**Haskell**

<aside> ⚙ **Commands on GHCI** ❯ **ghci:** open interative terminal ❯ **:r:** recompile file ❯ **:l File.hs:** run the file, add module Module where (it able to run file without extension) ❯ **:t Value:** checks the data type

</aside>

<aside> 📌 \*\*Basic Concepts

Types →\*\* Char: 'H' → [Char] or String: "Hello World!" \*\*\*\*→ Int: 1 → Bool: True/False → creating new data type: data Day = Monday | Sunday \*\*new type is Day and it has value constructors (Monday or Sunday) → Record syntax: data Point = Point {x, y :: Double} → creating new type with only one field: newtype Data = Data Int \*\*lazy evaluation

**Data Structure** → List: [1, 2, 3, 4] \*\*same element type and dynamic length → Tuples: (1, "Hello", 2, 'j') \*\*several element type and fixed-length

**Operators** → maths operations: + - \* / → conditions operations: True False not && || → equal, not equal: == /= \*\*→ sucessor: succ 8 → min max: min max [1, 2] → div (integral division): div 2 3 → mod (modular division): mod 2 3 → odd | even (check is number is odd | even): odd|even 2

\*\***Conditional Statement** *→ if x > 100 then x else x\*2*

**Functions** → nameFunction p1 p2 p3 = expression → call the function: nameFunction p1 p2 p3

**Pattern Matching;** → Inspect parts of a value constructor → h [] = 0 h (\_:[]) = 1 \*\* \_ ignore value h (\_:x:[]) = 2 + x \*\*x accepts any value

</aside>

**Typeclasses**

<aside> 💡 **Typeclasses Highlights** → A typeclass is a sort of interface that defines some behavior → If a type is a part of a typeclass, that means that it supports and implements the behavior the typeclass describes → **Eq:** used for types that support equality testing, implement == and /= → **Show:** all types covered so far except for functions are a part of this class, it takes a value whose type is a member of Show and presents it to us as a string → **Read:** sort of the opposite typeclass of Show, it takes a string and returns a type which is a member of **Read** → **Num:** numeric typeclass, its members have the property of being able to act as numbers. It has the functions: \*abs\* (transform negative values on positives one); *signum* (indicate if the number is positive, negative or zero); *fromInteger* (convert the interger number to other type) → **Fractional**: it provides a way to use the division for your type. It has the function \*fromRational\* (convert a fraction in another type) → **Real**: it provides the function \*toRational\* (convert the type to a Fractional) → **Integral**: includes only integral (whole) numbers. In this typeclass are Int and Integer. It has the functions \*quotRem\* (return the quotient and the rest); \*toInteger\* (convert the type to an Integer) → **Enum:** members are sequentially ordered types — they can be enumerated. It has the functions: \*succ\* (get the successors values) and *pred* (get ht predecessors values). → **Ord:** for types that have an ordering with >, <, >= and ≤, it has the \*compare\* that returns GT, LT or EQ; also *max* and \*min\* → **Bounded:** members have an upper and a lower bound. it has the *minBound* and *maxBound*

Creating a new class: *class SimNao a where simnao :: a -> Bool*

</aside>

**Lists**

<aside> 🌟 **List Highlights**

→ to join lists: *[1, 2] ++ [3, 4] (++) [1, 2] [3, 4] or "Hello" ++ " World!" →* to *push a new element on the first position: 3 : 4 : 4 : [10, 9, 4] → to get list length: length [1, 2, 3]* *→ to get the first element: head "ABCD" or head [1, 2] → to get last element: last "ABCD" or last [1, 2] → to get the list without first element: tail "ABCD" or tail [1, 2] \*\* head, last and tail are not refined to empty lists → to get all elements except the last one: init "ABCD" or init [1, 2] → to reverse a list: reverse "HASKELL" or reverse [1, 2, 3] → to extract elements from the beginning: take n [2, 4, 5] → to extract elements dropping n of the begin: drop n [1, 2] →* to get element on index: *[1, 2, 3] !! 2 →* to check if a element exist inside a list: \*4 elem [3,4,5,6]\* *→* to check if a list is empty: *null [1, 2] →* to sum list of numbers: *sum [1, 2] →* to multiply the list of numbers: *product [1, 2] →* to get the max or min of a list of numbers: *maximum | minimum [1, 2] →* get the element and the rest\*: x:xs\*

**Ranges** → to produce a value range: \*[1..20]\* → to cycle into an infinite list: \*take 12 (cycle "LOL ")\* → to produce an infinite list of just one element: \*take 10 (repeat 5)\* → to replicate a number into a list: \*replicate 3 10\*

\*\***List Comprehensions** → to build lists using expressions that will be distributed in each element → [EXPRESSION(var) | var←LIST, FILTER\_1, FILTER\_2...FILTER\_n] → doubleList xs = [2\*x | x←xs] → list = [2\*x+1 | x←[0 .. 10], x/=5]

</aside>

**Tuples**

<aside> 🌟 \*\*Tuples Highlights

→ tuples are immutable, each element is fixed and each index is called a coordinate → get the first coordinate of a tuple: fst ('H', "HELLO") → get the second coordinate of a tuple: snd ('H', "HELLO") → produce a list of pairs: zip [1, 2] [3,4] //[(1, 3) (2, 4)]

</aside>

**Functions**

<aside> 🌟 **Functions Highlights** \*\* → doc the type on functions: *biggerThan :: Int → Int → Bool biggerThan x y = x > y*  
→ infix function: \*biggerThan x y = (>) x y

→\* **Lambda**: anonymous functions, they can be executed as values and without an explicit context \*\\p1 p2 → EXP(p1 p2)

→\* **High Order Function**: it can be passed as parameters or return another function *ev :: (Int -> Int) -> Int* **- map:** takes a function and a list and applies that function to every element in the list, producing a new list *map (+3) [1,5,3,1,6] -* **filter:** takes a predicate and a list and then returns the list of elements that satisfy the predicate *filter (>3) [1,5,3,2] -* **foldr**: takes a binary function, a starting value (accumulator) and a list to fold up *foldl (\\acc x -> x : acc) []* \*\***foldl**: it does the same, but starts by the left side

\*\*→ **Currying**: it receives multiply arguments and returns a function's sequence evaluation *sum x y z = x + y + z* \*\*if a parameter was not passed, it returns a function with it

→ **Composition**: it's a functions chain *(funcA . funcB)*

→ **Function application ($):** the expression on its right of $ is applied as the parameter to the function on its left *sum $ map sqrt [1...5]*

→ **Guards (|):** a way of testing whether some property of a value (or several of them) are true or false \*age | age < underAge = "You're underaged" | otherwise = "You're an adult!" where underAge = 18\*

**Guards in Haskell**

While patterns are a way of making sure a value conforms to some form and de-constructing it, guards are a way of testing whether an argument (or several arguments) satisfies a property or not.  
  
This is very similar to an if statement, but guards are a lot more readable when we have several cascaded conditions we want to check. And they play really nicely with patterns.

#### A BMI function

We’re going to make a simple function that lists the range of a particular BMI (body mass index). The BMI is the weight (in kg) divided by height (in m) squared. If a BMI is less than 18.5, it’s in the underweight range. If it’s anywhere between 18.5 to 25, it’s in the normal range. 25 to 30 is overweight and more than 30 is obese.

ghci 67> let {bmiTell :: (Floating a, Ord a) => a -> String;

bmiTell bmi

| bmi 6 18.5 = "underweight range"

| bmi 6 25.0 = "normal range"

| bmi 6 30.0 = "overweight range"

| otherwise = "obese range"}

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. Note that there’s no = right after the function name and its parameters, before the first guard. Haskell newbies get syntax errors because they sometimes put it there.  
  
One way to remember that the =, i.e., the specification of the function value, follows the guard is to think of the guard as a presupposition that the argument of the function needs to satisfy before anything gets computed, i.e., before the function is actually applied to that argument (or arguments, as the case may be).  
  
If the pre-sup-positional requirement/guard is satisfied, we can go ahead and compute the value of the function, i.e., we can go ahead and assign a semantic value to the functional expression.  
  
A guard is 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.

ghci 68> bmiTell 24.3 "normal range"

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.

ghci 69> bmiTell 34.0

"obese range"

This is very reminiscent of a big if then else tree in imperative languages, only it is more readable. While big if-else trees are usually frowned upon, sometimes a problem is defined in such a discrete way that you can’t get around them. Guards are a nice alternative to this.  
  
Many times, the last guard is otherwise, which is just another word for True(it’s defined as otherwise = True) and therefore catches everything  
  
Thus, guards are very similar to patterns, only patterns check if the input has a particular form while guards check if the input satisfies boolean conditions.  
  
If all the guards of a function evaluate to False and we haven’t provided an otherwise catch-all guard, evaluation falls through to the next pattern. That’s how patterns and guards work together. If no suitable guards or patterns are found, an error is thrown.

#### Guards with multi-argument functions: reimplementing bmiTell :

We can use guards with functions that take as many parameters as we want. Instead of having the user calculate his own BMI before calling the function, let’s modify this function so that it takes a height and weight and calculates it for us.

ghci 70> let {bmiTell :: (RealFloat a) => a -> a -> String;

bmiTell weight height

| weight / height ↑ 2 6 18.5 = "underweight range"

| weight / height ↑ 2 6 25.0 = "normal range"

| weight / height ↑ 2 6 30.0 = "overweight range"

| otherwise = "obese range"}

#### Implementing the max function :

Another very simple example: let’s implement our own max function. If you remember, it takes two things that can be compared and returns the larger of them. This is how we can define it with guards:

ghci 71> let {max' :: (Ord a) => a -> a -> a;

max' a b

| a > b = a

| otherwise = b}

'

ghci 72> max' 2 5

5

Guards can also be written inline, but the definition is less readable. Here’s an example:

ghci 73> let {max" :: (Ord a) => a -> a -> a; max00 a b | a > b = a | otherwise = b}

ghci 74> max" 2 5

5

#### Implementing the compare function :

ghci 75> let {myCompare :: (Ord a) => a -> a -> Ordering;

a ‘myCompare‘ b

| a > b = GT

| a ≡ b = EQ

| otherwise = LT}

ghci 76> 3 ‘myCompare‘ 2

GT