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

FP 2009
Assignment 2

CAS 743 — Functional Programming

9 October 2009

1. Stack Machines

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”.

2. One-Dimensional Cellular Automata

A one-dimensional cellular automaton operates in a linear **space** of **cells**. For the purposes of this problem, only finite spaces are considered.

In every **generation**, every cell is assigned one of a set of **states**. Neighbouring the first and last cells of the space, one should assume “virtual cells” that are always in a **default state**.

For calculating from an (old) generation the **next generation**, there is a **rule** associated with the finite automaton. This rule prescribes what the state of every element is in the next generation, depending on its own state in the old generation, and on the states of its left and right neighbours in the old generation.

A **run** of such a linear cellular automaton is a sequence of successive state assignments of the space.

For example, the **Sierpinski cellular automaton** has two states, “empty” (default) and “full”, and the rule states that a cell is full in the next generation if out of the three considered cells (i.e., itself and its left and right neighbours), exactly one or two are full in the old generation.

If every **line** represents **one generation**, with full cells represented by '@' characters and empty cells by space characters, then an example run of the Sierpinski cellular automaton, starting with exactly the middle cell full, can be represented by the **sequence of lines** ***printed on the next page***.

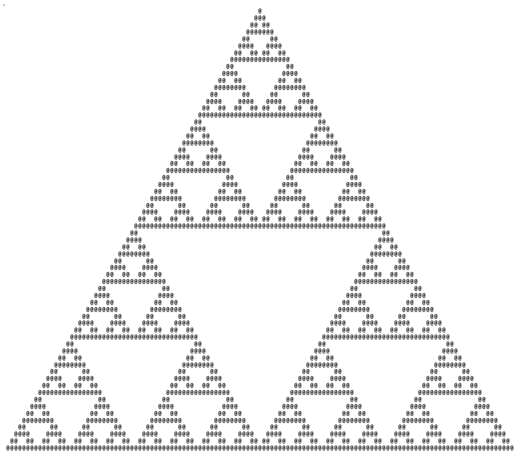
(a) Produce and document Haskell data type declarations for one-dimensional cellular automata.

(b) Give a type signature for the function calculating the next generation from an old generation for a one-dimensional cellular automaton.

(c) Give a definition for that function.

(d) Give a definition of an element of your one-dimensional cellular automata data type from (a) for the Sierpinski cellular automaton.

(e) Using the next-generation function from (b, c) and the Sierpinski automaton data from (d), define the function *runSierpinski* such that the invocation “*runSierpinski* 64” in Hugs or GHCi interaction in a sufficiently large terminal produces the below representation of the example run.



3. Two-Dimensional Cellular Automata

(a) Define data types and transition function for Conway’s “Game of Life”, a well-known two-dimensional cellular automaton. (You can find information for example from http://cafaq.com/.)

(b) Define a way to animate your “Game of Life” — an easy way would be to use <u>XTerm control sequences</u>. For example, in an xterm (or rxvt), try “*putStr* "\ESC[H\ESC[2J"”, (and read also about character literal escape sequences in the Haskell report).

(c) Define an interesting starting state involving for example at least two gliders as part of your solution. Program this starting state using appropriate abstractions (such as “glider”).