COGS Q350 Homework 4

Due Monday, October 19, 2015

- All 10 problems below plus #1-5 from section 8.1 of the attached Turing's World exerpt.

Prefix and Postfix Notation

Convert from prefix to infix notation. Draw a tree for each.

```
+ * A B / C D
```

```
/ * A + B C D
```

```
* A + B / C D
```

```
^ -> A B ~ v C D
```

Convert from postfix to infix notation. Draw a tree for each.

```
A B * C D / +
```

```
x y ^ 5 z * / 10 +
```

```
x y + z x / / w v - *
```

Convert from infix to prefix and to postfix. Draw a tree for each.

```
~(A -> B) v (C ^ D)
```

```
~(A -> B) v (C ^ D) -> ~E
```

```
((A * B) ^ 2)  + ((8 - A) / 2)
```

8

Additional Exercises and Projects

The authors would enjoy getting copies of particularly elegant solutions to any of these exercises, or other interesting machines the reader may build. Please send disks or hardcopy to either of us. We will be happy to return your disk if so requested.

8.1 Finite state machines

The problems in this section assume that you have read about finite state machines in Chapter 7. You can skip this section if you want to move directly to Turing machines.

Exercise 1 (Dragonslayer) Imagine a knight on a journey through a land infested with dragons (D's) and evil trolls (T's), as well as some friendly civilians (F's). He starts his journey with no weapons, but along the way he can find swords (→'s), used to slay dragons, and acquire spells (#'s), used to enchant the stupid trolls. He can carry at most one of each at any one time, and they are gone once he uses them. Luckily, people seem to have dropped a lot of swords and spells around the landscape.

Design a one-way finite state machine that accepts a string if and only if the string represents a journey on which the knight survives. Thus, for example, it should accept F→FD#→TFD but it should re-

ject F→FD#→TFDT, since the final troll gets the best of him.

Exercise 2 (Lights out) Imagine you have three light switches, numbered 1, 2, and 3, controlling two lights in a hallway, one at the north end, the other at the south end. The way the switches work is as follows:

1. Switch 1 turns the north light on or off, depending on whether it is currently off or on.
2. Switch 2 switches both of the lights, changing them from on to off or off to on.
3. Switch 3 turns the south light on or off.

Design a one-way finite state machine that checks to see if both the lights are off. That is, it should accept a string of 1's, 2's, and 3's if and only if the string represents a sequence of switch flips that results in both the lights being off (assuming they were off to start with). For example, it should accept the string 213 since this turns on both the lights and then turns off the north light and then the south light. By contrast, it should reject the string 23213 since this sequence of flips leaves the north light on.

Exercise 3 (0-1-2 Cube) Design a one-way finite state machine that accepts a string of 0's, 1's, and 2's if and only if the string contains an odd number of each numeral. Thus it should accept the string 1121022, but not 1120022.

Exercise 4 (Casting 3's) A well-known trick for seeing if a number expressed in base-ten notation is divisible by three is to add up the digits and see if the sum is divisible by 3. Using this trick, design a finite state machine that runs through a base-ten numeral and accepts it if and only if it represents a number divisible by 3. You should be able to do this with a machine having just three states.

Regular languages and finite state machines

A finite state machine using a given alphabet is thought of as determining a set of strings, the strings made up from letters in that alphabet that are accepted by the machine. This is called the *language* accepted by the machine. It is often denoted by $L(M)$, where M is the machine in question. A set of strings is said to be a *regular language* if it is the language accepted by some finite state machine. The theory of regular languages is a central topic in computer science.

Exercise 5 (Regular languages) Build one-way, deterministic finite state machines which show that the following are regular languages. Assume that the alphabet contains just A and B.

1. The set of strings that end in BBB.
2. The set of strings that do not end in BBB.
3. The set of strings that contain at least five consecutive B's. Thus it would accept BABBBBBAB but it would reject BABBBBAB.
4. The set of strings that do not contain at least five consecutive B's.

Exercise 6 (Equal A's and B's) Design a one-way deterministic finite state machine that operates on the set of strings of A's and B's, and accepts just those strings that contain the same number of A's and B's, and have the additional property that no prefix contains more than one more of one numeral than of the other. Thus, for example, it should accept BAABAB and ABABBA but it should reject ABBBAA since the prefix ABBB contains two more B's than A's. [This exercise is suggested by Exercise 2.7 in [Hopcroft & Ullman]. It is interesting because the set of strings that contain the same number of A's and B's is not a regular language. In other words, there is no finite state machine that accepts all and only these strings.]

Exercise 7 Design a one-way deterministic finite state machine to show that the set of all strings of A, B, (and) in which the parentheses are matched, but which contain no nestings of parentheses is a regular language.

Exercise 8 Design a one-way deterministic finite state machine to show that the set of all strings of A, B, (and) in which the parentheses are matched, but which contain no nestings of parentheses of depth greater than three is a regular language. For example A(((AB))A)() should be accepted but ((((A)B))) shouldn't. How could your machine be simplified if it were to accept only those strings which contain no nestings of depth greater than two?

The point of this exercise is to give you a feeling for why the set of matched strings is not a regular language. The greater the depth of nesting to be tested, the more states you would need for your machine. When you have finished building your finite state machine, go back and open Paren Check to remind yourself how this Turing machine manages to recognize this language.

Exercise 9 Build a one-way, deterministic finite state machine that accepts the same language accepted by the machine you built in Exercise 3, page 82.

Exercise 10 Build the two-way, deterministic finite state machine with only one state, an accepting state, with two arcs. Label one of these arcs (A, $\Rightarrow$) and the other (B, $\Leftarrow$). What language is accepted by this little machine? Design a one-way finite state machine that accepts the same language.

8.2 Turing machines

Exercise 11 (Food chain simulation) Design a Turing machine that will simulate action in the food chain. Imagine that a B always "eats" any A that it's facing and moves forward (that is, right), taking its place. (When the B moves forward, everyone behind him should follow.) Then, after all the B's have eaten, a C always eats any B that it's facing and moves forward. For example, the sequence CACBBAAA would turn into CAC. Your machine should transform any tape with a string of A's, B's and C's into one that represents the result of everyone's having eaten.

Exercise 12 (Copier) In building complex machines, it is often important to be able to copy a string of symbols. Build a machine which acts as follows. It starts on the left of a string α of A's and B's and ends on the left of the string $\alpha\alpha$, that is, it ends with two copies of α, with no space between them. You may use auxiliary symbols.

Exercise 13 Insert the copier into the holes in the Iterater schematic machine described in Chapter 6, page 74, and see if it works as expected.

Exercise 14 Redo Exercise 12 without using any symbols other than A and B. Compare the complexity of the two machines in terms of the number of states and arcs. Compare the time-space count of the two when they each copy the string

BBABAB

to get the string

BBABABBBABAB.

Exercise 15 (A^nB^n) Build a Turing machine that begins at the left of a (possibly empty) string of A's and B's, and writes GOOD if the string is of the form A^nB^n (i.e., has zero or more A's followed by the very same number of B's), but writes BAD otherwise. This is something that cannot be done by a finite state machine, as suggested in Exercise 6.

Exercise 16 (Cancelation) Exercise 4 asked you to combine our three prepackaged machines into one that

represented multiplication in any Abelian group with the two elements A and B. Now suppose that in addition we assume $A = B^{-1}$; that is, that AB is the multiplicative identity, so that you cancel any A with a B next to it. Design a machine that will accept any string of A's, B's, ('s, and)'s, and do as before, except this time canceling occurrences of AB. This exercise raises a general question about the computability of operations in a group. For more on this important topic, the reader is referred to Davis's chapter in [Barwise].

Exercise 17 (Coding) In Chapter 3, page 56, we described a way to encrypt the alphabet A, B, – with the alphabet *, –. Build a Turing machine to carry out this encryption, one that takes an arbitrary string of A's, B's and –'s and replaces it with the encrypted version. Assume that the input string is enclosed in parentheses, and likewise have the output delimited with parentheses. (What problems would come up if we didn't assume this?) Your machine should start and stop on the left parenthesis.

Exercise 18 (Decoding) Build a decoding machine for the system of encryption used in Exercise 17.

Exercise 19 Put together the machines designed in Exercises 17 and 18 so that they take a string, code it, and then decode it back into the original string.

Exercise 20 (Coding using the default alphabet) In Exercise 17, we really cheated, since we used auxiliary parentheses in our alphabet. Design a coding machine that will allow us to do without any auxiliary symbols. To do this, you need to assume that the input string does not contain two or more blanks in a row. In other words, A––B is not a legitimate input, though A–B–B is. Your machine should start on the first symbol of the input string (which could be a blank) and should halt on the first square of the resulting code (which again may be blank).

8.3 Numerical computation

Given any Turing machine M on the alphabet * and blank, we will say that M computes the (possibly partial) function f^2_M of two numerical arguments, where this function is determined as follows. Given numbers j, k, form a tape with $j+1$ *'s, a blank, then $k+1$ *'s. We use one extra * so that 0 is represented by something definite, a single *, rather than by the blank, which is used to separate the arguments. Position the read head at the leftmost * and run the machine. If the machine eventually halts with the read head at the left end of a string of $m+1$ *'s (on an otherwise blank tape), then we will say that $f^2_M(j,k) = m$. If the machine does not halt, or halts in some other configuration, then $f^2_M(j,k)$ is not defined. (If f^2_M is defined for all arguments, then it is said to be a *total* function; if not it is *partial.*)

We can generalize this for functions of other than two arguments in the natural way. Any given machine M can be said to compute a function f^n_M of n arguments: Given natural numbers $k_1, \ldots, k_n$, form a tape with k_1+1 *'s, a blank, then k_2+1 *'s, a blank, and so on. Position the read head at the leftmost * and run the machine. If the machine halts, and in doing so has the read head at the left end of a string of $m+1$ *'s (on an otherwise blank tape), then $f^n_M(k_1, \ldots, k_n) = m$. If the machine does not halt, or halts in some other configuration, then $f^n_M(k_1, \ldots, k_n)$ is not defined.

The conventions we've just described for determining the function computed by a Turing machine are called the *standard input-output conventions.* The leftmost nonblank symbol on the input tape is said to be the *standard starting position.* Since Turing machines simply manipulate marks on a tape, it's clear that we