Functions

Functions play a major role in Haskell, as it is a functional programming language. Like other programming languages, Haskell does have its own functional declaration and definition.

Functions declaration

Function declaration consists of the function name and its argument types list along with its output. The function declaration specifies the type of the function. The general syntax is:

```
myFunctionName :: [(Constraints) => typeOfInput1 -> typeOfInput2 -> ...
-> typeOfInputn] -> typeOfOutput
```

This is telling the compiler that myFunctionName is a function that takes n values as input (n can be zero) of type typeOfInput1, typeOfInput2, … typeOfInputn respectively and returns a value of type typeOfOutput. Constraints is a list comma separated of constraints.

Note: In Haskell we can have a function with no input. But a function always has an output.

Functions definition

Function definition include the following:

- The function name, which in Haskell always starts with a lowercase letter
- The list of parameters, each of which must also begin with a lowercase letter, separated from the rest by spaces (not by commas, like in most languages) and not surrounded by parentheses
- An equal (=) sign
- And the body of the function which defines the return expression. It is all what is at the right of the equal (=) sign.

Let us take a small example of the following function to understand this concept in detail. $addSqrt(x, y) = (x - 3)^2 + (y - 2)^2$ is a mathematical function, its declaration and definition in Haskell is:

```
addSqrt :: Int -> Int -> Int      --Function declaration
addSqrt a b = (a-3)^2 + (b-2)^2   --Function definition
```

The declaration tells the compiler that addSqrt is a function that takes two Int values as input and returns an Int value. But to be generalize the notion, we can consider the following:

```
addSqrt :: (Num a) => a -> a -> a  --Function declaration
addSqrt a b = (a-3)^2 + (b-2)^2    --Function definition
```

The previous declaration means that, the function $addSqrt$ take as input two things that are numeric values and also return a numeric value.

Activity:

1. Create a file called addSqrt.hs
2. Copy and paste this code in the file: `addSqrt  a b = (a-3)^2 + (b-2)^2`
3. Load the file from the REPL using the command
4. Query the type of the function addSqrt

Lesson: In your code you are not required to specify the function type. But it is best practice to specify it. As the declaration specifies the function type, when you don't specify it, Haskell will use inference to assign a type to your function. Haskell will always try to assign the most generic type possible.

Calling a function/Applying function to argument(s)

We call a function (or apply a function) when we want to get the output for a given input.

If you evaluate `addSqrt 4 7` in your REPL, you will get 26 as output. We also say that we applied the function addSqrt to the arguments 4 and 7.

Conditional expression

The right part of the equal (=) sign in the function definition is the expression returned by the function. The expression can be a conditional expression and is usually introduce by the if .. then … else construct. Its syntax is as follows:

```
if<Condition> then <True-Expression> else <False-Expression>
```

- Condition − It is the binary condition which will be tested. it is an expression that returns True or False.
- True-Expression − It refers to the output that is evaluated and returned as result when the Condition satisfies or evaluates to True
- False-Expression − It refers to the output that is evaluated and returned when the Condition does not satisfy evaluates to False

As Haskell codes are interpreted as mathematical expressions, the above statement will throw an error without the else block. The following code shows how you can use the if… then … else statement in Haskell.

```
a = 34
x = if (mod a 5 == 0) then 0 else 1
```

If you evaluate x in the REPL, you should get 1

Ways to define functions / functional patterns

Pattern matching

Pattern matching consists of specifying patterns to which some data should conform and then checking to see if it does and deconstructing the data according to those patterns.
When defining functions, you can define separate function bodies for different patterns.
We want a function that says the numbers (in letters) from 1 to 5 and says "Not between 1 and 5" for any other numbers?

Without pattern matching, we'd have to make a pretty convoluted if then else tree. However, with it we can do the following:

```
sayMe:: (Integral a) => a -> String
sayMe 1 = "One!"
sayMe 2 = "Two!"
sayMe 3 = "Three!"
sayMe 4 = "Four!"
sayMe 5 = "Five!"
sayMe x = "Not between 1 and 5"
```

The pattern is: for a given number, the number is between 1 and 5 or it is not.

wada

Guards

Whereas patterns are a way of making sure a value conforms to some form and deconstructing it, guards are a way of testing whether some property of a value (or several of them) are True or False. That sounds a lot like an if statement and it's very similar. The thing is that guards are a lot more readable when you have several conditions and they play really nicely with patterns. Instead of explaining their syntax, let's just dive in and make a function using guards. We're going to make a simple function that berates you differently depending on your BMI (body mass index). Your BMI equals your weight divided by your height squared. In most western cultures If your BMI is less than 18.5, you're considered underweight. If it's anywhere from 18.5 to 25 then your weight is considered normal. 25 to 30 is overweight and more than 30 is obese. So here's the function (we won't be calculating it right now, this function just gets a BMI and tells you off

```
bmiTellInCameroonContext :: (RealFloat a) => a -> String
bmiTellInCameroonContext bmi
    | bmi <= 18.5 = "You might be underweight, you should eat more!"
    | bmi <= 25.0 = "You are fine, I will talk to your dear kento!"
    | bmi <= 30.0 = "I see your dear kento is  taking good care of you!"
    | otherwise   = "You're wealthy and your kento takes care of you!"
```

Guards are indicated by pipes that follow a function's name and its parameters. Usually, they're indented a bit to the right and lined up. A guard is basically a boolean expression. If it evaluates to True, then the corresponding function body is used. If it evaluates to False, checking drops through to the next guard and so on. If we call this function with 24.3, it will first check if that's smaller than or equal to 18.5. Because it isn't, it falls through to the next guard. The check is carried out with the second guard and because 24.3 is less than 25.0, the second string is returned.

Another example:

```
myMax :: (Ord a) => a -> a -> a
myMax a b
  | a > b = a
  | otherwise = b
```

Where

Well, we can modify our function like this, passing to it height and weight:

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
      | bmi <= 18.5 = "You might be underweight, you should eat more!"
```

wada

```
    | bmi <= 25.0 = "You are fine, I will talk to your dear kento!"
    | bmi <= 30.0 = "I see your dear kento is  taking good care of you!"
    | otherwise   = "You're wealthy and your kento takes care of you!"
      where bmi = weight / height ^ 2
```

Let's see where I stand with my weight.

```
Prelude> bmiTell 85 1.90
"You are fine, I will talk to your dear kento!"
```

Let and In

Very similar to where bindings are let bindings. Where bindings are a syntactic construct that let you bind values or expressions to names. Let bindings let you bind to variables anywhere and are expressions themselves, but are very local, so they don't span across guards. Just like any construct in Haskell that is used to bind values to names, let bindings can be used for pattern matching. Let's see them in action! This is how we could define a function that gives us a cylinder's
surface area based on its height and radius:

```
cylinder :: (RealFloat a) => a -> a -> a
cylinder r h =
    let sideArea = 2 * pi * r * h
        topArea = pi * r ^2
    in sideArea + 2 * topArea
```

Case Expressions

Many imperative languages have Switch case syntax: we take a variable and execute blocks of code for specific values of that variable. We might also include a catch-all block of code in case the variable has some value for which we didn’t set up a case.
But Haskell takes this concept and generalizes it: case constructs are expressions, much like if expressions and let bindings. And we can do pattern matching in addition to evaluating expressions based on specific values of a variable. The syntax for case expressions is as follows:

```
case Expression of
    pattern -> result
    pattern -> result
    pattern -> result
    ...
```

wada

Expression is matched against the patterns. The pattern matching action is the same as expected: the first pattern that matches the expression is used. If it falls through the whole case expression and no suitable pattern is found, a runtime error occurs.

Example:

```
describeList :: [a] -> String
describeList xs =
    "The list is "
    ++ case xs of
         [] -> "empty."
         [x] -> "a singleton list."
         (y:ys) -> "a longer list."
```

Anonymous functions

λ-expressions (λ is the small Greek letter lambda) are a convenient way to easily create anonymous functions — functions that are not named and can therefore not be called out of context — that can be passed as parameters to higher order functions like map, zip, fold, etc.
An anonymous function is a function without a name. It is a Lambda abstraction and might look like this: \x -> x + 1. (That backslash is Haskell's way of expressing a λ and is supposed to look like a Lambda.)

Activity

1. Let consider the function `basicMax :: (Ord a) => a -> (a -> a)`. It takes as input a value that can be ordered and returns a function. If we want the function basicMax to be defined with the max function, we should define basicMax like this: `basicMax x = \y -> max x y` in this writing we are saying that `basicMax x` returns the function that for all variables y returns max x y. In the expression `\y -> max x y` x is a free variable.
2. In the REPL run the command: `(basicMax 7) 5`

Lesson:

We define anonymous functions by using the symbol \ and ->. The syntax is as follow:
$\backslash x_1\, x_2 \ldots x_n \; -> expression$ where x_1, x_2, …, x_n are the argument of the anonymous function. Arguments are spaces separated. $expression$ is the expression that defines the return value. All variable that do not belong to the set { x_1, x_2, …, x_n} are called free variables.

wada

Recursive function

What is recursion

Recursion is actually a way of defining functions in which the function is applied inside its own definition. Definitions in mathematics are often given recursively.

Example

GCD

What is GCD (Greatest Common Divismygcd m n
  | restDivmn == 0 = n
  | otherwise = mygcd n restDivmn
   where restDivmn = mod m n
or)?
In mathematics GCD or Greatest Common Divisor of two or more integers is the highest positive integer that divides both the number with zero as remainder.

Example: GCD of 20 and 8 is 4.
How do we determine the GCD of two numbers using the Euclidean Algorithm?
Description of the algorithm: The algorithm states that the GCD of two numbers a and b is the last not null remainder in the successive division of a and b. It uses the property:

$$GCD\ a\ b\ =\ GCD\ b\ r\ where\ r\ is\ the\ remainder\ of\ a\ divided\ by\ b.$$

for instance:
a = 782, b= 221
step1: a divided by b gives: quotient = 3 and remainder 119

a = 221, b = 119
Step2: a divided by b give: quotient = 1 and remainder 102

a = 119, b = 102
Step3: a divided by b give: quotient = 1 and remainder 17

a = 102, b = 17
Step4: a divided by b give: quotient = 6 and remainder 0 (null). The algorithm ends at this step. The last not null remainder is 17

GCD of 782 and 221 is **17** .

implementation in Haskell is very simple using recursion.

```
mygcd :: (Integral a) => a -> a -> a
mygcd m n
   | restDivmn == 0 = n
   | otherwise = mygcd n restDivmn
    where restDivmn = mod m n
```

Factorial

We start by saying that the factorial of 0 is 1. Then we state that the
a factorial of any positive integer is that integer multiplied by the factorial of its predecessor.
Here's how that looks translated in Haskell terms.

```
factorial :: (Integral a) => a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)
```