CENG 242

Homework #3

(Due: April 15th 2011, Friday 23:55)

In this homework, you will write an abstract data type, named DPDA, representing a deterministic pushdown automaton and provide interface functions for its manipulation and simulation.

A deterministic pushdown automaton is a finite state machine that can make use of a stack. A pushdown automaton can be represented as a 7-tuple $P = (Q, \Sigma, \Gamma, \delta, q_0, Z, F)$ where

- Q is a finite set of states.
- Σ is a finite set of input alphabet.
- Γ is a finite set of stack alphabet.
- $\delta: Q \times \Sigma_\varepsilon \times \Gamma \to P(Q \times \Gamma^*)$ where $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$, is the transition function. It represents the transitions between states.
- q_0 is the initial state, $q_0 \in Q$.
- Z is the initial stack symbol.
- F is the set of accepting states $F \subset Q$.

State transitions occur based on the current state, current input and the value at the top of the stack. Since DPDA is deterministic, there exists at most one transition from a state for each input, stack symbol tuple. The transition logic is follows:

The tuple (*p, a, A, q, B*), where $p, q \in Q$, $a \in \Sigma_\varepsilon$, $A \in \Gamma$, and $B \in \Gamma^*$ represents a transition from state *p* to state *q* when *a* is the current input and *A* is the topmost stack symbol, replacing *A* with *B* after the transition. If $B = \varepsilon$ (if B is the empty string), nothing is pushed onto the stack. Note that the topmost element is always popped from the stack in a transition, and a string (which may be empty) is pushed onto the stack. When stack contents are represented as a string at, $a \in \Gamma$, $t \in \Gamma^*$, the leftmost character a represent the element at the top, and string t represents the remaining elements in stack.

In this homework, the DPDA's won't have ε-transitions (i.e. transitions where $a = \varepsilon$). This assumption holds for both manipulation and simulation of DPDA's and you do not have to consider ε-transitions in your representation of DPDA's.

A deterministic pushdown automaton (without ε-transitions) $M = (Q, \Sigma, \Gamma, \delta, q_0, Z, F)$ computes as follows. It accepts input ω if ω can be written as $\omega = \omega_1 \omega_2 \ldots \omega_m$, where each $\omega_i \in \Sigma$ and sequences of states $r_0 r_1 \ldots r_m \in Q$ and strings $s_0 s_1 \ldots s_m \in \Gamma^*$ exist that satisfy the following three conditions. The strings s_i represent the sequence of stack contents that M has on the accepting branch of the computation.

1. $r_0 = q_0$ and $s_0 = Z$. This condition signifies that M starts out properly, in the start state and with an empty stack.

2. For $i = 0,1, \dots, m-1$, we have $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$ where $s_i = at$ and $s_{i+1} = bt$ for some $a \in \Gamma$, $b \in \Gamma^*$ and $t \in \Gamma^*$. This condition states that M moves properly according to the state, stack, and next input symbol.
3. $r_m \in F$. This condition states that an accept state occurs at the input end.

You are not given the internals of DPDA type; you are free to use any possible representation for pushdown automata. However, you should strictly obey to the interface specifications.

Your module should look like

```
module Hw3 (DPDA, interface functions) where
    data DPDA = internal definition of type
     definitions of interface and auxiliary functions
```

interface functions are given below and you should export them in your module. You can define arbitrary auxiliary functions and types.

Please implement the following the interface functions in your module:

- `createDPDA :: [Integer] -> [Char] -> [Char] -> Char -> Integer -> DPDA`

This function will create a DPDA. The first parameter is the set of states, the second parameter is the input alphabet, the third parameter is the stack alphabet, the fourth parameter is the initial stack symbol which will belong to the stack alphabet, and the fifth parameter is the initial state which will belong to the set of states. Initially, the stack should only have the initial stack symbol.

- `addState :: DPDA -> Integer -> DPDA`

This function will add the given state to the DPDA. If state is in the DPDA, function should return the same DPDA.

- `removeState :: DPDA -> Integer -> DPDA`

This function will delete the given state from the DPDA. All transitions involving the state should be removed as well. Note that the deleted state can also be an accepting state. If the state is not in the DPDA or the state is the starting state, function should return the same DPDA.

- `addTransition :: DPDA -> (Integer, Char, Char, Integer, String) -> DPDA`

This function will add the given transition to the DPDA. The transition is represented by a quintuple. Given a transition (*p, a, A, q, B*), you should add a transition from state *p* to state *q* when *a* is read from the input and *A* is the current stack symbol, replacing *A* with *B* (read the introduction for dealing with $B = \varepsilon$). If the transition elements are not from the corresponding sets (states and alphabets) or the transition is already present in the DPDA or adding the transition violates the definition of a DPDA, function should return the same DPDA.

- `removeTransition :: DPDA -> (Integer, Char, Char, Integer String) -> DPDA`

This function will delete the given transition from the DPDA. If the transition is not in the DPDA, the function should return the same DPDA.

- `setAcceptingState :: DPDA -> Integer -> DPDA`

This function will set the given state as an accepting state. If the state is not in the DPDA or it's already an accepting state, function should return the same DPDA.

- `unsetAcceptingState :: DPDA -> Integer -> DPDA`

This function will return a DPDA in which the given state is removed from the set of accepting states. If the state does not exist or it's not an accepting state, function should return the same DPDA.

- `accepts :: DPDA -> String -> Bool`

This function should return `True` if the given DPDA accepts the given string. Otherwise it should return `False`.

- `show :: DPDA -> String`

DPDA should be an instance of Show class. `show` function should produce an output as follows:

`initialState=`q_0

`inputAlphabet=[`a_1`,`a_2`,…]`

`stackAlphabet=[`a_1`,`a_2`,`a_3`,…]`

`states=[`$state_1$`,`$state_2$`,`$state_3$`,…]`

`transitions=[(p,a,A,q,B),…]`

`acceptingStates=[`$state_1$`,`$state_2$`,`$state_3$`,…]`

The lists in the representation should be sorted. Transitions should be sorted starting from the first (leftmost) element (e.g. `(2,'a','A',3,"B")<(2,'a','B',4,"C")`).

Example

```
p1 = createDPDA [0, 1, 2] ['0', '1', '2'] ['0', '$'] '$' 0

p2 = addState p1 3 –We add a new state to p1

p3 = setAcceptingState p2 3 –3 is the accepting state

p4 = addTransition p3 (0, '0', '$', 1, "0$")

p5 = addTransition p4 (1, '0', '0', 1, "00")

p6 = addTransition p5 (1, '1', '0', 2, "")
```

```
p7 = addTransition p6 (2, '1', '0', 2, "")

p8 = addTransition p7 (2, '2', '$', 3, "") –DPDA accepts 0^n1^n2

p9 = removeTransition p8 (3, '1', '0', 3, "") – p9 equals p8

accepts p9 "0001112" –evaluates to True

p9 –should produce the following output (when evaluated from
the command prompt)

initialState=0

inputAlphabet=[0,1,2]

stackAlphabet=[$,0]

states=[0,1,2,3]

transitions=[(0,0,$,1,0$),(1,0,0,1,00),(1,1,0,2,),(2,1,0,2,),(
2,2,$,3,)]

acceptingStates=[3]
```

Specifications:

1. All the work should be done individually.
2. Your codes should be written in Haskell and have the name "Hw3.hs"
3. In evaluation, black box method will be used. So be careful about the name of interface functions. You should create a module named Hw3 and export the necessary functions and DPDA type.
4. You will submit your code through Cow system.
5. You should test your codes in inek machines with hugs before submitting.