Aughdon Breslin and Bella Cruz
CS334 PS7
"I pledge my honor that I have abided by the Stevens Honor System."

Problem 1

A 2-stack PDA (2PDA for short) is a pushdown automaton with two stacks. In one step each stack can be popped or pushed independently. For example, the 2PDA may read an input symbol, pop the first stack, ignore the second stack and, based on the values seen, change state, push different symbols on the two stacks. The input tape is read-only.
Describe a 2-stack PDA that accepts the language L = $\{a^i b^k c^i d^k : i,k >= 0\}$. You do not need to give a PDA diagram – a high-level, but complete, description of how your PDA works will suffice.

Push $ onto stack 1 and stack 2 to signify the bottom of the stack.
Push a's on stack 1 until you're out of a's. Push b's on stack 2 until you're out of b's.
Pop a's from stack 1 until you're out of c's. Pop b's from stack 2 until you're out of d's.
If both stacks are empty after finishing the input, accept. If a stack empties and there are still a's on stack 1 or b's on stack 2, reject. If either $ is not reached when we are out of c's or d's, reject.

A:

On input $w = w_1 w_2 ... w_n$
Start with two empty stacks
If the input is empty, accept. Otherwise, maintain four states, listed below.
Push a $ onto stack 1 and stack 2 to signify when a stack is empty.
If at any point, the PDA tries to pop from an empty stack, signaling that there'd be more c's than a's or d's than b's, reject.
In state "reading a's":

- If w_i = a: push w_i onto stack 1; stay in state "reading a's"
- If w_i = b: push w_i onto stack 2; move to state "reading b's"
- If w_i = c: pop from stack 1; move to state "reading c's"
- If w_i = d: reject

In state "reading b's":

- If w_i = a: reject
- If w_i = b: push w_i onto stack 2; stay in state "reading b's"

- If w_i = c: pop from stack 1; move to state “reading c’s”
- If w_i = d: pop from stack 2; move to state “reading d’s”

In state “reading c’s”:

- If w_i = a: reject
- If w_i = b: reject
- If w_i = c: pop from stack 1; stay in state “reading c’s”
- If w_i = d: pop from stack 2; move to state “reading d’s”
- If both stacks are empty (the $s have been reached) and the entire input is read, accept

In state “reading d’s”:

- If w_i = a: reject
- If w_i = b: reject
- If w_i = c: reject
- If w_i = d: pop from stack 2; stay in state “reading d’s”
- If both stacks are empty and the entire input is read, accept

Problem 2

In Problem Set 6 you showed that the language $L_{mult} = \{a^i b^{ij} c^j : i,j >= 0\}$ is not context free. Either show that this language can be recognized by a 2PDA or explain why that is not possible.

A:

On input $w = w_1 w_2 ... w_n$

Start with 2 empty stacks. Push the dollar signs into each of them to signify the bottom of the stack.

Strategy: As you read each input, push it into the left stack (stack 1) until there are no more inputs. Then push and pop to simulate moving left and right as if in a Turing Machine. From there,

1. Cross off the first a
2. Cross off as many b's as there are c's
    a. Shuffle between the c's and b's, crossing off one of each, if all b's have been crossed off, but not all c's: REJECT.
3. Restore the c's.
4. Repeat from Step 1. If no a remains, and all b's are crossed off ACCEPT, else REJECT.

<u>Make sure that the Language is in a*b*c* form</u>

In state "reading a's":

- If w_i = a: push w_i into stack 1; stay in state "reading a's"
- If w_i = b: push w_i into stack 1; move to state "reading b's"
- If w_i = c: reject (we've read at least 1 a, and have now read a c, but no b's)

In state "reading b's":

- If w_i = a: reject
- If w_i = b: push w_i into stack 1; stay in state "reading b's"
- If w_i = c: push w_i into stack 1; move to state "reading c's"

In state "reading c's":

- If w_i = a: reject
- If w_i = b: reject
- If w_i = c: push w_i into stack 1; stay in state "reading c's"

When we have finished reading the input, we only work with the stack, treating that as a Turing Machine, popping into stack 1 and pushing into stack 2 to move left, and vice versa to move right.

<u>Starting Out</u>

From now on, we will never read any input.
Pop c from stack 1 and push c into stack 2. Do this for all c's.

The Big Loop
Pop b from stack 1 and push b into stack 2. Do this for all b's.
Pop a from stack 1.
Pop b from stack 2 and push b into stack 1. Do this for all b's.
Pop c from stack 2, pop b from stack 1, and push x into stack 2.
Pop x from stack 2 and push x into stack 1. Do this for all x's.

The Little Loop
Pop c from stack 2 and push x into stack 2.
Pop x from stack 1 and push x into stack 2. Do this for all x's.
Pop b from stack 1.
If we can Pop a from stack 1 and c from stack 2, reject. (This would mean all b's have been popped but not all c's.)
Pop x from stack 2 and push x into stack 1. Do this for all x's.
---End Little Loop---

Repeat The Little Loop until the right stack (stack 2) is empty.
Pop x from stack 1 and push c into stack 2. Do this for all x's.
If we can Pop a from stack 1, reject. (This would mean all b's have been popped but not all a's.)
---End Big Loop---

Repeat The Big Loop until the left stack (stack 1) is empty..
If it hasn't been rejected, accept.
The 2PDA recognizes the language $L_{mult} = \{a^i b^{ij} c^j : i,j >= 0\}$.

Problem 3
Prove that the intersection of a CFL and a Regular language is always context-free.

A:
Say “L” represents the CFL and “R” represents the regular language

PROVE: L ∩ R is a context-free language

Say “P” is a PDA that accepts L and “D” is a DFA that accepts R. The intersection of L and R will be represented by another PDA “A”. A will mimic the movements of P and D and accept a string if both P and D accept it.

When constructing A, it is important to note that every state of A will be represented by a pair of states: current state of P and current state of D. The set of accept states in A will be made up of pairs (state in P, state in D) where both states are accept states. The stack of A will equal the stack of P.

The definition of D = $(Q_D, \Sigma, \delta_D, q_D, F_D)$
The definition of P = $(Q_P, \Sigma, \Gamma_P, \delta_P, q_P, F_P)$

Since A is a PDA, it is defined but the 6-tuple: $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where:

$Q = Q_D \times Q_P$
$\Gamma = \Gamma_P$
$\delta ((p, q), x, a) = \{ ((p', q'), b) \mid p' = \delta_D(p, x) \text{ and } (q', b) \in \delta_P(q,x,a) \}$
$q_0 = (q_D, q_P)$
$F = F_D \times F_P$

Using induction on the number of transitions, we know that for any $s \in \Sigma^*$
$(q_0, \varepsilon) \rightarrow ((p,q), \sigma)$ if and only if $q_D \rightarrow p$ and $(q_P, \varepsilon) \rightarrow (q, \sigma)$

Thus, a string is accepted by the PDA A if and only if $(q_0, \varepsilon) \rightarrow ((p,q), \sigma)$ such that (p,q) is an accept state, if $(q_0, \varepsilon) \rightarrow ((p,q), \sigma)$ such that p is an accept state of M and q is an accept state of P, if $q_D \rightarrow p$ and $(q_P, \varepsilon) \rightarrow (q, \sigma)$ where p is an accept state of M and q is an accept state of P, and the string exists in the language of the DFA D and the PDA P.
Since this language can be represented by a PDA, it is context-free.

Optional Problem 4:
Prove that the language A\B = {w : wx ∈ A, x ∈ B}, where A is a CFL and B is regular, is a CFL.

A:
Let A be represented by a PDA P, and B be represented by a DFA D.
Let R be the PDA for A\B. The states of A\B would be equivalent to (the states of P) X (the states of D).

While we are still in the w part of the string, the input would be translated by P but not by D, since at this point we are only trying to make sure wx belongs to A and we haven't yet gotten to x. We do not care if w or wx belongs to B.

When we get to the x part of the string, the inputs affect both P and D since now both must accept the language, and we must reach an accepting state of P x D in both machines simultaneously.

R would operate using the same stack P would as if alone, but with the additional requirements of satisfying the conditions laid out by D. A state (p,q) in A\B would accept iff p accepts in P and q accepts in D. Any inputs that would only be valid in either P alone or D alone would be rejected by R.

Since this would be a valid PDA representation of the language A\B, A\B is a CFL.

Optional Problem 5:
A queue automaton is like a push-down automaton except that the stack is replaced by a queue. A queue is a tape allowing symbols to be written only on the left-hand end and read-only at the right-hand end. Each write operation (we'll call it a push) adds a symbol to the left-hand end of the queue and each read operation (we'll call it a pull) reads and removes a symbol at the right-hand end.
As with a PDA, the input is placed on a separate read-only input tape, and the head on the input tape can move only from left to right. The input tape contains a cell with a blank symbol following the input so that the end of the input can be detected. A queue automaton accepts its input by entering a special accept state at any time. Show that a language can be recognized by a deterministic queue automaton if the language is Turing-recognizable.

A:

Proving that a turing machine and a queue automaton have the same computational power.

Must prove TM → QA AND QA → TM :

The queue of the queue automaton will act like the tape of a turing machine. The symbol that is being read will be the symbol at the top of the queue. For the purpose of this proof, we will simulate the queue as a tape where the right end of the tape is the bottom of the queue (where elements get pushed) and the left end of the tape will be the top of the queue (where elements get popped).

To begin simulating a turing machine, you would first push every input symbol into the q, then push a $ at the very end to mark the beginning of the "tape". This will serve as the beginning of the "tape" because all elements to the left of the $ will be to the "right" of the tapehead and every symbol on the right of the S will be to the "left" of the tapehead.

TM → QA:

Must simulate all possible movements of a Turing machine with a queue.

1) 1 → a, right (move right while reading and writing a symbol)

2)2 → b, left (move left while reading and writing a symbol)

1) Move right while reading and writing a symbol

In terms of 1 → a, right, you would pop the symbol 1 from the queue and push the symbol a into the queue. This will move the tapehead right one box while leaving the rest of the simulated tape unchanged. If we end up popping a $, we would “undo” it by moving left, pushing a blank symbol, and then moving left again. This should allow us to effectively move right.

2) Move left while reading and writing a symbol

To move left, you’d have to move the character to the very right of the queue to the head. You would first push a symbol % into the queue. Then, until you reach the % you will pop each element and immediately push it back into the queue. Once you hit the % symbol, you would pop it from the queue and you would have successfully moved the element at the very bottom of the queue to the top.

This simulates moving the tapehead to the left by first reading the symbol, then shifting to the left twice (once to “undo” the fact that you needed to move right to read the symbol, and the second time to actually move left). The only case in which you would not make a second shift would be if you read the $ symbol.

QA → TM :

Must simulate all queue operations with a 2-tape turing machine:

1) push
2) pop

1) push

To “push” a symbol, the tapehead would move to the first blank symbol in the tape and write that symbol.

2) pop

To “pop” a symbol, the tapehead would move until it reaches the $ symbol, read the first symbol in the alphabet afterwards, then write a “X” in its place. This act of crossing the symbol out would indicate that it’s been “popped”.

Therefore, since you can simulate a turing machine with a queue Automaton and you can simulate a queue automaton with a turing machine, queue automatons and turing machines have the same computational power.