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

SFWR ENG 3E03
Exercise Sheet 5
Solution Hints

Design and Selection of Programming Languages

11 October 2006

Exercise 5.1 — Haskell Evaluation (36% of Midterm 1, 2004)

Assume the following Haskell definitions to be given:

```
succ n = n+1                    -- reduce in one step, e.g.: succ 5 → 6

take :: Int -> [a] -> [a]
take 0 _      =  []
take _ []     =  []
take n (x:xs) =  x : take (n-1) xs

feed h q y =  q : feed h (q + y) (h y)
```

Simulate Haskell evaluation for the following expression (write down the sequence of intermediate expressions):

```
take 3 (feed succ 0 1)
```

Note: You may introduce **abbreviations for repeated subexpressions**, or use **repetition marks for material that is unchanged from the previous line**. In particular, ***write*** "*s*" ***instead of*** "*succ*"!

Solution Hints

13 steps, 1 contractible arith

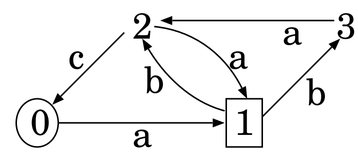
$\mathit{take}\ 3\ (\mathit{feed}\ \mathit{succ}\ 0\ 1)$
$= \mathit{take}\ 3\ (0 : \mathit{feed}\ \mathit{succ}\ (0+1)\ (\mathit{succ}\ 1))$
$= 0 : \mathit{take}\ (3-1)\ (\mathit{feed}\ \mathit{succ}\ (0+1)\ (\mathit{succ}\ 1))$
$= 0 : \mathit{take}\ 2\ (\mathit{feed}\ \mathit{succ}\ (0+1)\ (\mathit{succ}\ 1))$
$= 0 : \mathit{take}\ 2\ ((0+1) : \mathit{feed}\ \mathit{succ}\ ((0+1) + \mathit{succ}\ 1)\ (\mathit{succ}\ (\mathit{succ}\ 1)))$
$= 0 : (0+1) : \mathit{take}\ (2-1)\ (\mathit{feed}\ \mathit{succ}\ ((0+1) + \mathit{succ}\ 1)\ (\mathit{succ}\ (\mathit{succ}\ 1)))$
$= 0 : 1 : \mathit{take}\ (2-1)\ (\mathit{feed}\ \mathit{succ}\ (1 + \mathit{succ}\ 1)\ (\mathit{succ}\ (\mathit{succ}\ 1)))$
$= 0 : 1 : \mathit{take}\ 1\ (\mathit{feed}\ \mathit{succ}\ (1 + \mathit{succ}\ 1)\ (\mathit{succ}\ (\mathit{succ}\ 1)))$
$= 0 : 1 : \mathit{take}\ 1\ ((1 + \mathit{succ}\ 1) : \mathit{feed}\ (+)\ \mathit{succ}\ ((1 + \mathit{succ}\ 1) + \mathit{succ}\ (\mathit{succ}\ 1))\ (\mathit{succ}\ (\mathit{succ}\ (\mathit{succ}\ 1))))$
$= 0 : 1 : (1 + \mathit{succ}\ 1) : \mathit{take}\ (1-1)\ (\mathit{feed}\ \mathit{succ}\ ((1 + \mathit{succ}\ 1) + \mathit{succ}\ (\mathit{succ}\ 1))\ (\mathit{succ}\ (\mathit{succ}\ (\mathit{succ}\ 1))))$
$= 0 : 1 : (1 + 2) : \mathit{take}\ (1-1)\ (\mathit{feed}\ \mathit{succ}\ ((1 + 2) + \mathit{succ}\ (\mathit{succ}\ 1))\ (\mathit{succ}\ (\mathit{succ}\ 2)))$
$= 0 : 1 : 3 : \mathit{take}\ (1-1)\ (\mathit{feed}\ \mathit{succ}\ (3 + \mathit{succ}\ (\mathit{succ}\ 1))\ (\mathit{succ}\ (\mathit{succ}\ 2)))$
$= 0 : 1 : 3 : \mathit{take}\ 0\ (\mathit{feed}\ \mathit{succ}\ (3 + \mathit{succ}\ (\mathit{succ}\ 1))\ (\mathit{succ}\ (\mathit{succ}\ 2)))$
$= 0 : 1 : 3 : [\,]$

3% per necessary step:
- 1% for reducing the right redex
- 2% for performing the reduction correctly
- -1% for not writing down

Exercise 5.2 — Finite-State Machines (25% of Midterm 1, 2004)

Let the following type synonyms be given, as in the presentation in the first lecture:

type *State* = *Int*
type *Symbol* = *Char*
type *TransRel* = [(*State* , *Symbol* , *State*)]
type *FSM* = (*State* , *TransRel* , [*State*])



(a) Define *fsm1* :: *FSM* such that it represents the finite-state machine drawn above (with start state circled and end states in boxes):

(b) Define the Haskell function *isDet* :: *FSM* → *Bool* such that *isDet fsm* evaluates to the Boolean value indicating whether the finite-state machine *fsm* is deterministic or not.

For example, *isDet fsm1* = **False** since there are two *b*-edges from state 1 to different nodes.

Hint: Define auxiliary functions! For example:
- Calculate all start nodes of transitions in a *TransRel*.
- Given a state, calculate all edges leaving that state in a *TransRel*.
- Given a *Symbol* and a *TransRel*, find all target nodes of edges with that symbol.
- Given a *State* and a *TransRel*, find out whether any edges from that state violate determinacy.

Other functions may be useful, too. **Document your functions!**

Solution Hints

```
type State = Int
type Symbol = Char
type TransRel = [ ( State , Symbol , State ) ]

type FSM = ( State , TransRel , [ State ] )

fsm1 :: FSM                -- 6%
fsm1 = ( 0 , tr1 , [1] )
 where
  tr1 =
   [ ( 0 ,'a' ,1)
   , ( 1,'b' ,2 )
   , ( 1,'b' ,3 )
   , ( 2 ,'a' ,1)
   , ( 2 ,'c' ,0 )
   , ( 3 ,'a' ,2 )
   ]

edgeStarts tr = [ s | ( s , c , t ) ← tr ]   -- 3%
```

$outEdges\ tr\ s = [\,(c,t)\mid (s',c,t) \leftarrow tr,\ s' \equiv s\,]$ -- 3%

$isUnique\ es\ (c,t) = all\ (t\ \equiv)\ [\,t' \mid (c',t') \leftarrow es,\ c' \equiv c\,]$ -- 5%

$isDetState\ tr\ s = all\ (isUnique\ es)\ es$ -- 4%
 where $es = outEdges\ tr\ s$

$isDet\ (s0,\ tr,\ fin) = all\ (isDetState\ tr)\ (edgeStarts\ tr)$ -- 4%

Exercise 5.3 — Haskell Typing (19% of Midterm 1, 2004)

Provide **detailed derivations** of the Haskell types of the following functions:

```
swibble x y = [ ( x , y ) , ( x ++ "'", y + 1 ) ]
swoon g h = [ g ( (1 + ) . h ) ]
```

Solution Hints

Type classes have not been taught yet, only mentioned: Numeric types can be defaulted to *Integer* or *Int*.

$swibble :: (Num\ n) \Rightarrow String \rightarrow n \rightarrow [\,(String,\ n)\,]$

Assuming $1 :: Integer$, we must have $y :: Integer$ because of $y + 1$.

Since "" $:: String$, we also have $x :: String$ because of x ++ "" $:: String$.

Then $(x,y) :: (String,\ Integer)$, and the type of *swibble* follows easily.

$swoon :: (Num\ n) \Rightarrow ((a \rightarrow n) \rightarrow b) \rightarrow (a \rightarrow n) \rightarrow [\,b\,]$

Assuming $1 :: Integer$, we have $(1+) :: Integer \rightarrow Integer$, and because of the composition, we must have

$h :: a \rightarrow Integer$ for some type a.

Therefore, we have $((1+) \circ h) :: a \rightarrow Integer$, and may assume $g :: (a \rightarrow Integer) \rightarrow b$ for some type b.

Then we have $[\,g\ ((1+) \circ h)\,] :: b$, and therefore

$swoon\ g :: (a \rightarrow Integer) \rightarrow b$

and

$swoon :: ((a \rightarrow Integer) \rightarrow b) \rightarrow (a \rightarrow Integer) \rightarrow b$.

Exercise 5.4 (***Skeleton file is on the course page***)

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

type $Transition\ state\ input\ output = (state,\ input) \rightarrow (state,\ output)$

(a) Define a Haskell function

$process :: Transition\ state\ input\ output \to state \to [\,input\,] \to [\,output\,]$

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

Solution Hints

```
process tr s [ ] = [ ]
process tr s (input : inputs) = let
   (s', output) = tr (s, input)
 in output : process tr s' inputs
```

Using *process* from (b) 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 withline numbers as output.

Try: *runprocess countEcho* 0

Solution Hints

```
countEcho (count, input) = (count', shows count' (' ' : input))
  where count' = succ count
```

(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

Solution Hints

```
trAdd (s, input) = (s', show s')
 where
   n = read input
   s' = s + n
```

(d) 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* [ ]

Solution Hints

```
polish (n : m : ks, "+") = (k : ks, show k) where k = m + n
polish (n : m : ks, "-") = (k : ks, show k) where k = m − n
polish (n : m : ks, "*") = (k : ks, show k) where k = m ∗ n
polish (n : m : ks, "/") = (k : ks, show k) where k = m `div` n
polish (ks        , input) = (k : ks, show k) where k = read input
```