Programming Languages — Assignment 1
Haskell exercises

Due: Wednesday, 27 Feb 2013, 23:55

The following exercises are intended to give you some practice with the Haskell language and libraries. This assignment is graded out of 10 points.

1. [1 pt] **Feedback.** Complete the survey linked from the course web page after completing this assignment. Any non-empty answer will receive full credit.

2. [1 pt] **Haskell basics: binding and scope**

   For each of the following uses of names, give the line where the name is bound. Explain your reasoning in 1–2 sentences.

   (a) Consider the following Haskell code fragment.

```
1  tau = 6.28
2
3  circumference r =
4    let
5      tau = 6.283185
6    in
7      tau * r
8
9  area r = tau * r * r / 2
```

   - At which line is the use of `tau` at line 7 bound?
   - At which line is the use of `tau` at line 9 bound?

   (b) Consider the following Haskell code fragment.

```
 1  x = 3
 2
 3  foo x = case x of
 4    0 -> 0
 5    x -> (let
 6            x = y + 1
 7          in y) * foo (x - 1) where
 8            y = x + 1
 9
10  y = x + foo x
```

   At which lines are the following variable uses bound?

   - the use of `x` at line 3
   - the use of `y` at line 6
   - the use of `y` at line 7
   - the use of `x` at line 7
   - the use of `x` at line 8
   - the uses of `x` at line 10

3. [1 pt] **Haskell basics: typing**

   Provide an example for the following:

   (a) A three element tuple.

   (b) A list of all integers between 1 and 10 (inclusive).

   (c) A list of tuples, where each tuple has a string as its first element and a character as its second element.

   (d) A function that takes two integers as parameters and returns a boolean.

   (e) Unit.

   (f) `[[[Char]]]`.

4. [2 pts] **Library functions** These exercises are intended to give you practice writing Haskell functions. Do not implement them by simply calling library functions with the same behavior.

   To make the exercise more ~~painful~~ instructive, do not use the boolean operators (including `==`, `/=`, **`not`**, `&&`, `||`); instead, use only **`if–else`** expressions, pattern matching and boolean literals.

   For questions 4–6, use the file `hw1.hs` as a template. This file is available on the course webpage. Please ensure that when you submit your version of `hw1.hs`, that the code runs with `runhaskell`.

   (a) `cube :: Int -> Int`

   `cube x` returns the cube of `x`, `x`^{3}. For example, `cube 2 == 8`.

   (b) `absolute :: Int -> Int`

   `absolute x` returns the absolute value of `x`, |`x`|. For example, `absolute (-1) == 1`.

   (c) `xor :: Bool -> Bool -> Bool`

   `xor x y` returns the eXclusive OR of `x` and `y`. For example, `xor True False == True`.

5. [2 pts] **Recursion**

   The following should be written as recursive functions.

   (a) `largest :: [Int] -> Int`

   `largest xs` returns the largest element of `xs`. For example, `largest [-1, -3, -2] == -1`. Assume that the input list `xs` is not empty.

   (b) `fib :: Int -> Int`

   `fib n` returns the nth Fibonacci number, F_n, with $F_0 = F_1 = 1$ and $F_{n+1} = F_n + F_{n-1}$. Assume `n` ≥ 0.

   (c) Now, write a new version of `fib` function, called `fib'`, that returns the nth Fibonacci number, F_n, in $O(n)$ time. Hint: use a recursive helper function that returns the tuple (F_n, F_{n-1}) for input n.

   (d) This time write a version of the `fib` function called `fib''` that returns the nth Fibonacci number, F_n, in $O(\log n)$ time.

   Hint 1: Let

$$M = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

$$M^n M = \begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} F_n + F_{n-1} & F_n \\ F_{n-1} + F_{n-2} & F_{n-1} \end{pmatrix} = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = M^{n+1}$$

   You can represent a matrix as a 4-tuple.

   Hint 2: If n is even, $M^n = \left(M^{n/2}\right)^2$. If n is odd, $M^n = M M^{n-1}$.

6. [3 pts] **Data structures**

A binary search tree is a binary tree that satisfies an ordering invariant. If n is a node in the tree with data value x, left child l, and right child r, then all values in l must be $< x$ and all values in r must be $\geq x$.

We can represent a binary search tree with the following datatype:

```
data BST = Empty | Node BST Int BST
  deriving (Show, Eq)
```

A `BST` is either `Empty` or a `Node` with a value and two children. The "`deriving (Show, Eq)`" annotation defines **`show`** and `(==)` functions for the data type, allowing trees to be printed and compared.

Define the following four functions:

(a) `bstOk :: BST -> Bool`

`bstOk t` returns **`True`** iff `t` is a valid binary search tree. For example:

```
bstOk (Node (Node Empty 4 Empty) 3 (Node Empty 2 Empty)) == False
```

(b) `bstInsert :: BST -> Int -> BST`

The function `bstInsert t n` inserts integer `n` into tree `t`, returning a new tree. The function constructs and returns a new tree rather than destructively updating the input tree. Both the input tree and output tree should be valid binary search trees. For example:

```
bstInsert 1 Empty == Node Empty 1 Empty
```

(c) `bstDeleteMin :: BST -> (BST, Int)`

`bstDeleteMin t` deletes the minimum element of the tree, returning the new tree and the minimum element. Assume the input tree `t` is not `Empty`. Both the input tree and output tree should be valid binary search trees.

(d) `bstDelete :: BST -> Int -> BST`

`bstDelete t n` deletes the first node in the tree `t` whose value is `n`. If the `n` is not in the tree, the original tree should be returned. Both the input tree and output tree should be valid binary search trees. Hint: use `bstDeleteMin` as a subroutine.