CS 341 Homework 13
Pushdown Automata

1. Consider the pushdown automaton M = (K, Σ, Γ, Δ, s, F), where

K = {s, f},
F = {f},
Σ = {a, b},
Γ = {a},
Δ = {((s, a, ε), (s, a)), ((s, b, ε), (s, a)), ((s, a, ε), (f, ε)), ((f, a, a), (f, ε)), ((f, b, a), (f, ε))}.

(a) Trace all possible sequences of transitions of M on input aba.
(b) Show that aba, aa, abb ∉ L(M), but baa, bab, baaaa ∈ L(M).
(c) Describe L(M) in English.

2. Construct pushdown automata that accept each of the following:

(a) L = the language generated by the grammar G = (V, Σ, R, S), where

V = {S, (,), [,]},
Σ = {(,), [,]},
R = { $S \rightarrow \varepsilon$,
$S \rightarrow SS$,
$S \rightarrow [S]$,
$S \rightarrow (S)$}.

(b) L = $\{a^m b^n : m \leq n \leq 2m\}$.
(c) L = $\{w \in \{a, b\}^* : w = w^R\}$.
(d) L = {w ∈ {a, b}* : w has equal numbers of a's and b's}.
(e) L = {w ∈ {a, b}* : w has twice as many a's as b's}.
(f) L = $\{a^m b^n : m \geq n\}$
(g) L = {uawb: u and w ∈ {a, b}* and |u| = |w|}

3. Consider the following language : L = {w^Rw" : w ∈ {a, b}* and w" indicates w with each occurrence of a replaced by b, and vice versa}. In Homework 12, problem 5, you wrote a context-free grammar for L. Now give a PDA M that accepts L and trace a computation that shows that aababb ∈ L.

4. Construct a context-free grammar for the language of problem 2(b): L = ($\{a^m b^n : m \leq n \leq 2m\}$).

Solutions

1. (a) There are three possible computations of M on aba:

(s, aba, ε) |- (s, ba, a) |- (s, a, aa) |- (s, ε, aaa)
(s, aba, ε) |- (s, ba, a) |- (s, a, aa) |- (f, ε, aa)
(s, aba, ε) |- (f, ba, ε)

None of these is an accepting configuration.

(b) This is done by tracing the computation of M on each of the strings, as shown in (a).
(c) L(M) is the set of strings whose middle symbol is a. In other words,

L(M) = {xay ∈ {a, b}* : |x| = |y|}.

2. (a) Notice that the square brackets and the parentheses must be properly nested. So the strategy will be to push the open brackets and parens and pop them against matching close brackets and parens as they are read in. We only need one state, since all the counting will be done on the stack. Since ε ∈ L, the start state can be final. Thus we have M = ({s}, {(,), [,]}, {(, [}, Δ, s {s}), where (sorry about the confusing use of parentheses both as part of the notation and as symbols in the language):

Δ = {((s, (, ε), (s, ()), /* push (*/
((s, [, ε), (s, [)), /* push [*/
((s,), (),(s, ε)), /* if the input character is) and the top of the stack is (, they match */
((s,], [), (s, ε))} /* same for matching square brackets */

If we run out of input and stack at the same time, we'll accept.

(b) Clearly we need to use the stack to count the a's and then compare that count to the b's as they're read in. The complication here is that for every a, there may be either one or two b's. So we'll need nondeterminism. Every string in L has two regions, the a region followed by the b region (okay, they're hard to tell apart in the case of ε, but trivially, this even true there). So we need a machine with at least two states.

There are two ways we could deal with the fact that, each time we see an a, we don't know whether it will be matched by one b or two. The first is to push either one or two characters onto the stack. In this case, once we get to the b's, we'll pop one character for every b we see. A nondeterministic machine that follows all paths of combinations of one or two pushed characters will find at least one match for every string in L. The alternative is to push a single character for every a and then to get nondeterministic when we're processing the b's: For each stack character, we accept either one b or two. Here's a PDA that takes the second approach. You may want to try writing one that does it the other way. This machine actually needs three states since it needs two states for processing b's to allow for the case where two b's are read but only a single a is popped. So M = ({s, f, g}, {a, b}, {a}, Δ, s, {f, g}), where

Δ = {((s, a, ε), (s, a)), /* Read an a and push one onto the stack */
((s, ε, ε), (f, ε)), /* Jump to the b reading state */
((f, b, a), (f, ε)), /* Read a single b and pop an a */
((f, b, a), (g, ε)),/* Read a single b and pop an a but get ready to read a second one */
((g, b, ε), (f, ε))}. /* Read a b without popping an a */

(c) A PDA that accepts $\{w : w = w^R\}$ is just a variation of the PDA that accepts $\{ww^R\}$ (which you'll find in Lecture Notes 14). You can modify that PDA by adding two transitions ((s, a, ε), (f, ε)) and ((s, b, ε), (f, ε)), which have the effect of making odd length palindromes accepted by skipping their middle symbol.

(d) We've got another counting problem here, but now order doesn't matter -- just numbers. Notice that with languages of this sort, it's almost always easier to build a PDA than a grammar, since PDAs do a good job of using their stack for counting, while grammars have to consider the order in which characters are generated. If you don't believe this, try writing a grammar for this language.

Consider any string w in L. At any point in the processing of w, one of three conditions holds: (1) We have seen equal numbers of a's and b's; (2) We have seen more a's than b's; or (3) We have seen more b's than a's. What we need to do is to use the stack to count whichever character we've seen more of. Then, when we see the corresponding instance of the other character we can "cancel" them by consuming the input and popping off the stack. So if we've seen an excess of a's, there will be a's on the stack. If we've seen an excess of b's there will be b's on the stack. If we're even, the stack will be empty. Then, whatever character comes next, we'll start counting it by putting it on the stack. In fact, we can build our PDA so that the following invariant is always maintained:

(Number of a's read so far) - (Number of b's read so far)
=
(Number of a's on stack) - (Number of b's on stack)

Notice that $w \in L$ if and only if, when we finish reading w,

[(Number of a's read so far) - (Number of b's read so far)] = 0.

So, if we build M so that it maintains this invariant, then we know that if M consumes w and ends with its stack empty, it has seen a string in L. And, if its stack isn't empty, then it hasn't seen a string in L.

To make this work, we need to be able to tell if the stack is empty, since that's the only case where we might consider pushing either a or b. Recall that we can't do that just by writing ε as the stack character, since that always matches, even if the stack is not empty. So we'll start by pushing a special character # onto the bottom of the stack. We can then check to see if the stack is empty by seeing if # is on top. We can do all the real work in our PDA in a single state. But, because we're using the bottom of stack symbol #, we need two additional states: the start state, in which we do nothing except push # and move to the working state, and the final state, which we get to once we've popped # and can then do nothing else. Considering all these issues, we get M = ({s, q, f}, {a, b}, {#, a, b}, Δ, s, {f}), where

```
Δ = {((s, ε, ε), (q, #)),        /* push # and move to the working state q  */
     ((q, a, #), (q, a#)),       /* the stack is empty and we've got an a, so push it  */
     ((q, a, a), (q, aa)),       /* the stack is counting a's and we've got another one so push it  */
     ((q, b, a), (q, ε)),        /* the stack is counting a's and we've got b, so cancel a and b  */
     ((q, b, #), (q, b#)),       /* the stack is empty and we've got a b, so push it  */
     ((q, b, b), (q, bb)),       /* the stack is counting b's and we've got another one so push it  */
     ((q, a, b), (q, ε)),        /* the stack is counting b's and we've got a, so cancel b and a  */
     ((q, ε, #), (f, ε))}.       /* the stack is empty of a's and b's.  Pop the # and quit.  */
```

To convince yourself that M does the job, you should show that M does in fact maintain the invariant we stated above.

The only nondeterminism in this machine involves the last transition in which we guess that we're at the end of the input. There is an alternative way to solve this problem in which we don't bother with the bottom of stack symbol #. Instead, we substitute a lot of nondeterminism and we sometimes push a's on top of b's, and so forth. Most of those paths will end up in dead ends. The machine has fewer states but is harder to analyze. Try to construct it if you like.

(e) This one is similar to (d) except that there are two a's for every b. Recall the two techniques for matching two to one that we discussed in our solution to (b). This time, though, we do know that there are always two a's to every b. We don't need nondeterminism to allow for either one *or* two. But, because we no longer know that all the a's come first, we do need to consider what to do in the two cases: (1) We're counting b's on the stack; and (2) We're counting a's on the stack. If we're counting b's, let's take the approach in which we push two b's every time we see one. Then, when we go to cancel a's, we can just pop one b for each a. If we see twice as many a's as b's, we'll end up with an empty stack. Now what if we're counting a's? We'll push one a for every one we see. When we see b, we pop two a's. The only special case we need to consider arises in strings such as "aba", where we'll only have seen a single a at the point at which we see the b. What we need to do is to switch from counting a's to counting b's, since the b counts twice. Thus the invariant that we want to maintain is now

(Number of a's read so far) - 2*(Number of b's read so far)
=
(Number of a's on stack) - (Number of b's on stack)

We can do all this with M = ({s, q, f}, {a, b}, {#, a, b}, Δ, s, {f}), where

```
Δ = {((s, ε, ε), (q, #)),        /* push # and move to the working state q  */
     ((q, a, #), (q, a#)),       /* the stack is empty and we've got an a, so push it  */
     ((q, a, a), (q, aa)),       /* the stack is counting a's and we've got another one so push it  */
     ((q, b, aa), (q, ε)),       /* the stack is counting a's and we've got b, so cancel aa and b  */
     ((q, b, a#), (q, b#)),      /* the stack contains a single a and we've got b, so cancel the a and b
                                    and start counting b's, since we have a shortage of one a   */
     ((q, b, #), (q, bb#)),      /* the stack is empty and we've got a b, so push two b's  */
     ((q, b, b), (q, bbb)),      /* the stack is counting b's and we've got another one so push two  */
     ((q, a, b), (q, ε)),        /* the stack is counting b's and we've got a, so cancel b and a  */
     ((q, ε, #), (f, ε))}.       /* the stack is empty of a's and b's.  Pop the # and quit.  */
```

You should show that M preserves the invariant above.

(f) The idea here is to push each a as we see it. Then, on the first b, move to a second state and pop an a for each b. If we get to the end of the string and either the stack is empty (m = n) or there are still a's on the stack (m > n) then we accept. If we find a b and there's no a to pop, then there will be no action and so we'll fail. This machine is nondeterministic for two reasons. The first is that, in case there are no b's, we must be able to guess that we've reached the end of the input string and go pop all the a's off the stack. The second is that if there were b's but fewer than the number of a's, then we must guess that we've reached the end of the input string and pop all the a's off the stack. If we guess wrong, that path will just fail, but it will never cause us to accept something we shouldn't, since it only pops off extra a's, which is what we want anyway. We don't need a separate state for the final popping phase, since we're willing to accept either m = n or m > n. This contrasts with the example we did in class, where we required that m > n. In that case, the state where we run out of input and the stack at the same time (i.e., m = n) had to be a rejecting state. Thus we needed an additional accepting state where we popped the stack.

M = ({1, 2}, {a, b}, {a}, 1, {2}, Δ =

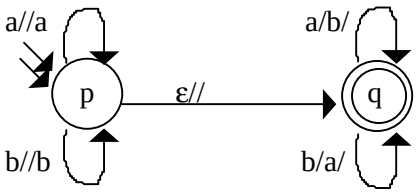
((1, a, ε), (1, a))	/* push an a on the stack for every input a
((1, b, a), (2, ε))	/* pop an a for the first b and go to the b-popping state
((1, ε, ε), (2, ε)	/* in case there aren't any b's -- guess end of string and go pop any a's
((2, b, a), (2, ε)	/* for each input b, pop an a off the stack
((2, ε, a), (2, ε)	/* if we run out of input while there are still a's on the stack, then pop the a's and accept

(g) The idea here is to create a nondeterministic machine. In the start state (1), it reads a's and b's, and for each character seen, it pushes an x on the stack. Thus it counts the length of u. If it sees an a, it may also guess that this is the required separator a and go to state 2. In state 2, it reads a's and b's, and for each character seen, pops an x off the stack. If there's nothing to pop, the machine will fail. If it sees a b, it may also guess that this is the required final b and go to the final state, state 3. The machine will then accept if both the input and the stack are empty.

M = ({1, 2, 3}, {a, b}, {x}, s, {2}, Δ =

((1, a, ε), (1, x))	/* push an x on the stack for every input a
((1, b, ε), (1, x))	/* push an x on the stack for every input b
((1, a, ε), (2, ε))	/* guess that this is the separator a. No stack action
((2, a, x), (2, ε)	/* for each input a, pop an x off the stack
((2, b, x), (2, ε)	/* for each input b, pop an x off the stack
((2, b, ε), (3, ε)	/* guess that this is the final b and go to the final state

3.



4. S → ε
S → aSb
S → aSbb