AIN SHAMS UNIVERSITY
FACULTY OF ENGINEERING
International Credit Hours Engineering Programs (i.CHEP)

Automata & Compatibility Assignment Documentation

Submitted to

Prof. Dr. Gamal A. Ebrahim

T.A Eng. Sally E. Shaker

Name & ID:

Name	ID
Ahmed Sameh Mohamed Mourad	*19P5861*

Table of Contents

Table of Figures:

1. DFA:

1.1 Code Documentation

1st Class: DfaState

src > DfaState.java

```java
import java.util.HashMap;

public class DfaState {
    String name;
    boolean isAccepted;

    char[] alphabet = {'0', '1'};

    HashMap<Character, DfaState> nextStates;

    public DfaState(String name, boolean isAccepted) {
        this.name = name;
        this.isAccepted = isAccepted;
    }

    public void addNextStates(DfaState[] states) throws Exception {

        this.nextStates = new HashMap<Character, DfaState>();

        if(states.length != alphabet.length)
            throw new Exception("Length Mismatch !");

        for(int i = 0; i < alphabet.length; i++) {
            nextStates.put(alphabet[i], states[i]);
        }
    }
}

```

Overview

The **DfaState** class represents a state in a Deterministic Finite Automaton (DFA). It contains information about the state's name, whether it is an accepted state, and its transitions to other states based on the input alphabet. The class provides methods to add next states based on the input alphabet. It's used in both questions 1 & 2.

Class Members

Properties

- **name**: A string that represents the name of the DFA state.
- **isAccepted**: A Boolean flag indicating whether the state is an accepted state.
- **alphabet**: A character array containing the valid input symbols for the DFA.

Methods

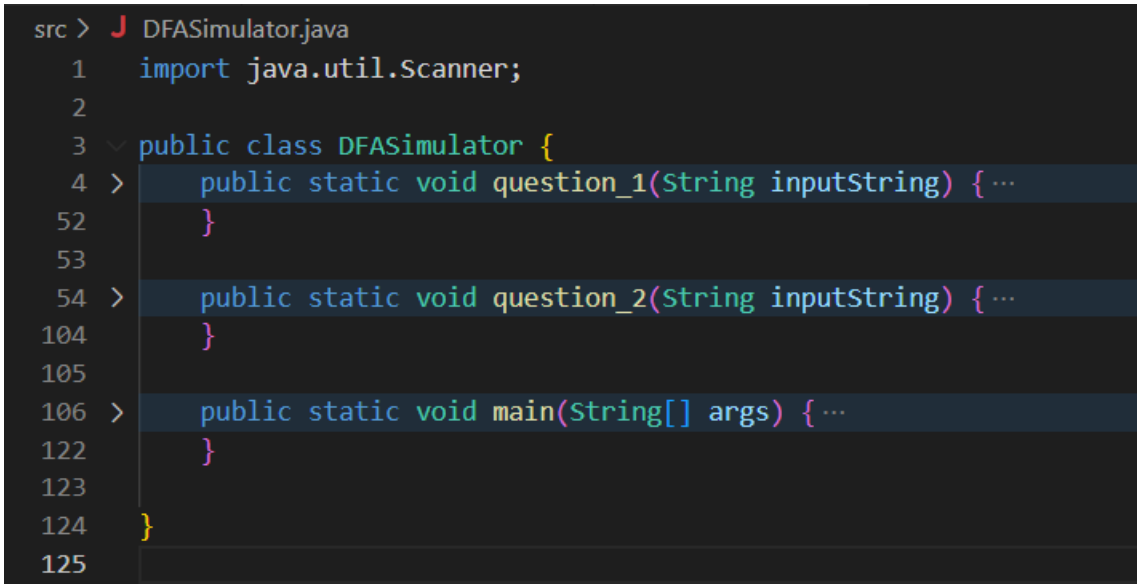
Constructor

- **DfaState(**String name, Boolean isAccepted**)**: Initializes a new instance of the **DfaState** class with the specified name and accepted flag. The constructor sets the initial values for the state's name and accepted flag.

Public Methods

- **void addNextStates(**DfaState[] states**)** throws Exception: Adds the next states for the DFA based on the input alphabet. This method takes an array of **DfaState** objects as input, representing the next states for each symbol in the DFA's alphabet. The method throws an exception if the length of the input states array does not match the length of the DFA's alphabet. The method iterates over the alphabet and associates each symbol with its corresponding next state using a **HashMap**.

2nd Class: DfaSimulator



Overview

The **DFASimulator** class implements a simulator for a DFA (Deterministic Finite Automaton). It provides a main function that interacts with the user to simulate the DFA on input strings.

It also provides two other functions (question_1, question_2) which accept or reject the inputted string.

question_1 accepts even number of 0's and even number of 1's.

question_2 accepts the set of all strings with three consecutive 0's followed by any number of 1’s.

1st Function: question_1()

```java
public static void question_1(String inputString) {

    char[] alphabet = {'0', '1'};

    // Accepted state
    DfaState q0 = new DfaState("q0", true);

    // Rejected states
    DfaState q1 = new DfaState("q1", false);
    DfaState q2 = new DfaState("q2", false);
    DfaState q3 = new DfaState("q3", false);

    try {
        // Transition table for the DFA
        q0.addNextStates(new DfaState[] {q1, q3});
        q1.addNextStates(new DfaState[] {q0, q2});
        q2.addNextStates(new DfaState[] {q3, q1});
        q3.addNextStates(new DfaState[] {q2, q0});
    }
    catch(Exception e) {
        System.out.println(e.getMessage());
        System.exit(-1);
    }

    DfaState curState = q0; // Starting state is q0

    System.out.print("\nTransitions: q0");

    // Traverse the input string and update the current state
    for (char c : inputString.toCharArray()) {
        if (c == '0') {
            curState = curState.nextStates.get('0');
            System.out.print(" -> " + curState.name);
        } else if (c == '1') {
            curState = curState.nextStates.get('1');
            System.out.print(" -> " + curState.name);
        } else {
            System.out.println("Invalid input character: " + c);
            System.exit(0);
        }
    }

    // Check if the final state is accepting
    if (curState.isAccepted) {
        System.out.println("\nThe input string is accepted at state " + curState.name);
    } else {
        System.out.println("\nThe input string is not accepted at state " + curState.name);
    }
}
```

Introduction

The **question_1** function implements a Deterministic Finite Automaton (DFA) for string acceptance. It takes an input string as a parameter and determines whether the string is accepted or rejected based on the defined DFA rules.

DFA Description

The DFA consists of four states: **q0**, **q1**, **q2**, and **q3**. The DFA's alphabet includes two characters: '0' and '1'. The DFA transitions between states based on the characters of the input string.

- **q0** is the initial state of the DFA and an accepted state.
- **q1**, **q2**, and **q3** are rejected states.

Transition Table

The DFA's transition table is defined as follows:

State	Input '0' Transition	Input '1' Transition
q0	q1	q3
q1	q0	q2
q2	q3	q1
q3	q2	q0

Input Processing

The **question_1** function processes the input string character by character. For each character in the input string, it performs the following steps:

1. If the character is '0', the DFA transitions to the state defined by the '0' transition of the current state. The transition is printed as **-> <nextState>**.
2. If the character is '1', the DFA transitions to the state defined by the '1' transition of the current state. The transition is printed as **-> <nextState>**.
3. If the character is neither '0' nor '1', an error message is displayed, indicating an invalid input character, and the program exits.

Output

After processing the entire input string, the function checks the final state to determine if it is an accepting state or a rejected state. The following messages are displayed accordingly:

- If the final state is an accepting state, the message is printed: "The input string is accepted at state **<stateName>**".
- If the final state is a rejected state, the message is printed: "The input string is not accepted at state **<stateName>**".

Error Handling

In case of any length mismatch in the construction of the DFA transition table, an error message is displayed, and the program terminates with an exit code of -1.

If an invalid input character is encountered during the evaluation, an error message is displayed, and the program terminates with an exit code of 0.

2nd Function: question_2()

```java
public static void question_2(String inputString) {

    char[] alphabet = {'0', '1'};

    // Rejected states
    DfaState q0 = new DfaState("q0", false);
    DfaState q1 = new DfaState("q1", false);
    DfaState q2 = new DfaState("q2", false);

    // Accepted states
    DfaState q3 = new DfaState("q3", true);
    DfaState q4 = new DfaState("q4", true);

    try {
        // Transition table for the DFA
        q0.addNextStates(new DfaState[] {q1, q0});
        q1.addNextStates(new DfaState[] {q2, q0});
        q2.addNextStates(new DfaState[] {q3, q0});
        q3.addNextStates(new DfaState[] {q3, q4});
        q4.addNextStates(new DfaState[] {q1, q4});
    }
    catch(Exception e) {
        System.out.println(e.getMessage());
        System.exit(-2);
    }

    DfaState curState = q0; // Starting state is q0

    System.out.print("\nTransitions: q0");

    // Traverse the input string and update the current state
    for (char c : inputString.toCharArray()) {
        if (c == '0') {
            curState = curState.nextStates.get('0');
            System.out.print(" -> " + curState.name);
        } else if (c == '1') {
            curState = curState.nextStates.get('1');
            System.out.print(" -> " + curState.name);
        } else {
            System.out.println("Invalid input character: " + c);
            System.exit(0);
        }
    }

    // Check if the final state is accepting
    if (curState.isAccepted) {
        System.out.println("\nThe input string is accepted at state " + curState.name);
    } else {
        System.out.println("\nThe input string is not accepted at state " + curState.name);
    }
}
```

Overview

The provided Java code implements a deterministic finite automaton (DFA) for evaluating input strings consisting of characters '0' and '1'. The DFA has a set of states, transition rules, and accepts or rejects input strings based on the final state reached after processing the input.

Function Description

The **question_2** function is the main entry point of the DFA string evaluation process. It takes an **inputString** parameter representing the string to be evaluated.

DFA Construction

The DFA is constructed using a set of states, both rejected and accepted. The DFA states are represented by the **DfaState** class, which encapsulates the state's name and acceptance status.

The rejected states are:

- **q0**: Initial state, not accepting.
- **q1**: Intermediate state, not accepting.
- **q2**: Intermediate state, not accepting.

The accepted states are:

- **q3**: Intermediate state, accepting.
- **q4**: Intermediate state, accepting.

Transition Table

The DFA's transition table is defined as follows:

State	Input '0' Transition	Input '1' Transition
q0	q1	q0
q1	q2	q0
q2	q3	q0
q3	q3	q4
q4	q1	q4

The DFA's transition rules are defined by specifying the next possible states for each state and input character combination. The transition table is built by assigning the next states to each state object.

DFA Evaluation

The evaluation of the input string is performed by traversing the characters of the **inputString** and updating the current state based on the transition rules defined in the transition table.

Starting from the initial state **q0**, each character of the input is processed. If the character is '0', the DFA transitions to the next state based on the '0' transition defined for the current state. If the character is '1', the DFA transitions to the next state based on the '1' transition. If the character is neither '0' nor '1', an error message is displayed, and the program terminates.

After processing the entire input string, the final state reached is checked for acceptance. If the final state is an accepting state (**q3** or **q4**), the input string is considered accepted. Otherwise, it is rejected.

Output

During the evaluation process, the DFA's transitions and the final result are printed to the console.

If the input string is accepted, the program displays: "The input string is accepted at state [final_state_name]"

If the input string is not accepted, the program displays: "The input string is not accepted at state [final_state_name]"

Error Handling

In case of any length mismatch in the construction of the DFA transition table, an error message is displayed, and the program terminates with an exit code of -2.

If an invalid input character is encountered during the evaluation, an error message is displayed, and the program terminates with an exit code of 0.

3rd function: main()

```
106     public static void main(String[] args) {
107         System.out.print("Question 1\n");
108 
109         Scanner input = new Scanner(System.in);
110 
111         System.out.print("Enter the input string: ");
112         String inputString = input.nextLine();
113 
114         DFASimulator.question_1(inputString);
115 
116         System.out.print("\nQuestion 2\n");
117 
118         System.out.print("Enter the input string: ");
119         String inputString2 = input.nextLine();
120 
121         DFASimulator.question_2(inputString2);
122     }
123 
124 }
```

Function Description

The **main** method is the entry point for the program. It allows the user to interact with the DFA simulator by providing input strings and executing specific questions.

Flow

1. Print "Question 1" to the console.
2. Create a **Scanner** object named **input** to read input from the user.
3. Prompt the user to enter an input string.
4. Read the input string provided by the user into the **inputString** variable.
5. Call the **question_1** method of **DFASimulator** and pass the **inputString** as an argument.
6. Print a newline character to separate the questions.
7. Print "Question 2" to the console.
8. Prompt the user to enter another input string.
9. Read the second input string provided by the user into the **inputString2** variable.
10. Call the **question_2** method of **DFASimulator** and pass **inputString2** as an argument.

1.2 Output Screenshots

```
Question 1
Enter the input string: 1010

Transitions: q0 -> q3 -> q2 -> q1 -> q0
The input string is accepted at state q0

Question 2
Enter the input string: 00001111

Transitions: q0 -> q1 -> q2 -> q3 -> q3 -> q4 -> q4 -> q4 -> q4
The input string is accepted at state q4
```

Figure 1 shows Accepted Strings in both questions (DFA)

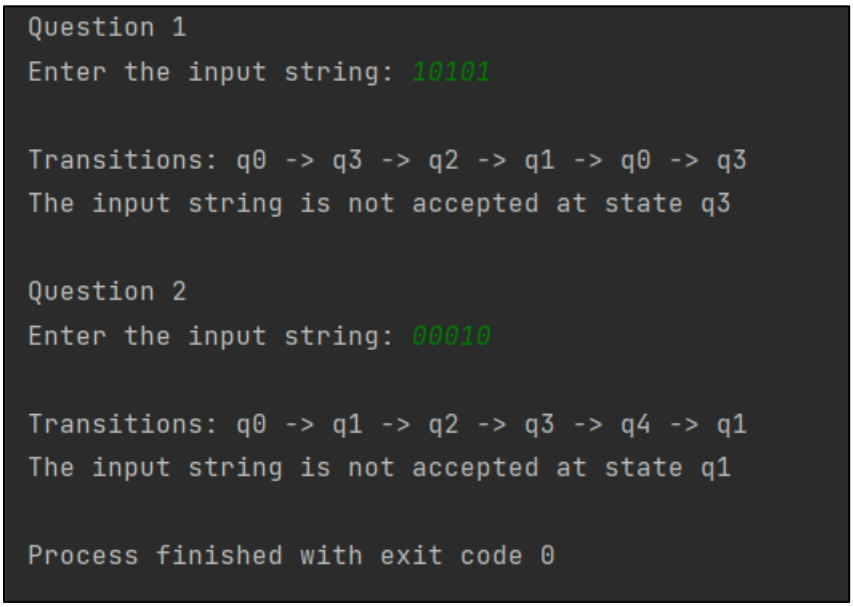


Figure 2 shows Rejected Strings in both questions (DFA)

2. PALINDROME PDA:

2.1 Code Documentation

1st Class: PdaInput

src > PdaInput.java

```java
import java.util.Objects;

public class PdaInput {
    char input;

    char popItem;

    char pushItem;

    public PdaInput(char input, char popItem, char pushItem) {
        this.input = input;
        this.popItem = popItem;
        this.pushItem = pushItem;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        PdaInput pdaInput = (PdaInput) o;
        return input == pdaInput.input && popItem == pdaInput.popItem && pushItem == pdaInput.pushItem;
    }

    @Override
    public int hashCode() {
        return Objects.hash(input, popItem, pushItem);
    }
}
```

The **PdaInput** class represents an input for a Pushdown Automaton (PDA) transition. In a PDA, transitions occur based on the current input symbol, the symbol at the top of the stack, and the symbol to be pushed onto the stack. It encapsulates three properties: **input**, **popItem**, and **pushItem**.

Properties

- **input**: Represents the input character for the PDA transition. It specifies the symbol that is read from the input sequence during the transition.

- **popItem**: Represents the character to be popped from the stack during the PDA transition. It indicates the symbol that should be removed from the top of the stack.
- **pushItem**: Represents the character to be pushed onto the stack during the PDA transition. It specifies the symbol that should be added to the top of the stack.

Constructors

- **PdaInput(**char input, char popItem, char pushItem**)**: Constructs a new **PdaInput** object with the specified input, popItem, and pushItem.

Methods

- **equals(**Object o**)**: Checks if this **PdaInput** object is equal to another object. Returns **true** if the objects are equal, and **false** otherwise.
- **hashCode()**: Returns the hash code value for this **PdaInput** object.

Overall, the **PdaInput** class provides a convenient and encapsulated representation of an input for a PDA transition, allowing you to work with PDA transitions more effectively in the Java programs.

2nd Class: PdaState

```java
import java.util.ArrayList;
import java.util.HashMap;

public class PdaState {
    String name;
    boolean isAccepted;

    char[] alphabet = {'a', 'b'};

    HashMap<PdaInput, PdaState> nextStates;

    ArrayList<PdaInput> transitionStates;

    public PdaState(String name, boolean isAccepted) {
        this.name = name;
        this.isAccepted = isAccepted;
    }

    public void addNextStates(PdaInput[] pdaInputs, PdaState[] pdaStates) throws Exception {

        this.nextStates = new HashMap<PdaInput, PdaState>();

        this.transitionStates = new ArrayList<PdaInput>();

        if(pdaInputs.length != pdaStates.length)
            throw new Exception("Length Mismatch !");

        for(int i = 0; i < pdaStates.length; i++) {
            nextStates.put(pdaInputs[i], pdaStates[i]);

            transitionStates.add(pdaInputs[i]);
        }
    }
```

```java


    public char getPushedItem(PdaInput pdaInput) {
        return pdaInput.pushItem;
    }

    public PdaState getNextState(PdaInput pdaInput) {
        return nextStates.get(pdaInput);
    }

    public String getGrammer(PdaInput pdaInput) {
        return pdaInput.input + ", " + pdaInput.popItem + " -> " + pdaInput.pushItem;
    }
}
```

The **PdaState** class represents a state in a Pushdown Automaton (PDA). It encapsulates properties and methods related to the state and its transitions.

Properties

- **name**: The name of the PDA state.
- **isAccepted**: A boolean value indicating whether the state is an accepted state.
- **alphabet**: An array of characters representing the alphabet of the PDA.
- **nextStates**: A **HashMap** that maps **PdaInput** objects to corresponding next states.
- **transitionStates**: An **ArrayList** containing the **PdaInput** objects representing the transition states.

Constructors

- **PdaState**(String name, boolean isAccepted): Constructs a new **PdaState** object with the specified name and acceptance status.

Methods

- **addNextStates**(PdaInput[] pdaInputs, PdaState[] pdaStates): Adds the next states to the current state. It takes an array of **PdaInput** objects and an array of **PdaState** objects as parameters. Throws an **Exception** if the lengths of the input arrays do not match.
- **getPushedItem(**PdaInput pdaInput**)**: Returns the character to be pushed onto the stack for a given **PdaInput** object.
- **getNextState**(PdaInput pdaInput): Returns the next state for a given **PdaInput** object.

- **getGrammer**(PdaInput pdaInput): Returns a string representing the grammar rule for a given **PdaInput** object. The string is in the format: **input, popItem -> pushItem**.

Functionality

The **PdaState** class represents a state in a PDA and provides methods to manage its transitions and access relevant information. Here's a brief overview of its functionality:

- The class maintains the **name** of the state and a boolean flag **isAccepted** to indicate whether it is an accepted state.
- The **alphabet** property stores an array of characters representing the PDA's alphabet.
- The **nextStates** property is a **HashMap** that maps **PdaInput** objects to the corresponding next states in the PDA. It allows you to define the possible transitions from the current state.
- The **transitionStates** property is an **ArrayList** that stores the **PdaInput** objects representing the transition states. It provides a list of the available transitions from the current state.

Overall, the **PdaState** class allows you to define and manage states in a PDA, handle transitions, and retrieve relevant information about the states and their transitions.

3rd Class: PalindromePDA

```java
import java.util.Scanner;
import java.util.Stack;

public class PalindromePDA {
    private Stack<Character> pdaStack;

    public PalindromePDA() {
        pdaStack = new Stack<Character>();
    }

    public boolean accept(String input) {

        // Rejected states
        PdaState qStart = new PdaState("qStart", false);
        PdaState qLoop = new PdaState("qLoop", false);
        PdaState q1 = new PdaState("q1", false);
        PdaState q2 = new PdaState("q2", false);
        PdaState q3 = new PdaState("q3", false);
        PdaState q4 = new PdaState("q4", false);
        PdaState q5 = new PdaState("q5", false);

        // Accepted state
        PdaState qAccept = new PdaState("qAccept", true);
```

```java

        try {
            // Transition table for the PDA
            qStart.addNextStates(new PdaInput[]{new PdaInput('ε', 'ε', '$')}, new PdaState[] {
                    q1
            });

            q1.addNextStates(new PdaInput[]{new PdaInput('ε', 'ε', 'S')}, new PdaState[] {
                    qLoop
            });

            q2.addNextStates(new PdaInput[]{new PdaInput('ε', 'ε', 'S')}, new PdaState[] {
                    q3
            });

            q3.addNextStates(new PdaInput[]{new PdaInput('ε', 'ε', 'a')}, new PdaState[] {
                    qLoop
            });

            q4.addNextStates(new PdaInput[]{new PdaInput('ε', 'ε', 'S')}, new PdaState[] {
                    q5
            });

            q5.addNextStates(new PdaInput[]{new PdaInput('ε', 'ε', 'b')}, new PdaState[] {
                    qLoop
            });
```

```
            qLoop.addNextStates(new PdaInput[]{
                    new PdaInput('ε', '$', 'ε'),
                    new PdaInput('ε', 'S', 'a'),
                    new PdaInput('ε', 'S', 'b'),

                    new PdaInput('ε', 'S', 'a'),
                    new PdaInput('ε', 'S', 'b'),
                    new PdaInput('ε', 'S', 'ε'),
                    new PdaInput('b', 'b', 'ε'),
                    new PdaInput('a', 'a', 'ε'),

            }, new PdaState[] {
                    qAccept,
                    q2,
                    q4,

                    qLoop,
                    qLoop,
                    qLoop,
                    qLoop,
                    qLoop,
            });
        }
        catch(Exception e) {
            System.out.println(e.getMessage());
            System.exit(-1);
        }
```

```
        PdaState curState = qStart;

        char pushedItem = '$';

        pdaStack.push(pushedItem);
        curState = curState.getNextState(new PdaInput('ε', 'ε', pushedItem));

        System.out.println("\nTransitions: " + curState.getGrammer(new PdaInput('ε', 'ε', pushedItem)));

        pushedItem = 'S';

        pdaStack.push(pushedItem);
        curState = curState.getNextState(new PdaInput('ε', 'ε', pushedItem));

        System.out.println(curState.getGrammer(new PdaInput('ε', 'ε', pushedItem)));
```

```
96          while(input.length() > 0) {
97              char symbol = input.charAt(0);
98
99              if(curState == qLoop) {
100                 if(symbol == pdaStack.peek()) {
101                     curState = curState.getNextState(new PdaInput(symbol, symbol, 'ε'));
102
103                     System.out.println(curState.getGrammer(new PdaInput(symbol, symbol, 'ε')));
104
105                     pdaStack.pop();
106
107                     // remove the first character of the original string
108                     input = input.substring(1);
109                 }
110
111                 else {
112                     if(pdaStack.peek() != 'S') {
113                         return false;
114                     }
115
116                     if(input.length() == pdaStack.size() - 2) {
117                         curState = curState.getNextState(new PdaInput('ε', pdaStack.peek(), 'ε'));
118
119                         System.out.println(curState.getGrammer(new PdaInput('ε', pdaStack.peek(), 'ε')));
120
121                         pdaStack.pop();
122                     }
123
124                     else if(input.length() < pdaStack.size()) {
125                         curState = qLoop;
126
127                         System.out.println(curState.getGrammer(new PdaInput('ε', pdaStack.peek(), symbol)));
128
129                         pdaStack.push(symbol);
130                     }
```

```
132                     else {
133                         if(symbol == 'a') {
134                             curState = q2;
135                         }
136                         else if(symbol == 'b') {
137                             curState = q4;
138                         }
139
140                         System.out.println(curState.getGrammer(new PdaInput('ε', pdaStack.peek(), symbol)));
141
142                         pdaStack.pop();
143
144                         pdaStack.push(symbol);
145                     }
146                 }
147             }
148
149             else {
150                 System.out.println(curState.getGrammer(curState.transitionStates.get(0)));
151
152                 pdaStack.push(curState.getPushedItem(curState.transitionStates.get(0)));
153
154                 curState = curState.getNextState(curState.transitionStates.get(0));
155             }
156         }
157
158         return pdaStack.peek() == '$';
159     }
```

Introduction

This documentation provides an overview and explanation of the Java code for the Palindrome Pushdown Automaton (PDA) program. The code is designed to determine whether an input string is a palindrome using a PDA implemented with a stack.

Code Description

The Java code consists of a single class named **PalindromePDA**. The class implements the logic for the palindrome recognition using a PDA.

Class Structure

The **PalindromePDA** class contains the following components:

1. **Fields:**
    - **pdaStack**: An instance variable of type **Stack<Character>** to represent the stack used in the PDA.
2. **Constructor:**
    - **PalindromePDA()**: Initializes the PDA stack by creating a new instance of **Stack<Character>**.
3. **Methods:**
    - **accept**(String input): This method takes an input string as a parameter and returns a boolean value indicating whether the input is a palindrome. It implements the palindrome recognition logic using a PDA approach.
    - **main**(String[] args): The main method of the class, which allows the program to be executed. It prompts the user to enter an input string and calls the **accept** method to check whether it is a palindrome.

4. **Helper Classes:**

   - **PdaState**: A helper class representing a state in the PDA. Each state has a name and a flag indicating whether it is an accepting state.
   - **PdaInput**: A helper class representing an input for the PDA transitions. Each input has three characters: a read symbol, a pop symbol, and a push symbol.

PDA Implementation

The PDA implementation in the **accept** method involves defining PDA states, transition rules, and processing the input string. The states are represented by instances of the **PdaState** class, and the transitions are defined by adding next states to each state with corresponding input symbols.

The **accept** method executes the PDA transitions based on the input string. It starts with an initial state (**qStart**) and processes the input symbols one by one, updating the current state and stack accordingly. The method keeps track of the PDA's stack operations and outputs the transitions as it progresses.

After processing the entire input string, the method checks whether the PDA stack contains only the initial stack symbol ('$'). If it does, the input string is considered a palindrome, and the method returns **true**; otherwise, it returns **false**.

1st function: accept()

Description: This method determines whether the given input string is accepted by a pushdown automaton (PDA) for checking palindromes. It uses a set of states and transitions to simulate the behavior of the PDA. The method returns **true** if the input string is accepted, and **false** otherwise.

Parameters:

- **input**: The input string to be checked for palindrome acceptance.

Return Type: **boolean**

- **true**: If the input string is accepted as a palindrome.
- **false**: If the input string is not accepted as a palindrome.

Explanation:

- Define and initialize multiple **PdaState** objects representing the various states of the PDA, including the rejected states (**qStart**, **qLoop**, **q1**, **q2**, **q3**, **q4**, **q5**) and the accepted state (**qAccept**).
- Set up the transition table for the PDA by specifying the valid inputs and the corresponding next states for each state in the PDA.
- In a **try-catch** block, attempt to add the transitions to the states. If an exception occurs, print the error message and exit the program with a status code of -1.
- Set the current state (**curState**) to **qStart**.
- Push the item **$** onto the PDA stack and update **curState** based on the transition specified by the input **(ε, ε, '$')**.
- Print the transition and grammar associated with the current state and input **(ε, ε, '$')**.
- Push the item **S** onto the PDA stack and update **curState** based on the transition specified by the input **(ε, ε, 'S')**.
- Print the transition and grammar associated with the current state and input **(ε, ε, 'S')**.
- Enter a loop that iterates until the input string is empty.
    - Extract the first character (**symbol**) from the input string.
    - If **curState** is **qLoop**, check various conditions and perform corresponding actions based on the symbol, stack top, and input length.
    - Otherwise, print the transition and grammar associated with the first transition of **curState**.
    - Push the item obtained from **curState.getPushedItem()** onto the PDA stack.
    - Update **curState** based on the first transition of **curState**.

- Remove the first character from the input string.

- Return **true** if the top of the PDA stack is **$**, indicating the input string is accepted as a palindrome; otherwise, return **false**.

2nd function: main()

```
161     public static void main(String[] args) {
162         PalindromePDA pda = new PalindromePDA();
163         Scanner scanner = new Scanner(System.in);
164 
165         System.out.print("Enter an input string: ");
166         String input = scanner.nextLine();
167 
168         if (pda.accept(input)) {
169             System.out.println("The input string is Accepted");
170         } else {
171             System.out.println("The input string is Rejected");
172         }
173     }
```

Function Description

The **main** method is the entry point for the program. It allows the user to interact with the PDA simulator by providing input strings and checking them.

Flow

1. Create a new instance of **PalindromePDA** called **pda**. This class represents a pushdown automaton for checking palindromes.
2. Create a new instance of **Scanner** called **scanner** to read user input from the command line.
3. Prompt the user to enter an input string by displaying the message **"Enter an input string: "** using **System.out.print**.
4. Read the input string entered by the user using **scanner.nextLine()** and store it in the variable **input**.

5. Call the **accept** method of the **PalindromePDA** object **pda** and pass the **input** string as an argument.
6. If the **accept** method returns **true**, print the message **"The input string is Accepted"** using **System.out.println**.
7. If the **accept** method returns **false**, print the message **"The input string is Rejected"** using **System.out.println**.

2.2 Output Screenshots

```
Enter an input string: abba

Transitions: ε, ε -> $
ε, ε -> S
ε, S -> a
ε, ε -> S
ε, ε -> a
a, a -> ε
ε, S -> b
ε, ε -> S
ε, ε -> b
b, b -> ε
ε, S -> ε
b, b -> ε
a, a -> ε
The input string is Accepted
```

Figure 3 shows an Accepted String (PDA).

```
Enter an input string: abbbaa

Transitions: ε, ε -> $
ε, ε -> S
ε, S -> a
ε, ε -> S
ε, ε -> a
a, a -> ε
ε, S -> b
ε, ε -> S
ε, ε -> b
b, b -> ε
ε, S -> b
ε, ε -> S
ε, ε -> b
b, b -> ε
ε, S -> ε
b, b -> ε
The input string is Rejected
```

Figure 4 shows a Rejected String (PDA).