CSci 555-01: Functional Programming
Assignment #3, Spring 2019

H. Conrad Cunningham

10 April 2019

Assignment #3: Mealy Machine Simulator

Super-Extended Deadline Thursday, 25 April 2019, 11:59 P.M.
(Originally due Monday, 8 April 2019)

Note: Students who turn in a "working" program by the "extended" deadline of 16 April will receive 5 points extra.

General Instructions

All homework and programming exercises must be prepared in accordance with the instructions given in the Syllabus. Each assignment must be submitted to your instructor by its stated deadline.

Citations: In accordance with expected scholarly and academic standards, if you reference outside textbooks, reference books, articles, websites, etc., or discuss an assignment with individuals inside or outside the class, you must document these by including appropriate citations or comments at prominent places in your submission such as in the header of the primary source file.

Identification: Put your name, course name, and assignment number as comments in each file you submit.

Assignment Description

1. This is an individual assignment.
2. When complete, submit your Scala source code files to the course Blackboard site for Assignment #3.

3. Be sure to document your source code appropriately using program comments. Give attention to the general instructions given above and in the Syllabus.

   Consider the source code with a constructive semantics (e.g. use preconditions, postconditions, and invariants with respect to an abstract model of the Mealy Machine).

4. Develop solutions to Mealy Machine Exercises below. A student taking the course for undergraduate credit may omit one exercise.

   This project is also given in the *Abstract Data Types* in Scala document.

5. Test your program thoroughly with a program that imports the Mealy Machine modules.

Mealy Machine Simulator Project

In this project, you are asked to design and implement Scala "modules" to represent Mealy Machines and to simulate their execution.

Mealy Machine

A Mealy Machine is a useful abstraction for simple controllers that listen for input events and respond by generating output events. For example in an automobile application, the input might be an event such as "fuel level low" and the output might be command to "display low-fuel warning message".

In the theory of computation, a *Mealy Machine* is a *finite-state automaton* whose output values are determined both by its current state and the current input. It is a *deterministic finite state transducer* such that, for each state and input, at most one transition is possible.

Appendix A of the Linz textbook [Linz 2017] defines a Mealy Machine mathematically by a tuple

$$M = (Q, \Sigma, \Gamma, \delta, \theta, q_0)$$

where

Q is a finite set of internal states
Σ is the input alphabet (a finite set of values)
Γ is the output alphabet (a finite set of values)
$\delta : Q \times \Sigma \longrightarrow Q$ is the transition function
$\theta : Q \times \Sigma \longrightarrow \Gamma$ is the output function
q_0 is the initial state of M (an element of Q)

In an alternative formulation, the transition and output functions can be combined into a single function:

$$\delta : Q \times \Sigma \longrightarrow Q \times \Gamma$$

We often find it useful to picture a finite state machine as a *transition graph* where the states are mapped to vertices and the transition function represented by directed edges between vertices labelled with the input and output symbols.

Mealy Machine Exercises

1. Specify, design, and implement a general representation for a Mealy Machine as a set of Scala definitions implementing an abstract data type. It should hide the representation of the machine behind an abstract interface and should have, at least, the following public operations.

    - Constructor `MealyMachine(s)` creates a new machine with initial (and current) state `s` and no transitions.
    - Mutator method `addState(s)` adds a new state `s` to this machine and returns an `Either` wrapping the modified machine or an error message.
    - Mutator method `addTransition(s1,in,out,s2)` adds a new transition to this machine and returns an `Either` wrapping the modified machine or an error message. From state `s1` with input `in`, the modified machine outputs `out` and transitions to state `s2`.
    - Mutator method `addResets` adds all reset transitions to this machine and returns the modified machine. This operation makes the transition function a total function by adding any missing transitions from a state back to the initial state.
    - Mutator method `setCurrent(s)` sets the current state of this machine to `s` and returns an `Either` wrapping the modified machine or an error message.
    - Accessor method `getCurrent` returns the current state of this machine.
    - Accessor method `getStates` returns a list of the elements of the state set of this machine.
    - Accessor method `getInputs` returns a list of the input set of this machine.
    - Accessor method `getOutputs` returns a list of the output set of this machine.
    - Accessor method `getTransitions` returns a list of the transition set of this machine. Tuple `(s1,in,out,s2)` occurs in the returned list if and only if, from state `s1` with input `in`, the machine outputs `out` and moves to state `s2`.

- Accessor method `getTransitionsFrom(s)` returns an `Either` wrapping a list of the set of transitions enabled from state `s` of this machine or an error message.

Note: It is possible to use a Labelled Digraph ADT module in the implementation of the Mealy Machine. A state is a vertex of the graph, transition is an edge of the graph, and an `(in,out)` is a label for an edge.

2. Given the above implementation for a Mealy Machine ADT, design and implement a separate Scala module that simulates the execution of a Mealy Machine. It should have, at least, the following public operations.

   - Mutator `move(m,in)` moves machine `m` from the current state given input `in` and returns an `Either` wrapping a tuple `(mm,out)` or an error message. The tuple gives the modified machine `mm` and the output `out`.

   - Mutator method `simulate(m,ins)` simulates execution of machine `m` from its current state through a sequence of moves for the inputs in list `ins` and returns an `Either` wrapping a tuple `(mm,outs)` or an error message. The tuple gives the modified machine `mm` after the sequence of moves and the output list `outs`.

References

[Linz 2017]: Peter Linz. *Formal Languages and Automata*, 6th Edition, Jones & Bartlett, 2017.