1 Introduction

Our not entirely unambitious goal in the next few lectures is to try to come to grips with the question:

> What do we mean by the phrase "mechanical computation"? In particular, what are some examples of problems that can, or more interestingly cannot be solved by mechanical computation?

In attacking this problem we will introduce an abstract version of a mechanical computer, the *Turing machine* (TM). The key insight is that, as far as anyone knows, this model and every other "sufficiently powerful" model of mechanical computing solve exactly the same set of problems. This idea is called the *Church-Turing thesis*:

> Any two models of mechanical computation solve the same set of problems.

A reason to believe in the truth of the Church-Turing thesis (aside from the fact that no one has ever found a counterexample) is that one of the things that mechanical computation devices can do is to *simulate* other such devices. If A can simulate B, and B can also simulate A, then their computational powers are equivalent.

2 Model matters

Before moving on to the definition of TMs let us return to push down automata and ask what happens if we replace the stack by a queue (resulting in a so-called queue machine)? The reason for doing this is the recognition that one of the problems with PDAs was the "burn after using" nature of the stack memory. In a queue model, we could at least retain some elements of the memory while exposing others.

Our acceptance criterion will be (as for PDAs) that we have exhausted the input, the memory is empty, and we are in an accepting state. I want to demonstrate that we can accept the language:

$$\text{POWERS-OF-2} = \{a^{2^n} \mid n \geq 0\}.$$

A high level description of the operation of a machine that accepts this language is as follows:

1. Copy the complete input onto the queue.
2. Pop elements off the queue, pushing on a b for every pair aa popped.
3. When you reach a b go through the queue, replacing each b by a.
4. Repeat from step 2.

The idea is that we will halt whenever we reach some unpaired a. If the queue is empty that was the only a on the stack and we should (and will) accept. Otherwise, there are some b's on the stack, meaning that we had an odd number of a's (and more than one) when we started this pass through the data, meaning that the number of a's in the input was of the form odd $\times 2^k$ for some odd number greater than one, meaning that we should (and will) reject.

We can describe this machine simply in a table (note that 'Read' means 'read from the input', while 'Pop' and 'Push' refer to operations on the queue).

State	Read	Pop	Push	New state
GET-INPUT	a	ϵ	a	GET-INPUT
	ϵ	ϵ	ϵ	GET-ODD-a
GET-ODD-a	ϵ	a	ϵ	GET-EVEN-a
	ϵ	b	b	REPLACE-b
GET-EVEN-a	ϵ	a	b	GET-ODD-a
REPLACE-b	ϵ	b	a	REPLACE-b
	ϵ	a	a	GET-ODD-a

The only accepting state is GET-EVEN-a and we accept there only if the input is empty (i.e. we consumed all the input in the GET-INPUT phase and the queue is empty.)

3 Definition of a Turing machine

The main thing that will ensure that the model we are working with remains 'mechanical' is that we will insist there only be finitely many possible internal 'states'. However, as in

the PDA and queue machine models we will conceive of a (potentially) infinite memory.

Furthermore, the other main change we will make is to remove the separation between the input (words to be recognised) and the contents of the memory. In the queue example above we already saw a simple version of this where we had a preprocessing phase of simply copying the entire input to memory and then manipulating it there.

The mental model of the memory that a TM uses is of a tape divided into cells where each cell can hold one symbol. Sitting above the tape is a read-write head. A single atomic step of a TM is:

- Read the current symbol on the tape
- Based on it and the current state,
  - Write a symbol in the current position,
  - Move the read-write head one cell to the right or left,
  - Change state.

So, to specify a TM we simply need to define the set of states and the transition function implicit in the second item above. Inputs to the transition function are pairs consisting of the current state and current symbol, and outputs are triples consisting of new symbol, motion direction, and new state. The transition function will almost always be partial since we want to allow the machine to halt, which happens when there is no transition defined for the current state and symbol pair. It may be deterministic or not – initially we will assume that it is.

The alphabet, Γ of a TM consists of an *input alphabet*, Σ, usually denoted using lower case letters, a special *blank* symbol representing an empty cell, denoted B, and other working or marker symbols denoted by other upper case letters or punctuation marks. The only restriction is that the complete alphabet be finite.

To process a word $w \in \Sigma^*$ we assume that the initial contents of the tape are a blank cell, followed by w (one character per cell), followed by an infinite sequence of blanks. We do not worry about what the machine might do to input of any other form. The TM begins in some initial state START with the read-write head over the leftmost (blank) cell.

We can be fairly flexible about the acceptance criteria, but they must involve the machine halting (i.e. there being no transition defined from the current state when reading the current symbol) and may involve being in an accepting state. One new thing that can happen is that we might crash the machine by trying to move the read-write head off the end of the tape. Such crashes are always considered to be rejecting computations.

4 An example

Consider the language $L = \{wcw \mid w \in \{a, b\}^*\}$ over $\Sigma = \{a, b, c\}$. We know that this cannot be recognised by a PDA. Can it be recognised by a TM? Of course.

Before designing the TM in detail let's consider how it could operate. Basically on an acceptable input (any word over Σ preceded by a blank) we want to read the first character, move forward past the first c and check that the next character matches. Then we need to mark those characters as 'processed' somehow and proceed in much the same fashion with the first unprocessessed character. If we ever find a mismatch we halt and reject. After processing all the characters before the first c, if there are any unprocessed characters remaining after it, we should also reject. Otherwise, we accept.

We can now assemble our machine in tabular form as we did in the queue example above:

State	Read	Write	Move	New state
START	B	B	R	READ
READ	a	X	R	SEEK-a
	b	X	R	SEEK-b
	c	c	R	CHECK
	X	X	R	READ
SEEK-a	a	a	R	SEEK-a
	b	b	R	SEEK-a
	c	c	R	LOOK-a
SEEK-b	a	a	R	SEEK-b
	b	b	R	SEEK-b
	c	c	R	LOOK-b
LOOK-a	X	X	R	LOOK-a
	a	X	L	MOVE-BACK
LOOK-b	X	X	R	LOOK-b
	b	X	L	MOVE-BACK
MOVE-BACK	a	a	L	MOVE-BACK
	b	b	L	MOVE-BACK
	c	c	L	MOVE-BACK
	X	X	L	MOVE-BACK
	B	B	R	READ
CHECK	X	X	R	CHECK
	B	B	R	ACCEPT

Note that there are no transitions defined for ACCEPT so as soon as we enter this state we will halt (and of course, accept). If we halt in any other situation (e.g. seeing a b when in state LOOK-a, or seeing anything other than X or B in CHECK) then we reject.

Of course you should trace the computation of this machine on various valid (though not necessarily acceptable) inputs such as:

$$BBB\cdots,\ BcB\cdots,\ BababB\cdots,\ BacaB\cdots,\ldots$$

5 Tutorial problems

Designing some Turing machines – it's important that you understand these, so if you're stuck – ask! For the simpler cases try to get every detail right – for the more complex one think first about the "big picture" of how the machine works before sweating out the details (but mainly, try to convince yourself that it's possible!)

I find using Anthony Morphett's Turing machine simulator (along with a text-editor to copy-paste from) is a good way to do/debug these.

Design Turing machines to accept the following languages (over $\{a, b\}$ unless obviously otherwise), either by final state or by halting.

1. $(\mathbf{a} \cup \mathbf{b})^*\mathbf{ab}(\mathbf{a} \cup \mathbf{b})^*$
2. $\mathbf{a}^*\mathbf{b}^*$
3. The set of all strings containing an even number of a's and an even number of b's.
4. The set of all palindromes (strings that read the same forwards as backwards).
5. $\{a^n b^n c^n \mid n \geq 0\}$.
6. $\{a^{2^n} \mid n \geq 0\}$.

As a more general problem think of how you could use a queue machine as described above to simulate the computation of any PDA. If you convince yourself that this is possible then you will know that queue machines are strictly more powerful than PDAs. What do you think the relationship in power between queue machines and TMs might be? What about machines that have two stacks as their memory (operations can push and pop from either or both)?