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

SFWR ENG 3E03
Exercise Sheet 6

Design and Selection of Programming Languages

18 October 2006

Exercise 6.1 — Haskell Evaluation (25% of 90 minutes Midterm 2, 2005)

Let the following Haskell definition be given:

```
from k = k : from (k+1)

prune True xs = []
prune False xs = xs

eat p [] = from (7 * 8)
eat p (x : xs) = x : prune (p x) (eat (not . p) xs)
```

Simulate Haskell evaluation for the following expression, i.e., write down **the complete sequence of intermediate expressions**:

```
eat (< 5) (from 5)
```

Note: You may introduce ***abbreviations for repeated subexpressions***, or use ***repetition marks for material that is unchanged from the previous line.***

Solution Hints

eat (< 5) (*from* 5)
--> eat (< 5) (5 : from (5 + 1))
--> 5 : prune ((< 5) 5) (eat (not . (< 5)) (from (5 + 1))) -- *
--> 5 : prune (5 < 5) (eat (not . (< 5)) (from (5 + 1)))
--> 5 : prune False (eat (not . (< 5)) (from (5 + 1)))
--> 5 : eat (not . (< 5)) (from (5 + 1))
--> 5 : eat (not . (< 5)) ((5 + 1) : from ((5 + 1) + 1))
--> 5 : (5 + 1) : prune ((not . (< 5)) (5 + 1)) (eat (not . (not . (< 5))) (from ((5 + 1) + 1)))
--> 5 : 6 : prune ((not . (< 5)) 6) (eat (not . (not . (< 5))) (from (6 + 1)))
--> 5 : 6 : prune (not ((< 5) 6)) (eat (not . (not . (< 5))) (from (6 + 1)))
--> 5 : 6 : prune (not (6 < 5)) (eat (not . (not . (< 5))) (from (6 + 1))) -- *
--> 5 : 6 : prune (not False) (eat (not . (not . (< 5))) (from (6 + 1)))
--> 5 : 6 : prune True (eat (not . (not . (< 5))) (from (6 + 1)))
--> 5 : 6 : []
= [5,6]

Exercise 6.2 — Haskell Typing (22% of Midterm 2, 2005)

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

```
maybe x f Nothing = x
maybe x f (Just y) = f y

keepof2 k h (x,y) = k (curry h x) y
```

Remember: $curry :: ((a, b) \to c) \to a \to b \to c$

Solution Hints

The prelude definition

data $Maybe\ a = Nothing \mid Just\ a$

implies the following types for the constructors of this datatype:

$Nothing :: Maybe\ a$
$Just :: a \to Maybe\ a$

Starting from the second equation and assuming $x :: q$ and $f :: a \to b$, we see that $y :: a$ and obtain:

$maybe :: q \to (a \to b) \to (Maybe\ a) \to b$

With the first equation, we see from the right-hand side that $x :: b$, too, so we have:

$maybe :: b \to (a \to b) \to (Maybe\ a) \to b$ ≈8%

Using $curry :: ((a, b) \to c) \to a \to b \to c$, we obtain $x :: a$ and $h :: (a,b) \to c$.

So $(curry\ h\ x) : b \to c$. Now let us assume that $y :: d$; then we have

$k :: ((b \to c) \to d \to e)$

for some e, and therefore:

$keepof2 :: ((b \to c) \to d \to e) \to ((a,b) \to c) \to (a,d) \to e$ ≈14%

Exercise 6.3 — Defining Haskell Functions (19% of Midterm 2, 2005)

Define the following Haskell functions (the solutions are independent of each other, but each can use functions specified in previous items):

(a) ≈5% $inits :: [a] \to [[a]]$

such that $inits\ xs$ evaluates to a list consisting of exactly all prefixes of xs (in which order is irrelevant).

E.g., $inits\ [1,2,3] = [[\,],[1],[1,2],[1,2,3]]$

(This is a function exported by the standard library module *List*.)

Solution Hints

$inits :: [a] \to [[a]]$ −− = List.inits
$inits\ [\,] = [[\,]]$
$inits\ (x:xs) = [\,] : map\ (x:)\ (inits\ xs)$

Or:

```
inits' :: [a] → [[a]]                      init' :: [a] → [a]        -- = Prelude.init
inits' [] = [[]]                           init' [x] = []
inits' xs = xs : inits' (init' xs)         init' (x:xs) = x : init xs
```

(b) ≈6% *fromThen* :: *Integer* → *Integer* → [*Integer*]

such that *fromThen x1 x2* = [*x1*, *x2* ..].

Solution Hints

```
fromThen :: Integer → Integer → [Integer]
fromThen x1 x2 = x1 : fromThen x2 (x2 + x2 − x1)
fromThen' x1 x2 = ft x1
  where
   ft x1 = x1 : ft (x1 + d)
   d = x2 − x1
```

(c) ≈8% *fromThenTo* :: *Integer* → *Integer* → *Integer* → [*Integer*]

such that *fromThenTo x1 x2 x3* = [*x1*, *x2* .. *x3*], e.g.:

```
fromThenTo 5 7  9 = [5,7,9]
fromThenTo 5 7 10 = [5,7,9]
fromThenTo 7 5 10 = []
fromThenTo 7 5  1 = [7,5,3,1]
```

Solution Hints

```
fromThenTo :: Integer → Integer → Integer → [Integer]
fromThenTo x1 x2 x3 = takeWhile p $ fromThen x1 x2
  where
   p = if x2 ≥ x1 then (≤ x3) else (≥ x3)
```

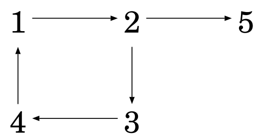
Or:

```
fromThenTo' :: Integer → Integer → Integer → [Integer]
fromThenTo' x1 x2 x3 = ftt x1
  where
   ftt x1 | p x1 = x1 : ftt (x1 + d)
          | otherwise = []
   d = x2 − x1
   p = if d ≥ 0 then (≤ x3) else (≥ x3)
```

Note: *fromThen* and *fromThenTo* are the functions underlying the syntactic sugar [1, 3 ..] and [1,3 .. 10] — you should not use this syntax to define these functions.

Exercise 6.4 — Simple Graphs (34% of Midterm 2, 2005)

A simple graph can be (naïvely) represented in Haskell as a list of pairs, where an edge from node *x* to node *y* is represented by the pair (*x*, *y*), and the sequencing of pairs in the list does not matter.



For example, one representation of the graph drawn to the left is

gr = [(1,2),(2,3),(2,5),(3,4),(4,1)]

Let the following type synonym be given:

type *Graph a* = [(*a*,*a*)]

(a) ≈6% Define *successors* :: *Eq a* ⇒ *Graph a* → *a* → [*a*] such that *successors g n* returns a list containing exactly the endnodes of those edges of the graph *g* that start at node *n*.

E.g., *successors gr* 2 = [3, 5] and *successors gr* 5 = []

Solution Hints

```
successors, successors' :: Eq a ⇒ Graph a → a → [a]
successors g n = [y | (x,y) ← g, x ≡ n] −− = map snd (filter ((n ==) . fst) g)

successors' [] n = []
successors' ((x,y):ps) n = if x ≡ n then y : successors' ps n else successors' ps n
```

(b) ≈10% *pathGraph* :: [*a*] → *Graph a*

such that *pathGraph*$[x_1, \ldots, x_n]$ evaluates to the list $[(x_1, x_2), \ldots, (x_{n-1}, x_n)]$ containing the pairs of immediately consecutive elements in *xs*, e.g.,

pathGraph [2,3,4,1,2,5] = [(2,3), (3,4), (4,1), (1,2), (2,5)], which is just another representation fo the graph drawn above.

Solution Hints

```
pathGraph :: [a] → [(a,a)]
pathGraph (x : xs≅(y : ys)) = (x,y) : pathGraph xs
pathGraph _ = []
```

(c) ≈8% A *path* in a simple graph can be represented as a list of nodes, as above in (b). Define the Haskell function *hasCycle* :: *Eq a* ⇒ [*a*] → *Bool* such that *hasCycle p* is true if path *p* contains a cycle, i.e., if there is a node that occurs at least twice in *p*. For example, the path [2,3,4,1,2,5] has a cycle around node 2.

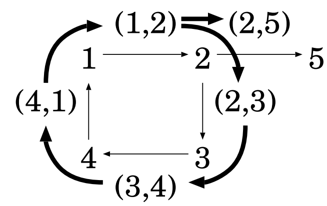
Solution Hints

```
hasCycle :: Eq a ⇒ [a] → Bool
hasCycle [] = True
hasCycle (x : xs) = x `elem` xs || hasCycle xs
```

(d) ≈10% Define *edgeGraph* :: *Eq a* ⇒ *Graph a* → *Graph* (*a*,*a*) such that *edgeGraph g* returns the *edge graph* of *g*. This edge graph has edges of *g* as nodes, and has an edge from *e1* to *e2* iff the end node of *e1* is equal to the start node of *e2* (as edges in *g*).

Solution Hints

```
edgeGraph :: Eq a ⇒ Graph a → Graph (a,a)
edgeGraph g = [(e1,e2) | e1@(_,x) ← g, e2@(y,_) ← g, x ≡ y
]
```



(e) new Define paths :: Eq a => Graph a -> [[a]] to calculate all non-empty cycle-free paths of a graph.

Solution Hints

We use induction over the number of edges: Adding an edge to a graph may combine two previously existing paths, or extend one previously existing path either at the beginning or at the end.

```
paths :: Eq a ⇒ Graph a → [[a]]
paths [] = []
paths (e@(x,y) : es) = let ps = paths es in
 ps ++
 [x : zs | zs@(z:zs') ← ps, y ≡ z, x `notElem` zs ]
 ++
 [zs ++ [y] | zs ← ps, x ≡ last zs, x `notElem` zs ]
 ++
 [zs ++ zs' | zs ← ps, x ≡ last zs, zs' ← ps, y ≡ head zs',
          all (`notElem` zs) zs' ]
 ++
 if x ≡ y then [] else [[x,y]]
```
