McMaster University
Department of Computing and Software
Dr. W. Kahl

FP 2006
Assignment 1

CAS 743 — Functional Programming

16 January 2006

We define a type of transition functions that define state transitions triggered by *input*s and also producing *output*s:

type *Transition state input output* = (*state* , *input*) → (*state* , *output*)

(a) Define a Haskell function

process :: *Transition state input output* → *state* → [*input*] → [*output*]

that calculates the list of outputs produced by a transition function given a starting state and a list of inputs.

Using *process* from (a) and prelude functions, the definition

```
runprocess :: Transition state String String → state → IO ()
runprocess tr s = do
  hSetBuffering stdout LineBuffering  -- requires: "import System.IO" at beginning of module
  interact (unlines ∘ process tr s ∘ lines)
```

allows *runprocess* to turn a transition with *String* inputs and outputs into a runnable program.

Try: *runprocess id* 0

(b) Define a transition function

countEcho :: *Transition Integer String String*

that keeps a counter as its state and otherwise just reproduces the input prefixed with line numbers as output.

Try: *runprocess countEcho* 0

(c) Define a transition function

trAdd :: *Transition Integer String String*

that uses the prelude functions *read* and *show* to add the *Integer* reading of the input to the accumulating state, and outputs that state as a string.

Try: *runprocess trAdd* 0

(d) For finite *state*, *input*, and *output* types, the *Transition* type defined above is the type of the transition function of a deterministic Mealy automaton.

Let us use the following type for explicit representations of such transition functions:

type *Mealy state input output* = [((*state* , *input*) , (*state* , *output*))]

Define a transition function generator

trMealy :: (*Eq state*) ⇒ *Mealy state String String* → *Transition state String String*

that turns a representation of a Mealy transition function with *String* inputs and outputs into the corresponding *Transition*.

Define a non-trivial Mealy transition function and try: *runprocess* (*trMealy myMealy*) *state0*

(e) Let the following type for representing finite-state machines (parameterised with the *state* type) be given:

type *FSM state* = (*state* , [*state*] , [((*state* , *String*) , *state*)])

Define a transition function generator

trDFSM :: (*Eq state* , *Show state*) ⇒ *FSM state* → *Transition state String String*

that, given a representation of a ***deterministic*** finite-state machine, produces a transition function that takes input symbols for the FSM as inputs, and *show*s the current state as output, together with information whether the current state is final.

Define a non-trivial FSM and try: *runprocess* (*trDFSM myFSM*) *myStartState*

(f) Produce *trNFSM* by modifying *trDFSM* to produce appropriate transitions also for ***non-deterministic*** finite-state machines.

(g) Define a transition function

polish :: *Transition* [*Integer*] *String String*

that implements a reverse Polish notation calculator by pushing number inputs on the stack, always outputing the top of the stack (if present), and interpreting `+`, `-`, `*`, `/` as taking their arguments from the stack and pushing the result back onto the stack.

Try: *runprocess polish* [ ]

(h) **(optional)** Instead of only showing the top element of the stack in your implementation of *polish*, output the top five elements on the stack (as far as present). Also include in your solution for *polish* some stack manipulation commands, such as

- "`dup`" pushes the top element of the stack onto the stack another time,
- "`exch`" swaps the two top element on the stack,
- "`rot`" pops the top element, say *n*, from the stack, and then rotates the top *n* remaining elements "downwards".