Homework 2: Lectures 3 & 4

CS 440: Programming Languages and Translators, Fall 2020

Due Fri Jan 31, 11:59 pm

1/25 p.2; 1/27 due date [website had right date], 1/30 pp.2,3

How to submit

See http://cs.iit.edu/~cs440 → Homework Policies for information on working with others, how to submit, etc. If you want to submit multiple files, zip them together and submit the zipped file.

Problems [50 points]

1. (2 points) Let `f x y z = x : ( [ y ] : [ z ] )`. What is the type of `f`?

   (This problem really goes with Lecture 2, so it should've been in HW 1.)

A. Lecture 3 [24 points]

Basic Typeclasses

2. (6 = 3 * 2 points) To answer the following questions, use `:info` *Type* to see what typeclasses *Type* is an instance of, then use `:info` *TypeClass* to see what functions / operators the different typeclasses support.

   a. The test `False` < `True` is allowed because < is provided by a typeclass that `Bool` is an instance of. What is the typeclass and what is the type of (<) (including the typeclass)?

   b. What are the functions that give the ASCII code for a character and give the ASCII character for an integer (if you use a type annotation `:: Char`)? (I.e., *fcn1* `'a'` yields `97`, *fcn2* `97 :: Char` yields `'a'`.) Also, what are their types (including the typeclass)?

   c. The functions in part (b) are provided by a typeclass that `Char` is an instance of. What is the typeclass?

Function Definitions; Patterns; Case Expressions

3. (18 points total) The function `twice` *list* should return true iff some value occurs twice in the list. E.g.,

```
> filter twice [[],[1],[1,2],[2,2],[1,2,3],[1,2,1],[1,1,2],
[1,2,2]]
[[2,2],[1,2,1],[1,1,2],[1,2,2]]
```

   a. (2 points) What is the type of `twice`? (Include the typeclass.)

   b. (4 points) Briefly describe the syntactic and semantic bugs in the program below. (For syntactic errors, don't just parrot the Haskell error messages; give a brief human-understandable description.)

```
:{
twice [] = False
twice [_] = False
twice [x,x] = True
twice ( _ ++ [x] ++ _ ++ [y] ++ _ ) = x == y
twice (h1 : h2 : t) == (h1 == h2 || twice h1 t)
:}
```

[1/30 -- oops! Had two part b's.] For parts c - f, feel free to give your functions different names (presumably variations on "twice"; twice_c, twice_d, e.g.)

c. (3 points) Rewrite `twice` to make it work. Keep using definition by cases; feel free to add/change/delete cases as you see fit.

d. (3 point) Write a definition by cases for `twice` that only has two cases (one recursive, one not).

e. (3 points) Rewrite your definition from part (c) using cases and guards; break up the 3-clause logical or test to use a sequence of guards. (Don't leave any `||` in the definition.)

```
twice x pattern
    | guard1 = result1      [1/25: need = not -> ]
    | guard2 = result2
  (omitted)
```

f. (3 points) Rewrite your definition from part (c) to be of the form `twice x = case x of` …. You can add guards to a case clause using the syntax

```
case expr of pattern | guard1 -> result1
                     | guard2 -> result2
                  (omitted)
```

B. Lecture 4 [24 points]

Higher-Order Functions

4. (2 points) Consider the following claim: "*A Haskell function is higher order if and only if its type has more than one arrow*." Is this correct? Give a brief argument.

Currying/Uncurrying

5. (4 = 2 * 2 points) Let `f :: (a -> a -> a) -> a -> a -> a`.

   a. Rewrite `f * (2 3)` so that it has no syntax errors and yields `6` if `f h x y = h x y`

   b. Write the definition of a function `g :: ((a, a) -> a, (a, a))-> a` so that `g` is an uncurried version of `f`. Calling your function on `*`, `2`, and `3` should yield `6`.

[1/30] ~~Map and~~ Filter

6. (4 = 2 * 2 points) Let `f1 = filter (\ x -> x > 0)` and `f2 = filter (\x -> x < 10)`, and let `nbrFilter g x = length (filter g x)`.

   a. Rewrite `f1(f2[-5..15])` so that it uses function composition to apply just one function to the list.

   b. Rewrite the `nbrFilter` function definition to have the form

      `nbrFilter g =` *function composition involving* `length` *and* `filter` … *and leaving out* `x`.

Lambda Functions

7. (8 = 6 + 2 points)

a. Rewrite `f g x y = g x (y x)` three ways, first `f g x` = unnamed lambda function, then `f g` = unnamed lambda function, and finally `f` = unnamed lambda function.

b. Briefly, how does *var* = *lambda function* relate to first-class functions in Haskell?

List Folding

8. (6 = 3 * 2 points) Let's re-implement the `foldl` function in multiple ways. Your `foldl` only needs to work on lists.

a. Write a definition for `foldl` using conditional expressions: `foldl1 f a x = if x == [ ]` then etc.

[1/30] Make it foldl1a, since there's already a foldl1 in the library.

b. Rewrite the definition using function definition by cases: `foldl2` …

c. Rewrite the definition using a `case` expression: `foldl3 f a x = case x` ....