COMP 200: Elements of Computer Science
Fall 2004
Lecture 33: November 19, 2004

Describing Syntax — Finite Automata

Reading Strings of Input

Many problems in Computer Science are characterized by recognizing specific patterns in a string of input. Recognizing a URL (uniform resource locator, as in a web browser — http://www.owlnet.rice.edu/~comp200 is a URL), understanding a typed command line, reading a line of Scheme code in DrScheme's definitions window, and parsing a string given to the Google or MSN search pages — all of these are simple input recognition problems.

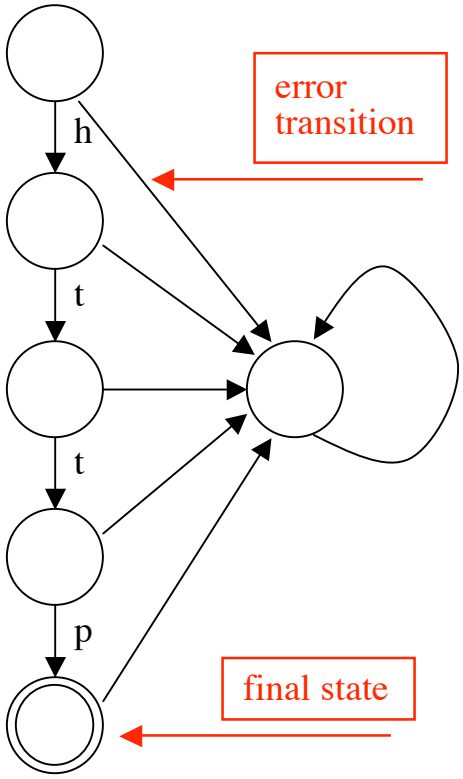
If we wanted to recognize some string, such as "http", we might write code such as

```
c ← read()
if c ≠ 'h'
   then report failure and stop

c ← read()
if c ≠ 't'
   then report failure and stop

c ← read()
if c ≠ 't'
   then report failure and stop

c ← read()
if c ≠ 'p'
   then report failure and stop
```



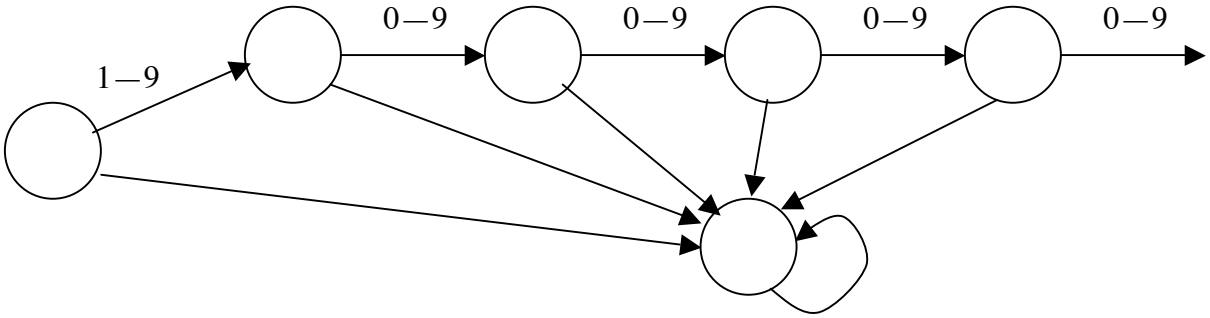
Rules of operation:

1. Start in the initial state
2. On each input character, follow the appropriate transition
3. When input is exhausted, if state is a final state, accept (else reject)

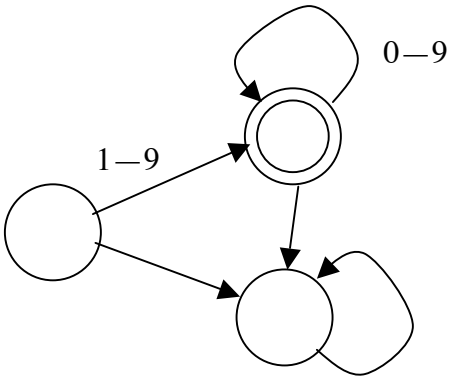
To recognize a second string, such as “fee”, we could build a second recognizer. If the string was related, as in “https” or “heat”, we could combine the recognizers so that they used common states. (A recognizer for “https” would simply add another final state, a transition using ‘s’ from the current final state to the new final state, and one more error transition.) We can design recognizers around the state transition diagrams. It turns out that we can easily and mechanically turn a state diagram into program text (code). If the code is written in some stylized, standard form, as our example is, we can also go from the code to the state diagram.

Recognizing More Complex Words

Consider a recognizer, or ***finite automaton***, that recognizes integers. It might look like



This one does not look ***finite***. In fact, it looks as if it goes on forever. And, where does the final state go? We can make them all final states, except for the initial state and the error state. If we bound the length of the integer, we get a ***finite*** state diagram. Another way to approach the problem of arbitrary length strings is to allow a state to transition back to itself.



This automaton is finite. It accepts any finite integer. (It doesn’t accept an infinite integer because it never stops running along the circular transition.)

It clearly requires a small finite amount of space. It makes one transition per input character, so it should be **O**(| *input* |).

These finite automata ae quire powerful. Given two finite automata, we can always construct a finite automaton that is their union. (Merge their start states, and we may need to merge more if they have common prefixes. The full proof is harder than that, but follows along those lines.) This simple property ensures us that we can recognize any word in a list of words in time proportional to the size of the word. (Build a finite automaton for each word; merge the automata; run the result. It always takes one transition per input character.)

This technology is used in products that block URLs and that recognize blacklisted email addresses. We cannot expect, in general, to recognize strings faster than **O**(| *input* |).

[It turns out, we can sometimes recognize a string in sublinear time – that is, without looking at all the characters. The ideas is simple, but the math is complex. We compute a failure function that tells us, for each position in the pattern, how far over we can skip in the input. After all, we know that we have matched some part of the pattern, so we know whether or not that matched part contains another prefix of the string. However, as the number of words in our list grows, the opportunities for skipping letters will diminish until, in the limit, we get back to **O**(| *input* |). URL blocking services have very large lists – hundreds of thousands of URLs.)

Formalism – What can a finite automaton do?

These finite automata are equivalent to Turing Machines with a one-way tape – that is, the tape is constrained to only move *left*. We talked about the fact that the Turing machines were fairly robust – one way infinite tapes are equivalent to two-way infinite tapes and so on. This restriction – one-way motion – severely limits the power of the Turing machine.

A finite automaton cannot count. It cannot match parentheses; that would involve encoding the open parentheses in some set of states and then backing up through them. (Draw a six-paren deep machine. The general machine needs one state per level it counts.)

We cannot recognize a palindrome. Again, that requires matching up symbols, which is akin to counting.

However, this subject is a little subtle. A finite automaton can recognize bounded sets and bounded differences – strings with alternating *a*'s and *b*'s or with even numbers of *a*'s and *b*'s.