Go + Pushdown Automata

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

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

Preliminaries: PDA concepts, 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 control (domain of values for the automaton’s single control register).

Every PDA has a finite set of states its 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. A PDA has a control (register) that has as value one of its states. The state of the PDA’s control and the 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 stack-top token respectively.

A 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 stack-top 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, stack-top token), the next state of the PDA’s control, and the token that replaces the current stack-top token. If the transition function does not provide a transition to take from the current configuration, the PDA hangs (e.g. is trapped in a special non-accepting (exception/crash) state and ignores any further input tokens).

Two special tokens, $\varepsilon$ and $\$, are assumed be in every PDA input and stack alphabet. The $\varepsilon$ token stands for the null (empty) token. The $\$ token that signifies the end of input token-stream or the bottom of the stack. When a PDA replaces a token $c$ at the top of its stack with $\varepsilon$, the PDA effectively pops $c$ from its stack; when the PDA “pretends” that $\varepsilon$ 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 $\varepsilon$ as the current input token, the PDA effectively makes a transition without consuming any actual input token.

An input token-stream presented to and consumed by the PDA is called a valid input token-stream. A valid input token-stream that causes the PDA to end up at an accepting state with a completely empty stack is called 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

Example: HelloPDA.
Suppose we seek a PDA that accepts all strings that start with 0s followed by an equal number of 1s, i.e. a PDA that recognizes the language $L = \{0^n1^n \mid n \geq 0 \}$. 
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 behavior, 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 this diagram, circles correspond to states, double circles indicate accepting states, an incoming arrow marks the start state, directed edges between circles labeled $x,y\rightarrow z$ 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 stack-top token, and token that will replace the current stack-top respectively).

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

```json
{"name": "HelloPDA",
  "states": ["q1", "q2", "q3", "q4"],
  "inputAlphabet": ["0", "1"],
  "stackAlphabet": ["0", "1"],
  "acceptingStates": ["q1", "q4"],
  "startState": "q1",
  "transitions": [
    ["q1", "", "", "q2", "$"],
    ["q2", "0", "", "q2", "0"],
    ["q2", "$", "0", "q3", ""]
  ],
  "eos": "$"}
```

What to do:

**Task 1.**
Implement a GoLang “class” (i.e. GoLang struct and associated functions methods) for executing/simulating PDAs. We refer to instances of your class as PDA processors. Each PDA processor has a source code program which is its JSON specification. Your class should have appropriate fields for the PDA’s control, stack, and source code. Further, your each PDA processor has a “clock” (initialized to 0) that ticks (advances by 1) after each transition taken.

Note the natural analogy/correspondence between the concepts of a program (in a high-level programming language) and a process (in an Operating System) and that of the PDA specification and PDA processor.

Your class should provide the following methods (API):
isValid()  
Return true iff its source code is valid, i.e. the start and accepting states are among the PDA’s states, and the values of the individual transitions are proper (i.e. among the PDA’s states or appropriate input or stack alphabets).

reset()  
initialize the PDA so that its control is at its start state, its stack is empty, its clock is 0. No return value.

clock()  
Return the current clock value (#transitions taken since latest reset)

control()  
Return the current state of the control.

peek(k)  
Return a sequence of (up to) k consecutive stack tokens without modifying the stack, and with the stack-top as the last token in the sequence. If k < 0, return all the stack tokens.

isAccepted()  
Return true iff the control is at an accepting state and the stack is empty.

open(spec)  
Process the given JSON specification string and set appropriate values for the PDA processor’s fields. Call reset(). No return value.

source()  
Return the JSON specification of the PDA.

feed(token)  
Present to 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 feed() call.

noMore()  
Announce the end of the input token-stream to the PDA. Return the #transitions that PDA takes (without consuming any further input tokens).

close()  
Garbage-collect/return any (re-usable) resources used by the PDA. No return value.

Your implementation may raise appropriate error/exceptions as needed.

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 an input token-stream from the standard-input.
- print on the stdout (standard-output) whether the PDA accepted/rejected the input token-stream
- print on the stderr (standard-error) the return values of the clock(), control(), and peek(3) calls after a noMore() call
- print on the stderr appropriate messages for any exceptions raised or transitions taken when calling API methods.
- for clarity, each message should be meaningfully tagged with the name the method called, the PDA, its current clock, control, stack-top, input token fed, #transition(s) taken, and return value as applicable (e.g. “pda=HelloPDA:method=isAccepted:clock=123:control=q3:stacktop=$:token=0:ntrans=0:return=false” 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; include the names and emails of all those that contributed to the project)
Project-specific assumptions.
For the purposes of tasks of this project, you may make the following assumptions:
1. PDAs are deterministic (e.g., at most one transition from a current state, input, stack-top triplet). In general, PDAs are non-deterministic devices.
2. A PDA is initially lazy until fed an input token or told no more tokens, when it becomes eager. A lazy PDA takes transitions only when fed a token. An eager PDA can instantaneously take all possible transitions that do not need the consumption of any tokens and do not pop the $ token from the stack (unless explicitly allowed).
3. An eager PDA that is told no more input is explicitly allowed to pop the $ token from the stack.
4. All (non-special) tokens in the input and stack alphabets are sequences of ASCII (printable) non-whitespace characters.
5. A token-stream is marshalled into an ASCII string by joining the sequence of tokens with a whitespace.
6. A token-stream is unmarshalled from an ASCII string by tokenizing (splitting) it on contiguous whitespaces (i.e. consecutive whitespaces are effectively suppressed into one whitespace).
7. PDA JSON specifications follow the structure of the example PDA above.
8. PDA JSON specifications are well-formed and valid.