1. Introduction to the Problem

This document details the design and implementation of a Pushdown Automata (PDA) simulation. This projects objective is to simulate a PDA's operations, taking as input a set of states, a stack and input alphabet, transition functions, start states, and final states as input, then simulating the PDA for given strings. The output includes the path traversed and the results, which are written to an output file.

2. Design Considerations

The design adopts an Object-Oriented approach to ensure a clear representation of the PDA's components. Emphasis was placed on readability and maintainability. The PDA class encapsulates all pertinent data and behaviors, and is structured to facilitate future enhancements and scalability. The current design uses very little memory and since the program is written in C++ with most of the operations being basic standard library methods, the runtime can be considered as near optimal as well.

In this project i used multiple structures:

- **PDA:** encapsulates the logic of the PDA with all the data necesessary.
- **Simulator:** Simulates the PDA on all of the given strings in the input file and writes the results to the output file.

3. Data Structures

- **Data Structures**:
    - `vector<string>`: Dynamic memory allocation was necessary for storing states and variables as we were reading the file.
    - `unordered_set<string>`: Utilized for final state look-up to optimize lookup times as last state's inclusion in the final states determined acceptance.
    - `map<tuple<string, char, char, char>, vector<string>>`: Enables efficient mapping from current states combined with the consumed input, the top of the stack, and what is being pushed to the stack, to subsequent set of states.
    - `stack<char>:` For stack operations.

4. Algorithm: The simulation operates as follows:

- **Base Case Checks**: It first checks if the end of the input string is reached and if the stack is empty. If these conditions are met along with the current state being a final state, the input is accepted.
- **Transition Processing**: The algorithm iteratively explores all possible transitions from the current state, based on the input character and the top of the stack.
- **Stack Operations**: Depending on the transition, it may pop from or push symbols onto the stack.
- **Recursive Calls**: For each viable transition, it updates the current state and recursively calls itself with the new state and the remaining input string (or the same string if the transition does not consume an input character).
- **Path Tracking**: Throughout this process, it keeps track of the path of states traversed.
- **Backtracking**: If a transition does not lead to an accepting state, the algorithm backtracks, undoing the transition and trying other possibilities.
- **Rejection**: If we are stuck in the current state the algorithms return a rejection.

In summary it is a recursive algorithm that explores all possible paths in the PDA based on the input string, stack operations, and transitions. It returns whether the input string is accepted by the PDA and the path of states traversed during the simulation.

5. Input/Output Format

- **Input**: A text file structured according to the project's outlined format. Notably, the description did not provide a suggestion to the reader on how or when should we stop reading the transitions and move on to the strings to be tested. That is why i used a simple END signal at the end of the transitions to specify where should the program stop treating the inputted characters as transitions.

- **Output**: An output file that lists the sequence of visited states for each input string, followed by an indication of the string's acceptance or rejection.

6. Testing

The program was tested against various PDAs to ensure robust handling of:

- Multiple alphabet sizes.
- Diverse sets of states and final states.
- Edge cases such as empty strings and empty input files.
- Incorrect input file formats.
- PDA's with different languages such as { 0^n 1^(2n) }
- PDA's with multiple transitions with the same signature (current state, pop , push) with different next states.

7. Error Handling

- **File I/O**: Verifies if opening of input and output files resulted in succes otherwise reports the failure to the user.

8. Code Structure

- **main()**: Manages the PDA and Simulator instance creation.

- **struct PDA**: Represents the PDA, holding all data and logic.

- **struct Simulator:** Simulates the PDA for all of the given strings in the input file.

- **Helper Functions**: `print(auto container)` for generic printing, and `print(const map<tuple<string, char,char,char>, vector<string>>& myMap)` specifically for the transition map.

Attributes of PDA:

- `number_of_states` (int): Total number of states in the PDA.
- `number_of_input_variables` (int): Number of input variables.
- `number_of_stack_variables` (int): Number of stack variables.
- `number_of_finals` (int): Number of final states.
- `states` (vector<string>): Stores all states.
- `final_states` (unordered_set<string>): Set of final states.
- `input_variables` (vector<char>): Input variables used in PDA.
- `stack_variables` (vector<char>): Stack variables used in PDA.
- `transitions` (map<tuple<string,char,char,char>,vector<string>>): Defines state transitions.
- `current` (string): Current state.
- `start` (string): Starting state.
- `initial_stack_symbol` (char): Initial stack symbol.
- `actions` (vector<string>): Actions for the PDA to perform.
- `pda_stack` (stack<char>): Stack used to simulate the PDA's memory.

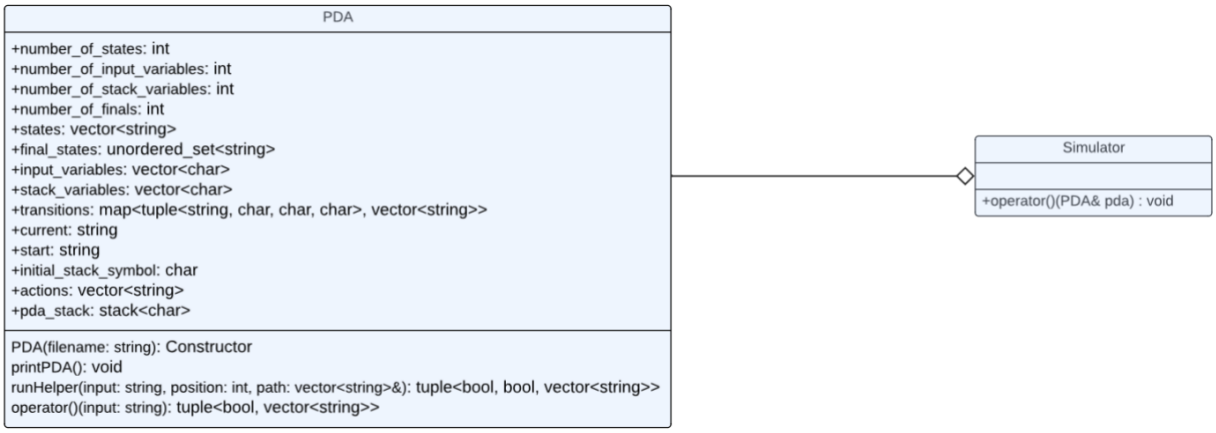
Methods of PDA:

- `PDA(string filename)`: Constructor to initialize PDA from a file.
- `printPDA()`: Prints the current state of the PDA.
- `runHelper(string, int position, vector<string>&)`: Recursive helper function for processing input strings.
- `operator()(const string&)`: Executes the PDA with a given input string.

Simulator Structure:

- `operator()(PDA& pda)`: Simulates the PDA with all the given strings and writes the path taken and the end result to an Output file.

The *Class Diagram* of the Program.



.