Winsor Tse, Robert Riccelli, Alexander Plaza, Albert Ezizov, Md Rahman

Dr. Frank Hsu

Theory Of Computation : CISC 4090

29 November 2021

Theory of Computation Final Project: Push-down Automata

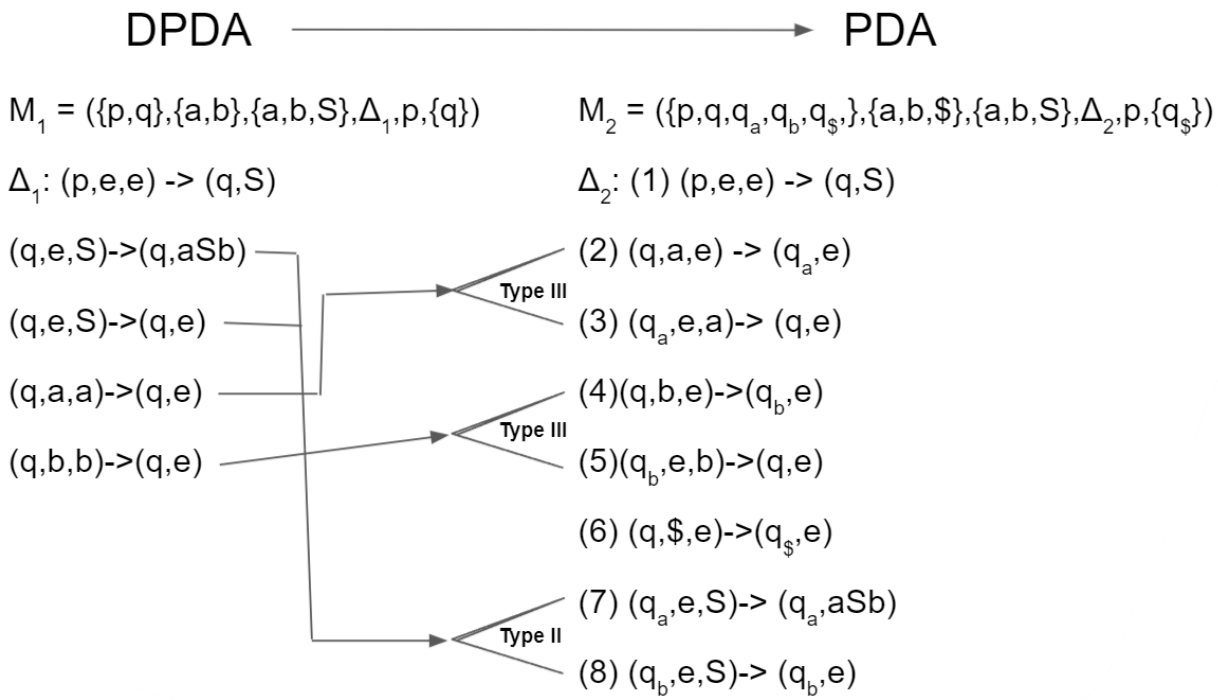
Our task in this semester project was to design and implement a push-down automata that recognizes the context-free language of $L = \{a^n b^n \mid n \geq 0\}$. This push down automata will also need to be deterministic and have the capability to look-ahead one step. The programming language we used was Java since the elements of our automata translated well into common structures like a stack and a hashmap. For the inputs to the automata, we used eight sets of strings including, $a^2b^2\$, a^3b^3\$, a^4b^4\$, a^5b^5\$, a^6b^6\$, a^7b^7\$, a^8b^8\$, a^9b^9\$, a^{10}b^{10}\$$. Our automata is able to recognize these patterns. For the output, we follow the format we use in class where we have a column for step, state, unread input, the top of the stack, the rule being used and the R rule used.

For this problem, we define a Context-Free Grammar G. G is a quadruple G = (V, Σ, R, S), where V is an alphabet, Σ is the set of terminals and a subset of V, R is the set of rules and is a finite subset of (V-Σ) x V*, and S is the start symbol and an element of (V - Σ). Our Context free Grammar consists of G = (V, Σ, R, S), V= {S, a, b, $}, Σ = {a, b, $}, R= {(S -> aSb), (S -> e)}.

There are a few important considerations to make when designing our program. The automaton needs to look ahead and keep track of where it is currently. This is done with two stacks where one keeps its current position and one looks ahead. A linked hashmap is used to keep track of the rules of the automata. The linked hashmap will map an arraylist of our current state and current string value to another arraylist that will contain the next state we should be at, what that current string value is made into, and the rule number that is used. Using these

structures we can efficiently take an input like aabb$ and output its result in a table that is divided into (a) step, (b) state, (c) unread input, (d) (Top of) stack, (e) Δ rule used, and (f) R rule used. The data structures we used are linked hashmap and arraylist. We used 5 strings to fill the hashmap with proper values and we used an Arraylist of strings using two string stacks.

Given the context-free language of $L = \{a^n b^n \mid n \geq 0\}$, we can generate a non-deterministic automaton M_1. We can make M_1 a deterministic automaton if we can predict the next input symbol. We do this by looking ahead and consuming an input symbol ahead of time. We can generate a non-deterministic automaton M_2.



We make some important decisions in our Java program to design our push down automata. First, we need to make sure to import the Java libraries for linked hash map, array list,

and scanner. Furthermore, we use linked hash maps to store the rules for the push down automata. The left of the hashmap is the state and the right is the new state/rule number. Next we define 5 strings to fill in the hashmap with the most common values that will be used. After this, we use fillHashMap() to fill the hashmap based on the given input strings. We defined an integer to record the number of a's and b's in user input and we store an arraylist of test cases to be matched with the user input. We use the function filltestcases() which creates test cases using a for loop from 2 occurrences of a/b to 11 occurrences. Using a scanner we take in the input test case from the user and we check to see if it is a valid input from the user that matches one of our defined test cases. If it is a valid test case we move to start processing the input. We start to output the table and we define some variables to keep track of step number and state. We define a variable input string to keep track of the input string entered by the user. While the input string is not empty, we process the user input and we update the stack. Finally, we format the output and print it to the user. Below we have an excerpt of the main logic of our program, which looks at the state and updates the input string or temp stack (stack2) if the state is p, q or any other state.

```
while(inputstring.isEmpty()==false){

            //current stack to search the hashmap and shorten the
userinput

            String stack2 = "";



            if(state.equals("p")){

                stack2="";

            }

            else if(state.equals("q")){
```

```
                stack2 = Character.toString(inputstring.charAt(0));

            }

            else{

                stack2 = Character.toString(stack1.charAt(0));

            }
```

Below, our program searches the linked hashmap using an arraylist of strings. The search arraylist adds the state and temp stack to search for the elements of the right part of the hashmap and then it stores it.

```
 //Arraylist that stores the right part of the Hashmap eg. [p, ]= *[q, S, 1]

            ArrayList<String> nextStep = new ArrayList<String>();

            //Arraylist that searches Hashmap using the left part of the Hashamp eg. *[p, ]= [q, S, 1]

            ArrayList<String> search = new ArrayList<String>();



            //add the current state and current stack (stack2)

            search.add(state);

            search.add(stack2);

            //searches through the left part of the Hashmap and stores it in nextStep

            nextStep = map.get(search);
```

Below we have the part of the logic where we actually shorten the input string and store it into the stack. We are essentially mimicking the stack.pop() function but instead with the string version. We see that if the state is q, we then shorten/pop the input string and store it into the temp stack. We increase the step number for the output. Then, inputString is officially shortened after passing the if/else. The conditions that follow check if the state is not empty. If it is not, the temp stack is updated to the popping of stack1 (in string form). Else, temp stack is just the stack.

```
if(state.equals("q")){
                stack2  = inputstring.substring(1, inputstring.length());
            }
            else{
                stack2 = inputstring;
            }
            //increase stepnumber
            stepnumber++;
            //inputstring is shortened by stack2 or current stack
            inputstring = stack2;

            if(state.substring(1, state.length()).isEmpty()==false){
                stack2  = stack1.substring(1, stack1.length());
            }
            else{
                stack2 = stack1;
```

```
            }
```

In this part of the program, state is updated by the nextStep arraylist found in the previous step. Stack1 is updated by including the second element of the arraylist and adding temp stack. The following steps are one time conditionals just for output. The last lines are for the ending of the push down automata, where it follows specific steps to end the program.

```
 //get the state eg. [q, S, 1] which would be q
            state = nextStep.get(0);
            //get the stack eg. [q, S, 1] which would be S plus the
current stack (stack2)
            stack1 = nextStep.get(1) + stack2;


            if((nextStep.get(2)).equals("7")){
                R = "S -> aSb";
            }
            else if((nextStep.get(2)).equals("8")){
                R = "S -> e";
            }
            else {
                R = "";
            }


            //Strings used for the end of the PDA
```

```
            String tempInputString = "";
            String tempStacks = "";

            if(inputstring.isEmpty()==false){
                tempInputString = inputstring;
            }
            else{
                tempInputString = "e";
            }

            if(stack1.isEmpty()==false){
                tempStacks = stack1;
            }
            else{
                tempStacks = "e";
            }
            //print function that formats the output
            print_output(Integer.toString(stepnumber), state,
tempInputString, tempStacks, nextStep.get(2), R, n);
```

Our task in this semester project was to design and implement a push-down automata that recognizes the context-free language of $L = \{a^n b^n \mid n \geq 0\}$ and we have implemented that through

our program in Java. After defining the grammar and deterministic pushdown automata, we were able to implement our automaton in Java with some data structures including linked hashmaps, array lists and stacks. Our push-down automata is deterministic and is able to take input ranging from $a^2b^2\$$ to $a^{10}b^{10}\$$ and generate the desired output.