Modelling the World

Preparation

It is recommended that you read the following chapter before doing this tutorial:

- Types

You should also be able to:

- Navigate around Linux
- Use GHCi, especially loading and reloading source files in GHCi.

Post your questions on Piazza.

1 Basic Types in Haskell

Haskell has a very sophisticated *type system*. For now, you only need to know that **every** value has an associated type: for example, some values are integers, `5 :: Int`, some are characters, `'c' :: Char`, some are lists of integers, `[1,2,3] :: [Int]`, and some are functions, `words :: String -> [String]`.

For those who have programmed in languages like Python or JavaScript, *typing* might be a rather unfamiliar concept. To declare `a has a value of 5`, we'd write:

```
a = 5
```

Python will read this as an integer, but will automatically coerce the value to a different type when we try to add values of different types (integers, floating point numbers etc.).

In languages like Java, we have to give the type when we declare a variable; to declare (roughly) the same thing, we need to write:

```
int a = 5
```

We explicitly mention that `a` is an integer, but Java will still coerce the value if we try to add it to a floating point value.

In Haskell, we may declare the type; if we do not Haskell will infer the most general type consistent with the context, but we may also specify the type. We write something similar[1]:

```
a :: Int  -- We read this as "a has type Int"
a = 5
```

You can test what type a value has in GHCi by using the `:type` command (or abbreviate this as `:t`).

For example:

```
GHCi> :type True
True :: Bool
```

1. Your first task is to find the types of the following values:

Value	Type	Value	Type	Value	Type	Value	Type
`True`	`Bool`	`3.14`		`[1, 2, 3]`		`(1, 2)`	
`False`		`'A'`		`[1, 2, 3.14]`		`(1, 2, 3)`	
`1`		`'b'`		`['A', 'b', ' ']`		`Just "Me"`	
`2`		`' '`		`"Ab "`		`Nothing`	

Some answers you get from GHCi might be somewhat cryptic, or sometimes bewideringly verbose, bring them to the tutorials and discuss with your tutors and friends.

[1] It's a good practise always to include type declarations.

2 Rock, Paper, Scissors

In this exercise, we will model the game Rock, Paper, Scissors. There are three *moves* in the game, namely *rock*, *paper*, and *scissors*. Rock blunts scissors; scissors cut paper; paper wraps rock. In each of these cases the first-mentioned wins.

We can enumerate them to create a new type in Haskell using the `data` keyword (we are making a new data type):

```
data Move = Rock | Paper | Scissors deriving (Eq, Show)
```

Haskell also has a data type, called `Ordering`:

```
data Ordering = LT | EQ | GT    -- Less than, equal to, greater than
```

and we can use it to represent the outcome of a game.

1. Load the file Tutorial1.hs[2] into GHCi and your task is to complete the function `order` that takes two moves and output an `Ordering`:

   ```
   order :: Move -> Move -> Ordering
   ```

   Below is the expected behaviour of this program for some arguments:

   ```
   GHCi> order Rock Scissors
   GT
   GHCi> order Rock Rock
   EQ
   GHCi> order Scissors Rock
   LT
   ```

   Rock is greater than scissors, rock is equal to rock, scissors is less than rock etc.

There are a few different ways to define this function, it is perfectly valid as long as you get all the possible cases covered. Compare your solution with your peers'.

[2]It should be bundled in the same zip file as this tutorial sheet. If not found, you can get it here: https://gavinphr.github.io/INF1A/tutorial1/Tutorial1.hs

3 Fruits and their Colours

Continue to use Tutorial1.hs in this exercise.

We have 7 different fruits:

- Strawberry
- Plum
- Pear
- Mandarin
- Grape
- Apple
- Banana

And 5 different colours:

- Red
- Purple
- Yellow
- Green
- Orange

Our task is to write a Haskell function that takes a value of type `Fruit` and returns its `Colour`.

1. First make two new data types `Fruit` and `Colour`. Complete the `data` declarations in Tutorial1.hs:

   ```
   data Fruit = {--Complete this part--} deriving (Show)
   data Colour = {--Complete this part--} deriving (Show, Eq)
   ```

   The final result should look similar to the `data` declarations in the previous exercise.

2. Now we can make a function that takes in a fruit and outputs a colour. Fill in the function declaration of `colourOf`:

   ```
   colourOf :: Fruit -> Colour
   ```

   Your answer should cover the 7 different cases that match the 7 different fruits. Some fruits can have multiple colours, just choose the one you like.

   Here is an example of the expected behaviour:

   ```
   GHCi> colourOf Banana
   Yellow
   ```

Using the `colourOf` function is like asking a wh-question in the sense that:

```
Q: What colour is a pear?          GHCi> colourOf Pear
A: Green                           Green
```

What if we are only allowed to ask yes-no questions? You might say, that is simple we can just keep asking questions until we get a positive response. For example:

```
Q: Is a pear Red?     A: No.
Q: Is a pear purple?  A: No.
Q: Is a pear yellow?  A: No.
Q: Is a pear green?   A: Yes.
```

3. Your task is to implement this exact feature, so that we can ask Haskell:

```
GHCi> isRedFruit Pear
False
GHCi> isPurpleFruit Pear
False
GHCi> isYellowFruit Pear
False
GHCi> isGreenFruit Pear
True
```

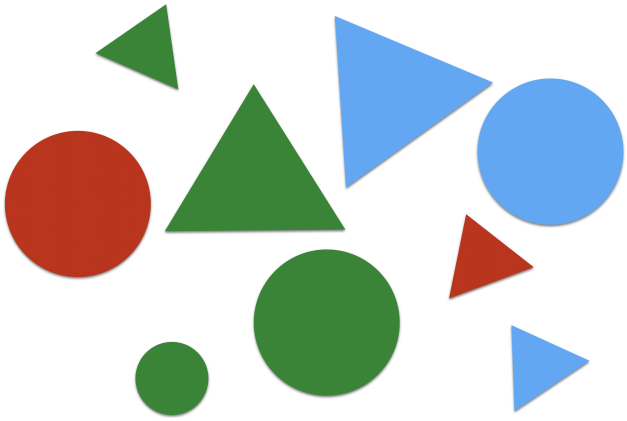
The type declarations are already written for you in Tutorial1.hs. You can make use of the `colourOf` function.

Remark:

We will be dealing with yes-no questions (or rather predicates that are functions returning a boolean value, either true or false) for the majority of this course. The key takeaway from this exercise is that we can acquire the same information by asking either a wh-question, or a series of yes-no questions.

4 A Universe of Discs and Triangles

The figure below shows our universe of **things**:



Every **thing** in this universe is red or blue or green, small or big, a disc or a triangle.

1. Determine whether each of the statements below is true or false.

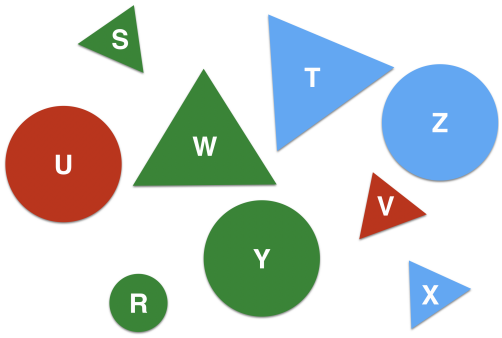
   - Some big disc is red.
   - Every red triangle is small.
   - Some big triangle is green.
   - Every small disc is red.
   - No red thing is blue.

   For this small universe it is easy to work out which are true, by inspection; it might not be so easy for a larger universe. We can write Haskell functions to decide whether statements similar to the ones above are true or false.

   Let's first introduce names for the things in our universe, and a new type to represent the universe of things:

```
data Thing = R | S | T | U | V | W | X | Y | Z deriving ( Eq, Show )

things :: [Thing]
things = [R, S, T, U, V, W, X, Y, Z]
```



Then we define 7 functions[3]:

```
isRed      :: Thing -> Bool
isBlue     :: Thing -> Bool
isGreen    :: Thing -> Bool
isDisc     :: Thing -> Bool
isTriangle :: Thing -> Bool
isSmall    :: Thing -> Bool
isBig      :: Thing -> Bool
```

Now we can test whether a thing has each specific property.

```
GHCi> isRed U
True
GHCi> isDisc U
True
GHCi> isBlue Y
False
```

Now we have everything we need to put Haskell to work. For the first statement from question 1,

Some big disc is red.

First we find all the things that are discs and big:

```
GHCi> [x | x<-things, isBig x, isDisc x]
[U,Y,Z]
```

For each red disc, test whether it is red:

```
GHCi> [isRed x | x <- things, isBig x, isDisc x]
[True,False,False]
```

Then test whether at least one of them is red, using the `or` function:

```
GHCi> or [isRed x | x <- things, isBig x, isDisc x]
True
```

2. Continue to use Haskell to find the truth values of the remaining statements from question 1. You might find the functions `and` and `not` helpful.
   - Every red triangle is small.
   - Some big triangle are green.
   - Every small disc is red.
   - No red thing is blue.
3. Our Universe has three colours, two sizes, and two shapes, but only 9 of the 12 possible kinds of individual.

   Find examples of statements of the form "Every X Y is Z" that are true, and of the form "Some X Y is Z" that are false in our toy universe. X, Y, and Z may be any predicates; Z may also be negated, e.g. not red, not disc.

   Can you list **all** the statements of the form "Some X Y is Z" that are false?

[3]Function definitions are not shown here but you can find them in Tutorial1.hs.