<0 \end{cases}$$

There are at least two other representations you might consider using.

Another thing to keep in mind is that after a tape or a machine has been specified, it should also be easy to correct any errors in the initial specification. Experiments are usually wrong the first time they are stated.

Finally, one comment about the use of a conversational language. Contrary to popular belief, it is difficult and time consuming to compose a program at the terminal. This is especially true if you are not very familiar with the language. Your time will be most productive if you have your program entirely composed BEFORE you sit down at the terminal.

Remember, this program should be as convenient as possible for you to use.

References

[1] Blum, Manuel, "A Machine Independent Theory of the Complexity of Recursive Functions," JACM, Vol. 14, No. 2, pp. 322-336.

[2] Lin, Shen and Tibor Rado, "Computer Studies of Turing Machine Problems," JACM, Vol. 12, No. 2, pp. 196-213.

[3] Minsky, Marvin, Computation: Finite and Infinite Machines, Prentice-Hall, Englewood Cliffs, 1967, pp. 117-119.

[4] Turing, Alan M., "On Computable Numbers, with an Application to the Entscheidungsproblem," Proc. London Math. Soc., 1936, Sec. 2-42, pp. 230-265.

```
AMT

-------------------------------------------------------------
THIS PROGRAM SIMULATES A TURING MACHINE WITH A TWO-WAY TAPE.
FACILITIES ARE PROVIDED FOR DEFINING MACHINES, RUNNING EXPER-
IMENTS AND DEBUGGING EXPERIMENTS.  THE FOLLOWING COMMANDS ARE
USED TO CONTROL THE SIMULATION:
   S:XXXX INDICATES THAT A SPECIFICATION OF XXXX IS TO BE MADE
   P:XXXX INDICATES THAT THE VALUE OF XXXX IS BEING REQUESTED
   N:YYYY INDICATES A NEW YYYY IS TO BE SPECIFIED
   GO     STARTS THE TURING MACHINE
   END    TERMINATES THIS PROGRAM
   ?      PRINTS THIS DESCRIPTION AGAIN
   @      IS USED FOR COMMENTS
THE FOLLOWING ARE VALID ENTRIES FOR XXXX ABOVE:
   ALPHABET           SPECIFIES THE TAPE ALPHABET
   QUINTUPLES         SPECIFIES THE STATE QUINTUPLES
   TAPE               SPECIFIES THE TAPE CONFIGURATION
   STATE              SPECIFIES THE STATE OF THE MACHINE
   CELL               SPECIFIES THE HEAD POSITION ON THE TAPE
   TRACE              SPECIFIES THE OPTION TO TRACE STATE TRANS.
   TRANSITION LIMIT   MAXIMUM ALLOWABLE STATE TRANSITIONS WITHOUT
                      INTERVENTION
   STORAGE            SPECIFIES THE MAXIMUM NUMBER OF TAPE CELLS
THE FOLLOWING ARE VALID ENTRIES FOR YYYY ABOVE:
   MACHINE            INDICATES A NEW MACHINE IS TO BE SPECIFIED
   EXPERIMENT         INDICATES A NEW EXPERIMENT IS TO BE SPECIFIED
WHEN AN EXPERIMENT HAS BEEN SPECIFIED, GO STARTS IT GOING.
-------------------------------------------------------------
```

```
)
A  LET'S DEFINE A TURING MACHINE TO COMPUTE
A  EXCLUSIVE-OR OF TWO BINARY STRINGS.  OUR
A  TAPE WILL HAVE THE FOLLOWING FORMAT:
A  <BIN STRING 1> ≠ <BIN STRING 2> → <RESULT>
A  WITH □'S AND ○'S AS MARKERS FOR PROCESSED
A  PORTIONS OF STRINGS
)
N:MACHINE
THE ALPHABET CURRENTLY CONTAINS: B
PLEASE ENTER TAPE ALPHABET: SINGLE CHARACTERS SEPERATED BY COMMAS
0,1,□,○,→,≠
ENTER STATE QUINTUPLES: STATE, READ, NEW STATE, WRITE, MOVE
SEPERATED BY COMMAS, SO THAT THE FOLLOWING DOMAINS APPLY:
STATE, NEW STATE ARE POSITIVE INTEGERS, 0 FOR HALT
READ, WRITE ∈ B01□○→≠
MOVE ∈ L,R,-
..... ENTER DONE TO TERMINATE STATE ASSIGNMENT
|
1,1,2,□,R
|
1,0,3,□,R
|
2,0,2,0,R
|
2,1,2,1,R
|
2,≠,4,≠,R
|
3,0,3,0,R
|
3,1,3,1,R
|
3,≠,5,≠,R
|
4,○,4,○,R
|
4,0,6,○,R
|
4,1,7,○,R
|
5,○,5,○,R
|
5,0,7,○,R
|
5,1,6,○,R
|
6,0,6,0,R
|
6,1,6,1,R
|
6,→,6,→,R
|
6,B,8,1,R
```

```
|
7,0,7,0,R
|
7,1,7,1,R
|
7,→,7,→,R
|
7,B,8,0,L
|
8,0,8,0,L
|
8,1,8,1,L
|
8,○,8,○,L
|
8,→,8,→,L
|
8,≠,8,≠,L
|
8,□,1,□,R
|
DONE
∘
A  WE MAY NOW SPECIFY AN EXPERIMENT
∘
N:EXPERIMENT
SPECIFY INITIAL TAPE INPUT (BEGINNING ON CELL [0])
1101≠1000→
SPECIFY CELL ON WHICH READ HEAD SHOULD BE POSITIONED
[]:
     0
SPECIFY FIRST STATE
[]:
     1
SPECIFY TRACE: 0 NO TRACE, 1 TRACE
[]:
     1
SPECIFY MAXIMUM STATE TRANSITIONS
[]:
     5
∘
A  OK, WE ARE READY TO GO
∘
GO
112□R
2121R
2020R
2121R
2≠4≠R
TRANSITION LIMIT REACHED
∘
A  OUR TURING MACHINE HAS RUN FOR FIVE TRANSITIONS AND HAS
A  STOPPED TO ALLOW US TO LOOK AT SOME OF THE VALUES.  WE
```

```
⍝  MAY PRINT THE CURRENT VALUE OF THE TAPE.
⍝
P:TAPE
10 TAPE CELLS WERE USED
[0]□101≠1000→
⍝
⍝  THE [0] INDICATES THAT THE PORTION OF THE 'INFINITE' TAPE
⍝  WHICH HAS BEEN PRINTED BEGINS WITH THE CELL  0 .
⍝
P:CELL
CURRENT HEAD POSITION IS: 5
⍝
P:STATE
CURRENT STATE IS: 4
⍝
⍝  NOT MUCH HAS HAPPENED, LET'S CONTINUE
⍝
GO
4170R
7070R
7070R
7070R
7→7→R
TRANSITION LIMIT REACHED
⍝
⍝  THIS IS TOO TEDIOUS, LETS CHANGE SOME PARAMETERS
⍝
S:TRANSITIONS
SPECIFY MAXIMUM STATE TRANSITIONS
□:
     100
⍝
S:TRACE
SPECIFY TRACE: 0 NO TRACE, 1 TRACE
□:
     0
⍝
GO
MACHINE HALTED
⍝
P:TAPE
12 TAPE CELLS WERE USED
[0]□□01≠0000→01
⍝
⍝  OOPS, WE GOOFED SOMEWHERE!
⍝  WHERE IS THE READ HEAD?
⍝
P:CELL
CURRENT HEAD POSITION IS: 12
⍝
⍝  LET'S PRINT OUT THE MACHINE
⍝
P:QUINTUPLES
STATE TRANSITION MATRICES
```

```
1 0 3 □ R
1 1 2 □ R

2 0 2 0 R
2 1 2 1 R
2 ≠ 4 ≠ R

3 0 3 0 R
3 1 3 1 R
3 ≠ 5 ≠ R

4 0 6 0 R
4 1 7 0 R
4 0 4 0 R

5 0 7 0 R
5 1 6 0 R
5 0 5 0 R

6 B 8 1 R
6 0 6 0 R
6 1 6 1 R
6 → 6 → R

7 B 8 0 L
7 0 7 0 R
7 1 7 1 R
7 → 7 → R

8 0 8 0 L
8 1 8 1 L
8 □ 1 □ R
8 0 8 0 L
8 → 8 → L
8 ≠ 8 ≠ L

°
A  IF WE DIDN'T KNOW WHAT WAS WRONG, WE WOULD PROBABLY RERUN
A  THE EXPERIMENT WITH THE TRACE ON.  HOWEVER, I HAVE REASON
A  TO BELIEVE THAT THE ERROR IS IN THE FIRST QUINTUPLE OF STATE 6.
°
S:QUINTUPLES
ENTER STATE QUINTUPLES: STATE, READ, NEW STATE, WRITE, MOVE
SEPERATED BY COMMAS, SO THAT THE FOLLOWING DOMAINS APPLY:
STATE, NEW STATE ARE POSITIVE INTEGERS, 0 FOR HALT
READ, WRITE ∈ B01□0→≠
MOVE ∈ L,R,-
..... ENTER DONE TO TERMINATE STATE ASSIGNMENT
|
6,B,8,1,L
|
DONE
°
A  LET'S SEE IF THAT FIXES IT.
```

```
A  WE'LL MANUALLY MOVE THE READ HEAD LEFT ONE CELL AND START
A  THE MACHINE IN STATE 8 (THE SKIP LEFT LOOP)
o
P:CELL
CURRENT HEAD POSITION IS: 12
o
S:CELL
SPECIFY CELL ON WHICH READ HEAD SHOULD BE POSITIONED
[]:
     11
o
P:STATE
CURRENT STATE IS: 0
o
S:STATE
SPECIFY FIRST STATE
[]:
     8
o
A  WE SHOULD BE READY TO CONTINUE
o
FO
WHAT?
o
A SORRY ABOUT THAT
o
GO
MACHINE HALTED
o
P:TAPE
14 TAPE CELLS WERE USED
[0][][][]≠0000→0101
o
P:CELL
CURRENT HEAD POSITION IS: 4
o
A  THAT'S THE END OF THE EXPERIMENT
A  THANX FOR TURING WITH US!
o
END
```

6 TURING MACHINES

6.0 INTRODUCTION

A Turing machine is a finite-state machine associated with an external storage or memory medium. This medium has the form of a sequence of *squares*, marked off on a linear *tape*. The machine is coupled to the tape through a *head*, which is situated, at each moment, on some square of the tape (Fig. 6.0-1). The head has three functions, all of which are exercised in each operation cycle of the finite-state machine. These functions are: *reading* the square of the tape being "scanned," *writing* on the scanned square, and *moving* the machine to an adjacent square (which becomes the scanned square in the next operation cycle).

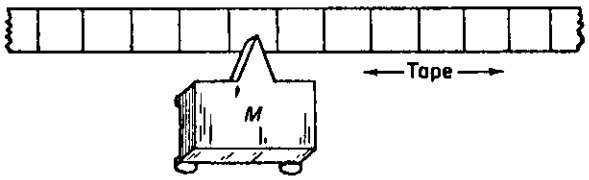


Fig. 6.0-1

It will be recalled from section 2.2 that a finite-state machine is characterized by an alphabet $(s_0, \ldots, s_m)$ of input symbols, an alphabet $(r_0, \ldots, r_n)$ of output symbols, a set $(q_0, \ldots, q_p)$ of internal states, and a pair of functions

$$Q(t + 1) = G(Q(t), S(t))$$

$$R(t + 1) = F(Q(t), S(t))$$

which describe the relation between input, internal state, and subsequent behavior.

In order to attach the external tape, it is convenient to modify this description a little. The input symbols $(s_0, \ldots, s_m)$ will remain the same, and it will be precisely these that may be inscribed on the tape, one symbol per square. The input to the machine M, at the time t, will be just that symbol printed in the square the machine is scanning at that moment. *The resulting change in state will then be determined, as before, by the function G. The output of the machine M has now the dual function* of (1) *writing on the scanned square* (perhaps changing the symbol already there) and (2) *moving the tape one way or the other.*

Thus R, the response, has *two* components. One component of the response is simply a symbol, from the same set $(s_0, \ldots, s_m)$, to be printed on the scanned square; the second component is one or the other of two symbols '0' (meaning "Move left") and '1' ("Move right"), which have the corresponding effect on the machine's position. Accordingly, it is convenient to think of the Turing machine as described by *three* functions

$$\begin{aligned} Q(t+1) &= G(Q(t), S(t)) \\ R(t+1) &= F(Q(t), S(t)) \\ D(t+1) &= D(Q(t), S(t)) \end{aligned}$$

where the new function 'D' tells which way the machine will move.

In each operation cycle the machine starts in some state q_i, reads the symbol s_j written on the square under the head, prints there the new symbol $F(q_i, s_j)$, moves left or right according to $D(q_i, s_j)$, and then enters the new state $G(q_i, s_j)$.

When a symbol is printed on the tape, the symbol previously there is erased. Of course, one can preserve it by printing the same symbol that was read, i.e., if $F(q_i, s_j)$ happens to be s_j. Because the machine can move either way along the tape, it is possible for it to return to a previously printed location to recover the information inscribed there. As we will see, this makes it possible to use the tape for the storage of arbitrarily large amounts of useful information. We will give examples shortly.

The tape is regarded as infinite in both directions. But we will make the restriction that *when the machine is started the tape must be blank, except for some finite number of squares.* With this restriction one can think of the tape as really finite at any particular time but with the provision, whenever the machine comes to an end of the finite portion, someone will attach another square.

Formal mathematical descriptions of Turing machines may be found in Turing [1936], Post [1943], Kleene [1952], Davis [1958]. There are unimportant technical differences in these formulations. For our purposes it will usually be sufficient to use pictorial state diagrams. Our immediate

purpose is to show how Turing machines, with their unlimited tape memory, can perform computations beyond the capacity of finite-state machines; it is usually easier to understand the examples in terms of diagrams than in terms of tables of functions. While it is fresh in our minds, however, let us note that the finite-state parts of our machines can be described nicely by sets of *quintuples* of the form

(old state, symbol scanned, new state, symbol written, direction of motion)

i.e.,

$$(q_i, s_j, G(q_i, s_j), F(q_i, s_j), D(q_i, s_j))$$

or

$$(q_i, s_j, q_{ij}, s_{ij}, d_{ij})$$

i.e., as quintuples in which the third, fourth, and fifth symbols are determined by the first and second through the three functions G, F, and D mentioned above.†

Thus a certain Turing machine (section 6.1.1 below) would be described by the following six quintuples:

$(q_0$,	0,	q_0,	0,	$R)$	$(q_1$,	0,	q_1,	0,	$R)$
$(q_0$,	1,	q_1,	0,	$R)$	$(q_1$,	1,	q_0,	0,	$R)$
$(q_0$,	B,	HALT,	0,	-)	$(q_1$,	B,	HALT,	1,	-)

or just

(0,	0,	0,	0,	1)	(1,	0,	1,	0,	1)
(0,	1,	1,	0,	1)	(1,	1,	0,	0,	1)
(0,	B,	H,	0,	-)	(1,	B,	H,	1,	-)

where we have reserved the symbol 'H' (or 'HALT') to designate a halting state.

One more remark. When we dealt with finite-state machines and the things they could do, we had to regard the input data as coming from some *environment*, so that the description of a computation was usually not contained completely in the description of the machine and its initial state. With a Turing machine tape we have now a *closed* system, for the tape serves as environment for the finite-state machine part. *Hence we can specify a "computation" completely by giving* (1) *the initial state of the machine* and (2a) *the contents of the tape.* Of course we have also to say (2b) *which square of the tape the scanning head sees at the start.* We will usually assume the machine starts in state q_0.

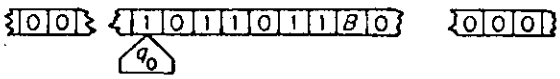
†The state denoted by q_{ij} is defined to be that one of the q_i's given by the function $G(q_i, s_j)$ and similarly for s_{ij} and for d_{ij}.

6.1 SOME EXAMPLES OF TURING MACHINES

The remainder of this chapter shows some of the things Turing machines can do to the information placed on their tapes, and contrasts these processes with those obtainable from finite-state machines. (For the comparison, one may think of a finite-state machine as a specially restricted kind of Turing machine which can move in only one direction.)

6.1.1 A parity counter

We will set up a machine whose output is 1 or 0 depending on whether the number of 1's in a string of 1's and 0's is odd or even. The input string is represented on the Turing machine's tape in the form



where we have printed the sequence in question followed by a B. The machine starts (in state q_0) at the beginning of the sequence; the B is to tell the machine where the sequence ends. The machine needs two states, one for odd and one for even parity, and it changes state whenever it encounters a 1. The associated finite-state machine is represented by Table 6.1-1.

Table 6.1-1. QUINTUPLES FOR PARITY COUNTER

q_i	s_j	q_{ij}	s_{ij}	d_{ij}
0	0	0	0	1
0	1	1	0	1
0	B	H	0	-

q_0

q_i	s_j	q_{ij}	s_{ij}	d_{ij}
1	0	1	0	1
1	1	0	0	1
1	B	H	1	-

q_1

If we trace the operation of the machine we find that it goes through the configurations at the top of p. 121.

The machine ends up at the former site of the terminal B which it has replaced by the answer. The input sequence has been erased.

PROBLEM. Change the quintuples so that the sequence is not erased.

In this simple example the machine always moves to the right. In such a case there is no possibility of recording information on the tape and returning to it at a later time. Hence one could not expect it to do anything that could not also be done by an unaided finite-state machine (with sequential input) and we know already, from section 2.2, that this is true for this computation.

THE FIRING SQUAD SYNCHRONIZATION PROBLEM

Tim Teitelbaum

This is a problem within a problem, which combines a small piece of the theory of finite state machines with the practice of interactive programming and system building.

First of all, we have the firing squad problem itself as devised by Myhill and described in Moore[3]:

Consider a finite (but arbitrarily long) one-dimensional array of finite-state machines all of which are alike except the ones at each end. The machines are called soldiers, and one of the end machines is called a General. The machines are synchronous, and the state of each machine at time t + 1 depends on the states of itself and of its two neighbors at time t. The problem is to specify the states and transitions of the soldiers in such a way that the General can cause them to go into one particular terminal state (i.e., they fire their guns) all at exactly the same time. At the beginning state (i.e., t = 0), all the soldiers are assumed to be in a single state, the quiescent state. When the General undergoes the transition into the state labeled "fire when ready," he does not take any initiative afterwards, and the rest is up to the soldiers. The signal can propagate down the line no faster than one soldier per unit of time, and their problem is how to get all coordinated and in rhythm. The tricky part of the problem is that the same kind of soldier with a fixed number K of states is required to be able to do this, regardless of the length n of the firing squad. In particular, the soldier with K states should work correctly, even when n is much larger then K. Roughly speaking, none of the soldiers is permitted to count as high as n.

Two of the soldiers, the General and the soldier farthest from the General, are allowed to be slightly different from the other soldiers in being able to act without having soldiers on both sides of them, but their structure must also be independent of n.

A convenient way of indicating a solution of this problem is to use a piece of graph paper, with the horizontal coordinate representing the spatial position, and the vertical coordinate representing time. Within the (i,j) square of the graph paper a symbol may be written, indicating the state of the ith soldier at time j. Visual examination of the pattern of propagation of these symbols can indicate what kinds of signaling must take place between the soldiers.

* * *

Since the solution of this problem involves considerable busy-work, it will be convenient for you to have the aid of a computer program. What this program does constitutes the second part of this problem and is entirely up to you. It could only verify your candidate solutions or, at the opposite extreme, it might (try to) generate the entire solution for you.

Such a program, if written in an interactive programming language, could be used to develop the solution strategy incrementally. Thus, you could first concentrate on developing a conversational system for programming, debugging, and editing the soldiers' rules; then you can use your system to work on the firing squad problem per se.

Consider the task of optimizing your own total time. What is the trade-off between time spent incorporating features in your computer program versus effort expended directly on the design of the soldiers program?

If, after all due effort, you haven't made any progress, you may wish to toss in the towel and refer to the solution strategy description

given in Minsky[2]. (But do yourself a favor and don't give up until desperate.)

If, on the other hand, you have found a solution, you may wish to consider finding solutions which optimize the time or number of states required. An eight-state minimum time solution (2n-2) may be found in the CMU thesis by Balzer[1].

References

[1] Balzer, R. M., "Studies Concerning Minimal Time Solutions to the Firing Squad Synchronization Problem," CMU Computer Science Department Ph.D. thesis, 1966.

[2] Minsky, M., Computation, Finite and Infinite Machines, Prentice Hall, p. 282.

[3] Moore, E. F., Sequential Machines, Selected Papers, Addison-Wesley, 1964, pp. 213-214.

TREES, TREES, TREES

Tim Teitelbaum

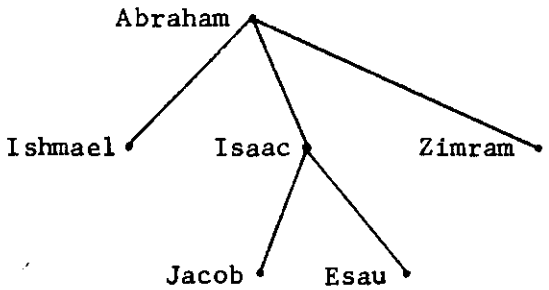
Question. Could you help me--I'm a little confused?

Answer. Sure, what's your problem?

Q. What kind of data objects are manipulated by LISP programs?

A. Trees.

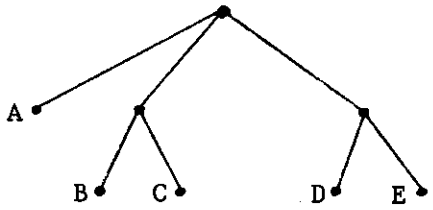
Q. Oh, I get it. Something like:



A. Not exactly. More like: (A (B C) (D E)) .

Q. I don't get it. Why is that a tree?

A. Because you can think of it as being:

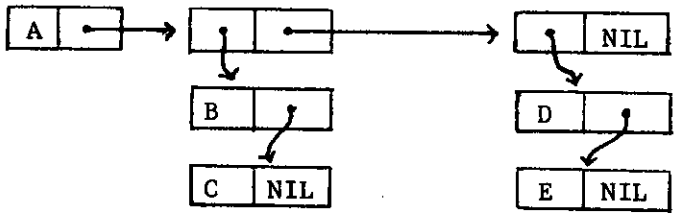


Q. But only the terminal nodes of your tree have data on them.

A. Tough! Those are the rules.

Q. So a LISP tree really looks like that?

A. No. It really looks like:



Q. OK, Forget it.

As you can see, there is no one data type which is a tree. There are, in fact, many species of trees, each with its own sub-species and mutations. The subject of tree structures (and related objects like lists) is confusing but very important. The purpose of the following problem is twofold:

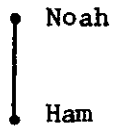
1) It is a means of helping you understand and differentiate between various tree structures.
2) It is a small (though non-trivial) exercise in the LISP programming language.

Problem

Consider the list L of father-son pairs:

```
L = ( (Isaác  Esau)   (Abraham  Ishmael)   (Abraham  Zimram)
      (Noah    Ham)   (Isaac      Jacob)   (Abraham   Isaac) ).
```

This list corresponds in a fairly obvious way to a forest of two trees -- the one given above and the other, a separate family tree:



However, since the information is distributed throughout the list, L is a fairly useless representation. This is especially true if we wish to perform operations like:

Extract the decendants of x

Extract the linage of x

Form a list of all first cousins.

Assuming these are the types of operations desired, your problem is:

1) To specify a suitable format for representing in LISP a forest of family trees (ie., trees with data at all nodes).

2) To program in LISP a function tree(x) to transform a list of father-son pairs (like L above) to the format specified in part 1) above. Note that L is not sorted in any particular order -- it's harder this way. (It would be very educational to code tree(x) twice: once in "pure LISP" and once using the full power of LISP 1.5, eg., the prog feature, property lists, rplaca, rplacd, etc.)

DEPARTMENT OF COMPUTER SCIENCE

GRADUATE QUALIFYING EXAMINATION

TAKE HOME EXAM

NOVEMBER 3, 1970

The four functions described on pages 3 and 4 provide a behavioral definition of a "register" module, i.e., some object called a register which holds a string of characters, each character being a non-negative integer. The register may hold p_1 characters (p_1 may be quite large). Each character comes from an alphabet containing p_2 distinct characters.

Naturally these functions can be implemented in many ways. The main parameter of implementation is the representation of the register. Two important criteria governing choice of implementation are:

1. Execution speed of the functions.

2. Space requirement for the register.

1. Select at least two representations of the register and for each implement in BASIC on the PDP-10 the four functions such that there is extensive variation in their performance with respect to the two criteria, i.e., different representations should tend to optimize different criteria. Make some sensible assumptions about the ranges of p_1 and p_2.

2. Suppose you are given:

   A. "Frequencies" of the use of the four functions x_1, x_2, x_3 and x_4, e.g., as arising in some application, i.e.,

$$0 \leq x_i \leq 1, \quad i = 1,2,3,4$$

$$\sum_{1}^{4} x_i = 1$$

B. p_1 and p_2.

C. "Costs" c_t and c_s representing the costs associated with use of time and space in the application.

Write a BASIC program SELECT which solves the implementation selection problem expressed as the integer programming problem

$$\underset{I}{\text{minimize}} \quad Z(I) = \sum_{i=j}^{n} \sum_{j=i}^{4} (c_t t_{ij} + c_s s_{ij}) x_j I_i$$

$$\text{subject to} \quad \sum_{i=1}^{n} I_i = 1$$

$$I_i = 0 \text{ or } 1, \quad i = 1,2,\ldots n$$

where t_{ij} and s_{ij} are the execution times and the storage requirements, respectively, of the j^{th} function in the i^{th} implementation. Remember:

n should be ≥ 2.

Naturally, t_{ij} and s_{ij} are functions of p_1 and p_2.

3. How would you alter the formulation of section 2 to take into account:

A. Space and time constraints which might be imposed by the application?

B. Transformations of the register representation? Over the space of implementations and for some sets $\{p_1, p_2, x_1, x_2, x_3, x_4\}$ a smaller value of Z(I) is attainable if trans-

formations of the register representation are added to the set of four functions. How would you assign frequencies to these transformations? Specify the transformation functions in the manner in which the four basic functions are given. No programs are required for the answers to section 3.

4. Your examination answers should include:

   A. Listings of the BASIC programs for SELECT and for the four functions in their several guises. Each should contain sufficient comments to explain their operation. You need not write the various error routines but you must include their calls.

   B. Specification of the costs s_{ij} and t_{ij} and how they were arrived at. Hint: How can these be obtained from the PDP-10 runs? You may assume:

      CPU run time = \$.08 per second

      Core usage cost = \$.01 per K or core per second of CPU time (\$.01 per Kilocore second)

   C. Several representative runs using meaningful p_1, p_2, x_1, x_2, x_3, x_4, c_s and c_t values.

   D. Whatever verification (formal proof, test data, etc.) you may be able to obtain in the time available that your programs are correct.

DEFINITION OF A REGISTER MODULE

Notation: In describing the effect of a procedure we shall describe the value of certain functions in terms of the values which existed before the execution of the function. We shall use the function call enclosed in quotes, e.g., 'sin(x)' to indicate the old or original value of the function, the unquoted version, e.g., sin(x) to indicate its value after the function being defined is executed once.

In the following definition we leave two parameters open and refer to them as p_1 and p_2. When integer values are supplied the definition is complete.

Function: LENGTH

possible values: an integer $0 \leq \text{LENGTH} \leq p_1$

effect: no changes or effect on values of any other functions

parameters: none

initial value: 0

Function: GETCHAR (I)

possible values: an integer $0 \leq \text{GETCHAR} \leq p_2$

parameters: I must be an integer

effect: no changes to other functions in module

if $I \leq 0 \vee I > \text{LENGTH}$ then a subroutine call to a routine GETERR is performed.

initial value: undefined since initial value of LENGTH = 0

Function: INSERT (I,J)

possible values: none (in BASIC terms this is a subroutine)

parameters: I must be an integer
J must be an integer

effect:

if I <0 ∨ I> LENGTH ∨ J <0 ∨ J> p_2 then a subroutine call to a routine INSERTER is performed.

else LENGTH = 'LENGTH' + 1. If 'LENGTH' ≥ p_1 a subroutine call to a function LENGER is performed.

GETCHAR (k) =

if k < I, 'GETCHAR(k)'

if k = I, J

if k > I, 'GETCHAR(k-1)'

Function: DELETE (I,J)

possible values: none

parameters: I must be an integer
J must be an integer

effect:

if I ≤ 0 ∨ J < 1 ∨ I+J-1 > LENGTH then a subroutine call to a routine DELERR is performed.

else

LENGTH = 'LENGTH' - J.

GETCHAR(k) = if k < 1 then 'GETCHAR(k)'

if k ≥ I then 'GETCHAR(k+J)'

Note: Definitions are due to Professor D. L. Parnas.

QUALIFYING EXAMINATION

May 11, 1971

24 Hour Take Home Exam

A programming language L is given a rating R(L) = j if j is the largest integer for which there is a fixed non-recursive expression in the language L whose value is the value of A(j,n) for all integral n > 0 where A(j,n) is the Ackerman function defined by:

$$A(0,n) = n+1$$
$$A(j,0) = A(j-1,1),\ j > 0$$
$$A(j,n) = A(j-1,A(j,n-1)),\ n > 0$$

In APL:

$$A(0,n) = n+1$$
$$A(1,n) = n+2$$
$$A(2,n) = 3 + 2\times n$$
$$A(3,n) = -3 + 2*3+n$$
$$A(4,n) = -3 + */(3+n)\rho 2$$

Consequently R(APL) ≥ 4.

1. What is R(Algol 60)?

2. What is R(PURE LISP)?

3. What is R(SNOBOL4)?

Justify your answers to the above.

By adding to APL an execute operator ∈ which evaluates character strings the following has been found:

$$A(5,n) = -3 + \in((n+2)\rho'*/2\rho'),\ '2'$$

(Recall that in APL the scope of every unary operator, such as ∈, is the entire expression to its right.)

4. The following has been hypothesized: If any programming language L can transform strings of data into statements of the language and repeatedly execute them $R(L) = \infty$.

   Sketch a proof (or counterexample) of this proposition.

5. One can also provide a simple expression for A(j,n) by adding a new operator $\underline{\rho}$ and extending an old one (., the inner product) as follows:

$$2 \times (n+3) = +/(n+3)\rho 2$$
$$2 * (n+3) = \times/(n+3)\rho 2$$

Ha! $2w[j](n+3) = w[j-1]/(n+3)\rho 2$

where $w[1] = +$ $w[2] = \times$ $w[3] = *$

Then $2w[j](n+3) = 2w[j-1]\ w[j-1]/(n+2)\rho 2$

$$-3 + 2w[j](n+3) = -3 + 2w[j-1](-3 + 2w[j](n-1 + 3) +3)$$

or $A(j,n) = A(j-1, A(j,n-1))$

$$2w[j](n+3) = w[j-1]/(n+3)\rho 2$$
$$= w[j-1]/2\ \underline{\rho}(n+3)$$

(definition of $\underline{\rho}$: $a\rho b = b\underline{\rho}a$)

$$= 2w[j-1].\ \underline{\rho}(n+3)$$

Formally: $w[j] = w[j-1].\underline{\rho}$

(inner product of w[j-1] and $\underline{\rho}$)

$$w[j] = w[1].\underbrace{\underline{\rho}.\underline{\rho} \ \ldots \ .\underline{\rho}}_{j-1 \text{ times}}$$
$$= +.\underline{\rho}.\underline{\rho} \ \ldots \ .\underline{\rho}$$

It is now possible to give a closed non-recursive expression for A(j,n) for every j and n.

$$A(j,n) = -3 + \in'2+', ((j-1)\rho'.\underline{\rho}'), 'n+3'$$

Using the development above as a model, write a LISP function for A(j,n), which does not call on itself. Evaluate it for some small arguments.

6. This suggests that by adding $\in$ and a few fixed operators to a language L we can represent many recursive functions by non-recursive expressions. Consider the language L whose data structure is strings of characters over a finite alphabet $\mathcal{A} \cup \phi$ (ϕ is the null character) on which are defined functions:

(1) head(x) (first character in x)
(2) rest(x) (all but the first character in x)
(3) x·y (the string xy)

and strings obtained by

(4) subst(x,v,y) (the string obtained by substituting the string x for the character v in the string y)
(5) and functions F defined by the primitive recursive schema:

$$F(u,\phi) = \phi$$
$$F(u,v\cdot x) = B(x,v,F(u,x))$$

where u is a parameter string
v is a character in $\mathcal{A}$
and B is a given primitive recursive function.

Then by adding a few additional functions show how one may represent F(u,v·x) by

$$\underbrace{\tilde{B}(\ldots(\tilde{B}\phi\ldots)}_{\text{m times}}$$

i.e., $F(u,v\cdot x) = \in(m\rho'\tilde{B}('),'\phi',m\rho')'$ where m is a primitive recursive function of the length of x, itself a function of the above type. For each B there is a $\tilde{B}$, which may be defined in terms of the basic primitive recursive functions plus these added functions.

Hint:

One possible way is to add a few new operators and encode strings as integers represented in base k, if there are k characters in the alphabet $\mathcal{A}$, and to consider ordinary primitive functions over the integers, and decode back to strings. Thus use the known results: Over the integers, all primitive recursive functions of the form

$$F(u,0) = u$$
$$F(u,Sx) = B(x,F(u,x))$$

(S the successor function)

can be obtained from the scheme:

$$G(0) = 0$$
$$G(Sx) = \underbrace{B(B(\ldots(B(0)))\ldots)}_{x \text{ times}}$$

by the following constructions over the non-negative integers (the so-called Gödel pairing function):

(a) Let $J(u,v) = ((u+v)^2 + u)^2 + v$

(b) $Ex = x - [\sqrt{x}]^2$

(c) $Kx = E[\sqrt{x}]$, $Lx = Ex$

then (d) $J(0,0) = 0$, $K(0) = 0$, $L(0) = 0$

and if $L(S(x)) \neq 0$ then $K(S(x)) = K(x)$ and $L(S(x)) = S(L(x))$

also (e) $K(J(u,v)) = u$ $L(J(u,v)) = v$ and $J(K(x), L(x)) = x$

(f) Then $F(u,0) = u$, $F(u,S(x)) = B(x,F(u,x))$ has a

$\widetilde{F} \ni \widetilde{F}(0) = 0$, $\widetilde{F}(S(x)) = \widetilde{B}^x(0)$

and $F(u,x) = L(\widetilde{F}(J(u,x))$

and $\widetilde{B}(x) = J(S(K(x)), 0^{L(S(K(x)))} \cdot K(S(K(x)))$

$+ \operatorname{sgn} L(S(K(x))) \cdot B(L(K(x)),L(x)))$

where

$u^x = 1$ if $x = 0$, all u

$= 0$ if $u = 0$ and $x \neq 0$

and $\operatorname{sgn} x = 0^{0^x}$

and $[x]$ = the greatest integer $\leq x$

7. What do you conjecture about classes of recursive functions in LISP for which such iterative expressions would exist?

BUSY BEAVER PROBLEM

Background

This writeup of the Busy Beaver Problem is taken from Korfhage[1].

Turing machines are constructed to perform specific tasks such as addition or multiplication. Part of the construction is the tacit assumption of a standard format for the input string. Thus one is naturally led to question the performance of the machine on a non-standard input string. This is the halting problem: given a Turing machine and an arbitrary tape, to determine whether or not the machine would eventually halt using the given tape as input. This and the related Busy Beaver problem have been shown to be unsolvable by any Turing machine (or algorithm). That is, it is not possible to design an algorithm which will solve this problem. The essential word here is "eventually." It is easy to determine whether or not a given machine using a given tape will halt within 1,479,641 or any other given number of steps: just try to run the machine for 1,479,642 steps. But with "eventually," we have no limit on the possible number of steps which may occur.

There are only a finite number of Turing machines of a given size (that is, number of states and symbols). For example, if we allow n states (not counting the halt state), two moves, and two symbols (0 and 1), then each block in the table describing a machine may be filled in $4(n + 1)$ ways (the extra one is for the halt state). Since there are $2n$ blocks in the table, if we require

that each block be filled there are exactly $N = (4(n + 1))^{2n}$ n-state two-symbol Turing machines. The Busy Beaver problem (of class (n,2)) is to determine which of these machines will, when started with a blank tape, halt with the highest possible number of 1's on its tape. This is thus a specialized halting problem, which has been shown by Rado [2] to be unsolvable. Nevertheless, some work has been done on this with interesting results [3]. It is known that for two-symbol machines, the highest possible number of 1's obtainable with a halting machine of 1 state is 1, 2 states--4, and 3 states--6. Table 1 shows one of the 3-state machines which will halt with six 1's. Four other such machines exist.

Table 1

A machine solving the three-state Busy Beaver problem

	0	1
q_0	$1Rq_1$	$1Rq_0$
q_1	$1Lq_2$	$1Rq_3$
q_2	$1Rq_0$	$1Lq_1$

For Turing machines having more than three states or operating on more than two symbols, the maximum possible score is not known. Nor has anyone solved the related problem of determining the maximum number of moves or shifts which is possible in a machine which halts. The known results are given in Table 2, where $\Sigma(n)$ denotes the maximum possible score, and SH(n) denotes the maximum possible number of shifts.

To indicate the magnitudes which must be considered in this problem, let us look at the 100-state machines. There are $163,216^{100}$ of these, some of which will halt when started with a blank tape, and some of which will not, It is known that one of these will halt with $(((7!)!)!)!$ or approximately $10^{10^{10^{15000}}}$ 1's on the tape. Thus the maximum number of ones attainable is at least that large, and probably considerably larger. Yet if we use ten billion years as an estimate of the age of the universe and assume that one billion 1's can be printed per second (somewhat faster than current digital computers), only approximately 3.15×10^{26} of these 1's could have been printed since the universe began.

Table 2

The known results in the Busy Beaver problem*

n =	1	2	3	4	5	6	7	8
Two-symbol machines								
$\Sigma(n)$	= 1	= 4	= 6	$\geq$ 13	$\geq$ 17	$\geq$ 35	$\geq$ 22,961	$\geq 3(7.3^{92} - 1)/2$
SH(n)			= 21	$\geq$ 107				
Three-symbol machines								
$\Sigma(n)$			$\geq$ 12					
SH(n)			$\geq$ 57					

* These results were communicated to the author in February 1966 by C.Y. Lee of Bell Telephone Laboratories, and are due to Lee, Tibor, Shen Lin, Patrick Fischer, Milton Green, and David Jefferson.

Problem

The problem is to find $\Sigma(n)$ and SH(n) for as many two-symbol machines as you can. Use the Turing machine simulator you built for an earlier

problem, or borrow one from a friend, or get the simulator written by the author of the earlier problem.

References

[1] Korfhage, Robert R., Logic and Algorithms, Wiley, 1966.

[2] Rado, Tibor, "On Non-Computable Functions," Bell System Technical Journal, 41 (1962), pp. 877-884.

[3] Lin, Shen and Rado, Tibor, "Computer Studies of Turing Machine Problems," JACM, 12 (1965), pp. 196-212.

ANALYSIS OF ALGORITHMS

Background

This description of the mathematical analysis of algorithms is taken from Knuth[1].

The general field of algorithmic analysis is an interesting and potentially important area of mathematics and computer science that is undergoing rapid development. The central goal in such studies is to make quantitative assessments of the "goodness" of various algorithms. Two general kinds of problems are usually treated:

Type A. Analysis of a particular algorithm. We investigate important characteristics of some algorithm, usually a frequency analysis (how many times each part of the algorithm is likely to be executed), or a storage analysis (how much memory it is likely to need). For example, it is possible to predict the execution time of various algorithms for sorting numbers into order.

Type B. Analysis of a class of algorithms. We investigate the entire family of algorithms for solving a particular problem, and attempt to identify one that is "best possible". Or we place bounds on the computational complexity of the algorithms in the class. For example, it is possible to estimate the minimum number $S(n)$ of comparisons necessary to sort n numbers by repeated comparison.

Type A analyses have been used since the earliest days of computer programming; each program in Goldstine and von Neumann's classic memoir[2] on "Planning and Coding Problems for an Electronic Computing Instrument" is accompanied by a careful estimate of the "durations" of each step and of the total program duration. Such analyses make it possible to compare different algorithms for the same problem.

Type B analyses were not undertaken until somewhat later, although certain of the problems had been studied for many years as parts of "recreational mathematics". Hugo Steinhaus analyzed the sorting function S(n), in connection with a weighing problem[3]; and the question of computing x^n with fewest multiplications was first raised by Arnold Scholz in 1937[4]. Perhaps the first true study of computational complexity was the 1956 thesis of H. B. Demuth[5], who defined three simple classes of automata and studied how rapidly such automata are able to sort n numbers, using any conceivable algorithm.

It may seem that Type B analyses are far superior to Type A, since they handle infinitely many algorithms at once; instead of analyzing each algorithm that is invented, it is obviously better to prove once and for all that a particular algorithm is the "best possible". But this is only true to a limited extent, since Type B analyses are extremely technology-dependent; very slight changes in the definition of "best possible" can significantly affect which algorithm is best. For example, x^{31} cannot be calculated in fewer than 9 multiplications, but it can be done with only 6 arithmetic operations if division is allowed.

These are the most important points about algorithmic analysis:

1) Analysis of algorithms is an interesting activity which contributes to our fundamental understanding of computer science. In this case, mathematics is being applied to computer problems, instead of applying computers to mathematical problems.
2) Analysis of algorithms relies heavily on techniques of discrete mathematics, such as the manipulation of harmonic numbers, the solution of difference equations, and combinatorial enumeration

theory. Most of these topics are not presently being taught in colleges and universities, but they should form a part of many computer scientists' education.

3) Analysis of algorithms is beginning to take shape as a coherent discipline. Instead of using a different trick for each problem, there are some reasonably systematic techniques which are applied repeatedly. (Numerous examples of these unifying principles may be found by consulting the entries under "Analysis of algorithms" in the index to [6].) Furthermore, the analysis of one algorithm often applies to other algorithms.

4) Many fascinating problems in this area are still waiting to be solved.

Problem

Choose three or four algorithms for a single task (such as sorting or searching a table) and compare their efficiencies for various assumptions about the data. (Type A analysis.)

OR

Attempt a Type B analysis. The precise specification of the class of algorithms and the measure of efficiency are extremely important.

References

[1] Knuth, Donald E., Mathematical Analysis of Algorithms, Computer Science Department, Stanford University. STAN-CS-71-206.

[2] Goldstine, Herman H. and John von Neumann, "Planning and Coding Problems for an Electronic Computing Instrument," in John von Neumann's Collected Works, A. H. Taub, ed., 5 (Pergamon Press, 1963), 80-235.

[3] Steinhaus, Hugo, Mathematical Snapshots, (Oxford University Press, 1950), 38-39.

[4] Scholz, Arnold, "Aufgabe 253," Jahresbericht der deutschen Mathematiker-Vereinigung, class II, 47 (1937), 41-42.

[5] Demuth, Howard B., Electronic Data Sorting (Ph.D. thesis, Stanford University, 1956), 92 pp.

[6] Knuth, Donald E., The Art of Computer Programming (Addison-Wesley Publishing Corporation: Volume 1, 1968; volume 2, 1969; volume 3, 1972).

SIMULATION OF A SMALL COMPUTER

Motivation

It is important for every Computer Scientist to understand the issues associated with machine language programming. You should write a few programs in assembly language at some point, but by the end of the IC you should at least understand what a machine language is and how instructions are interpreted by the hardware. This knowledge will be presumed by core courses in hardware, programming languages, and operating systems.

This problem requires you to write a simulator for a small computer. This is not an artificial task; simulators are often written for mini-computers in order to construct software before the machine is actually available and to debug software using the facilities available only in the larger machine.

The Problem

1. Obtain a description of the DEC PDP-8 from the instructor for this problem.

2. Write a program which simulates the behavior of the PDP-8. If you need to make simplifying assumptions, be sure to justify them carefully.

3. Include facilities for obtaining simulated timings--the amount of time a program would take to execute if it were really being run on a PDP-8. See if you can make the simulator efficient enough to attain a 50:1 simulation ratio.

4. Write three or four small programs (and debug them) to test the simulator.