Homework 2 Solution

CS 440: Programming Languages and Translators, Fall 2019

Problems

1. (Polymorphic type)

   If `f x y z = x : ( [ y ] : [ z ] )`, then `f :: [a] -> a -> [a] -> [[a]]`

2. (Typeclasses)

   a. For `False < True`, `<` is provided by typeclass `Ord` and has type `(<) :: Ord a => a -> a -> Bool`.

   b. The ASCII integer for a character `c` is `fromEnum c`. The ASCII character for an integer `n` is `toEnum n :: Char`. (You need the type annotation because otherwise `toEnum` doesn't know what target type it should be aiming for.) Their types are

   ```
   fromEnum :: Enum a => a -> Int

   toEnum :: Enum a => Int -> a
   ```

   c. `Char` is an instance of `Enum`, which provides `fromEnum` and `toEnum`.

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?

3. (`twice x` = does some value occur twice in `x`?)

   a. (Code problems) Let me number the lines of code to make referencing them easier:

   ```
   1.   twice [] = False
   2.   twice [_] = False
   3.   twice [x,x] = True
   4.   twice ( _ ++ [x] ++ _ ++ [y] ++ _ ) = x == y
   5.   twice (h1 : h2 : t) == (h1 == h2 || twice h1 t)
   ```

   So for errors,

   - Line 3: Can't use x twice in the pattern [x,x]. One fix: Replace the line by

     ```
     twice {x,y] = (x == y)
     ```

   - Line 4: Uses a bad pattern: You can only try to match lists against `[…]` and `(:)`; you can't use a general function. The whole line needs to be deleted.

- Line 5: `twice h1 t` should be `twice (h1 : t) because` twice is supposed to be passed a list. We're also missing a test: We need to check for `h2` occurring twice in `h2 : t`, otherwise we'll miss lists like `[1,2,2]` where `h1` = 1 ≠ 2 ≠ `h2` and `twice(h1:t)` = `twice [1,2]` = `False`, but `twice(h2:t)` = `twice [2,2]` = `True`. The ful line becomes

```
twice (h1 : h2 : t) = (h1 == h2 || twice (h1:t) || twice (h2:t))
```

b. (Rewrite using function definition by cases)

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

c. (Rewrite using only two cases)

```
twice (h1 : h2 : t) = (h1 == h2 || twice (h1:t) || twice (h2:t))
twice _ = False  -- for [] and [_] arguments
```

d. (Rewrite using cases and guards) The `otherwise -> False` case can be omitted; we'll just fall into the `twice _ = False` case.

```
twice (h1 : h2 : t)
    | h1 == h2 = True
    | twice (h1:t) = True
    | twice (h2:t) = True
    | otherwise = False
twice _ = False
```

e. (Rewrite using a case expression and guards)

```
twice x = case x of
    (h1 : h2 : t)
        | h1 == h2 -> True
        | twice (h1:t) -> True
        | twice (h2:t) -> True
    _ -> False
```

4. Say we have `f` :: *type1* `->` *type2*. For `f` to be higher-order, either *type1* or *type2* (or both) involve functions. In the first case, *type1* contains an arrow; in the second, *type2* contains an arrow. So `f` is higher-order iff it has at least two arrows.

5. (Function `f :: (a -> a -> a) -> a -> a -> a` .)

   a. If `f h x y = h x y`, then `f (*) 2 3` has no syntax errors and yields `6`.

   b. Define `g (h, (x,y)) = h (x,y)`, then `g :: ((a, a) -> a, (a, a))-> a` and `g (uncurry (*), (2,3)) = 6`.

6. (Map & 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. `f1(f2[-5..15]) = (f1 . f2)[-5..15]`

   b. `nbrFilter g x = length (filter g x)` is equivalent to

   `nbrFilter g = length . (filter g)`. To see this, we can use substitutions:

   `nbrFilter g = length . (filter g)` is the same as

   `nbrFilter g x = length . (filter g) x` is the same as

   `nbrFilter g x = length ((filter g) x)`.

7. (`f g x y = g x (y x)`.)

   a. `f g x = \ y -> g x (y x)`

   `f g = \x y -> g x (y x)`

   `f = \g x y -> g x (y x)`

   b. Having *var = unnamed lambda function* means that giving a function a name is just like giving any other kind of expression a name; treating functions like any other kind of value is what first-class functions are about. (The `f x =` *expr* syntax is there to make life easier.)

8. (Redefine `foldl` on lists)

   a. (With a conditional expression)

```
foldl1 f a x
    = if x == [ ] then a else foldl1 f (f a (head x)) (tail x)
```

   b. (Function definition by cases)

```
foldl2 _ a [] = a
foldl2 f a (h : t) = foldl2 f (f a h) t
```

   c. (Using a `case` expression)

```
foldl3 f a x = case x of [] -> a
                         h : t -> foldl3 f (f a h) t
```


