IC STUDY PROBLEMS

Mary Shaw (ed.)
August 1973

First Edition August 1971
Revised and Expanded August 1972
Revised and Expanded August 1973 (J. W. McCredie)

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

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. A second contest was held in the spring of 1972. Five prizes of $100.00 were awarded in the second contest.

The specifications for both contests 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. A problem should be elegant and have an elegant solution.

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

4. A problem should involve a non-trivial programming effort, which should be an integral part of obtaining the solution.

5. A 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, a 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.

Use of the Problems in the IC

Each problem will be presented by its author or another interested member of the department at a morning lecture. We have scheduled the problems so that either a language appropriate for its solution will already have been introduced in the IC or almost any language you know will be adequate. There will be a chart in the teletype room on which each student will indicate which problems he is working on. The chart will provide for communication among the students working on a problem; they may meet among themselves or with the person who presented the problem to discuss difficulties, solutions, extensions, or other topics.

Completed answers should be submitted to the instructor for that problem. Solutions should be finished within two weeks of the initial presentation to keep them from dragging on forever and creating a massive workload at the end of the IC. The instructor will comment to the student on his solution and select the best of the solutions for possible publication.

A Note to the Students

We expect that you will be able to obtain complete solutions to five of the problems discussed during the IC and to do enough work on the others to understand the issues evoked by the problems. 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 elegant solution for a new problem than you could do 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. If you already know two of the three languages APL, ALGOL, and LISP, learn SNOBOL or BLISS and work on one of Markov algorithms,"One Man's Program ---," or "BLISSful Cooperation."

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 mini-compiler.

4. If you have never experimented with finite-state machines or Markov algorithms, try to do either the firing squad problem or the Markov algorithm problem, or both.

5. If you are already an expert programmer in a variety of languages, work on the analysis of algorithm.

6. Some of the Learning Laboratories may refer you to specific problems. If you are participating in such a lab, solve those problems.

7. If you are in doubt about which problems are the best ones for you to solve, talk to your advisor or to Mary Shaw.

The Awards

The problems awarded prizes in the 1970 contest were:

Birol Aygun	The Mutation Problem
Hans Berliner	The Power of Heuristics
Susan Gerhart	Hamming Codes
Anita Jones	One Man's Program is Another Man's Data
Rudy Krutar	Polynomial Manipulation with Fast Multiplication
Åmund Lunde	Lexical Analysis with Coroutines
Leroy Richardson and Richard Young	Area of a Region
George Robertson	A Problem in Simple Languages
Larry Snyder	Turing Machine Simulation
Tim Teitelbaum	Firing Squad Synchronization Problem
Tim Teitelbaum	Trees, Trees, Trees

The problems awarded prizes in the 1972 contest were:

Hans Berliner and Bruce Lowerre	Brute Force Has its Limitations
Roger Fajman	A Mini-compiler
Jim Gillogly	Maximizing the Payoff of α-β
Anita Jones	Markov Algorithms
Åmund Lunde	BLISSful Cooperation - or Speed vs. Security

Mary Shaw
August, 1972

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. Consider M as being built by adding to a skeleton of N found in M, where a skeleton is a sequence of subsequences of N with their ordering preserved. What can you say about the size of such a skeleton? How is it related to the maximal match between N and M?

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?

BRUTE FORCE HAS ITS LIMITATIONS

Hans Berliner and Bruce Lowerre

We are used to using computing power and taking for granted that whatever tasks we give the computer will be accomplished in a reasonable amount of time. For this reason we seldom give much thought to the efficiency of the programs we write and even more importantly to the efficiency of the algorithms we use to produce our solutions. This is because almost all of the problems we encounter are basically small problems. Thus we use interpretive languages which run one or two orders of magnitude slower than a compiled language would, but because we usually deal with small problems it is hardly noticeable. Likewise certain efficiencies which can be applied to repetitive operations such as sorting and matrix manipulations are seldom appreciated until one encounters a large problem.

However, there are problems in which the effect of computing efficiency can be observed in very drastic fashion. These are problems in which the computing time varies as a second order or higher function of the problem size. Sorting, certain matrix operations, and heuristic searches are in this class. In the latter type of problem, the effort involved in the complete enumeration (generate and test) approach will usually be a power function of the number of steps required in the solution.

The purpose of this problem is to allow the student to get a hands-on appreciation of the effects on computing efficiency that can be brought about by the efficiency of the algorithm that is used to do the computation. As a by product it teaches how to set up and conduct a tree search. The problem is:

Create a "knight's tour" of the chess board

A knight's tour is defined to be a closed chain of knight moves (a knight's move is two squares along one rectangular coordinate and one square along the other) which touches every square on the chess board once and only once, and after 64 moves returns to a square which is a knight's move away from the starting square. For any square on the board, there are from 2 to 8 legal moves initially with an average of 5.25. This makes the size of the solution space 5.25^{64} or approximately 10^{46}. This estimate is high because as the tour progresses, the number of choices from each square diminishes. A better estimate would be to assume 4.25 moves per square since one cannot go back to the square from which one just came. This gives an estimate of 4.25^{64} or 10^{40}. This estimate is still high. A conservative estimate would be about 10^{30}. The number of solutions contained in this space is extremely difficult to estimate. The final version of the authors' program produced about 600 solutions in about 2 minutes of CPU time. The first 30 moves of all these solutions were identical! This suggests an extremely dense solution space. If we assume that there are 600 solutions for any set of first 30 moves which do not violate the constraints in the authors' program, then we can estimate the total number of solutions to be $600*4.25^{30}$ or approximately 10^{20}. This number is undoubtedly still high, but indicates that there is no more than one solution for every 10^{20} points in the solution space. Therefore a program which generates 10,000 solution attempts per second, running for 1000 seconds (17 minutes), would only have 1 chance in 1000 of generating a correct solution.

In order to investigate this problem we will need a move generator which takes as input the name (coordinate) of a square, and generates as output the legal squares to which a knight can move from there. Note that it is not legal to move to a square which has been used earlier in the solution, so that some provision will have to be made to keep track of the used squares and have the legal move generator check this array before pronouncing a move legal. We also must have a way of remembering the sequencing of our current solution attempt since if we run into a dead end, we will need to have a way of backing up to try a different move at some previous decision point.

This can be done by creating a tree structure in your program, possibly in the form of a stack, which has the facility of remembering the current try at any point in the chain, and the other alternatives at that point which have not yet been tried. Then if the program reaches depth 64 in the chain successfully, it will have a correct solution. If it reaches an impasse (no further legal move possible) before this, then it must backtrack to the previous level and try another legal move.

The above is the basic structure of a brute force program to solve the "knight's tour" problem. Your first step should be to design and program such a solution. You may work with one other person on this project. Instrument your program so that it can output the current state of the solution at any time. This will allow you to observe it in operation and get some ideas about the adequacy of the algorithms that you will be trying. One good idea is to have the program print out the state of the solution every time you extend the solution chain to a greater length than it has ever been before.

When the program is running try it on the "knight's tour" problem using the generate and test algorithm that you now have implemented. Do not allow your program to run more than 5 minutes of CPU time. It is extremely unlikely that you will have found a solution in this time (nor in 10 or 100 times this amount of time). Look at your printout of the current solution that the program is working on and see if you can get any ideas for some simple rules that will keep the program from wasting its time without keeping it away from any of the solutions.

A MINI-COMPILER

Roger Fajman

The problem is to write a program to translate simple arithmetic expressions into machine code for a single-address computer.

The Algorithm:

The Algorithm is taken from a paper by Wirth and Weber (1). This algorithm is designed to recognize a particular class of context-free languages very quickly and simply. It has actually been used in practical compilers (2). First, we must define a simple-precedence grammar.

A grammar is a quadruple of the form G=(T,N,P,S). T is the set of terminal symbols of the language generated by G (i.e., the symbols which make up the sentences of the language). N is the set of non-terminal symbols of G. P is a set of productions which tell how to generate the sentences of the language. For a context free language, the productions have the form $A \rightarrow u$, where A is a non-terminal symbol and u is a non-empty string of terminals and non-terminals. S is the start symbol, from which all sentences of the language are generated. The canonical parse of a sentence is that parse in which the leftmost possible reduction is made first. A grammar is unambiguous if and only if there is only one canonical parse for each sentence generated by the grammar.

A parsing algorithm for a language generated by a grammar is a procedure for finding the canonical aprse, given a sentence of the language as input. In accordance with the definition, a parsing algorithm must first detect the leftmost substring of the sentence to which a reduction is

applicable. Then the reduction is performed and the same principle applied to the new sentence. In order to detect the leftmost reducible substring, Wirth and Weber's algorithm makes use of previously established non-commutative relations between the symbols of the grammar:

(a). The relation = holds between all adjacent symbols within a string which is directly reducible.

(b). The relation < holds between the symbol immediately preceding a reducible string and the leftmost symbol of that string.

(c). The relation > holds between the rightmost symbol of a reducible string and the symbol immediately following that string.

A simple precedence grammar is defined by Wirth and Weber as a context-free grammar in which at most one of the above relations holds between each pair of symbols of the grammar. Most context-free grammars are not simple precedence grammars, but it is usually possible to take a grammar for a programming language and turn it into a simple precedence grammar by appropriate manipulations. Wirth and Weber give an algorithm determining whether a grammar is a simple precedence grammar and for finding the precedence relations from the grammar.

The process for detecting the leftmost reducible substring consists of scanning the sentence from left to right until the first symbol pair is found so that S(I)>S(I+1), then to retreat back to the last symbol pair for which S(J-1)<S(J) holds. S(J)...S(I) is then the sought substring; it is replaced by the symbol resulting from the reduction. The process is then repeated. It is not necessary to restart scanning at the beginning of the sentence. Since all symbols S(K) for K less than J have

not been altered, the search for the next $>$ can start at the place of the previous reduction.

In the following description of the algorithm in pseudo ALGOL the original sentence is denoted by P(1)...P(N). K is the index of the last symbol scanned. For practical reasons, all scanned symbols are copied and renamed S(1)...S(I). The reducible substring therefore will always be S(J)...S(I) for some J. Internal to the algorithm, there exists a symbol $\perp$ (end of file) initializing and terminating the process. To any symbol of the grammar it has the relations $\perp > S$ and $S > \perp$. We assume $P(0)=P(N+1)=\perp$.

```
S(0):=P(0); I:=0; K:=1;
while P(K) notequal "⊥" do begin
     I:=J:=I+1; S(I):=P(K); K:=K+1;
     while S(I)>P(K) do begin
          while S(J-1)=S(J) do J:=J-1
          S(J):=LEFTPART(S(J)...S(I));
          I:=J;
          end;
     end;
```

The function denoted by LEFTPART(S(J)...S(I)) requires that the reducible substring be identified in order to obtain the symbol resulting from the reduction. If the parsed sentence is to be evaluated, then the interpretation rule corresponding to the syntactic rule $U \rightarrow S(J)...S(I)$ must be identified and executed. Wirth and Weber prove the following theorem:

> Theorem. The given parsing algorithm yields the canonical form of the parse for any sentence of a precedence language, if there exist no two syntactic rules with the same right part. Furthermore, this canonical parse is unique.

EXAMPLE:

G=(T,N,P,S)
T={(,),@}
N={H,S}
P: S→H)
H→(
H→H@
H→HS

The language defined by G is a sequence of zero or more string elements enclosed in parentheses, where an element is another string or @. G is a precedence grammar. The precedence relations are given by the matrix:

	S	H	@	(	)
S	>	>	>	>	>
H	=	<	=	<	=
@	>	>	>	>	>
(	>	>	>	>	>
)	>	>	>	>	>

As an illustration of the parsing algorithm, the sentence (@(@)) is parsed.

Stack	Relation	Input
⊥	<	(@(@))⊥
⊥(	>	@(@))⊥
⊥H	=	@(@))⊥
⊥H@	>	(@))⊥
⊥H	<	(@))⊥
⊥H(	>	@))⊥
⊥HH	=	@))⊥
⊥HH@	>	))⊥
⊥HH	=	))⊥
⊥HH)	>	)⊥
⊥HS	>	)⊥
⊥H	=	)⊥
⊥H)	>	⊥
⊥S		⊥

The Problem:

Consider the following grammar for arithmetic expressions:

G=(T,N,P,program)

T={variable,constant,+,-,*,/,(,),←}

N={program,expression,expression',sum,sum',term,term',factor}

P: program→⌊expression⌋
expression→expression'
expression'→sum
expression'→variable←expression'
sum→sum'
sum'→term
sum'→+term
sum'→-term
sum'→sum'+term
sum'→sum'-term
term→term'
term'→factor
term'→term'*factor
term'→term'/factor
factor→variable
factor→constant
factor→(expression)

G is a simple precedence grammar. The precedence relations are given by

	⊥	program	expression	expression'	sum	sum'	term	term'	factor	variable	constant	←	+	-	*	/	(	)
⊥			<	<	<	<	<	<	<	<	<		<	<			<	
program																		
expression	>																	=
expression'	>																	>
sum	>																	>
sum'	>												=	=				>
term	>												>	>				>
term'	>												>	>	=	=		>
factor	>												>	>	>	>		>
variable	>											=	>	>	>	>		>
constant	>												>	>	>	>		>
←				=	<	<	<	<	<	<	<		<	<			<	
+							=	<	<	<	<						<	
-							=	<	<	<	<						<	
*									=	<	<						<	
/									=	<	<						<	
(			=	<	<	<	<	<	<	<	<		<	<			<	
)	>												>	>	>	>		>

Note that the blank entries in the table denote combinations that should never occur in legal sentences.

Now suppose that you have a computer with a single accumulator (AC) and the following instruction set:

Instruction		Meaning
LOAD	X	AC←X
STORE	X	X←AC
ADD	X	AC←AC+X
SUB	X	AC←AC-X
MUL	X	AC←AC*X
DIV	X	AC←AC/X
LOADI	C	AC←C
ADDI	C	AC←AC+C
SUBI	C	AC←AC-C
MULI	C	AC←AC*C
DIVI	C	AC←AC/C
NEG		AC←-AC

X is a storage location. C is a constant which is contained in the instruction.

The problem, then, is to read in sentences in this language and output the appropriate machine code to compute the value of the expression. Variables are any one of the letters A,...,Z. Constants are a single digit. You may use temporary locations from the stack T1,T2,.... Do not worry about generating "optimal" code.

Optional Work:

1. You may have noticed that the parsing algorithm does not provide for input strings which are not syntactically correct. Modify the algorithm to handle errors.

2. Try to generate better code.

3. Extend the instruction set of the computer so as to permit the generation of faster and more compact code.

4. What would happen if the computer had more than one accumulator?

References:

1. Wirth, N. and Weber, H. EULER: A generalization of ALGOL and its formal definition. CACM, 9, 1 (January 1966), 13-23.

2. McKeeman, W., et al. A Compiler Generator.

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 a transmitted n + k - bit sequence into an n - bit message sequence and a k - bit sequence indicating absence or position of error.

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 $k_3\ k_2\ k_1$ where

$$k_3 = y_4 \oplus y_5 \oplus y_6 \oplus y_7$$

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

$$k_1 = y_1 \oplus y_3 \oplus y_5 \oplus y_7$$

If one and only one digit is transmitted incorrectly, say $y_j \neq x_j$, then $k_3\ k_2\ k_1$ will give the binary representation of j or 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 assigmment 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:

↓	1 2 3 4 5 6 7	bit positions
0 0 1 1		
0 1 1 3	p x x x	
1 0 1 5		
1 1 1 7		
↓		p = check position
0 1 0 2	p x x x	x = included in equations
0 1 1 3		
1 1 0 6		
1 1 1 7		
↓		
1 0 0 4	p x x x	
1 0 1 5		
1 1 0 6		
1 1 1 7		

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.

Let M be the NTK-bit sequence to be transmitted. Let U be a matrix where column i is the binary representation of i to K places. The sequence is constructed to satisfy the matrix equation

$$UM = \bar{0}$$

where $\bar{0}$ is a vector of length K of all zeros.

Now, the received message may be represented as M+E, where E is an error vector with 1's in every position where an error occurs. The

$$U(M+E) = (UM)+(UE) = \bar{0}+UE = D$$

If E is all zeros (no errors in transmission) then $D=\bar{0}$ and if E has a 1 in position i then E selects the binary representation of i from U.

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.

MAXIMIZING THE PAYOFF OF α-β

Jim Gillogly

Problem Statement

(1) Understand the α-β procedure used in game trees as described in "Experiments with Some Programs That Search Game Trees," by Slagle and Dixon (available in the AI library).

(2) Write a program in any suitable language to simulate searching a game tree with fixed depth, fixed branching factor, and randomly assigned values as a terminal evaluation function. (If you can get an analytic solution, you win. But don't spend all your time trying.)

(a) The random evaluation function must produce values from a finite range specified by the experimenter; the seed for the random number generator must also be supplied by the experimenter.

(b) The effort required to search the tree will be measured by the number of bottom positions (NBP) in the simulated tree. You should be able to cycle through the program for a specified number of iterations and output the mean and standard deviation of the NBP over that sample.

(c) In order to debug your program you should be able to turn α-β on or off and you should be able to print the tree if necessary. To help in debugging, start with a range of 1 (perfect ordering) and compare your results with those predicted by Theorem 1 (Slagle and Dixon).

(3) Use this model to determine the effects on NBP of selecting values from sets of different size: choose a fixed depth and branching factor commensurate with the efficiency of your program and graph the values of the mean and standard deviation of NBP as the range increases from 1 (perfect ordering) to ∞ (the largest range for which you can detect no significant difference). What is the difference between the NBP in the best and worst cases? Try for statistical significance. (Notice that even the worst case is far better than a minimax search without α-β.) If you have time, compare the results you obtained in this experiment with the curves for different depths and branching factors.

(4) Extend the program to allow a variable branching factor drawn from a given distribution (e.g., binomial distribution with mean at the value used in (3)). What effect does this have on the mean and variance for a case comparable to the one you analyzed in (3)?

Conceptual Foundations

The purpose of this problem is to teach modeling and simulation techniques in a (hopefully) interesting environment. The student must abstract a situation and analyze the abstraction. The problem requires considerable programming ability in some language. Because of the recursive nature of the problem it is somewhat easier to write in a language which allows recursion, but a reasonably straightforward program can also be written in FORTRAN. Some slight knowledge of statistics would be helpful, though not necessary.

You should be able to understand and apply the results of a scientific paper. Since the Slagle and Dixon paper assumes no prior knowledge this should not be a problem.

An attack on the problem should begin with an understanding of the nature of game trees, minimax, and α-β. You should pay particular attention to the difference between "deep" and "shallow" cutoffs as indicated in Slagle and Dixon.

The α-β procedure is only one example of an analytical advance which cuts down search spaces. Nearly all AI problems use a horrendous search through some kind of a graph, and those searches can be abstracted and simulated in the same way as this problem. The direct analogue of α-β in general graph traversing problems is the branch-and-bound procedure.

Besides the benefits accrued from doing a good simulation, the problem itself has vast implications for game programmers. The moral of the problem is that your terminal evaluation function should be as discrete as possible within the constraints of information loss. If you use a continuum you may have to evaluate more than 400% more positions, with a much higher variance.

MARKOV ALGORITHMS

Anita Jones

Several different representations of machines for performing computation have proved useful for research into the art (and artifice) of computation. One representation, Markov algorithms, were introduced in Theory of Algorithms written by the Russian A. A. Markov in 1954.[1] A Markov algorithm is an ordered set of productions (discussed later) to be applied to a valid input string by observing the ordering of the productions in a plausible way.

Markov algorithms can compute any computable function (intuitive proof: given that a Turing machine can compute any computable function, a Turing machine description may be represented as a Markov algorithm in a straightforward fashion.) In fact Markov algorithms are simpler to grasp and invent than Turing machines because of their use of variables and context sensitivity.

This problem requires the construction of a Markov algorithm interpreter which accepts an algorithm description and an input argument, applies the algorithm to the argument and outputs the transformed input string. For ease of use, the program should then query the user to determine if he wishes to specify another input for the same machine, another machine description or to quit. The interpreter should then be used on several algorithms, e.g., a string duplicator, gcd computer, string reverser ....

> WARNING: Social responsibility requires that the reader be informed that after being introduced to Markov algorithms, some computer science students exhibit an obsessive, almost addictive interest in devising these 'delightful little algorithms.' Addiction has been known to last for weeks.

For those willing to risk the consequences:

A Markov algorithm is an ordered set of rules called productions designed to perform a transformation on any input string formed from symbols in a known input alphabet.

A production is of the form $\alpha \rightarrow \beta$ where α, β are strings of symbols. Markov algorithms use the antecedent portion, α, to specify a pattern. If the pattern is found in the (input) string being processed, it is replaced by a string determined by the consequent portion of the production, β. α, β may include any symbol.

To apply a production $\alpha \rightarrow \beta$ to an input string S, two alphabets must be known: the input alphabet and the variable alphabet. These alphabets must be disjoint. α matches a substring in S if there is some assignment of input alphabet symbols to the variables in α so that if the variables are replaced by their assigned value (from the input alphabet), the resulting string duplicates a substring in S. In this case β is then used to construct the string resulting from replacing occurrences of variables in β by their assigned values. This resulting string is used to replace the matched substring in S.

What if an antecedent can be used to match two substrings in S? Then replace the leftmost substring in S.

Applying a Markov algorithm is done in steps: Each step applies the productions sequentially, beginning with the first of the ordered set. When a production is applied successfully and the string replacement is performed, that step is complete. If no production can be successfully applied in a step, then either the algorithm or the input string is in error.

How does the algorithm's application terminate? A special symbol '.' appearing as the first symbol in a consequent will not be part of the replacement string created when the consequent is used. Instead it causes termination of the algorithm's application.

Examples of Markov algorithms are given below.

Solving this problem introduces the student to

1. construction of a powerful but miniature interactive interpreter driven by productions
2. the consequences of designing an interpreter which must react to storage variations dependent upon the user input algorithm
3. practice in writing and exercising productions as they will appear in formal languages and language syntax descriptions
4. non sequential control structure.[2]

SNOBOL and APL provide suitable environments for building this interpreter.

---

[1] This problem is brought to you by

Markov, A. A., Theory of Algorithms, Office of Technical Services, U. S. Department of Commerce (translation), 1969.

and

Jones, A. K., this document.

[2] The author also believes that Markov algorithms are particularly appropriate for the analysis of algorithms: the order of trying productions in each step introduces additional information to aid in the analysis, for a later production is applied ONLY if all previous productions failed. This problem could be extended to use Markov algorithms as the substrate for introducing algorithm analysis.

```
TYPE ODDEVE.MRK
00200    Y10 ——— Internal Alphabet
00300    X
00400    10 ——— Input Alphabet
00500    Y111↑Y1;
00600    Y11↑.0;  } ——— Productions
00700    Y1↑.1;
00800    ↑Y;
```

```
.R SNOBOL 41
*MARK3
HI!  I'M MARK.
SPECIFY MACHINE DESCRIPTION FILE:
ODDEVE.MRK
STRING:
1111
#4: SUBSTITUTE Y FOR  IN 1111
#1: SUBSTITUTE Y1 FOR Y111 IN Y1111
#2: SUBSTITUTE 0 FOR Y11 IN Y11
STRING: 0
WHAT NEXT: MACH,STR,END
STR
STRING:
111111
#4: SUBSTITUTE Y FOR  IN 111111
#1: SUBSTITUTE Y1 FOR Y111 IN Y111111
#1: SUBSTITUTE Y1 FOR Y111 IN Y1111
#2: SUBSTITUTE 0 FOR Y11 IN Y11
STRING: 0
WHAT NEXT: MACH,STR,END
STR
STRING:
1
#4: SUBSTITUTE Y FOR  IN 1
#3: SUBSTITUTE 1 FOR Y1 IN Y1
STRING: 1
WHAT NEXT: MACH,STR,END
END
*
```

Given a unary representation of a number, compute if it is odd (and output '0') or even (and output '1').

EXAMPLE 2

Duplicate a numeric string.

```
TYPE DUP.MRK
00100
00200    GAB
00300    SN ————————— Variable Alphabet
00400    1234567890 —— Input Alphabet
00500    GS↑SASBG;
00600    ANBS↑SANB;
00700    ANB↑N;          ————————Productions
00800    G↑.;
00900    ↑G;
```

```
.R SNOBOL 41
*MARK3
HI!  I'M MARK.
SPECIFY MACHINE DESCRIPTION FILE:
DUP.MRK
STRING:
23
#5: SUBSTITUTE G FOR  IN 23
#1: SUBSTITUTE 2A2BG FOR G2 IN G23
#1: SUBSTITUTE 3A3BG FOR G3 IN 2A2BG3
#2: SUBSTITUTE 3A2B FOR A2B3 IN 2A2B3A3BG
#3: SUBSTITUTE 2 FOR A2B IN 23A2BA3BG
#3: SUBSTITUTE 3 FOR A3B IN 232A3BG
#4: SUBSTITUTE  FOR G IN 2323G
STRING: 2323
WHAT NEXT: MACH.STR.END
STR
STRING:
3336
#5: SUBSTITUTE G FOR  IN 3336
#1: SUBSTITUTE 3A3BG FOR G3 IN G3336
#1: SUBSTITUTE 3A3BG FOR G3 IN 3A3BG336
#1: SUBSTITUTE 3A3BG FOR G3 IN 3A3B3A3BG36
#1: SUBSTITUTE 6A6BG FOR G6 IN 3A3B3A3B3A3BG6
#2: SUBSTITUTE 3A3B FOR A3B3 IN 3A3B3A3B3A3BG6A6BG
#2: SUBSTITUTE 3A3B FOR A3B3 IN 33A3BA3B3A3BG6A6BG
#2: SUBSTITUTE 3A3B FOR A3B3 IN 33A3B3A3BA3BG6A6BG
#2: SUBSTITUTE 6A3B FOR A3B6 IN 333A3BA3BA3BG6A6BG
#2: SUBSTITUTE 6A3B FOR A3B6 IN 333A3BA3BG6A3BA6BG
#2: SUBSTITUTE 6A3B FOR A3B6 IN 333A3BG6A3BA3BA6BG
#3: SUBSTITUTE 3 FOR A3B IN 3336A3BA3BA3BA6BG
#3: SUBSTITUTE 3 FOR A3B IN 33363A3BA3BA6BG
#3: SUBSTITUTE 3 FOR A3B IN 333633A3BA6BG
#3: SUBSTITUTE 6 FOR A6B IN 3336333A6BG
#4: SUBSTITUTE  FOR G IN 333633336G
STRING: 33363336
WHAT NEXT: MACH.STR.END
END
*
```

Reverse a string.

```
 TYPE REV.MRK
00100    ABCD
00200    SV
00300    0123456789
00400    BS;SB;
00500    B;C;
00600    ASV;VAS;
00700    ASC;CBS;
00800    SD;DS;
00900    D;A;
01000    AS;.;
01100    ;AB;

.R SNOBOL 41
*MARK3
HI!  I'M MARK.
SPECIFY MACHINE DESCRIPTION FILE:
REV.MRK
STRING:
12
#0: SUBSTITUTE AB FOR  IN 12
#1: SUBSTITUTE 1B FOR B1 IN AB12
#1: SUBSTITUTE 2B FOR B2 IN A1B2
#2: SUBSTITUTE C FOR B IN A12B
#3: SUBSTITUTE 2A1 FOR A12 IN A12C
#4: SUBSTITUTE DC1 FOR A1C IN 2A1C
#5: SUBSTITUTE D2 FOR 2D IN 2DC1
#6: SUBSTITUTE A FOR D IN D2C1
#4: SUBSTITUTE DC2 FOR A2C IN A2C1
#6: SUBSTITUTE A FOR D IN DC21
#7: SUBSTITUTE  FOR AC IN AC21
STRING: 21
WHAT NEXT: MACH.STR.END
STR
STRING:
369
#0: SUBSTITUTE AB FOR  IN 369
#1: SUBSTITUTE 3B FOR B3 IN AB369
#1: SUBSTITUTE 6B FOR B6 IN A3B69
#1: SUBSTITUTE 9B FOR B9 IN A36B9
#2: SUBSTITUTE C FOR B IN A369B
#3: SUBSTITUTE 6A3 FOR A36 IN A369C
#3: SUBSTITUTE 9A3 FOR A39 IN 6A39C
#4: SUBSTITUTE DC3 FOR A3C IN 69A3C
#5: SUBSTITUTE D9 FOR 9D IN 69DC3
#5: SUBSTITUTE D6 FOR 6D IN 6D9C3
#6: SUBSTITUTE A FOR D IN D69C3
#3: SUBSTITUTE 9A6 FOR A69 IN A69C3
#4: SUBSTITUTE DC6 FOR A6C IN 9A6C3
#5: SUBSTITUTE D9 FOR 9D IN 9DC63
#6: SUBSTITUTE A FOR D IN D9C63
#4: SUBSTITUTE DC9 FOR A9C IN A9C63
#6: SUBSTITUTE A FOR D IN DC963
#7: SUBSTITUTE  FOR AC IN AC963
STRING: 963
WHAT NEXT: MACH.STR.END
END
*
```

One Man's Program is Another Man's Data

Anita Jones

Information used within a computer is represented in a symbolic form. Whether the representation is interpreted as program,data or both, format or content alteration of the representation is a symbol manipulation or string processing task. To provide an environment in which to consider the concepts and mechanisms germane to interpretation independent manipulation of symbols [1,3], a simple line editor is to be implemented. Text is written in some alphabet of symbols and the editor permits insertion, deletion and replacement of symbol strings located by context within the text. A few basic notions:

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 here !)

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

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 (denoted: 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 separated by line feeds.

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 be imagined to move 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 still subject to alteration by MERGE.

To process a single command, MERGE inputs the next command from MODIFY. The cursor moves through TEXT until it passes over LF [<num>] where line <num> is the first line possibly affected by the current command.

---

* The statement of the problem is phrased with a conversational system environment in mind. The reader may substitute his own line delimiter (e.g. 'new card') if appropriate.

MODIFY Command formats:

DELETE <num> <num 1>

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

MERGE performs the DELETE command by erasing all characters on the input side of the cursor up to and including LF[<num 1> +1], if <num 1> was specified or up to and indlucing LF[<num> +1], if <num 1> is null.

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

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

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

MERGE performs the INSERT command by placing the delimited insertion string (which may contain LF) followed by the carriage return and line feed characters into the TEXT string to the output side of the cursor.

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

MERGE uses the REPLACE command to edit a single TEXT line. LF may not appear within the body of the strings α_J or β_J.

1. Let the first non-blank, non-numeric, non-LF character after <num> be the delimiter. Set J = 1.

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

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

4. Scanning the line <num>, replace each occurrence of the current recognition string with the corresponding replacement string.

5. J←J + 1. Go to step 2.

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

One Solution

The algorithm MERGE may be profitably considered at an "abstracted operation" level before binding language and representation details. A first pass expressed in imperative English phrases might be:

```
MERGE:  locate TEXT, MODIFY and NEWTEXT files
        copy TEXT to NEWTEXT altering lines as directed by MODIFY.
```

The problem (REPLACE in particular) requires that MODIFY commands be applied to TEXT sequentially. Expanding the abstraction of MERGE to define a sequential use of the MODIFY commands we get:

```
                                                              Label in Example solution
MERGE:  locate TEXT, MODIFY, and NEWTEXT files                START
   repeat
      get next MODIFY command:  C <num>...                    MAIN
      move cursor to line <num>                               NUMCHECK
      in case C = INSERT do insert text from MODIFY           INSERT
              C = DELETE do erase prescribed TEXT lines       DELETE
              C = REPLACE do iteratively replace substrings   REPLACE
   until MODIFY file exhausted
   move cursor to end of TEXT file                            EXIT
```

So far only logically separate operations are specified. We have yet to bind the representation of input or intermediate data strings. These, as well as a pattern match strategy, may be affected by the expressive power of the language in which the final program is coded so it seems appropriate to select a language at this point.

The example solution is in SNOBOL [2], a string manipulation language which provides string variables and pattern recognition facilities. (Any language which permits easy representation and manipulation of variable length strings could be used.)

The label name listed with each abstracted operation in the last version of MERGE indicates the beginning of the implementation of that operation in the example solution.

To get from the abstracted version of the MERGE algorithm to the example solution requires binding:

1. data representation: All intermediate processing data is maintained in SNOBOL string variables. The line currently under the cursor is in T and the current edit command is in COMMAND. (Each command processing operation has local variables in which it may place portions of the content of the COMMAND variable for convenient use. Lines to the input and output sides of the cursor reside in the files TEXT and NEWTEXT respectively.

2. domain restrictions: Specify what action is to be taken if the values of inputs do not fall within the expected domains. In response to command errors descriptive messages are reported to the user's terminal, the editor ignores the remainder of the current command and attempts to continue.

3. program usability expansions: The MERGE algorithm is imbedded in a conversational loop allowing the terminal user to conveniently request repeated edits.

Only a single implementation of the editor is included here because the well formed structure of the problem encourages similarly structured solutions !

References

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

[2] Griswold,R.E., Poage,J.F. and Polonsky, I.P., The SNOBOL 4 Programming Language, Prentice-Hall, 1968.

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

Example Solution:

```
***          L I N E       E D I T O R
***
         &TRIM = 1 ;   &ANCHOR = 1
         IOFORMT = '(1X,20(A5))'
         INPUT('TTYIN','2',80)
         INPUT('TEXT','21',80)
         INPUT('MODIFY','22',80)
         OUTPUT('OUTPUT','23',IOFORMT)
         OUTPUT('TTYOUT','2',IOFORMT)
         DEFINE('NUMCHECK()')
         DEFINE('INTEXT()')
         DIGITS = '0123456789'
         REPL = BREAK(*DELIM) . R LEN(1) BREAK(*DELIM) . P LEN(1)
         BLANK = SPAN(' ') | NULL
         DEL = LEN(1) . DELIM
         IDR = 'INSERT' | 'REPLACE' | 'DELETE'
         MODCMD = POS(0) BLANK IDR . WORD BLANK SPAN(DIGITS) . NUM BLANK
START    TTYOUT = 'INPUT TEXT FILE: '
         IFILE('21',TTYIN)
         TTYOUT = 'MODIFY FILE: '
         IFILE('22',TTYIN)
         TTYOUT = 'OUTPUT TEXT FILE: '
         OFILE('23',TTYIN)
         TEXTLN = -1 ;   INTEXT() ;  OUTLN = -1
***
*** CYCLIC MAIN PROGRAM WHICH CALLS NUMCHECK, INSERT, DELETE AND
***      REPLACE OPERATIONS
***
MAIN     COMMAND = ' ' MODIFY                                  :F(EXIT)
         C = COMMAND
         COMMAND MODCMD =                                      :F(CMDFAIL)
         NUMCHECK()                                            :($(WORD))
EXIT     OUTPUT = T                                            :S(EX1)
EX1      OUTPUT = TEXT
CYCLE    ENDFILE('21') ; ENDFILE('22') ; ENDFILE('23')
         TTYOUT = 'EDIT COMPLETE. NEW FILES:  Y  OR  N'
         TTYIN 'Y'                                             :S(START)F(END)
***
*** PROCESS:      INSERT <NUM> <DELIM> STRING <DELIM>
***
INSERT   COMMAND DEL =                                         :F(NODELIM)
IDEL     COMMAND BREAK(DELIM) . I                              :F(ILP)
         OUTPUT = I                                            :(MAIN)
ILP      OUTPUT = COMMAND
         COMMAND = MODIFY                                      :S(IDEL)F(INCCMD)
***
*** PROCESS:      DELETE <NUM> N<NUM2>
***
DELETE   NUM2 = DIFFER(COMMAND) COMMAND                        :S(D2)
         NUM2 = NUM                                            :(DLP)
D2       INTEGER(NUM2)  GE(NUM2,NUM)                           :F(DELFAIL)
DLP      INTEXT()  LT(NUM2,TEXTLN)                             :F(DLP)S(MAIN)
```

```
***
*** PROCESS:      REPLACE <NUM> <DELIM>R1<DELIM>R2<DELIM>...
***
REPLACE  COMMAND DEL =                                          :F(NODELIM)
RLP      R = 0 ;   COMMAND DEL =                                :F(REP)
         DIFFER(R)                                              :F(RNULL)
REAPPLY  T (LEN(N) ARB) . X R =                                 :F(RCHK)
         T = X R T
         N = SIZE(X) + SIZE(R)                                  :(REAPPLY)
RCHK     DIFFER(COMMAND)                                        :S(RLP)F(MAIN)
***
*** SERVICE SUBROUTINES
***
NUMCHECK LT(NUM,TEXTLN)                                         :S(NUMDEL)
NUMLP    EQ(NUM,TEXTLN)                                         :S(RETURN)
         OUTLN = TEXTLN ;  OUTPUT = T INTEXT()                  :(NUMLP)
NUMDEL   WORD 'INSERT' GT(NUM,OUTLN)                            :S(RETURN)F(NUMFAIL)
INTEXT   T = TEXT                                               :F(NOTEXT)
         TEXTLN = TEXTLN + 1                                    :(RETURN)
***
*** ERROR MESSAGES
***
NODELIM  TTYOUT = C '    :: NO DELIMITER DETECTED'              :(CONT)
CMDFAIL  TTYOUT = C '    :: KEYWORD FAILURE'                    :(CONT)
NUMFAIL  TTYOUT = C '    :: LINE ALREADY EDITED'                :(CONT)
INSCMP   TTYOUT = C '    :: INSERT INCOMPLETE-EXITING'          :(EXIT)
REP      TTYOUT = C '    :: REPLACE INVALID'                    :(CONT)
DELFAIL  TTYOUT = C '    :: INVALID DELETION PARM'              :(CONT)
NOTEXT   TTYOUT = 'EXHAUSTED TEXT BEFORE COMMANDS'
         OUTPUT = T                                             :(CYCLE)
RNULL    TTYOUT = C '    :: REPLACE STRING NULL'                :(CONT)
CONT     TTYOUT = 'CONTINUING.'                                 :(MAIN)
END
```

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 OR Y >= 2100 THEN BEGIN
                                   M := 0;
                                   D := 0
                                 END
END CALENDAR
```

and these commands in the MODIFY file:

```
REPLACE 0 /L/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 OR (Y/100) * 100 NE Y THEN T ELSE 0;!
  REPLACE 6 $- (M*3066) / 100$* 100) / 30558$8$$
REPLACE 7 ZMZ(MZ
DELETE 9 12
```

result in the NEWTEXT file containing ACM Algorithm 398: Tableless Date Conversion by Richard A. Stone:

```
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 OR (Y/100) * 100 NE 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 two replacements correct errors resulting from the first two replacements.

'INSERT 2...', inserts the appropriate non-printing 'carriage return' and LF characters (1) between lines as included within the MODIFY file and (2) following the inserted text as required by the definition of INSERT.

There are two commands used on line 4.

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

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

Example: 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: (Note the multiple replacement of 's' by '&'.)

```
REPLACE 2 *ship*bit*wax*tracks*s*&*
  REPLACE 2  =&ea=whir=
REPLACE  3  404       04c484
```

results in the edited NEWTEXT:

```
The time has come the walrus said
  To speak of many things
  Of &ailing bit& and whirling track&
      Of Babbages and kings.
```

Using NEWTEXT as a file to be edited again and the following REPLACE commands

```
REPLACE 2     ' O'    O'&ailing'SLT'&'s'bit'chip'
REPLACE 3 !klr!
```

yields

```
The time has come the walrus said
  To speak of many things
    Of 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:

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

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:

$x = 1 \cdot 0$

$x^2 = 0 \cdot (1 \cdot 0)$

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

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

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

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

and this is represented as:

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

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

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

and this is represented as:

$(-4 \cdot (6 \cdot 2)) \cdot 0$

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.

LEXICAL ANALYSIS WITH COROUTINES

Åmund 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.

BLISSful COOPERATION -- OR SPEED VS. SECURITY
A PROBLEM INVOLVING COROUTINES AND GENERAL LANGUAGE ISSUES

Åmund Lunde

Problem Statement

The solution to this problem has 3 distinct parts, 2 experimental and one theoretical. The experimental parts consist of coding in BLISS two different algorithms for the same problem, using two different control-structures. The theoretical part is to compare the solutions and then discuss how the comparison might be different if you had used a different language. Obviously the experimental parts are independent of each other and of the theoretical part. However, if you can present a good answer to the theoretical part without doing the programming you will be wasting your time doing that, and may be when going to any of the programming sections of the IC. You should attempt all or parts of the problem depending on your previous experience.

If you do not know what coroutines are, you should now read the section 'The coroutine concept' in the description of the problem 'LEXICAL ANALYSIS WITH COROUTINES.'

Problem to be programmed:

To print all different subsets of M numbers from the set of the N first natural numbers, $1 \leq M \leq N$.

NOTE: By definition of sets each member occurs only once. Hence if N = 5, M = 3, the sets [1 2 3] and [1 2 4] are subsets to be printed, but not the sequences [1 1 2] or [1 5 5]. Also, of course, each subset should be printed exactly once.

The problem may be attacked in at least 2 ways, by recursive routines and by coroutines. In case you don't see a natural coroutine solution read the hint. Program these two solutions in BLISS and compare their execution speeds. If a coroutine facility were added to ALGOL, (like in SIMULA-67), and the two programs translated into that language, how do you think the relation between the execution times would change?

Hint: (Try before reading).

There will be a chain of M+1 (or maybe M+2) coroutines, the main-program and M coroutine instances of the same routine, one for each position of the selection. Each has pointers to its predecessor and successor in the chain. The main program will do the printing. Each time it needs a new selection it will activate the coroutine for the last position in the selection. This will increase its selected number by 1, check if this is legal (i.e., not too large), and then activate the main program or its predecessor depending on the result of the test. Figure out on your own which variables you need in each co-routine instance, and how they should be initialized. Maybe you will want an extra coroutine at the end of the chain to tell you when you are through.

Historical note:

This coroutine solution to this problem was written in 1971 in SIMULA-67 by Mr. Dag Belsnes at the University of Oslo, Norway. At that time it was (and maybe it still is) the winning entry for this problem in their continuously ongoing 'Code it neater and faster in SIMULA' contest.

The current formulation and the BLISS versions are due to the present author.

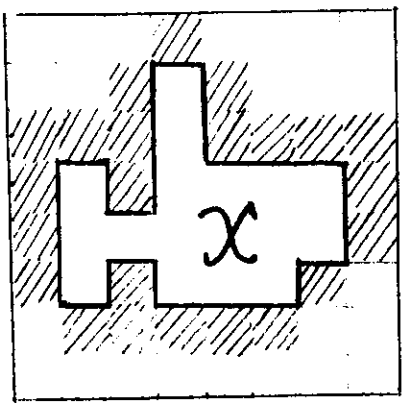
AREA OF A REGION

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. But there may be more than one white region in the two-dimensional space.

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 simple 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 "$a_1a_2\ldots a_k$", 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.

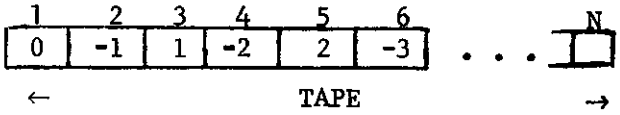
3.) Accepting a unary sequence if it has 2^i marks, for any non-negative i. This one is easy.

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.)

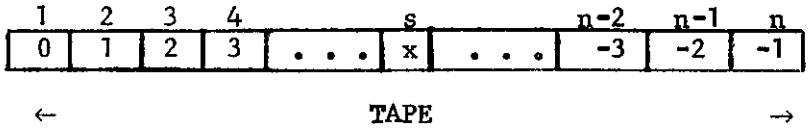


The tape vector is a vector of length n. TAPE[1] is the 0 cell, all odd numbered elements are positive cells, all even numbered elements are negative cells, such that:

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

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
   A      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 o'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,□,o,→,≠
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□o→≠
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,o,4,o,R
|
4,0,6,o,R
|
4,1,7,o,R
|
5,o,5,o,R
|
5,0,7,o,R
|
5,1,6,o,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
```

```
      A  MAY PRINT THE CURRENT VALUE OF THE TAPE.
      ∘
      P:TAPE
      10 TAPE CELLS WERE USED
      [0]□101≠1000→
      ∘
      A  THE [0] INDICATES THAT THE PORTION OF THE 'INFINITE' TAPE
      A  WHICH HAS BEEN PRINTED BEGINS WITH THE CELL  0 .
      ∘
      P:CELL
      CURRENT HEAD POSITION IS: 5
      ∘
      P:STATE
      CURRENT STATE IS: 4
      ∘
      A  NOT MUCH HAS HAPPENED, LET'S CONTINUE
      ∘
      GO
      4170R
      7070R
      7070R
      7070R
      7→7→R
      TRANSITION LIMIT REACHED
      ∘
      A  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
      ∘
      A  OOPS, WE GOOFED SOMEWHERE!
      A  WHERE IS THE READ HEAD?
      ∘
      P:CELL
      CURRENT HEAD POSITION IS: 12
      ∘
      A  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  O  R
4  1  7  O  R
4  O  4  O  R

5  0  7  O  R
5  1  6  O  R
5  O  5  O  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  O  8  O  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□O→≠
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).

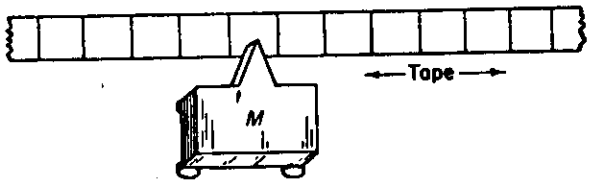


Fig. 6.0-1

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:

$(q_0, 0, q_0, 0, R)$ $(q_1, 0, q_1, 0, R)$
$(q_0, 1, q_1, 0, R)$ $(q_1, 1, q_0, 0, R)$
$(q_0, B, \text{HALT}, 0, -)$ $(q_1, B, \text{HALT}, 1, -)$

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.

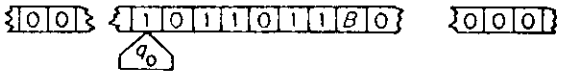
†The state denoted by q_{ij} is defined to be that one of the q_i's given by the function $G(q_i, s_j)$ and similarly for s_{ij} and for d_{ij}.

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

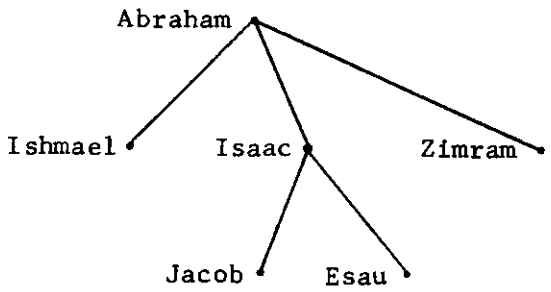
Question. Could you help me--I'm a little confused?

Answer. Sure, what's your problem?

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

A. Trees.

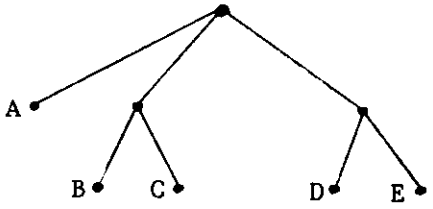
Q. Oh, I get it. Something like:



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:

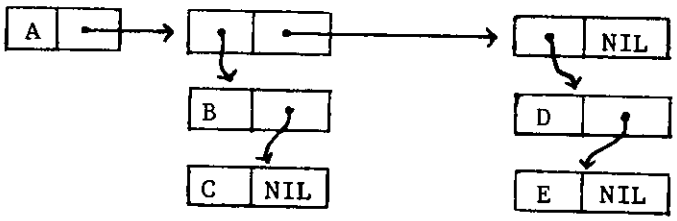


Q. But only the terminal nodes of your tree have data on them.

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:

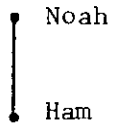
1) It is a means of helping you understand and differentiate between various tree structures.
2) It is a small (though non-trivial) exercise in the LISP programming language.

Problem

Consider the list L of father-son pairs:

```
L = ( (Isaac   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.)

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).

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.

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.

CIRCUIT DESIGN

Elaine Rich

The purpose of this problem is to illustrate different ways to build a binary adder using Digital Equipment Corporation (DEC) logic labs. It also provides an alternative to writing programs. Instead you can plug in wires and watch lights flicker.

A DEC logic lab is a box containing the circuits to perform logical functions. These circuits can be connected together by wires plugged into the front panel. It has eight j-k flip-flops, eight two input nand gates, six three input nand gates, four four input nand gates, and four two input exclusive or gates. There are three pulser switches and eight rocker switches for input, and eight lamps for output. The logic lab contains all of the functions necessary to build a digital computer. Needless to say, to do so would take quite a few labs. It is possible, however, to build an adder using just one. For a detailed description, see the manual which accompanies it, called the Computer Lab Workbook.

There are several ways to build a binary adder. One obvious method is to construct a combinatorial circuit which produces the desired output. Another approach is to build the add circuitry only once and use shift registers so that each pair of bits can be added using it. Still another technique is to build only a half adder between each pair of bits and then do the addition in two steps, the first of which adds the two input bits and produces a carry while the second adds the carry to the answer from the first step. Some of the tradeoffs involved are obvious. The pure combinatorial circuit is the fastest, but also the most expensive. The serial adder, i.e., the one with

only one adder, is cheaper but also slower, especially if the number of bits to be added is very large.

Constructing circuits on a device like the logic lab which has only a specific set of gates available creates considerations in design other than the obvious ones mentioned above. Some of these should become obvious as you try to build the different types of circuits.

For this problem, construct adders using the techniques mentioned above (and any others you can think of). In each case, try to build the largest one possible, i.e., one that can add the largest numbers. There should be a significant difference in the size of the adders you can build using the different techniques.

The fact that the logic lab has only nand gates (and a few not too generally useful exclusive or gates), rather than an assortment of and, or, and not gates, provides an additional challenge. At the same time it illustrates the advantages of a universal function, since with the nand gates you will never find yourself with nothing left but or gates when what you really need is an inverter.

Chapters 6 and 7 of the Computer Lab Workbook illustrate the construction of different types of adders. It would be good to do as much as you can without too much help from the manual, but if you are confused at this point, at least look at it to find out what the ideas are. It does show circuits for the adders described above, but these aids should be used only as a last resort.