1 Finite State Machines (FSMs)

1.1 What is an FSM?

An FSM is a mathematical model for a particularly simple type of computation. As we shall see later, it doesn't capture the full complexity of other models of computation.

Okay, what's a model of computation, and why would anyone use one?

A model of computation is a simple mathematical model which retains the key elements of computation. (It will probably help when we see an example.) It is useful to have simple mathematical models, because they allow us to prove theorems about what sort of problems can and cannot be solved by computer (or brain).

Formally, a FSM f has an input alphabet A. For the sake of discussion, let $A = \{a,b\}$.

Inputs to f consist of strings over the alphabet A. Thus: a, b, aa, ab, ba, bb, etc. are all legal inputs to f.

For each input I, a string over the alphabet A, the FSM f either *accepts* or *rejects* the input I.

1.1.1 Examples

An FSM that accepts all strings over the alphabet { a, b } that end with a.

An FSM that accepts all strings over the alphabet { a, b } that contain *at least* one a.

An FSM that accepts all strings over the alphabet { a, b } which contain an odd number of a's.

An FSM that accepts all strings over the alphabet { a, b } of the form ab, abab, ababab, ...

1.2 Regular Expressions

The set of languages that are accepted by FSMs are called *regular expressions*.

Formally, a *regular expression* is defined (recursively) by:

1) The empty string ϵ is a regular expression.

2) For every character c in the input alphabet, c is a regular expression.

3) If X and Y are regular expressions, then X OR Y is a regular expression. (I.e. if you can construct an FSM to accept strings of the form X and you can construct a FSM to recognize strings of the form Y, then you can construct one to recognize strings of either form.)

4) If X and Y are regular expressions, then the concatanation, XY is also a regular expression.

1.3 DFSMs vs. NDFSMs

So far I've steered away from introducing too much complexity. When I've referred to FSMs, I've really been talking about Deterministic Finite State Ma-

chines, DFSMs. There is another class of FSMs called NDFSMs which are Non-Deterministic Finite State Machines. What's the difference?

For a DFSM, whenever you have an input, you know exactly which transition to make, i.e. the transition is deterministic. For an NDFSM, there can be different transitions for the same input, and there can be transitions for no input. An NDFSM *accepts* a language if *any* possible set of transitions accepts the language.

With the concept of NDFSMs under our belt, we're ready to ask the tough question — are some FSMs more powerful than others?

(What do we mean by *more powerful*? It means that the set of languages which can be recognized is larger.)

How about if we allow multiple stop states? This is not more powerful - we can add one single stop state, and put an ϵ transition from each previous stop state to the new stop state.

Now for the $64,000 question — are NDFSMs more powerful than DFSMs?

Surprisingly, no. It turns out that we can convert any NDFSM of size N into a DFSM of size $\leq 2^N$. All we have to do is create new states which correspond to *sets* of possible states in the NDFSM.

So, DFSMs and NDFSMs are equally powerful. Is that it? Are there problems they can't compute?

Actually, yes. Try building an FSM to recognize languages of the form $a^n b^n$. It is provable that this is not possible, but I won't include the proof here. (If you're interested, look up *The Pumping Lemma*.)

2 Turing Machines

2.1 What is a Turing Machine?

A different model of computation, and, as we shall see, more powerful.

A Turing machine consists of:

1) An infinitely long tape, which consists of characters and blanks.
2) A finite input alphabet.
3) A finite output alphabet.
4) A special *blank* character, usually denoted B.
5) A finite set of states.
6) A start state.
7) A (finite) set of accepting states.
8) Rules of transition. Each rule is of the form:

If you're in state S and you see input I, write O from the output alphabet, move { left, right }, and go to state $S2$.

2.1.1 Examples

A Turing Machine to recognize strings of the letter a.

A Turing Machine to add one to a number.

A Turing Machine to double a number.

2.2 The Power of Turing Machines

Example:

A Turing Machine to recognize strings of the form $a^n b^n$.

Turing Machines are more powerful than FSMs.

Formally, *every language that can be recognized by an FSM can be recognized by a TM.*

2.3 The Church/Turing thesis

Every language that can be recognized by any computational model *can be recognized by a TM.*

Proof left as an exercise for the reader. :-)

2.4 Limits of Turing Machines

Can a Turing Machine compute *everything*?

No, there are some things it can't compute. But no known computational model can compute them either.

For example, the *Halting Problem*: Given as input some sort of representation of a Turing Machine and an input to that TM, will the machine halt (either accept or reject the input) in a finite amount of time.

Proof is tricky. If you really want to see it, let me know.

Equivalently, given a computer program (possibly with inputs) and a certain line in that program, will the program ever execute that line?

There are other things that TMs can't compute, but none come to mind right now.