CS 341 Homework 18
Computing with Turing Machines

1. Present Turing machines that decide the following languages over {a, b}:

(a) $\varnothing$

(b) {ε}

(c) {a}

(d) {a}*

2. Consider the simple (regular, in fact) language L = {w ∈ {a,b}* : |w| is even}

(a) Give a Turing machine that decides L.

(b) Give a Turing machine that semidecides L.

3. Give a Turing machine (in our abbreviated notation) that accepts L = $\{a^n b^m a^n : m > n\}$

4. Give a Turing machine (in our abbreviated notation) that accepts L = {ww : w ∈ {a, b}*}

5. Give a Turing machine (in our abbreviated notation) that computes the following function from strings in {a, b}* to strings in {a, b}* : f(w) = ww^R.

6. Give a Turing machine that computes the function f: {a,b,c}* → N (the integers), where f(w) = the number of a's (in unary) in w.

7. Let w and x be any two positive integers encoded in unary. Show a Turing machine M that computes

f(w, x) = w + x.

Represent the input to M as

◊❑w;x❑

8. Two's complement form provides a way to represent both positive and negative binary integers. Suppose that the number of bits allocated to each number is k (generally the word size). Then each positive integer is represented simply as its binary encoding, with leading zeros. Each negative integer n is represented as the result of subtracting |n| from 2^k, where k is the number of bits to be used in the representation. Given a fixed k, it is possible to represent any integer n if $-2^{k-1} \le n \le 2^{k-1} - 1$. The high order digit of each number indicates its sign: it is zero for positive integers and 1 for negative integers.

Examples, for k = 4:

0 = 0000, 1 = 0001, 2 = 0010, 3 = 0011, 4 = 0100, 5 = 0101, 6 = 0110, 7 = 0111

-1 = 1111, -2 = 1110, -3 = 1101, -4 = 1100, -5 = 1011, -6 = 1010, -7 = 1001, -8 = 1000

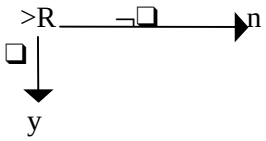
Since Turing machines don't have fixed length words, we'd like to be able to represent any integer. We will represent positive integers with a single leading 0. We will represent each negative integer n as the result of subtracting n from 2^{i+1}, where i is the smallest value such that $2^i \ge |n|$. For example, -65 will be represented as 1111111, since 2^7 (128) ≥ 65, so we subtract 65 (01000001 in binary) from 2^8 (in binary, 100000000). We need the extra digit (i.e., we subtract from 2^{i+1} rather than from 2^i) because, in order for a positive number to be interpreted as positive, it must have a leading 0, thus consuming an extra digit.

Let w be any integer encoded in two's complement form. Show a Turing machine that computes f(w) = -w.

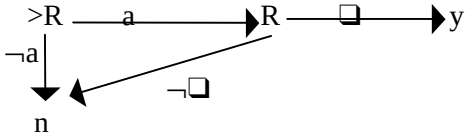
Solutions

1 (a) We should reject everything, since no strings are in the language.

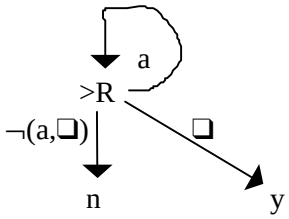
(b) Other than the left boundary symbol, the tape should be blank: ◊❑❑❑



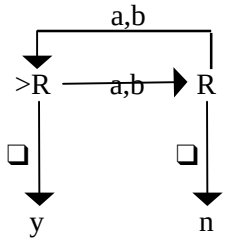
(c) Just the single string a: ◊❑a❑



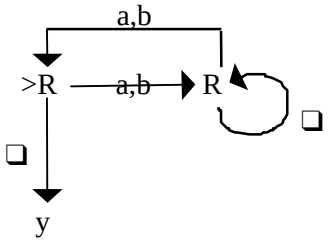
(d) Any number of a's:



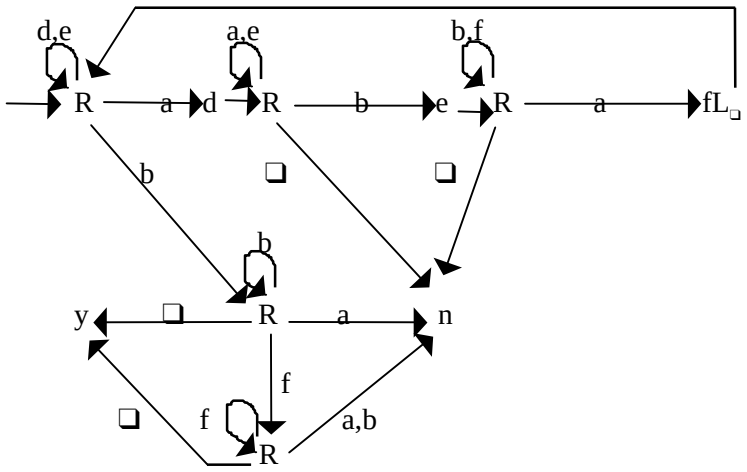
2. (a)



(b)



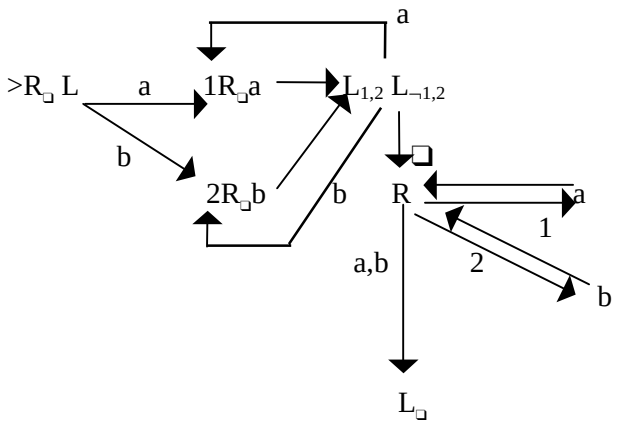
3. The idea is to make a sequence of passes over the input. On each pass, we mark off (with d, e, and f) a matching a, b, and a. This corresponds to the top row of the machine shown here. When there are no matching groups left, then we accept if there is nothing left or if there are b's in the middle. If there is anything else, we reject. It turns out that a great deal of this machine is essentially error checking. We get to the R on the second row as soon as we find the first "extra" b. We can loop in it as long as we find b's. If we find a's we reject. If we find a blank, then the string had just b's, which is okay, so we accept. Once we find an f, we have to go to the separate state R on the third row to skip over the f's and make sure we get to the final blank without either any more b's or any more a's.



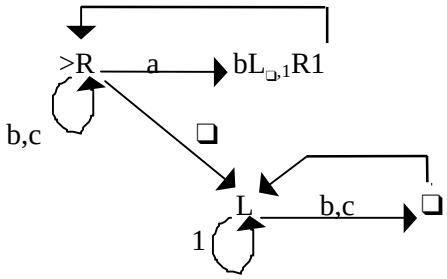
4. The hard part here is that we don't know where the middle of the string is. So we don't know where the boundary between the first occurrence of w ends and the second begins. We can break this problem into three subroutines, which will be executed in order:
(1) Find the middle and mark it. If there's a lone character in the middle (i.e., the length of the input string isn't even), then reject immediately.
(2) Bounce back and forth between the beginning of the first w and the beginning of the second, marking off characters if they match and rejecting if they don't.
(3) If we get to the end of the w's and everything has matched, accept.

Let's say a little more about step (1). We need to put a marker in the middle of the string. The easiest thing to do is to make a double marker. We'll use ##. That way, we can start at both ends (bouncing back and forth), moving a marker one character toward the middle at each step. For example, if we start with the tape ◊❑aabbaabb❑❑❑, after one mark off step we'll have ◊❑a#abbaab#b❑❑❑, then ◊❑aa#bbaa#bb❑❑❑, and finally ◊❑aabb##aabb❑❑❑. So first we shift the whole input string two squares to the right on the tape to make room for the two markers. Then we bounce back and forth, moving the two markers toward the center. If they meet, we've got an even length string and we can continue. If they don't, we reject right away.

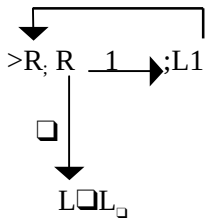
5. The idea is to work from the middle. We'll scan right to the rightmost character of w (which we find by scanning for the first blank, then backing up (left) one square. We'll rewrite it so we know not to deal with it again (a's will become 1's; b's will become 2's.) Then we move right and copy it. Now if we scan back left past any 1's or 2's, we'll find the next rightmost character of w. We rewrite it to a 1 or a 2, then scan right to a blank and copy it. We keep this up until, when we scan back to the left, past any 1's or 2's, we hit a blank. That means we've copied everything. Now we scan to the right, replacing 1's by a's and 2's by b's. Finally, we scan back to the left to position the read head to the left of w.



6. The idea here is that we need to write a 1 for every a and we can throw away b's and c's. We want the 1's to end up at the left end of the string, so then all we have to do to clean up at the end is erase all the b's and c's to the right of the area with the 1's. So, we'll start at the left edge of the string. We'll skip over b's and c's until we get to an a. At that point, rewrite it as b so we don't count it again. Then (remembering it in the state) scan left until we get to the blank (if this is the first 1) or we get to a 1. In either case, move one square to the right and write a 1. We are thus overwriting a b or a c, but we don't care. We're going to throw them away anyway. Now start again scanning to the right looking for an a. At some point, we'll come to the end of string blank instead. At that point, just travel leftward, rewriting all the b's and c's to blank, then cross the 1's and land on the blank at the left of the string.



7. All we have to do is to concatenate the two strings. So shift the second one left one square, covering up the semicolon.



8. Do a couple of examples of the conversion to see what's going on. What you'll observe is that we want to scan from the right. Initially, we may see some zeros, and those will stay as zeros. If we ever see a 1, then we rewrite the first one as a 1. After that, we're dealing with borrowing, so we swap all digits: every zero becomes a one and every one becomes a zero, until we hit the blank at the end of the string and halt.