CSCI 365 problem set 4

Due Wednesday 19 February, 2020

Revision 1: compiled Tuesday 11th February, 2020 at 10:25

Idiomatic Haskell

Exercise 1 Here is a list of standard library functions that every Haskell programmer ought to be familiar with:

```
fst snd max min even odd
flip (.) show
head last tail init (++) null concat
map foldr concatMap
product sum and or all any length maximum minimum
filter find
iterate repeat replicate cycle take drop splitAt takeWhile dropWhile
reverse sort
zip zipWith unzip
maybe either
```

(There are lots of other useful functions in the standard library, but this is a good start!)

Pick three functions from the above list that you are unfamiliar with. For each function, do the following:

1. Read the documentation for the function at `https://hackage.haskell.org/package/base/docs/Prelude.html`.
2. Write down the function's type and explain in your own words what the function does.[1]
3. Give an example expression which uses the function.

[1] For maximum learning, try this: read the function's documentation, then put it aside and do something else for five minutes. Then try to write down as much about the function as you can remember, without peeking (its type, a description of what it does, *etc.*). Then check to see how accurate you were and whether you missed anything.

Exercise 2 Each of the following function definitions is needlessly verbose. Write an equivalent definition for each that is shorter or more idiomatic. For example, sometimes this may involve simplifying redundant expressions; sometimes it may involve replacing manual recursion with calls to library functions.

(a)
```
f x = \y -> x + y
```

(b)
```
g n
  | n > 2 = True
  | otherwise = (n - 3)^2 > 10
```

(c)
```
p a b = if a > 2*b then False else True
```

(d)
```
fun1 :: [Integer] -> Integer
fun1 []     = 1
fun1 (x:xs)
  | even x    = (x - 2) * fun1 xs
  | otherwise = fun1 xs
```

Hint: (d) and (e) can both be implemented in a single line of code.

(e)
```
fun2 :: Integer -> Integer
fun2 1 = 0
fun2 n | even n    = n + fun2 (n `div` 2)
       | otherwise = fun2 (3 * n + 1)
```

Hint: You may want to use `iterate` and `takeWhile`.

List induction

On this problem set, when you are asked to *prove* something, you should give a formal-style proof using a structured proof format and equational reasoning. On the other hand, if you are asked to *show* or *justify* something, an informal (but still convincing) argument will suffice.

The exercises may refer to the following standard definitions:

```
length :: [a] -> Integer
length []     = 0
length (_:xs) = 1 + length xs

map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs

(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)

(.) :: (b -> c) -> (a -> b) -> a -> c
(g . f) x = g (f x)
```

Exercise 3 You must complete **at least three** out of the following seven proofs involving list induction. You need not do more than three, although you may do more if you feel that the practice is useful.

(a) Prove that `map id = id`.

(b) Prove that for all functions `f` and `g` of appropriate type,

```
map (f . g) = map f . map g.
```

To prove that two functions are equal, it suffices to show they have the same result on all inputs. That is, to prove `f = g`, it suffices to prove that for all `x` of the appropriate type, `f x = g x`.

(c) Give a function `f` such that

```
length = foldr f 0,
```

and prove it.

(d) Give an alternate definition of `(++)` via `foldr`, and prove that your implementation gives the same results as the standard definition.

(e) Prove that `(++)` is associative. That is, prove that for all lists `xs`, `ys`, and `zs`,

```
(xs ++ ys) ++ zs = xs ++ (ys ++ zs).
```

(f) State and prove a theorem relating `map` and `(++)`.

Tree induction

Exercise 4 Consider the following definitions.

```
data Tree a where
  Empty :: Tree a
  Node  :: Tree a -> a -> Tree a -> Tree a

foldTree :: b -> (b -> a -> b -> b) -> Tree a -> b
foldTree e n Empty = e
foldTree e n (Node l a r) = n (foldTree e n l) a (foldTree e n r)

isBST :: Tree Integer -> Bool
isBST Empty = True
isBST (Node l a r) = allTree (<a) l && allTree (>a) r && isBST l && isBST r

allTree :: (a -> Bool) -> Tree a -> Bool
allTree p = foldTree True (\l a r -> l && p a && r)

insert :: Integer -> Tree Integer -> Tree Integer
insert x Empty = Node Empty x Empty
insert x (Node l a r)
  | x == a    = Node l a r
  | x <  a    = Node (insert x l) a r
  | otherwise = Node l a (insert x r)
```

Give a formal proof that `insert` preserves binary search trees—that is, for all `t :: Tree Integer` and for all `x :: Integer`, if `isBST t` then `isBST (insert x t)`.