Computer Science 320 (Fall Term, 2016)
Concepts of Programming Languages

Solutions for Mid-Term Exam

Tuesday, October 25, 2016

Problem 1. ***Recursion and List Comprehension.*** Consider the following function definitions:

```
fooA []  _      = []
fooA [y] zs     = map (*y) zs
fooA (y:ys) zs  = (fooA [y] zs) ++ (fooA ys zs)

fooB x
  | (x <= 1)                                    = []
  | elem x (fooA [2..(x-1)] [2..(x-1)])         = x : fooB (x-1)
  | otherwise                                   = fooB (x-1)

fooC x = [ a | a <- [2..x] , not (elem a (fooB x)) ]
```

Note that `fooC` depends on `fooB` and that `fooB` depends on `fooA`. So, try to understand what `fooA` does first, then go to `fooB`, and `fooC`, in this order.

(a) Evaluate `(fooA [1,2] [2,3,4])`. Give a one-sentence explanation, in case you get it wrong.

Answer:

`(fooA [1,2] [2,3,4])` ⇒ `[2,3,4,4,6,8]`

`(fooA xs ys)` multiplies every integer in `xs` with every integer in `ys`.

(b) Evaluate `(fooB 10)`. You may add one-sentence explanation, in case you get it wrong.

Answer:

`(fooB 10)` ⇒ `[10,9,8,6,4]`

`(fooB x)` produces the list of all composite numbers up to, and including `x`, in decreasing order.

(c) Evaluate `(fooC 10)`. You may add one-sentence explanation, in case you get it wrong.

Answer:

`(fooC 10)` ⇒ `[2,3,5,7]`

`(fooC x)` produces the list of all prime numbers up to, and including `x`, in increasing order.

Problem 2. ***Datatypes and Binary Trees.*** Consider the following polymorphic datatype, representing binary trees where every leaf and every internal node is labelled:

```
data Tree a = Leaf a | Node a (Tree a) (Tree a)
```

In parts (a), (b) and (c) below, we ask you to define appropriate Haskell fuctions: To get full credit, you need to use **pattern-matching** and **wildcards** as much as possible.

(a) Define a function that counts the number of internal nodes (excluding leaves) in a tree:

Answer:

```
numberOfNodes :: Tree a -> Int
numberOfNodes (Leaf _) = 0
numberOfNodes (Node _ t1 t2) = 1 + numberOfNodes t1 + numberOfNodes t2
```

(b) Define a function that counts the number of leaves in a tree:

Answer:

```
numberOfLeaves :: Tree a -> Int
numberOfLeaves (Leaf _) = 1
numberOfLeaves (Node _ t1 t2) = numberOfLeaves t1 + numberOfLeaves t2
```

(c) Recall the definition of the function `zip` on two lists `xs` and `ys`: It pairs off corresponding elements in `xs` and `ys`, and returns a list which is as long as the shorter of the two input lists. Generalize the definition of `zip` to work on trees, with the output being another tree that matches as much as possible the shapes of the two input trees:

Answer:

```
zipTree :: Tree a -> Tree b -> Tree (a,b)
zipTree (Leaf x) (Leaf y) = Leaf (x,y)
zipTree (Leaf x) (Node y _ _) = Leaf (x,y)
zipTree (Node x _ _) (Leaf y) = Leaf (x,y)
zipTree (Node x t1 t2) (Node y t3 t4) =
                      Node (x,y) (zipTree t1 t3) (zipTree t2 t4)
```

Problem 3. ***Infinite Lists of Integers.*** There are four independent parts.

(a) What is the infinite list defined by the following declaration?

```
infListA :: [Int]
infListA = 1 : (zipWith (+) ones infListA)
    where ones = 1 : ones
```

Answer: 1, 2, 3, 4, 5, ... (the positive integers)

(b) What is the infinite list defined by the following declaration?

```
infListB :: [Int]
infListB = 1 : (zipWith (+) infListB infListB)
```

Answer: 1, 2, 4, 8, 16, ... (the powers of 2)

(c) What is the infinite list defined by the following declaration?

```
infListC :: [Int]
infListC = 1 : (zipWith (*) (tail nats) infListC)
    where nats = 0 : (map succ nats)
```

Answer: 1, 1, 2, 6, 24, 120, 720, ... (the list of factorials)

(d) What is the infinite list defined by the following declaration?

```
infListD :: [Int]
infListD = 0 : 1 : (zipWith (+) infListD (tail infListD)
```

Answer: 0, 1, 1, 2, 3, 5, 8, 13, ... (the list of Fibonacci numbers)

Problem 4. ***Types and Type Checking.*** The infix operator for composition is just "." so that, for example, the expression "`(even . ceiling) 4.2`" is equivalent to "`even (ceiling 4.2)`", which evaluates to `False`. Using composition as a prefix operator, the preceding expression is also equivalent to "`(.) even ceiling 4.2`".

(a) Write the type of the prefix operator for composition.

Answer:

```
(.) :: (a -> b) -> (c -> a) -> c -> b
```

The library function `flip` is a higher-order function which takes as input a curried function `f` of two arguments `x` and `y` such that the evaluation of the expression "`f x y`" produces the same result as the evaluation of the expression "`(flip f) y x`". For example, the two expressions below:

`map even [1,2,3]` and `(flip map) [1,2,3] even`

are equivalent and therefore evaluate to the same final result `[False,True,False]`.

(b) Write the type of the library function `flip`.

Answer:

```
flip :: (a -> b -> c) -> (b -> a -> c)
```

In the remaining parts of this problem, take the type of the library function `even` to be `Int -> Bool`, and the type of the library function `ceiling` to be `Float -> Int`.

(c) Does `((.) even ceiling)` type-check? If it does, write its type. If it does not, give a reason.

Answer: The expression does type check, and its type is:

```
((.) even ceiling) :: Float -> Bool
```

(d) Does `((.) ceiling even)` type-check? If it does, write its type. If it does not, give a reason.

Answer: The expression does not type check, because `(.) ceiling :: (a -> Float) -> (a -> Int)` cannot be applied to `even :: Int -> Bool` which follows from the fact that `(a -> Float)` cannot be instantiated to `(Int -> Bool)`.

Problem 5. ***User-Defined Polymorphic Lists.*** Instead of the native (i.e., pre-defined) datatype of polymorphic lists, we want to use the following user-defined datatype:

```
data List a = Nil | Cons a (List a)
```

To get full credit, you need to use **pattern-matching** and **wildcards** as much as possible throughout. In parts (a), you must use **recursion** explicitly; in parts (b), (c) and (d), you must use `foldList` explicitly.

(a) Define the function `foldList` which acts on user-defined lists just as `foldr` acts on native lists.

Answer:

```
foldList :: (a -> b -> b) -> b -> List a -> b
foldList f init Nil = init
foldList f init (Cons x xs) = f x (foldList f init xs)
```

(b) Define the function `mapList` on user-defined lists, the counterpart of `map` on native lists.
You must use `foldList` to get credit.

Answer:

```
mapList :: (a -> b) -> (List a) -> (List b)
mapList f = foldList (Cons . f) Nil
```

(c) Define the function `sumList` which adds up the entries in an argument of type `(List Int)`.
You must use `foldList` to get credit.

Answer:

```
sumList :: (List Int) -> Int
sumList = foldList (+) 0
```

(d) Define the function `toZeroOneList` which takes a list of type `(List Int)` then turns every *even* integer to 0 and every *odd* integer to 1. For example, the list "`(Cons 3 (Cons 2 (Cons 7 Nil)))`" is transformed to "`(Cons 1 (Cons 0 (Cons 1 Nil)))`".
You must use the function `foldList` to get credit.

Answer:

```
toZeroOneList :: (List Int) -> (List Int)
toZeroOneList = foldList help Nil
                where
                help m
                  | (even m)     = Cons 0
                  | (odd  m)     = Cons 1
```