Burt Rosenberg

Problem Set 10

OUT: 16 NOVEMBER, 1993
DUE: 3 DECEMBER, 1993

Reading Assignment

Read Chapters 19–21 from *Algorithms.*

Turing Machines

A Turing Machine is a very simple type of computer. Although simple, it is as powerful as any computer and any known computational device, and its simple programming language is as powerful as any known programming language. In fact, in close inspection, the machine we are about to present is *more* powerful than any computer or computer language available today.

The Turing Machine consists of two parts:

- a read-write tape,
- a finite-state automata.

The read-write tape is a one-dimensional array of cells, infinitely extending both to the left and to the right. Each cell is a place to store a single symbol. A symbol is simply a character from some pre-arranged class of characters, called an *alphabet.* A special *blank* character is the value of a cell if the cell is otherwise empty.

The finite-state automata can write, and later read, a symbol in each cell of the tape. It can overwrite the cell with a new symbol, just as computers can put a new value into a variable, overwriting the old value. A movable *head* is positioned over the cell that the automata is at that moment capable of reading or writing. The automata can move the head to the left or to the right, one step at a time.

A finite-state automata is a simple machine consisting of a set of *states* and a pre-arranged network of state interconnections. One imagines a single token moving about the network of states, The movement of a token from one state to another is called a *state transition.* State transitions are triggered by the contents of the tape cell over which the head is now positioned. Each state transition has three effects,

1. The token is moved to a new state.
2. The contents of the tape cell is optionally changed.
3. The head is optionally moved one step to the right or left.

The details of the state transition is in fact the program the Turing Machine carries out. To describe a Turing Machine is to give the set of states, noting which state is the special *start state* and which is the special *halt state*, as well as the set of state-transitions, including the following five items:

1. the *from* state of the transition,
2. the *to* state of the transition,
3. the tape symbol that would cause the transition,
4. if the tape symbol is to be changed, the new tape symbol,
5. if the head is to be moved, the direction that the head will move one step.

The halt state has no outward going state transitions. When in the halt state, the Turing Machine is stopped.

A computation on a Turing Machine requires intervention from the operator for four actions. First, the operator writes the input into the cells of a completely blank tape. Second, he/she places the head over the left-most character of the input and places a token in the start state. Third, he/she pushes the start button. The Turing Machine goes through its state transitions, moving the head to and fro, tirelessly writing symbols, reading symbols, following its prearranged set of state transitions. Perhaps it is computing Pi to a million places, perhaps it is discovering a new melodic pattern in Bach's Prelude and Fugue.

Perhaps eventually the operator intervenes for the fourth time, noticing that the Turing Machine has gone into its halt state and is no longer computing. The operator observes what is left written on the tape: this is the output or the result of the computation. A Turing Machine might never halt. It may be impossible to predict whether a Turing Machine will ever halt. That is, it may be impossible to decide if said Turing Machine is working out some mysterious internal computation on its way to discovering a

new melodic pattern in Bach's Prelude and Fugue, or maybe it has just gone bananas. How is anyone to know?

Programming Assignment

Write a Turing Machine simulator. The input should be a description of the Turing Machine followed by the input on which the Turing Machine is to run. The output is (at least) a print out of the final contents of the tape.

The description of the Turing Machine assumes that states are labeled with nonnegative integers, that state 0 is the halt state, and state 1 is the start state. Each transition is described by a line of the format:

```
I J x y d
```

where I is the index of the "from" state, J is the index of the "to" state, x is the character of the alphabet which triggers this transition, y is the new character written, and d is either L, R or N for move left, move right or no movement. By convention, the "-" symbol will represent a blank character on the tape. After all the transitions have been entered, a line starting with a period (.) signals the end of the state transition input.

Let lines beginning with # be comments. After the state transitions, the next non-blank, non-comment line is the input to start the Turing Machine on.

For instance, here is a program which adds 1 to a number in binary. Remember, the Turing machine is started with the head over the leftmost character of the input.

```
# Turing Machine to increment a number written in Binary
#
# In state 1, there is a carry. If you see a 1, change
# to a 0, move the carry right:
1 1 1 0 R
# When there is a carry, and you see a 0, change to
# a 1 and you are done:
1 2 0 1 N
# Or, may be there is a carry out of the most sig. bit
1 2 - 1 N
```

```
# Clean up, addition done, move head back to leftmost bit
2 2 0 0 L
2 2 1 1 L
2 0 - - R
.
#input
11011001
```

This machine should terminate and print out,

```
00111001
```

- Now use your Turing Machine simulator on a program you write to add two numbers in binary. That is, the input is something like,

  ```
  001101101 110111
  ```

  two strings on 0-1 separated by a blank, and the output is the sum,

  ```
  111001011
  ```

- Now write a multiplication routine.

HINT: As with most of this programming business, the key is to write small, simple, reusable routines, which can be joined together to accomplish larger and larger goals. But there is no real way of "calling" a routine. Each time you want to call a routine, you must make of copy of all its transitions using a fresh set of states.

You might think of writing a program to do this for you: Given a set of Turing Machine transitions including a mechanism for having a group of states collected under a single "procedure" name, and a "call" operation, the program rewrites the transitions, making copies of blocks of transitions as needed, to remove the call operations. That is, it replaces calls with copies of the procedure, renumbering states so that fresh state names are used each time.