ASGS1115 - Support for Science

Semester 1 - 2019

LAB week 4: Syntax in Functions

Typeclasses Reminders

- Num
- Integrals
- Enums
- Floating
- Ord
- Read
- Show

Pattern Matching

Load the Haskell script PatternMatching.hs which contains the following Haskell functions:

```
1  weightedDice :: (Integral a) => a -> Double
2  weightedDice 6 = 0.05
3  weightedDice 5 = 0.05
4  weightedDice 4 = 0.1
5  weightedDice 3 = 0.2
6  weightedDice 2 = 0.3
7  weightedDice 1 = 0.3
8  weightedDice a = 0
9
10 dotProduct :: (Num a) => (a,a) -> (a,a) -> a
11 dotProduct (v1,v2) (u1,u2) = (v1*u1) + (v2*u2)
12
13 firstmult :: (Num a) => (a,a) -> (a,a) -> a
14 firstmult v1 v2 = v11 * v21
15     where (v11,_) = v1
16           (v21,_) = v2
```

Try putting inputs to the functions and observe the outputs.

- What do you think this functions does?
- How What does each lines in the function definition mean?
- What does **where** do?

Guards

Load Haskell script Guards.hs, which contains the following function definition:

```
grade :: (RealFloat x) => x -> String
grade x
    | x > 100 = "Impossible"
    | x >= 80 = "HD"
    | x >= 70 = "D"
    | x >= 60 = "C"
    | x >= 50 = "P"
    | x >= 0 = "F"
    | otherwise = "Invalid"
```

Try putting inputs to the function and observe the outputs.
Pipe symbols | on each line of the function definition indicates the 'guard' for the function.

- In what cases do you think the guard can be useful?
- what will happen if some parts of the input domain is not defined in the guard?

Challenge:
What if the grading scheme is not from 0-100? Write a function called `gradearb` which takes three argument, the score, upper bound of the score, and lower bound of the score. It should still gives the same grade for the same percentage (e.g 'HD' for $\geq 80\%$).
Hint: use **where**

Control Flows - If statements and Cases

Load Haskell script FunctionConditionals.hs, which contains the function defnitions below:

```
reLu :: (RealFloat x) => x -> x
reLu x = if (x < 0) then 0 else x

lenl :: (Num x) => [x] -> x
lenl xs =
    case xs of  [] -> 0
                (x:xs) -> 1+ (lenl xs)
```

Try inputting values to these functions and observe the outputs.

- In what use case will **if** and **case** better suited compared to the other?

Challenge:
Rewrite these following functions either with **case** or **if**.

```haskell
topsecret :: (String s) => s -> s
topsecret password = "Haskell is awesome!"
topsecret s = "Incorrect password"
    where password = "1234"

suml :: (Num x) => [x] -> x
suml [] = 0
suml (x:xs) = x + (suml xs)

averagel :: (Num x) => [x] -> x
averagel [] = 0
averagel xs = (suml xs) / (lenl xs)
```