PROBLEMS

The *Nice* Programming Language

1. Define a model of computation which does not depend on computers or programming.

2. We used floating point numbers instead of real numbers in our programming language. Why?

3. Add the data types character and string to the *NICE* language. Describe the operations which are needed to manipulate them.

4. Examine the following program:

```
program superexpo(x, y)
var m, n, w, x, y, z: integer;
begin
  w = 1;
  for m = 1 to y do
      begin
        z = 1;
        for n = 1 to w do z = z*x;
        w = z
      end;
  halt(z)
end
```

What are the values of the function it computes when y equals 1, 2, and 3? Describe this function in general.

5. How many multiplications are performed by the programs expo and superexpo? (Express your answer in terms of x and y.)

6. We have seen that exponentiation can be programmed with the use of one for-loop and that superexponenentiation can be done with two for-loops. What can be computed with three nested for-loops? How about four?

7. Suppose you are given a program named big(x) and you modify it by replacing all *halt*(z) statements with the pair:

```
z = z + 1.
halt(z)
```

   What does the new program compute? Would you believe anyone who told you that they have written a program which computes numbers larger than those computed by any other program?

8. Write a program which computes the function

$$\text{bar}(x) = \begin{cases} x \text{ if } x \text{ is odd and positive} \\ \text{undefined otherwise} \end{cases}$$

   Combine this with the program for the function fu(x) from the *NICE* language section to get an identity function program.

9. Suppose you have a program which computes the characteristic function for the predicate (or relation) P(x, y). This program provides a 1 as output when P(x, y) is true and a 0 whenever P(x, y) is false for x and y. Can you modify it so that your new program finds and returns for any y, the least x such that P(x, y) is true if there is one? This function is usually called $\mu x P(x, y)$ and defined:

$$\text{least}(y) = \begin{cases} \text{the least } x \text{ for which } P(x, y) \text{ is true} \\ \text{undefined if there is no such } x \end{cases}$$

10. Why is the "undefined" clause needed in the above definition of $\mu x P(x, y)$?

Turing Machines

1. What does the Turing machine of figure 2 which adds 1 to its input do when given #000 as input? What about the inputs: #bbb, #b011, and #11b10?

2. Examine the following Turing machine:

I1	0	1	right	same
	1	0	right	same
	b	b	left	next
	#	#	right	same

I2	0	1	halt	
	1	0	left	same
	#	#	halt	

   What does it do when presented with the inputs #1011, #1111, and #0000? In general, what does this machine accomplish?

3. Design a Turing machine that subtracts 1 from its input.

4. Design a Turing machine that recognizes inputs which read the same forwards and backwards. (The machine should halt with its output equal to 1 for inputs such as #101101 or #11011, and provide the output 0 for #1101 or #001110.)

5. How many instructions does the machine of the previous exercise execute on inputs which are n symbols in length?

6. Design a Turing machine which receives #xby (x and y are binary integers) as input and computes x + y. (You may use the machines that add and subtract 1 as subroutines.)

7. Write down the instructions for a Turing machine which determines whether its input is zero. What happens when this machine is given a blank tape as input?

8. How many instructions are executed by the Turing machines of the last two problems on inputs of length n?

9. Design a Turing machine which computes the fu(x) function of the *NICE* language section.

10. A ***0-1 valued Turing machine*** is a machine which always provides outputs of 0 or 1. Since it halts for all inputs, it computes what is known as a ***total function.*** Assume that $M_i(x, y)$ is a 0-1 valued Turing machine and design a machine which receives y as input and halts if and only if there is an x for which the machine $M_i(x, y) = 1$.

A Smaller Programming Language

1. Describe the ways in which division must change after floating point numbers have been replaced by triples of integers which denote their signs, absolute values and decimal points.

2. Assume that you have programs for the following functions:

```
prime(i) = the i-th prime number
expo(x, y) = x raised to the y-th power.
```

   A pair such as (x, y) can be uniquely encoded as:

```
expo(prime(1),x)*expo(prime(2),y)
```

   and decoded by a suitable division routine. In a similar way, any single dimensional array might be encoded. Describe the way in which an array can be encoded as a single integer. Then write a select(a, k) function which provides the k-th element of the array encoded as the integer a, and a replace(a, k, x) program which sets the k-th element of a to the value of x.

3. How might a two dimensional array be encoded as a single integer? What about an n-dimensional array?

4. Arrays of programming languages have declared bounds on each dimension. Is this restriction needed for our purposes? How might the routines of the last two problems be affected if they were to be written for arbitrarily large arrays?

5. Define integer division. Show that division can be replaced by subtraction in much the same way that multiplication was replaced by addition.

6. If we allow the predecessor operation (x = x - 1) to be included in a programming language, then subtraction is not needed. Show this.

7. Suppose procedure calls had been included in our original programming language. How might they have been eliminated? What about recursive calls?

8. Many modern programming languages include pointers as a data type. Do they allow us to compute any functions that cannot be computed in our simple language? Why?

9. Dynamic storage allocation is provided by some languages that let programs call for new variables and structures during runtime. Is this necessary for increased computational power?

10. The size of a program could be defined as the number of symbols in the program. (In other words: the length of the program.) Consider two programs (one in the extended language and one in the simplified language) that compute the same function.

    a) How might their sizes differ?
    b) Compare their running times.

11. Let's consider programs in our *SMALL* language which have no input (and thus need no titles). The only one line program that computes a defined function is:

```
1: halt(x)
```

and this program outputs a zero since we assume that all variables are initialized to zero. The two-line program which computes a larger value than any other two-line program is obviously:

```
1: x = x + 1;
2: halt(x)
```

and this outputs a 1. We could go to three lines and find that an even larger number (in fact: 2) can be computed and output. We shall now add a little computational power by allowing statements of the form:

```
k: x = y
```

and ask some questions. What are the maximum values computed by 4 and 5 line programs of this kind. How about 6? 7? etc.? Do you see a pattern emerging? How about a general formula?

(Years ago Rado at Bell Laboratories thought up this famous problem. He called it the **Busy Beaver Problem** and stated it as:

"How many 1's can an n-instruction Turing machine print before halting if it begins with a blank tape as input?"

12. Suppose that someone gives you a *SMALL* language program called beaver(x) which computes the Busy Beaver function of the last exercise. You find that it has exactly k+1 lines of code and ends with a halt(z) statement. After removing the title and the *halt*, it can be embedded on lines k+13 through 2k+12 in the following code to make the program:

```
   1: x = x + 1;
   2: x = x + 1;
          •
          •
          •
 k+7: x = x + 1;
 k+8: y = x;
 k+9:  y = y - 1;
k+10: x = x + 1;
k+11: if y = 0 then goto k+13;
k+12: if w = 0 then goto k+9;
k+13: 
        Program for beaver(x)
2k+12:
2k+13: z = z + 1;
2k+14: halt(z)
```

Now let's ask some questions about this new program.

a) What value does x posses just before line k+13 is executed?
b) What value is output by this program?
c) What is the value (in words) of beaver(x)?
d) What is the value of z (in words) at line 2k+12?
e) How many lines does this program have?

Comment on this!

Equivalence of the Models

1. Design a "blank-squeezing" Turing machine. That is, a machine which converts #xbby to #xby.

2. Translate the program instruction $x_i = x_k$ into Turing machine instructions.

3. If a *SMALL* program that has n variables executes k instructions, how large can the values of these variables be? How much tape must a Turing machine have to simulate the program?

4. Compare the running times of Turing machines and *SMALL* programs. Assume that one instruction of either can be executed in one unit of time.

5. Translate the Turing machine instructions of problem 2 ($x_i = x_k$) into *NICE* program instructions. Comment on mechanical translation procedures.

6. In the translation from Turing machines to programs an array was used to hold the Turing machine tape. How might scalar variables be employed instead? How would reading, writing and head movement take place?

7. Discuss size trade-offs between Turing machines and programs that compute the same function.

Machine Enhancement

1. Turing machines have often been defined so that they can remain on a tape square if desired. Add the command *stay* to the Turing machine moves and show that this new device is equivalent to the ordinary Turing machine model.

2. Post machines are very much like Turing machines. The major difference is that a Post machine may write or move, but not both on the same instruction. For example, the instruction:

0	**left**	**next**
1	**0**	**same**
b	**halt**	

tells the machine to move if it reads a 0 and to write if it reads a 1. Show that Post machines are equivalent to Turing machines.

3. Endmarkers on our Turing machine tapes were quite useful when we wished not to fall off the left end of the tape during a computation. Show that they are a bit of a luxury and that one can do without them.

4. How much more tape does a two symbol (0, 1, and blank) machine use when it is simulating an n symbol machine? Can this extra space be reduced? How?

5. Design a Turing machine that receives a binary number as input and transforms it into encoded decimal form. (Use the encoding of the machine enhancement section.)

6. Describe the process of changing an encoded decimal into the equivalent binary number.

7. Show that Turing machines which use one symbol and a blank are equivalent to ordinary Turing machines.

8. Describe instructions for multi-tape Turing machines. Specify input and output conventions. Prove that these machines are equivalent to one tape Turing machines.

9. Consider Turing machines that operate on two dimensional surfaces that look something like infinite chessboards. They now require two additional moves (*up* and *down*) in order to take advantage of their new data structure. Prove that these machines are equivalent to standard Turing machines.

10. Turing machines need not have only one head per tape. Define multiheaded Turing machines. Demonstrate their equivalence to Turing machines that have one head.

11. Consider the problem of recognizing strings which consist of n ones followed be n zeros. How fast can this set be recognized with Turing machines that have:

    a) Two tapes with one head per tape.
    b) One tape and one tape head.
    c) One tape and two tape heads.

    Describe your algorithms and comment on the time trade-offs that seem to occur.

12. A wide-band Turing machine is able to scan several symbols at one time. Define this class of machines and show that they are equivalent to standard Turing machines.

13. Can wide-band Turing machines compute faster than standard Turing machines? Discuss this.