Problem Set 4

Fall 09

Due: Thursday Nov 5 at 11:00 AM in class (i.e., Room 103 Talbot Lab)
Please follow the homework format guidelines posted on the class web page:
`http://www.cs.uiuc.edu/class/fa09/cs373/`

1. [**Points**: 10] Suppose G is a CFG with p rules and the body of each rule is not longer than n (Note: the rule $S \to aSbSA$ has a body length of 5). If we can derive ε from non-terminal S, then prove that we can derive it from S in no more than $\frac{n^p-1}{n-1}$ steps.

Solution:

Consider all parse trees that derive ε from S and let T be one of them that has the minimum number of nodes. On every path from root to some leaf of T, no non-terminal appears twice (if it appears twice, then we can remove all nodes between the two occurrences of that non-terminal and still have a valid parse tree for ε). Therefore the height of the tree is at most $p-1$. Since each node in the tree has at most n childs, the k-th level of T has at most n^k nodes. Therefore the total number of nodes (which is equal to the total number of steps in the derivation) is no more than $1+n+n^2+\cdots+n^{p-1} = (n^p-1)/(n-1)$.

2. [**Points**: 10]

   (a) Prove that this language is not context free:

$$L = \{a^i b^j c^{\max(i,j)} \mid i,j \geq 0\}$$

Solution:

Assume to the contrary that it is context free. Therefore it follows the pumping lemma for CFLs and has a pumping length p. Consider the string $s = a^p b^p c^p \in L$. Since $|s| \geq p$, pumping lemma applies to s and we should be able to break s into $s = xuyvz$ such that $|uyv| < p$, $|uv| \neq 0$ and $s_i = xu^i yv^i z \in L$ for every $i \geq 0$. We consider two cases (note that one of these cases always happens):

- If $|u| > 0$ and u contains some c's, or $|v| > 0$ and v contains some c's, then s_0 has exactly p a's (note that $|uyv| < p$ and since uyv has some c's it cannot "reach" the area of a's in the string) but less than p c's therefore $s_0 \notin L$.
- Otherwise consider s_1. Since u and v do not intersect c's, s_1 has exactly p c's. But the number of a's or b's in s_1 is more than p (since one of u and v has positive length and it intersects a's or b's). Therefore $s_1 \notin L$.

So L does not follow the CFL-pumping lemma and therefore is not a CFL.

   (b) Prove that context free languages are not closed under the operator max, as defined below:

$$\max(A) = \{w \in A \mid \text{if } wx \in A \text{ for some string } x, \text{ then } x = \varepsilon\}.$$

Hint: part(a) might be useful.

Solution:

Consider this language A:

$$A = \{a^i b^j c^k \mid k \leq i \text{ or } k \leq j\}$$

We have seen a PDA for this language, so we know that A is context free. But note that if $w = a^i b^j c^k \in A$ and $k < \max(i, j)$, then $wc \in A$, therefore $w \notin \max(A)$. So the only possibility for w to be in $\max(A)$ is when $k = \max(i, j)$. This means that $\max(A) = L$, where L is the language in part(a) that we know is not context free.

3. [**Points**: 10] Describe an algorithm that given a grammar G determines whether $L(G)$ has at least 373 strings or not.

Solution:

First we describe an algorithm A that given grammar G and a string $w \in L(G)$, computes a grammar G' so that $L(G') = L(G) - \{w\}$: $\overline{\{w\}}$ is a regular language so it is the language of some DFA M. From textbook we know how to construct a PDA D, given G, that accepts $L(G)$. Again from textbook we know how to construct a PDA D' that accepts the language $L(G) \cap \overline{\{w\}} = L(G) - \{w\}$ (cross product of PDA D and DFA M). Again(!) from textbook we know how to build a grammar G' equivalent to a given PDA D'. Note that $L(G') = L(G) - \{w\}$.

Second, the textbook presents a procedure B that given some grammar G, determines whether $L(G)$ is empty or it produces a string w in $L(G)$ (proving that $L(G)$ is not empty). Now to answer our question, using B we generate some string w in $L(G)$ (or we fail to do). The using B we generate a grammar that its language just misses w and we repeat the same process as before (to obtain a new string in $L(G)$ and remove that new string from the language and repeating the same process ...). If we can repeat this till we produce 373 strings, the answer to our question is positive, otherwise $L(G)$ has less than 373 strings in it.

4. [**Points**: 10] Given a grammar G in Chomsky normal form:

   (a) Design an algorithm (via modifying the CYK algorithm) that for any given number $n \in N \cup \{0\}$ answers in $O(n^2)$ time whether there is a string w generated by G with length n.

Solution:

The CYK algorithm will be modified in the following way. Instead of having variables X_{ik} which stores all the non-terminals that can derive the substring $w_i w_{i+1} \dots w_{i+k}$ (of input w), we will have variable X_i which will store all the non-terminals that can derive a string of length i. If finally, X_n contains the start symbol, then it means the grammar can derive a string of length n. The detailed algorithm is below.

   (b) Now give another algorithm that does the same in $O(n)$ time.

Solution:

For this problem, **giving the idea** of identifying all terminals and that this leads to a regular language will get the full credit.

Modified_CYK
Input n
for $i = 1$ to n do
    $X_i = \emptyset$
for every production do
    if $A \to a$ is a production then $X_1 = X_1 \cup \{A\}$
for $i = 2$ to n do
    for $j = 1$ to $i - 1$ do
        if $A \to BC$ is a production $B \in X_j$ and $C \in X_{i-j}$ then $X_i = X_i \cup \{A\}$
if $S \in X_n$ then output **yes**
else output **no**

Since we are interested only in whether a string of certain length exists in a context free language, we may effectively identify all the terminals in G. Let us assume we change all terminals in G to the symbol 0 and obtain a grammar G'. If G generates $w = w_1 w_2 \dots w_n$, then 0^n must be in G'. On the other hand, if G' generates 0^m for some m, then G must generate some string of length m.

Having established that that we may work with G' instead of G, we want to show that G' describes a regular language and therefore can be converted into a regular expression. This is a direct application of Parikh's Theorem, which states that a CFL is regular *if we ignore the order of letters in the words.* A simple version that applies to our problem is that if $L(G') = A \subset \{0\}^*$ is context free, then A is regular. The sketch of proof of this is outlined below (for the case in which A is infinite).

1 Let $K > 1$ be the constant of the pumping lemma for G', and let $r = K!$. Show that for every $w \in A$, if $|w| \geq K$ then

$$\{wa^{rm} \mid m \geq 0\} \subset A.$$

2 For every i such that $0 \leq i < r$, let

$$A_i = \{a^m \mid a^m \in A, m \geq K, m \equiv i \bmod r\}.$$

clearly,

$$A = \{a^m \mid a^m \in A, m < K\} \cup \bigcup_{i=0}^{r-1} A_i.$$

If $A_i \neq \emptyset$, let z_i be the shortest string in L_i. Prove

$$A_i = \{z_i a^{rm} \mid m \geq 0\}.$$

3 Conclude that A is regular.

Note that the proof is also an algorithm that "dissect" A into pieces. The algorithm may take a lot of time to run but it is a one time job and does not depend on the input n. Given any n, we only need to check a constant number of "pieces" of A to see whether $0^n \in A$. This yields an algorithm of complexity less than $O(n)$.

5. [**Points**: 15] Design a Turing machine M that given a string w on its tape (as input), halts with a string w_1 on one of its tapes (as its output) such that:

(a) $w \in \{a, b\}^* \backslash \{\varepsilon\}$ and w_1 is the next string in the dictionary (lexicographical) order after w.

Solution:

Since there are confusions about what lexicographic ordering is, the following two solutions are both accepted:

1 The Turing machine moves to the end of w, appends a a.

2 The Turing machine first moves to the right end of its input and then, scanning to the left, looks for the first a, changing every b in its way to a a. When it finds the first a, it replaces it with a b and halts. If on the other hand it reaches the blank cell on the left end of the input, it moves back to the right end and adds a a.

(b) $w \in L(a^*), |w| = n$, $w_1 \in L(a^*)$ and $|w_1| = \lceil \sqrt{n} \rceil$, where $\lceil x \rceil$ is the smallest integer number that is greater than or equal to x.

Solution:

We use 3 tapes here. Let tape 1 be the input tape. Let tape 2, 3 be blank initially. We use two operations to build the TM.

- **INCREMENT**. Move pointer on tape 2 and 3 left until they hit blank. Then move the pointer right until it hits a blank. Write an a in place of this blank.
- **MULTICOMP**. Move all tape pointers to the left most a. We then move tape 1, 3 pointers synchronously on each a read from both tape 1 and 3, until one of the pointers hits a blank. If tape 1 pointer hits a blank during the process, the machine accepts (with tape 2 containing w_1). If it is tape 3 pointer that hits blank first, we move tape 3 pointer to the leftmost a and move the tape 2 point to the right once. If tape 2 pointer points to a blank, then we do **INCREMENT** again.

The machine starts with **INCREMENT** and then moves to **MULTICOMP**. It will eventually accept.

(c) $w \in L(a^*), |w| = n$, $w_1 \in L(a^*)$ and $|w_1| = F_n$, where F_n is the n-th Fibonacci number ($F_0 = 0, F_1 = 1, F_2 = 1, F_3 = 2, \ldots, F_{k+1} = F_k + F_{k-1}, \ldots$).

Solution:

We use 3 tapes here. Let the first tape contain the input. To start, let tape 2, 3 be blank. Define a high level operation **ADD2TO3** as: Move head of tape 2 to the leftmost a and move head of tape 3 just beyond the rightmost a. For each a read on tape 2, write an a on tape 3, and move heads of tape 2, 3 to the right. We may define **ADD3TO2** similarly: Move head of tape 3 to the leftmost a and move head of tape 2 just beyond the rightmost a. For each a read on tape 3, write an a on tape 2, and move heads of tape 2, 3 to the right. The machine works as this

1 On reading a first a on input, write a on tape 3. Move heads of tape 1, 3 right.

2 On reading each additional a on tape 1, move tape 1 heard to the right, then do either **ADD3TO2** or **ADD2TO3**. We need one bit to record which addition to use next.

3 Once the input tape is exhausted, we have w_1 on either tape 2 or tape 3 (the longer of the two).

6. [**Points**: 10] A MidPDA is an extended PDA that can read the middle of its stack. Formally, the next transition in a MidPDA is determined based on four things: the currect symbol from the input string, the symbol on top of the stack, the current state of MidPDA, and the symbol in the middle of the stack (when the stack has k symbols in it, the $\lceil k/2 \rceil$-th symbol from the bottom of the stack is defined as the middle of the stack. If the stack is empty, the middle of the stack is ε). When MidPDA makes a transition, it may change its state, may push/pop something to/from the stack, and may read at most one symbol from the input.

   Is a MidPDA stronger than a normal PDA? Prove or disprove.

Solution:

Obviously a MidPDA is as strong as a normal PDA since it can just ignore the extra data that it receives from the middle of its stack (pretending that it is a normal PDA). We show that MidPDA can accept the non-CFL $L = \{a^n b^n c^n \mid n > 0\}$ and therefore it is stronger that a normal PDA : As MidPDA reads a's, it just pushes A onto the stack. As it reads b's, if the middle of stack is an A, it just pushes B onto the stack (and if it sees an a after a b just rejects), and if the middle of stack is not A (signaling that the number of b's is more than a's), then it just rejects. As it reads the first c, if the middle of stack is B (signaling that the number of a's and b's are equal), then it pops B from the stack, otherwise it rejects. As it reads the rest of c's, it pops B from the stack. If the top of the stack is an A and the input string is finished (singnaling equal number of b's and c's), it accepts.

7. [**Points**: 25]

   Pushdown automata use stack as their information storage, but we can imagine a similar machine that uses other data-structures. *Queue automaton* is a finite state machine that uses a queue (instead of a stack) as its storage. A *two-stack PDA* is a PDA with two distinct stacks. It makes transitions based on its current state, the two symbols at the top of its stacks, and one symbol from its input (possibly ε). When a transition is being made, it may change its state, may push/pop something to/from its first stack, may push/pop something to/from its second stack (independently from the first stack), and may read at most one symbol from its input.

   (a) Formalize the definitions of the queue automaton and the two-stack PDA.

Solution:

A queue automaton (QA) is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where:

i. Q is a finite set of states
ii. Σ is a finite input alphabet
iii. Γ is a finite stack alphabet
iv. $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathbb{P}(Q \times \Gamma_\varepsilon)$ is the transition function, where $A_\varepsilon = A \cup \{\varepsilon\}$
v. $q_0 \in Q$ is the start state
vi. $F \subseteq Q$ is the set of accept states

A QA M accepts the string w iff w can be written as $w = w_1 w_2 \dots w_m$, where each $w_i \in \Sigma_\varepsilon$ and sequences of states $r_0, r_1, \dots, r_m \in Q$ and strings $s_0, s_1, \dots, s_m \in \Gamma*$ exist that satisfy the following conditions:

i. $r_0 = q_0$ and $s_0 = \varepsilon$
ii. For $i = 0, \ldots, m-1$, there exist $a, b \in \Gamma_\varepsilon$ and $t \in \Gamma^*$, such that $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, $s_i = at$ and $s_{i+1} = tb$ (this one is the only thing that differs from PDA)
iii. $r_m \in F$

A two-stack PDA (2PDA) is a 7-tuple $(Q, \Sigma, \Gamma, \Theta, \delta, q_0, F)$, where:

i. Q is a finite set of states
ii. Σ is a finite input alphabet
iii. Γ is a finite alphabet of the first stack
iv. Θ is a finite alphabet of the second stack
v. $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \times \Theta_\varepsilon \to \mathbb{P}(Q \times \Gamma_\varepsilon \times \Theta_\varepsilon)$ is the transition function, where $A_\varepsilon = A \cup \{\varepsilon\}$
vi. $q_0 \in Q$ is the start state
vii. $F \subseteq Q$ is the set of accept states

A 2PDA M accepts the string w iff w can be written as $w = w_1 w_2 \ldots w_m$, where each $w_i \in \Sigma_\varepsilon$ and sequences of states $r_0, r_1, \ldots, r_m \in Q$ and strings $s_0, s_1, \ldots, s_m \in \Gamma*$, $p_0, p_1, \ldots, p_m \in \Theta*$ exist that satisfy the following conditions:

i. $r_0 = q_0$, $s_0 = \varepsilon$, $p_0 = \varepsilon$ (we begin with both stacks being empty)
ii. For $i = 0, \ldots, m-1$, there exist $a, b \in \Gamma_\varepsilon$, $c, d \in \Theta_\varepsilon$, $t \in \Gamma^*$ and $u \in \Theta^*$, such that $(r_{i+1}, b, d) \in \delta(r_i, w_{i+1}, a, c)$; $s_i = at$, $s_{i+1} = bt$, $p_i = ct$ and $p_{i+1} = dt$
iii. $r_m \in F$

(b) Describe how to construct a two-stack PDA that accepts an intersection of two CFLs. (Note: this proves that two-stack PDA is more powerful than PDA).

Solution:

Suppose we have two CFLs A and B, and two corresponding PDAs $P^A = (Q^A, \Sigma, \Gamma^A, \delta^A, q_0^A, F^A)$ and $P^A = (Q^B, \Sigma, \Gamma^B, \delta^B, q_0^B, F^B)$. We show how to construct a 2PDA P from P^A and P^B. The idea is similar to construction of an NFA for the intersection of 2 regular languages: we run P^A and P^B simultaneously, using the first stack of P as the stack of P^A and the second stack as the stack of P^B, and each state P being a pair of states of P^A and P^B.
$P = (Q, \Sigma, \Gamma^A, \Gamma^B, \delta, q_0, F)$ where:

i. $Q = Q^A \times Q^B$
ii. $\delta((q,p), x, a, b) = \{((u,v), c, d) \mid (u,c) \in \delta^A(q, x, a) \text{ and } (v,d) \in \delta^B(p, x, b)\}$
iii. $q_0 = (q_0^A, q_0^B)$
iv. $F = \{(q,p) \in Q \mid q \in F^A \text{ and } p \in F^B\}$

(c) Prove that a queue automaton is at least as powerful as a PDA by showing how to emulate a PDA using it.

Solution:

In QA we can only enqueue to the end of the queue and dequeue from the beginning. That is, if we have

$w = abc$, $w \in \Gamma^*$ as the contents of the queue, we can get $w' = bc$ after dequeuing or $w' = abcd$ after enqueuing. However, we want to "push" the symbol b to the beginning of the queue, as in a PDA. We can do this by adding b to the end, and then "rewinding" the queue, until b is the first symbol, by getting symbols (one by one) from beginning and putting them to the end: $abcb \to bcba \to cbab \to babc$. The problem here is that we need to know when to stop, because we already have some other b in the queue. We achieve this by adding a special marker # right before b, and then removing it: $abc\# \to abc\#b \to bc\#ab \to c\#abb \to \#babc \to babc$. Now, suppose we have a PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$. We show how to construct a QA $M = (Q', \Sigma, \Gamma', \delta', q_0', F')$, that will accept exactly the same language as P.

i. $Q' = Q \cup E$, where E is a set of intermediate states (see below)
ii. $\Gamma' = \Gamma \cup \{\#\}$
iii. $q_0' = q_0$
iv. $F' = F$

We define δ in the following way: for every transition $(q \in Q, a \in \Sigma_\varepsilon, x \in \Gamma_\varepsilon) \to (p \in Q, y \in \Gamma_\varepsilon)$ in P we introduce a sequence of transitions in M that emulate pushing by "rewinding" the queue (adding new intermediate states r and l to E):

i. $\delta'(q, a, x)$ contains $(r, \#)$
ii. $\delta'(r, \varepsilon, \varepsilon) = \{(l, y)\}$
iii. $\delta'(l, \varepsilon, c) = \{(l, c)\}$ for each $c \in \Gamma$
iv. $\delta'(l, \varepsilon, \#) = \{(p, \varepsilon)\}$

(d) Prove that a queue automaton is strictly more powerful than a PDA by presenting an example of a language that can be recognized by a queue automaton, but not by any PDA.

Solution:

Consider $L = \{ww \in \{0,1\}^* \mid w \in \{0,1\}^*\}$. We know that this language is not CFL, however we can construct a QA that recognizes it. The idea is similar to constructing PDA for $\{ww^R \mid w \in \{0,1\}^*\}$, we enqueue symbols to the queue, try to "guess" (non-deterministically) the middle of the string and then switch from enqueuing to dequeuing.
$M = (Q, \Sigma, \Gamma, \delta, q_0, F)$, where:

i. $Q = \{q_1, q_2, q_3, q_4\}$
ii. $\Sigma = \{0, 1\}$
iii. $\Gamma = \{0, 1, \$\}$
iv. $q_0 = q_1$
v. $F = \{q_1, q_4\}$

We define δ as following:

i. $\delta(q_1, \varepsilon, \varepsilon) = \{(q_2, \$)\}$
ii. $\delta(q_2, 0, \varepsilon) = \{(q_2, 0)\}$ and $\delta(q_2, 1, \varepsilon) = \{(q_2, 1)\}$
iii. $\delta(q_2, \varepsilon, \varepsilon) = \{(q_3, \varepsilon)\}$
iv. $\delta(q_3, 0, 0) = \{(q_2, \varepsilon)\}$ and $\delta(q_2, 1, 1) = \{(q_2, \varepsilon)\}$
v. $\delta(q_3, \varepsilon, \$) = \{(q_4, \varepsilon)\}$

(e) Prove that a two-stack PDA is at least as powerful as a queue automaton by showing how to emulate it on a two-stack PDA.

Solution:

Observe that popping elements from one stack and pushing them to another reverses the order of the elements. We can use this fact to emulate queue using two stacks: we push elements in the second stack (emulating "enqueue") and pop from the first ("dequeue"); when the first one is empty, we "pour" elements from the second one to the first one. Doing so, we reverse the order, and the elements that we pushed last, now are at the bottom of the stack and will be popped last, as in the queue. We will use this idea to construct a 2PDA that emulates QA.

Suppose we have a QA $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$. We show how to construct a 2PDA $P = (Q', \Sigma, \Gamma', \Theta', \delta', q_0', F')$, that will accept exactly the same language as M.

i. $Q' = Q \cup E$, where E is a set of new intermediate states

ii. $\Gamma' = \Theta' = \Gamma \cup \{\$\}$, i.e. both stack use the same alphabet, \$ is used to indicate the bottom of the stack.

iii. q_0' is the new start state

iv. $F' = F$, accept states remain the same

We define δ' as the following:

We initialize both stacks with \$ on the bottom: $\delta'(q_0', \varepsilon, \varepsilon, \varepsilon) = \{(q_0, \$, \$)\}$.

For every transition $(q \in Q, a \in \Sigma_\varepsilon, x \in \Gamma_\varepsilon) \to (p \in Q, y \in \Gamma_\varepsilon)$ in M we introduce several transitions in P that emulate dequeuing x and enqueuing y from the queue. If we have x on the top of the first stack, we pop it and push y onto the second stack: $\delta'(q, a, x, \varepsilon)$ contains (p, ε, y). We also need to check whether the first stack is empty, and if it is, then we need to transfer all symbols from the second stack to the first one. For this purpose we add this sequence of transitions (we add the new states l and r to E):

i. check that the first stack is empty (by testing for the \$), and go to l if it is:

$$\delta'(q, \varepsilon, \$, \varepsilon) = \{(l, \$, \varepsilon)\}$$

ii. transfer elements from the second stack to the first (reversing the order):

$$\delta'(l, \varepsilon, \varepsilon, c) = \{(l, c, \varepsilon)\} \text{ for every symbol } c \in \Gamma$$

iii. check whether we've reached the bottom of the second stack and go to r if we did:

$$\delta'(l, \varepsilon, \varepsilon, \$) = \{(r, \varepsilon, \$)\}$$

iv. continue normal execution by popping x from the first stack and pushing y to the second:

$$\delta'(r, a, x, \varepsilon) = \{(p, \varepsilon, y)\}$$

8. [**Points**: 10] Extend the PDA definition from class so that an entire string can be pushed onto the stack in one step. This means $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to Pow(Q \times \Gamma^*)$. It turns out that to recognize CFLs, very few states are needed. Find the minimum k such that for any CFL, there exists an extended PDA that recognizes it with no more than k states. Prove your answer.

Solution:

We know that for any CFL L there exists a CFG G describing it. We show how to construct an extended PDA (EPDA) P from G. This construction resembles the one for CFG to PDA conversion, except that we do not need intermediate states.
$P = (Q, \Sigma, \Gamma, \delta, q_{start}, F)$, $Q = \{q_{start}, q_{loop}, q_{accept}\}$, $F = \{q_{accept}\}$, Γ is the set of all non-terminals in G plus \$. δ is:

(a) $\delta(q_{start}, \varepsilon, \varepsilon) = \{(q_{loop}, S\$)\}$, S - is the start symbol in G

(b) For each non-terminal A in G, $\delta(q_{loop}, \varepsilon, A) = \{(q_{loop}, w) \mid A \to w \text{ is a rule in } R\}$

(c) For each terminal a in G, $\delta(q_{loop}, a, a) = \{(q_{loop}, \varepsilon)\}$

(d) $\delta(q_{loop}, \varepsilon, \$) = \{(q_{accept}, \varepsilon)\}$

This shows that we only need only a maximum of 3 states to handle every CFL. No we show that we can't handle every CFL with less.
Obviously, it's not enought to have only one state. This only state has to be either accept state or not. In the latter case our EPDA will only accept an empty language, while in former case it will always accept an empty string, but not every CFL contains an empty string.
Now we show that 2 states will not suffice. Suppose that for every language L we can construct an EPDA with only 2 states. Let's consider a $L = \{a^n b^n \mid n \geq 0\}$ and a corresponding EPDA P with 2 states q_{start} and q_{accept}. Since P accepts $a^k b^k$ (for some k), this means (by definition), that there's a path from q_{start} to q_{accept} on $a^k b^k$, starting with an empty stack and ending with $w \in \Gamma^*$ on the stack. This also means that P can accept $a^k b^k$ starting with $w' \in \Gamma^*$ on the stack, and accepting with $w'w$.
Consider the string $a^{k+1}b^k = aa^k b^k$. After consuming the first a, P must be in q_{start}. It may not be in q_{accept}, because it means that P accepts the string a, which is not in L; and there must be at least one transition from q_{start} on input a, otherwise P will reject any string starting with a, such as a^1b^1, a^2b^2, etc. We also know that there's a path from q_{start} to q_{accept} on the remaining part of the string and any stack contents. Hence, P accepts $a^{k+1}b^k$.
The idea is similar to pumping lemma: we show that if P accepts $a^n b^n$ then we can "pump" as many as to the front as we want. We obtained the contradiction, therefore 2 states is not enough. Hence, 3 is the minimum number of states we need to accept any CFL.