

Test 1: Solutions and Comments

CS 2MJ3 Fall 2009

Nick James

October 28, 2009

1 Question 1

See Prof. Soltys' solutions for the DFA diagram. Almost everyone did this question perfectly. The only attempts which were a little disturbing used the Pumping Lemma and began with the statement, "Assume L is a regular language." I do hope it is clear—now you're not racing to finish a test—that if you're asked to prove that a statement, P is true, you are instantly *doomed* if you begin with, "Assume P is true." No point in writing anything beyond that.

Slightly less disturbing (but no more correct) were attempts to use the Pumping Lemma without the assumption that L is regular. This doesn't work because the Pumping Lemma defines a rather complicated property (there is an n such that every $z \in L$ longer than n can be blah, blah, blah...) and says *only* that every regular language satisfies that property. In other words, if L is regular, it satisfies that property. If it isn't regular, it may or may not satisfy the property. So if you take a language and show that it satisfies the Pumping Lemma (rather than contradicting it), that doesn't actually give us any meaningful information about it; it STILL may be non-regular¹.

2 Question 2

The language, $L_{eq} = \{x \in \{0,1\}^* : x \text{ has the same number of 0s as 1s}\}$, is not regular

Proof. Suppose L_{eq} is regular. Let $n > 0$ and let $z = 0^n 1^n$. Let $u, v, w \in \{0,1\}^*$ such that $z = uvw$, $|uv| \leq n$, and $v \neq \varepsilon$. Then v must consist entirely of zeros since the first n symbols of z are all zeros and the length of the uv part of z is no longer than n . Thus, $\exists m > 0$ such that $v = 0^m$. Hence $uv^2w = 0^{n+m}1^n \notin L_{eq}$. But this contradicts the Pumping Lemma for Regular Languages. Therefore, L_{eq} cannot be regular. □

To anyone who didn't get full marks on this question, PLEASE see my little tutorial on using the Pumping Lemma at,

¹A natural follow-up question, however, would be a request for such a counterexample. Unfortunately, none come to mind immediately, but it's an interesting question. Can you think of a language which is non-regular but cannot be proven so via the Pumping Lemma?

http://www.cas.mcmaster.ca/~soltys/cs2mj3-f09/a1_comments.pdf

If you don't want to read all that, just look at the step-by-step summary on the last page.

3 Question 3

I described one solution in detail in my document on PDAs (see #2 and just replace the a 's and b 's with 0's and 1's):

<http://www.cas.mcmaster.ca/~soltys/cs2mj3-f09/pdas.pdf>

Some of you (including Prof. Soltys) wrote a solution that is elegant and easy to understand, but it has one subtle flaw. First, let's review the solution in question:

1. *If the stack is empty*, push the current symbol on the stack.
2. If the current symbol is the same as the one on the top of stack, push another copy of that symbol on the stack.
3. If the current symbol is different from the one on the top of stack, (pop and) discard the symbol on the top of the stack.
4. The PDA accepts the string iff the stack is empty when the string has been read completely.

The problem is the condition in Step 1: PDAs cannot “detect” whether the stack is empty while reading a string because of the way the PDA model is designed. That design limitation is inherent in both acceptance criteria (*empty stack* and *final state*), but the text in the course doesn't begin to cover the level of detail necessary to see this (so no marks off for having written this solution).

This is an easy problem to circumvent, but before we see how that's done, I can sense that some of you have seen an apparent contradiction in what I just said and you're clamouring to get a word in edgewise. If a PDA can't detect whether its stack is empty, then—uh... huh, huh... just HOW is a PDA supposed to *accept by empty stack* then, Huh? Hmm? Care to enlighten us about that, Mr. “TA” (if that *is* your real name)?

Yes, good catch. You're obviously very bright for having spotted it! The answer is that acceptance is not a state transition, it's an interpretation of the state of the entire PDA. It's easy to define acceptance criteria that involve the contents of the stack. If we try to design a state transition that can take place only when the stack is empty, however, the only conceivable way to do so would be like this:

$$(p, a, \varepsilon) \rightarrow (q, A)$$

But remember that our PDAs have to be nondeterministic! So a PDA with a state transition like this could use this transition whenever it's in state p and reads an a —regardless of what is on the stack. The ε there just means it doesn't pop anything from the stack when it uses that transition. It does NOT mean that the transition can be taken only when the stack is empty.

As I said, this is an easy problem to circumvent, though, and it's a trick that's used often: start by pushing a special symbol that isn't used for anything else on the (initially empty) stack. A PDA can't check whether its stack is empty, but it *can* check whether the top symbol is something specific.

Let's use the symbol, \perp for this purpose and modify the approach above (which was *almost* perfect):

1. Push \perp on the stack.
2. If the top symbol on the stack is \perp , push the current symbol on the stack.
3. If the current symbol is the same as the one on the top of stack, push another copy of that symbol on the stack.
4. If the current symbol is different from the one on the top of stack, (pop and) discard the symbol on the top of the stack.
5. The PDA accepts the string (either by entering an accepting state or by emptying the stack) iff the symbol on the top of the stack is \perp when the string has been read completely.

So that works fine, and if you're happy, go ahead and move on. The reason you may not be happy with this solution is that it's actually impossible to build using Fernández's PDA model. What's the problem? Well, the problem is that if you write out the actual state transitions, the ones described by 2 and 3 are actually pushing TWO symbols on the stack (again, see my solution in the PDA tutorial I wrote on the web site).

Fernández's PDA can't do that because the codomain of her δ function is $\mathcal{P}(Q \times (\Gamma \cup \{\varepsilon\}))$. The $(\Gamma \cup \{\varepsilon\})$ part means that it can push AT MOST one symbol on the stack during each state transition. Similarly, you can pop at most one symbol from the stack during state transitions. So to build up the stack in Fernández's model, you cannot look at the stack when you add something to it. As soon as you look, you pop. And then all you can do is replace the thing you popped (with either it or something else).

This means we have a much greater reliance on states. Here is my Fernández-style solution:

$$M = (\{p, q, r\}, \{0, 1\}, \{0, 1, \perp\}, \delta, p, \{r\})$$

where δ consists of the following transitions:

$$\begin{array}{ll} (p, \varepsilon, \varepsilon) \rightarrow (q, \perp) & (q, 1, 0) \rightarrow (q, \varepsilon) \\ (q, 0, \varepsilon) \rightarrow (q, 0) & (q, 0, 1) \rightarrow (q, \varepsilon) \\ (q, 1, \varepsilon) \rightarrow (q, 1) & (q, \varepsilon, \perp) \rightarrow (r, \varepsilon) \end{array}$$

We begin in state p and can't do ANYTHING other than push \perp on the stack using the first transition. This is why states are so essential in Fernández's model. Typically you can get away with only one state in Kozen's model (the previous solution), but the limitation of being able to push at most one stack symbol is incredibly restrictive.

The two transitions under that one can be taken whenever we read either 0 or 1. In such cases we *cannot* examine the stack because to do so requires popping from it, and in Fernández’s model, once we pop, we can only replace the top symbol; we can’t make the stack bigger. If we could push more than one symbol, these two transitions would be,

$$(q, 0, 0) \rightarrow (q, 00) \quad (q, 1, 1) \rightarrow (q, 11)$$

So we must rely on nondeterminism. In a sense, this solution is less deterministic than the previous one.

The next two transitions (top-right) are simply copied from the previous solution, and from these you can see what I mean when I say this solution is “less deterministic.” In our Kozen-style solution the middle four transitions were completely deterministic (read 0, stack 0: push 0; read 1, stack 1: push 1; read 0, stack 1: pop; read 1, stack 0: pop). In this solution, whenever we read a 0, we can either blindly push 0 on the stack (even though the top may be a 1—which would be a “mistake” in the sense that this attempt won’t end in acceptance and the machine will have to try again), or we can pop as long as the top of the stack is a 1.

The last transition can be taken whenever \perp is on the stack—even if there is more of the string left to read. This is fine since taking the transition leaves the machine in state r from which it can do nothing. In order to accept a string, it has to both be in state, r , AND have read the entire string. If there’s anything left to read when the machine is in state, r , well, too late now, Machine—shoulda turned left at that last exit like I told you to, but did you listen?? *Ohhhh noooo*, you know *everything!* *You’re smarter than Deep Blue!* (Ugh... I swear it’s a miracle of nature you can even keep your stack in LIFO order. I think your mother must have dropped you on your start state when you were being initialized.)

4 Question 4

There are three stages involved with simulating a Turing machine, M_0 , with another Turing machine, M_1 . First, we must have a systematic way (an algorithm, if you will) to copy the contents of M_0 ’s tape to M_1 ’s tape. Second, we must have a systematic way to translate the state transitions of M_0 for M_1 . Finally, we must have a systematic way to copy the contents of M_1 ’s tape back to M_0 ’s tape. With those three things in place, we have a complete simulation. It’s almost as though M_1 is doing M_0 ’s homework.

There are myriad ways to do this particular simulation, but two inherently different approaches (each of which has infinitely many variations) occurred to me upon reading the question. I’m sure there are more, but we don’t have all day here. Only one student used a solution like the one Prof. Soltys offered, but that solution is actually much easier (there’s much less maintenance in involved) than the other one.

No matter which solution you choose, one thing we must demand is that there must be a unique “blank” symbol that fills the unoccupied portions of the tape

and may not appear anywhere within the area of input data. Essentially, we require that the region of the tape occupied by the input data be *connected*².

4.1 Diagonal Solution

In Prof. Soltys' solution, we enumerate the cells on the 2D tape like this:

```

1  2  4  7  11 16 ...
3  5  8  12 17
6  9  13 18
10 14 19
15 20
21
⋮

```

That's just one of many possible enumerations, and it's one of the simplest, so we'll go with that one. This enumeration takes care of Step 1 and Step 3: copying M_0 's tape to M_1 's tape, and then copying the resulting tape back (in the event that M_1 manages to halt). There are two ways to do this: (a) copy the cells exactly as shown above:

(a)

(1,1)	(1,2)	(2,1)	(1,3)	(2,2)	...
-------	-------	-------	-------	-------	-----

or (b) copy the cells as shown, inserting a delimiter after each diagonal line of cells copied:

(b)

(1,1)	△	(1,2)	(2,1)	△	(1,3)	...
-------	---	-------	-------	---	-------	-----

We'll look at approach (a). If you'd like to know more about how (b) works, please ask. I'd be happy to show you, but I figure this document is already getting pretty long and I'd like to wrap it up soon.

There are no limitations on the alphabets, the states used, or the transitions, so the only thing that must be translated here are the directional instructions and we need a few variables to handle that: one to keep track of our current row, one for the current column, and one for counting moves. These variables can be maintained in any number of ways. Probably the most straightforward is to use a separate tape for each one. We already know that Turing machines with 4 tapes are equivalent to those with only 1 tape, so let's use a 4-tape machine to make it easy. Then we can write those three variables using unary notation (i.e. $1 = 1$, $2 = 11$, $3 = 111$, $4 = 1111$, etc.).

When M_0 wants to move left, M_1 moves left by $row + column - 2$ cells. For example, cell (3,3) is at position 13 in the linear tape. The cell to its left, (3,2), is at position 9. So we need to move left by $13 - 9 = 4$ cells, and $row + column - 2 = 3 + 3 - 2 = 4$.

Similarly, when M_0 needs to move right, M_1 moves right by $row + column - 1$ cells. To move "up," M_1 moves left by $row + column - 1$ cells, and to move

²Depending on how we define a "simulation," I suppose one could argue that the original tape could be consulted on the fly, and then such a requirement would not be necessary.

“down,” M_1 must move $row + column$ cells right. After performing any of these movements, the row or column variable must be incremented or decremented accordingly.

We need the “counting” variable in order to implement instructions like “move right $row + column$ times.” That particular instruction, for example, could be done by temporarily going into an “add row + column” state. In that state, we move heads 2 and 3 (the row and column variables) along with 4 (the counter) all the way to the left, then copy the 1s from head 2 to tape 4. Move head 4 one space right (off the last “1” written), then copy the 1s from head 3 to tape 4. Move head 4 right, and write an *end* delimiter (e.g. \triangle).

Now tape 4 contains $row + column$ 1s. At this point, we enter a “move right *counter* number of times” state. Tape 4 is rewound to the beginning. While tape 4’s symbol is “1,” move tapes 4 and 1 to the right.

WHAT A PAIN IN THE SOMETHING–OR–OTHER! Am I right? Yeah. Still, I wanted to show you some details from a fully developed simulation. So, let’s have a look at the kind of solution the large majority of the class used. If you thought the diagonal thing was complicated and tedious, this one is absolutely horrifying if you try to implement every detail. So we won’t try to implement every detail; we’ll just look at the main pieces.

4.2 Row-by-row solution

While the 2D tape itself is infinite, the initial data on it is not. Thus, each row (or column, if you prefer) has an end and we can copy them, one by one, onto the linear tape, with or without delimiters separating them. The best approach is to treat the input data as if it occupies a rectangle: find the longest row and column and treat those as defining the region of input data.

The reason for having all the rows be the same length are the U and D commands. As in the previous solution, we need to maintain some variables, but this time the variables we maintain are the number of rows and the number of columns. To move “down,” (or “up”) we move *number of cols* to the right (or left). Add 1 extra move if using row delimiters. One thing we need to watch for is moving down when we’re already on the last row. In such an event, we must be sure to increase the row count.

Moving left actually requires no translation whatsoever. M_0 moves left: M_1 moves left. Simple as that. M_0 will be already set up to deal with the left edge of its tape. A typical approach would be for M_0 to put a special sentry symbol (e.g. “┌”) at the leftmost cell of each row in use and check for that symbol whenever it moves left. Any such instructions and initialization will be carried over to M_1 automatically.

Moving right, however, is much more complicated, and this is where many marks were lost—probably the majority, in fact. M_0 can move right as far as it wants: the tape just keeps going. But M_1 has taken these infinite rows and made them finite (remember we had to find the longest row and use that length). If M_0

moves to the right beyond the length of the (initially) longest row, M_1 is in trouble: it will accidentally move into the beginning of the next row down. So for every move right, M_1 must check that this move will leave it still within the bounds of the current row. If not, it must ENLARGE EVERY ROW by one cell. To do so, it must move to the end of the tape and copy the last n cells m positions forward (where m is the number of rows³ and n is the number of columns). Then the next n cells are copied $m - 1$ positions forward. The very first n cells are left in place at the beginning of the tape. After all this, the tape head can finally return to the position at which it was intended to be when the "move right" command was first encountered.

I felt the enlargement of the rows was one of the most important features of this kind of simulation because without it, M_1 has only a finite tape. And a "Turing machine" with a finite tape is nothing more than a DFA in disguise. I also felt that it was not a completely unreasonable detail to expect in a solution—even with the very little time available for the test. So I attributed a fairly nontrivial amount of marks to this aspect of the simulation. The little mechanical details could be somewhat off, but that Turing machine *definitely* has to be prepared to lengthen its rows dynamically.

³This is why we had to maintain the number of rows! See two paragraphs before this.