Project 0

Go + Pushdown Automata

A Pushdown Automaton (PDA) is a simple computing device capable of performing quite interesting and powerful computations (eg parsing programming languages). Computer Scientists design PDAs to recognize whether a token-stream is in some desired language (set of token-streams).

In this project, you will implement a PDA processor in Go.

Preliminaries: PDA specifications and operation.

A PDA is a Finite State Automaton (FSA) endowed with an unlimited-size stack. The stack essentially extends the finite amount of memory available to the FSA in the form of finite set of possible states for its *state control* (domain of values for the automaton's single register).

Every PDA processes token-streams and has a state control, input and stack alphabets, finite set of states, start and accepting states, a transition function, and an initially empty stack as described below.

Consider a PDA.

The PDA has a finite set of *states* its state control may be at, a finite set of valid input tokens (its *input alphabet*), a finite set of valid tokens that can be placed in the stack (its *stack alphabet*), and designates one state as the *start* state and a subset of states as *accepting*. The state of its state control and token at the top of its stack, after the latest transition (or the start state if no transitions were made), are called its *current state* and *current top* (stack) token respectively.

The PDA processes an input sequence of tokens (*token-stream*) as follows. The PDA is presented with a single token at a time from the input token-stream in left-to-right order. The PDA, upon presented with the *current* (next) input token, inspects its current state and top stack token, makes a transition as determined by its transition function, and consumes the current input token. Another input token may be presented to the PDA only after the latest presented token is consumed.

The *transition function* (program/code) of the PDA specifies, based on the *current configuration* (state, input token, top stack token), the next state of the PDA's control, and the token that replaces the current top stack token. If the transition function does not provide a transition to take from the current configuration, the PDA *hangs* (eg. is trapped in a special non-accepting (exception) state, refuses to consume any further input tokens, crashes, fail-stops, etc).

Two special tokens are assumed be in all PDA input and stack alphabets: the ε token which stands for the null (empty) token, and the $ token that signifies the end of input token-stream. When a PDA replaces a token c at the top of its stack with ε, the PDA effectively pops c from its stack; when the PDA "pretends" that ε is at the top of its stack and replaces it with a token c, the PDA effectively pushes c to the top of the stack. When the PDA "pretends" to have been presented with ε as the current input token, the PDA effectively makes a transition without consuming any actual input token.

We refer to an input token-stream presented to and consumed by the PDA as a *valid* input token-stream. We refer to a valid input token-stream that causes the PDA to end up at an accepting state with a completely empty stack as an *accepted* token-stream; all other input token-streams are called *rejected*. The set of accepted input token-streams constitute the *language recognized (accepted)* by the PDA.

For additional details on PDAs, refer to:

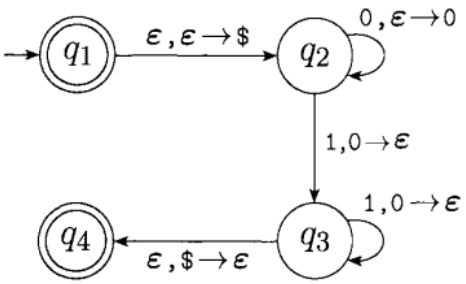
- Chapter 2.2 from Introduction to Theory of Computation, Michael Sipser 2nd edition

- https://www.geeksforgeeks.org/introduction-of-pushdown-automata/

Example: HelloPDA.
Suppose we seek a PDA that accepts all strings that start with 0s followed by an equal number of 1s, ie a PDA that recognizes the language L = $\{0^n1^n \mid n >= 0\}$.

Intuitively, at a high-level, such a PDA can accept input token-streams in L by doing roughly the following: start by pushing each encountered 0 to the stack; upon encountering the first 1, change behaviour, and pop a 0 off the top of the stack for each encountered 1.

The precise (graphical) specification of such a PDA is given in the diagram below. In the diagram, circles correspond to states, double circles indicate accepting states, an incoming arrow marks the start state, directed edges between circles labeled x,y->z indicate the transitions in the transition function (where edge's source and destination indicate the current and next state of the transition, and x, y, and z are the current input token, current top stack token, token that will replace the current top stack token respectively).



A plausible (suggested) JSON specification (encoding of the diagram) for the PDA is

```
{"name": "HelloPDA",
 "states": ["q1", "q2", "q3", "q4"],
 "input_alphabet": ["0", "1"],
 "stack_alphabet" : ["0", "1"],
 "accepting_states": ["q1", "q4"],
 "start_state": "q1",
 "transitions": [
        ["q1", null, null, "q2", "$"],
        ["q2", "0", null, "q2", "0"],
        ["q2", "1", "0", "q3", null],
        ["q3", "1", "0", "q3", null],
        ["q3", null, "$", "q4", null]],
 " eos": "$"}
```

What to do:

Task 1.
Implement a GoLang class for executing/simulating PDAs. We refer to instances of your class as *PDA processors*.
Your class should provide the following methods (API):

- **open(spec):** load and parse/process spec as the JSON specification string of a PDA. Return True on success.
- **reset()**: initialize the PDA to be at its start state with an empty stack.
- **put(token):** present token as the current input token to the PDA. The PDA consumes the token, takes appropriate transition(s), and returns the #transitions taken due to this put() call.

- **eos():** announce the end of input token-stream to the PDA.
- **is_accepted():** return True if PDA is currently at an accepting state with empty stack; False otherwise.
- **peek(k):** return up to k stack tokens from the top of the stack (default k=1) without modifying the stack.
- **current_state():** return the current state of the PDA's control.
- **close():** garbage-collect/return any (re-usable) resources used by the PDA.

The methods reset(), eos(), and close() do not return any values to their caller.
Your class implementation may throw appropriate Exceptions when errors occur.

Task 2.

Implement a driver GoLang program that uses your class to simulate the computation of a PDA on an input token-stream. Your driver should

- read the JSON specification of a PDA from a file whose pathname is provided as the first mandatory command-line argument
- read the character string of the marshalled input token-stream either from a file whose pathname is provided as an optional second command-line argument or from the standard-input.
- print on the standard-output whether the PDA accepted/rejected the input token-stream
- print on the standard-error the return values of the current() and top(5) calls after a done() call
- print on the standard-error appropriate text messages for any exceptions thrown by the calls to your class methods.
- for clarity, each printed message should be meaningfully tagged with the name of the PDA, call number of method call, and method name that the message is associated with (eg "pda=HelloPDA:call_no=123:method=is_accepted:: True" and so on)

What to submit:

Submit a .tar.gz archive with your

- complete GoLang code
- sample input files (with PDA specs or marshalled token-streams)
- a Bash script file with commands demonstrating the execution of your driver with different command-line arguments
- two files capturing the standard-output and standard-error of the execution of your script
- README file (with relevant documentation and usage guidance)

Project-specific assumptions.

For the purposes of tasks of this project, you may make the following assumptions:

1. PDAs are deterministic (eg, at most one transition from a current state, input, top of stack token triplet) . In general, PDAs are non-deterministic devices.
2. PDAs are eager and instantaneously take all possible transitions from a current state that do not need the consumption of any input tokens.
3. All (non-special) tokens in the input and stack alphabets are sequences of ASCII (printable) non-whitespace characters.
4. A token-stream is marshalled by joining the sequence of tokens with a whitespace into an ASCII string
5. A token-stream is unmarshalled from an ASCII string by tokenizing (splitting) it on contiguous whitespaces (ie. consecutive whitespaces are effectively suppressed into one whitespace).
6. PDA JSON specifications follow the structure of the example PDA above.
7. PDA JSON specifications are well-formed and valid.