Theory of Computation Assignment no. 7

Goktug Saatcioglu

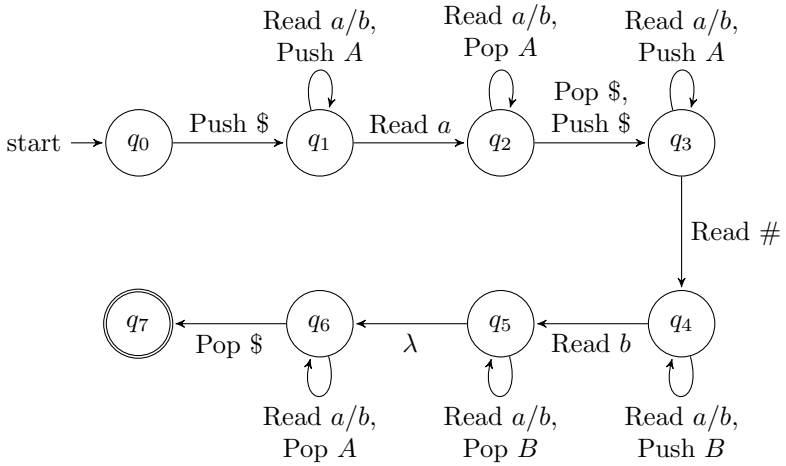
(1) a. This CFG describes the language of balanced (legal) parantheses. That is, each opening bracket "(" has a corresponding closing bracket ")". Notice that the empty string, λ, is not a part of this language thus the language requires at least a single pair of balanced paranthesis.

b. Assuming that each "(" must have a corresponding ")" and each "[" must have a corresponding "]", consider the following CFG G':

$$S \to SS \mid (S) \mid [S] \mid () \mid []$$

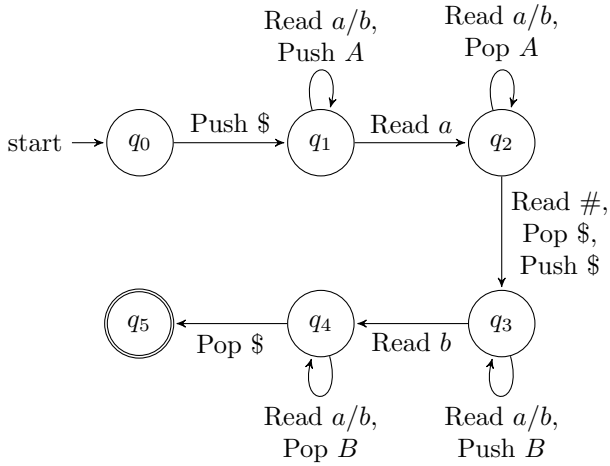
which derives the language described in the question.

(2) i. To build this PDA we first construct three different PDA's based on the different cases that can occur in A and take the union of them. The first case is if $|u| - |v| < 0$, then we should only accept words that are $|x| - |y| < 0$ and $|u| - |v| = |x| - |y|$. The PDA that recognize this case, say A_1, is given below.

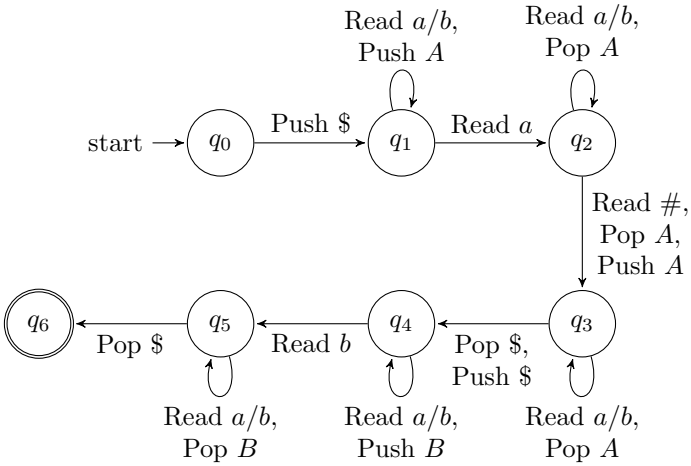


For A_1 we can interpret $|u| - |v| = |x| - |y|$ as $|u| - |v| - |x| + |y| = 0$ and since we know that $|u| - |v| < 0$ we will have to push to stack the extra v's as we can't have a negative stack size. Thus, q_0 is the empty stack with nothing being read yet, q_1 is the stack with the shielding symbol and we push A's as we read u. Then we transition to q_2 after reading the seperating letter of u and v, which in this case is a, and we pop from the stack A's as we read v. Since we have more letters in v than in u, after we reach the \$ symbol in the stack at q_2 we transition to q_3. At q_3 we push the "excess" A's onto the stack and upon reaching the # symbol move to q_4. Using a similar logic as we used to process uav, we first push B's onto the stack in state q_4 as we read x. Then we read the sepearting letter of x and y, which in this case is b, and we move to q_5. As we read letters from y we pop B's from the stack until we reach an A. Then we start popping the remaining A's in state q_6. If we can pop the remaining A's we know that $|u| - |v| - |x| + |y| = 0$ for the case $|u| - |v| < 0$ and $|x| - |y| < 0$. Thus, we finally pop the shielding symbol \$ and get to q_7 which is our accepting state.

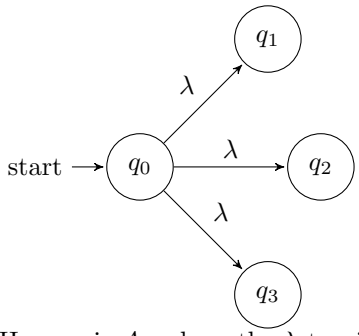
Next we consider the simpler case which is $|u| - |v| = 0$ and $|x| - |y| = 0$. The PDA that recognize this case, say A_2, is given below.



For A_2 the states are same as A_1 except we remove the states that count extra letters as $|u|-|v| = 0$ and $|x|-|y| = 0$. We start at state q_0 which is the empty stack and nothing has been read and at q_1 we push the shielding symbol and start counting the letters of u by pushing A's. After reading the seperating letter of a, we count the letters of v by popping A's in state q_2. We know this case only happens when $|u| - |v| = 0$, thus when we see # we make sure we can pop \$ before moving to q_3 and then push \$ back on the stack. Similarly, for q_3 and q_4 we count the letters of x and y and respectively and if $|x| - |y| = 0$ we can transition into the accepting state q_5 by popping \$.
Finally we consider the case whhere is $|u| - |v| > 0$ and $|x| - |y| > 0$. The PDA that recognize this case, say A_3, is given below.

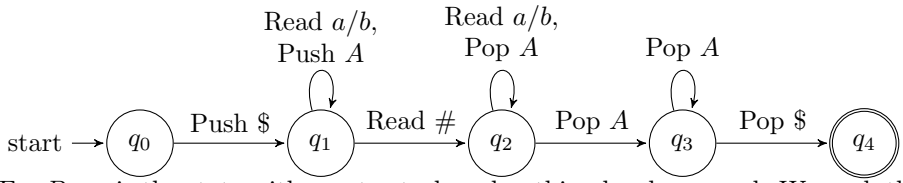


For A_3 again we have a similar construction thus there is no need to explain all the states. However, notice that the transition from q_2 to q_3 guarantees that we only consider the case where $|u| - |v| > 0$ as we pop an A and then push that A back. Then we empty the stack of A's as we read x for state q_3. Finally, q_4 and q_5 checks if the remaining x and all of y have the same amount of letters and if so we move to the accepting state q_6.
We can combine A_1, A_2 and A_3 by taking the union of each PDA to get a PDA, say A_P that recognizes the language A. The PDA A_P is given below.

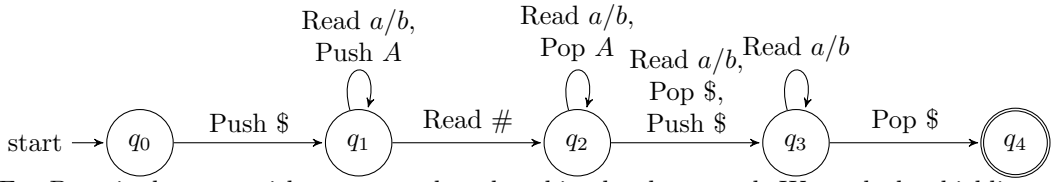


Here q_1 is A_1 where the λ-tranisiton goes to the starting state of A_1 and the starting state of A_1 is no longer a starting state. Similarly, q_2 is A_2 where the λ-tranisiton goes to the starting state of A_2 and the starting state of A_2 is no longer a starting state and q_3 is A_3 where the λ-tranisiton goes to the starting state of A_3 and the starting state of A_3 is no longer a starting state. We keep the accepting states of A_1, A_2 and A_3. Since A_P covers all cases of A, we can say that A_P is a PDA that recognizes A.

ii. Using a similar logic to what we did for i. we can consider two possible cases. The first is when $|w| > |z|$ and the PDA that recognizes this case, say B_1, is given below.

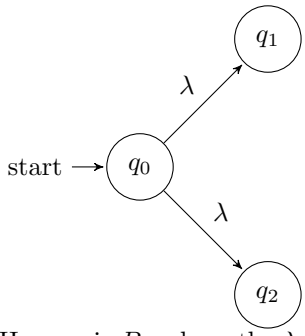


For B_1 q_0 is the state with empty stack and nothing has been read. We push the shielding symbol \$ onto the stack and move to q_1 where we start counting the number of letters of w by pushing A's onto the stack. After seeing the # letter we move to onto q_2 where we start popping A's from the stack for each letter of z. If there is at least one A left in the stack after reading all of z we know that $|w| > |z|$ and we can move to q_3. Here q_3 is necessary since there may be more A's left on the stack which we'd like to empty before popping the \$ and moving to the accepting state q_4. Similarly, we consider the case when $|w| < |z|$ and the PDA that recognizes this case, say B_2, is given below.



For B_2 q_0 is the state with empty stack and nothing has been read. We push the shielding symbol \$ onto the stack and move to q_1 where we start counting the number of letters of w by pushing A's onto the stack. After seeing the # letter we move to onto q_2 where we start popping A's from the stack for each letter of z. If at one point while reading z we get to the shielding symbol \$ and we can read at least one more letter in z we know that $|w| < |z|$ and we can move to q_3. Here q_3 is necessary since there may be more letters left to read for z and we'd like to read all of them before popping \$ and moving to the accepting state q_4.

We can combine B_1, and B_3 by taking the union of each PDA to get a PDA, say B_P that recognizes the language B. The PDA B_P is given below.



Here q_1 is B_1 where the λ-tranisiton goes to the starting state of B_1 and the starting state of B_1 is no longer a starting state. Similarly, q_2 is B_2 where the λ-tranisiton goes to the starting state of B_2 and the starting state of B_2 is no longer a starting state. We keep the accepting states of B_1 and B_2. Since B_P covers all cases of B, we can say that B_P is a PDA that recognizes B.

iii. We can show that $A \cup A' \cup B = C$ where A' is the language defined in the same way as A except the letters a and b are switched and $C = \{s\#t \mid s, t \in \{a, b\}^* \text{ and } s \neq t\}$ by considering the cases where a word is in C. We know that $s \neq t$ if $|s| \neq |t|$. This is simply the language given by B (would also work for the languages given by A and A') and is recognized by the PDA B_P as shown in part ii. Now it may be also true that $|s| = |t|$ but $s \neq t$. Here it is enough to show that s and t differ by one letter. We split this into two further cases with the first being that s has an a somewhere where t has a b in that place. This is the language A since if $(|u| - |v|) = (|x| - |y|)$ then $(|u| - |v|) + |a| = (|x| - |y|) + |b|$ and we are guaranteed that s and t differ in a single position when $|s| = |t|$ by splitting s into uav and t into xby. A is recognized by the PDA A_P as shown in part i. Without loss of generality, we can also consider the converse case where s is ubv and t is xay such that $|s| = |t|$ but $s \neq t$. This is the language given by A' and a relevant PDA $A_{P'}$ can be constructed by making minor modifications to A_P. Thus, we have shown that $A \cup A' \cup B = C$ and a PDA can be constructed to recognize the language given by C by simply taking the union of PDA's that recognize A, A' and B.

(3) We begin by noting that regular expressions are made up from three base cases ϕ (the empty set), λ (the languge containing the empty string) and a (the language containing a). There are also three recursive cases where r and s represent languages R and S respectively and we have $r \cup s$ $(R \cup S)$, $r \circ s$ $(R \circ S)$ and r^* (R^*). Thus, to construct a CFG that transofmrs a regular expression w to a CFG G such that $L(G) = L(w)$, we must handle each recursive case of regular expression. We start off by defining the rule for $a \cup b$, where a and b represent the languages A and B respectively. The corresponding CFG in this case becomes $S \to a$ and $S \to b$ because the union means A or B thus we can either choose the rule leading to a or the rule leading to b. This can be simplified to just $S \to a \,|\, b$. Then we define the rule for $a \circ b$ where a and b represent the languages A and B respectively. The corresponding CFG in this case becomes $S \to ab$ since we wish to concatenate two languages A and B meaning we should concatenate them in the CFG. Finally, for a^* where a represents the language A we will need to define the CFG rule as $S \to \lambda$ and $S \to aS$ because we can either obtain the empty string λ for a^* or we can derive a, aa, aaa... and so on for a^*. This can be simplified to just $S \to aS|\lambda$. For the base cases of regular expressions it is obvious that ϕ becomes $S \to S$ (i.e. the empty set), λ becomes $S \to \lambda$ and a (the single letter) becomes $S \to a$.
Now that we have estabilished all the rules we must combine them together into an algorithm that will translate a regular expression w to a CFG G such that $L(G) = L(w)$. The algorithm will start reading the regular expression from the left (i.e. the beginning) and continue until it encounters one of the recursive cases described above. If a base case is encountered then the algorithm will correctly translate the regular expression to a CFG. However, if a recursive case is seen then we must act according to each case. If the union operator is seen then we recursively run the algorithm on the left of the union and on the right of the union and define a rule for the left and right as described above. Similarly, for concatention we do the same but instead of using the union rule we use the concatenation rule. For the

star operator we apply its rule and apply the algorithm recursively to the expression contained in the star. Thus, as we sweep to the left we make recursive calls on each recursive operation and apply the same left sweeping algorithm to the recursive cases. Furthermore, we must stop sweeping as soon as we make a recursive call to avoid having duplicate rules which will in turn represent a different regular expression. Finally, when no recursive calls can be made we terminate using the base cases and after all calls are finished we will have converted the regular expression into a CFG. As a note, a unique naming scheme may also be used for each rule that goes to another rule to avoid duplicates.
This algorithm works because it works the same way a regular expression is recursively generated by using recursive rules that are defined above and resolving all rules until termination cases are met. Furthermore, it takes care of duplication issues which in turn makes sure our translation is correct. Finally, termination of the algorithm is gauranteed by the base cases defined above.