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***.

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

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*.)

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

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

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

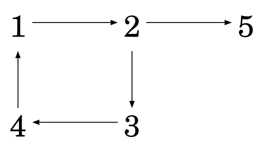
such that *fromThenTo x1 x2 x3* = [*x1*, *x2* .. *x3*], e.g.:

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]

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 = [ ]

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

such that *pathGraph*[x_1, …, x_n] evaluates to the list [(x_1, x_2), …, (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.

(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.

(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*).

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