1 Introduction

We will break the shackles of "accepting languages" and think about some more general ways that TMs can be used, and what different kinds of acceptance criteria we might have.

2 TMs as function computers

Instead of using TMs as language acceptors we can use them to compute functions. That is, we could say that a TM computes the function f (with domain some subset of Σ^*) if, on input w from its domain it halts with the contents of the tape being $f(w)$ (or possibly $w\#f(w)$). For input not in the domain of f we would generally require that the machine should crash or run forever (signifying the fact that f is not defined there).

As an example consider the problem "compute 2^k". In other words we want to design a TM that on input a^k produces an output tape of $a^k\#b^{2^k}$. In designing this machine it's useful (as usual) to think in slightly higher level terms first:

- Find the end of input and replace it with $\#$, followed by b
- For each a in the input, double the number of b's past the $\#$

How does a "double the b's" module work? Take each b, replace it by X, and add an X at the end. When no more b's are left, change all the X's to b's.

Let's see if that is enough to put together a transition table:

State	Read	Write	Move	New State
START	B	B	R	FIND-END
FIND-END	a	a	R	FIND-END
	B	$\#$	R	ADD-b
ADD-b	B	b	L	FIND-a
FIND-a	B	B	R	HALT-ACCEPT
	a	A	R	START-DOUBLE
	b	b	L	FIND-a
	A	A	L	FIND-a
	$\#$	$\#$	L	FIND-a
START-DOUBLE	A	A	R	START-DOUBLE
	$\#$	$\#$	R	FIND-b
FIND-b	b	X	R	ADD-X
	X	X	R	FIND-b
	B	B	L	CHANGE-X-TO-b
ADD-X	b	b	R	ADD-X
	X	X	R	ADD-X
	B	X	L	FIND-$\#$
CHANGE-X-TO-b	X	b	L	CHANGE-X-TO-b
	$\#$	$\#$	L	FIND-a
FIND-$\#$	b	b	L	FIND-$\#$
	X	X	L	FIND-$\#$
	$\#$	$\#$	R	FIND-b

This simple version of the machine halts in some abnormal way on 'bad' input, but we can't really recognize that (unless we agree that only HALT-ACCEPT is an accepting state). However, we could fix this by sending it into a loop or crash whenever some unexpected input was received. For instance, if in FIND-END we see something other than an a or B we could add a transition to CRASH. The transitions from CRASH are 'whatever you see leave it alone and move left, staying in CRASH'. This will eventually drive the head off the left hand end of the tape resulting in a crash. If we prefer to leave the computer looping we could move right instead (forever).

3 Accepting variations

The machines we've looked at so far accept by halting in some designated accepting state(s). Is this really necessary? Can we dispense with the need for specifying accepting states? Yes we can.

Theorem 3.1. *The collections of languages accepted by Turing machines with designated final states and those accepted by Turing machines by halting are the same.*

One direction is clear since if we have a language accepted by halting we can simply designate all the states as accepting states. The problem is to show how to transform a machine in which "abnormal" i.e. non-accepting halts might occur into one which crashes or loops instead. But, we already saw how to do this in the example above. Simply add a new state CRASH which causes leftward movement (and remains in CRASH) on any input. Then, take any non-defined transition which represents an abnormal halt, and replace it by one which enters the CRASH state.

4 Testing primality

To begin to establish the idea that perhaps TMs are rather powerful computing device, let's ask if we can recognise the language PRIME consisting of all strings a^p where p is a prime number. The purpose of this example is not so much to show that it can be done, but rather to illustrate that the time has definitely come to extend the allowed operation of our 'mechanical computing devices' to make this sort of thing easier!

The basic idea of testing a number n for primality is: 'for each possible factor, k, determine whether or not k is a divisor of n'. If no such test produces a factor then the number is prime.

We can see how to do this using some basic ideas that we've already explored. Start with an input tape containing a^n. First, transform that to $a^n\#X^2$. The idea in general now is starting from a tape in the form $a^n\#X^k$ we're going to check whether or not k is a factor of n. We do this by 'pairing off a's (changing them to A's) and X's (changing them to Y's). So, while there are still a's and X's on the tape change an X to Y and an a to A. When we can't do this any more:

- If there are no a's left, but still some X's, then n was not a multiple of k. Change all the A's back to a's, all the Y's to X's and add one more X. Repeat.
- If there are no X's left, but still some a's, then change all the Y's back to X's and start pairing X's with a's again.
- If there are neither X's nor a's left, then we found a factor. Check whether this factor is n or something smaller. In the former case, accept, and in the latter reject.

5 Tutorial problems

Design Turing machines to accept the following languages either by final state or by halting. We will generally use the symbol # as a divider between parts of the input. Remember that you may use (a finite set of) additional symbols to write on the tape if that is convenient.

1. The language consisting of all strings of the form $w\#c$ where $w \in \{a, b\}^*$ is non-empty, and c is the first character of w.
2. The language consisting of all strings of the form $w\#v$ where the length of v is greater than the length of w.
3. Modify the design of the 'compute 2^k' machine into a 'compute k^2' machine.
4. Like ordinary computers, Turing machines can also be thought of as machines that transform input into output. Design a Turing machine which, on input $w \in \{a, b\}^*$ halts with the tape showing $a^k b^l$, where k is the number of a's in w, and l is the number of b's (i.e., it sorts w.)
5. (*) Write a machine that converts unary to binary. That is, on input Ba^k it should halt and the final tape state should be $Ba^k\#k_2$, where k_2 is the representation of k in binary. Specifically, $Baaaaa$ should produce $Baaaaa\#101$.

There are a number of Turing machine simulators available on the internet (as google knows). As of April 2020, two that look pretty good are the Turing machine simulator by Anthony Morphett, and the Online Turing Machine Simulator by Martin Ugarte. In particular, the former has a syntax that's pretty close to what I've been using in the examples.