CENG 242

Hw #3

Spring 2006/2007

(Due: April 15th, 2007 Sunday 23:59)

In this homework, you will write a Haskell code simulating a Deterministic Finite Automaton (DFA).

The states will be represented by numbers; alphabet will be a subset of English lowercase letters; transitions will be represented by triples (number, character, number) (e.g. (1,'a',3) means a transition occurs from state 1 to state 3 when an 'a' comes. Let's say the first argument (1) "*from state*", second argument ('a') "*action*" and the third argument "*to state*"); start state will be given while creating the automaton; and accept states can be added to an automaton.

The operations on an automaton will be:

- `createDFA :: Integer -> [Char] -> DFA`

This function gets two arguments. The first one is for start state, and the second one is for alphabet. All DFAs in our homework is assumed to have a dead state numbered as 0 (zero) and all undefined transitions from any state is assumed to go to 0 and all transitions from 0 is to itself. None of the operations given will be called with 0. After call to createDFA, as you can understand, all the transitions from start state will go to 0. This method returns the resulting DFA.

- `addState :: DFA -> Integer -> DFA`

This function will add the given state to the automaton. If the state exists in given DFA, you will not add the state again, just return the same DFA.

- `addTransition :: DFA -> (Integer, Char, Integer) -> DFA`

This function will add the transition to the automaton if the transition from *from state* with *action* goes to 0. Otherwise return the same DFA.

- `deleteState :: DFA -> Integer -> DFA`

This function will delete the state from automaton. All the transitions from/to this state should also be deleted. If the given state is not in the automaton, return the same automaton.

- `deleteTransition :: DFA -> (Integer, Char) -> DFA`

This function will delete the transition from automaton. If the transition is invalid (no such state, no such action, transition goes to 0 etc.), return the same DFA.

- `addFinalState :: DFA -> Integer -> DFA`

This function will add the given state to automaton as a final state. If the state exists in the given automaton, do not add the state and return the same automaton.

- `setFinalState :: DFA -> Integer -> DFA`

This function will set the given state of the automaton as a final state. If the given state does not exist in the given automaton, return the same automaton.

- `unsetFinalState :: DFA -> Integer ->DFA`

This function will remove the given state from final states set. If the state is not a final state, return the same automaton.

- `accept :: DFA -> String -> Bool`

This function checks whether the given string is accepted or not by the given automaton.

- `show :: DFA -> String`

This is the instance of Show, so that it will print the DFA in the given format:

`[StartState=`*`theStateNumber`*`]`
`[AcceptStates=`*`listOfAcceptStates`*`]`
`[Transitions=`*`setOfTransitions`*`]`

Examples:

```
a1 = createDFA 3 [‘a’,’b’,’c’]          -- should create the DFA and return it as a1
a2 = addState a1 5                      -- adds a state numbered 5
a3 = addState a2 3                      -- since 3 is in DFA, a3 is same as a2
a4 = addTransition a3 (3,’a’,5)         -- adds a transition from 3 to 5 with ‘a’
a5 = addTransition a4 (3,’a’,3)         -- a5 = a4 since a transition from 3 with ‘a’ exists
a6 = addTransition a5 (3,’c’,3)         -- adds a transition from 3 to itself with ‘c’
a7 = setFinalState a6 5                 -- sets the state 5 as a final state
s1 = show a6                            -- result of s1 is given later
a8 = deleteState a7 7                   -- a7 = a6 since a state numbered 7 does not exist
a9 = deleteState a8 5                -- deletes the state 5 and all transitions related with 5
a10 = deleteTransition a9 (3,’b’)       -- a9 = a8 since no transition exist from 3 with ‘b’
a11 = deleteTransition a10 (3,’a’)      -- deletes the transition from 3 with ‘a’
a12 = addFinalState a11 4               -- add a state numbered 4 and set it as a final state
```

a13 = addFinalState a12 3 -- a12 = a11 since a state numbered 3 exists
a14 = setFinalState a13 3 -- set the state 3 as a final state
a15 = setFinalState a14 5 -- a14 = a13 since state 5 does not exist
a16 = unsetFinalState a15 6 -- a15 = a14 since state 6 does not exist
a17 = unsetFinalState a16 3 -- 3 is now not a final state
b1 = accept “ba” -- return false

The string s1 will be:
“[StartState=3]
[AcceptStates=[5]]
[Transitions=[(0,’a’,0),(0,’b’,0),(0,’c’,0),(3,’a’,5),(3,’b’,0),(3,’c’,3),(5,’a’,0),(5,’b’,0),(5,’c’,0)]]”

If there is more then one accept states, separate them with commas, e.g. [AcceptStates=[1,2]]. Print accept states in increasing order if there are more than one and print transitions in increasing state numbers and increasing characters (actions) as given above. As can be seen above, there is no tabs or spaces between any characters, there will be just two newlines as separators (after]’s).

Specifications:

- For the definition of DFA etc., you can use internet, do not ask questions like what a DFA is.
- You can use any representation for DFA, no restriction but the name.
- You will submit a single file named Hw3.hs including all your definitions. There must be a module called Hw3. Your file should begin with the line:

```
module Hw3 (DFA, createDFA, addState, addTransition, deleteState,
deleteTransition, addFinalState, setFinalState, unsetFinalState,
accept, show) where
```

- You will submit your codes through cow system. Specifications (file name, function name, module name etc.) are strict. Breaking any of them will cost you getting a 0 from the homework since black box method is used.