The Fundamentals

What is Programming

Programming

Programming is the process of creating a series of instructions that tell a computer how to perform a task.

Instructions that tell a computer how to perform a task are written in a *programming language*.

Programing language

A programming language is a computer language, through which a human being can give instructions to a computer.

Source code

Instructions that tell a computer how to perform a task and written in a programming language are called source code.

Source File

Files in which the source code is stored are called source files.

It should be noted that the computer at the lower level only understands the binary language, meaning zeros and one.

Compilation

The process of transforming the source code into a binary file (which can be assimilated by the processor) is called compilation.

After a source code (located in a source file) is successfully compiled, a file containing its executable version is generated. Failed compilation occurs when the source code does not adhere to the rules of the programming language.

Execution

The computer will only run the instructions recorded in the source file after it executes the file obtained from compiling the source code file.

Programing Paradigms

There are several programming languages. These languages can be grouped into several programming paradigms.

Paradigms in programming refer to the fundamental principles of software development. At best, it is the classification, style or way of programming. It is an approach to solve problems by using programming languages.

Example of Paradigm

Below are some programming paradigms.

1. Imperative programming
   They include:
   - Structured programming
   - Procedural programming

2. Object-oriented programming
   They include:
   - Class-oriented programming
   - Component-oriented programming

3. Declarative programming
   They include:
   - Functional programming
   - Logic programming

Some programming languages:

Language	Paradigm
Java	Object-oriented programming
C++	Object-oriented / procedural
C	procedural
PHP	Object-oriented/ procedural
Perl	Object-oriented / procedural
Haskell	Functional oriented
F#	Functional oriented
Clojure	Functional oriented
JavaScript	Functional oriented & Procedural

Maths Prerequisites

Elements of Set Theory

Definition

A set is a collection of pairwise distinct objects called elements.
A set can be defined in two ways:

- in extension: we give the list of elements;
- in comprehension: we give a common property verified by the elements of the set.

Example

Let E be the set of even integers between 0 and 10. Then the elements of E are 0, 2, 4, 6, 8 et 10. We write E = {0, 2, 4, 6, 8, 10} = $\{x \in IN \,/\, (\exists k \in IN,\ x = 2k) \ \wedge \ (0 \leq x \leq 10)\}$.

Notes

To say that a mathematical object x is an element of a set A, we write: x ∈ A. When x is not an element of A, we write: x ∉ A.
With E = {0, 2, 4, 6, 8, 10}, we have : 4 ∈ E et 5 ∉ E.

Ω is the set univers. All element belong to Ω

Empty set

There exists a set which does not contain any element, it is the empty set denoted ∅.

Sets Cardinality

The cardinality of a set is the number of elements in that set. The cardinal of a set A is denoted card(A).

Subsets

Set A is a subset of Set B if all the elements of A are elements of B, in other words
$\forall\, x,\ x \in A \Rightarrow x \in B$
We note it A ⊆ B (A included in B).
Example: {0, 1, 2} ⊆ {0, 1, 2, 3} ⊆ N ⊆ Z ⊆ Q ⊆ IR ⊆ C
Note:

- $\emptyset \subseteq A \ for\ all\ set\ A$
- $A \subseteq A \ for\ all\ set\ A$

Sets Equality

Two sets A and B are equal when they have the same elements. We write : A = B. Thereby, A = B if and only if A ⊆ B and B ⊆ A.
Example: If E = {0, 2, 4, 6, 8, 10} and F = {6, 8, 10, 0, 2, 4, 6, 8}, we have : E = F.

Set of Subsets

Let A be a set, the set of parts of A denoted P(A) is the set of subsets of A.

Note

We always have:

- $\varnothing \in P(A)$ because $\varnothing \subseteq A$,
- $A \in P(A)$ because $A \subseteq A$.

Set Operations

Intersection

If A and B are two sets, we denote $A \cap B$ the set of mathematical objects that belong to A and B. $A \cap B$ reads « A inter B » or « the intersection of A and B ».
Note: We note that $A \cap B$ is a subset of A and a subset of B.

Example

if $E = \{0, 2, 4, 6, 8, 10\}$ and $F = \{3, 10, 2, 8, 8, 5\}$, then $E \cap F = \{2, 8, 10\}$.

Union

If A and B are two sets, we denote $A \cup B$ the set of mathematical objects that belong to A or B. $A \cup B$ reads «A union B» or «the union of A and B».
Note: We note that A and B are subsets of $A \cup B$.

Example

if $E = \{0, 2, 4, 6, 8, 10\}$ and $F = \{3, 10, 2, 8, 8, 5\}$, then $E \cap F = \{0, 2, 3, 4, 5, 6, 8, 10\}$.

Properties

In mathematics, a property is any characteristic that applies to a given set
Idempotence : $A \cap A = A$
Commutativity : $A \cap B = B \cap A$
Associativity : $A \cap (B \cap C) = (A \cap B) \cap C$
Neutral element : $A \cap \Omega = \Omega \cap A = A$

Idempotence : $A \cup A = A$
Commutativity : $A \cup B = B \cup A$
Associativity : $A \cup (B \cup C) = (A \cup B) \cup C$
Neutral element : $A \cup \varnothing = \varnothing \cup A = A$

Distributivity

A ∪ (B ∩ C) = (A ∪ B) ∩ (A ∪ C) et A ∩ (B ∪ C) = (A ∩ B) ∪ (A ∩ C)

Difference

A\B = {elements in A but not in B}

Symmetric difference

A ∆ B = {elements in A ∪ B but not in A ∩ B} = (A ∪ B) \ (A ∩ B)

Complementary: $\overline{A}$ = Ω \ A

Properties

Involution : $\overline{\overline{A}} = A$
Morgan's Law : $\overline{A \cap B} = \overline{A} \cup \overline{B}$; $\overline{A \cup B} = \overline{A} \cap \overline{B}$

Cartesian product

The Cartesian product of two sets A and B noted A × B is the set defined by:
A × B = {(a, b) where a ∈ A et b ∈ B}.
In general, given k set A 1 , · · · , A k the Cartesian product of t A 1 , · · · , A k is the set defined by:
A 1 × · · · × A k = {(a 1 , . . . , a k) where ai ∈ Ai for all i ∈ {1, . . . , k}}.

Example :

For the RGB computer colour coding system (“Red, Green, Blue”), a colour is an element of [0, 255] × [0, 255] × [0, 255] = [0, 255]3
two colours that have the same triplet are equal; we can define colour sets:

{predominantly green colour} = {(r , g , b) / g ≥ (r + b)}

Overview of functions

Definition of function

Let A and B two non empty sets. Let $\mathfrak{R}$ be a relation from A to B. The relation $\mathfrak{R}$ is a function if any element x of A is related to at most one element y of B. i.e: For any element x of A, for

any element y of B and for any element z of B if x is related to y and x is related to z then y = z.

$\forall x \in A, \forall y \in B, \forall z \in B, \; x \Re y \wedge x \Re z \Rightarrow y = z$

In other words, the same causes are said to produce the same effects.

Note

Let f be a function from A to B. if x is related to y by f, then:
We denote f (x) = y.
y is called image of x by f and x is called antecedent of y by f.
A is called the starting set and B is called the ending set.
We write $f: A \rightarrow B$
$x \rightarrow f(x)$

Example

Let E be the relation from the IR set of real numbers to the Z set of relative integers which to any real number associates its integer part (the highest of the integers smaller than this number). The relation E is a function. 3.2 is related to 3.$E(3,2) = 3$, $E(-6,2) = 7$. At any time, $E(3,2)$ Always gives 3.

$E: IR \rightarrow Z$
$x \rightarrow E(x)$

Counterexample

Consider the relationship $\Re$ from Z (set of relative integers) to Z which to any relative integer x associates the relative integer y such that x is the square of y. This relation is not a function. Because :
4 is the square of 2. So 4 $\Re$ 2
4 is the square of -2. So 4 $\Re$ -2
4 is related to more than one element (-2 and 2). The relation $\Re$ is therefore not a function.
The same causes do not produce the same effects. Indeed, if $\Re$ is a function, what would be $\Re(4)$? 2 or -2? At one time we will have 2, at another time we will have -2!

Function Range

Let f be a function from set A to set B. The range of the function f, denoted Im(f) is the subset of B made up of all the images of the elements of the starting set A.

The inverse range of an element y of the target set B, denoted $f^{-1}(y)$, is the set of elements of A whose image by the function f is y.

The inverse image of a part P of the target set B, denoted f^{-1} (P), is the set of elements of A whose image by the function f is contained in P.

Example

Let f be the function defined by f(x) = x2 + 2, complete relationships

- f(0) = ……..
- f-1(6) = ……..
- Im(f) = ……..
- f-1([11; 27]) = …...

Function Domain of Definition

The definition set of a function f (or the domain of function f) is the set of elements x for which f(x) exists. We denote D(f) this set (or simply Df).

Function Operations

Let f and g be two functions defined from A to B. we consider that on B are defined the operators ($+$, $-$, $\times$, $\div$). We define their sum, difference, product and quotient by stating that:

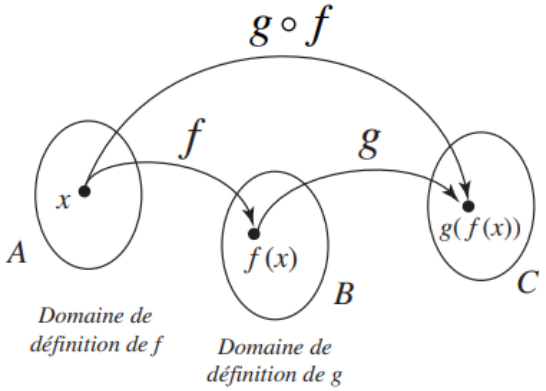
- $(f + g)(x) = f(x) + g(x)$
- $(f - g)(x) = f(x) - g(x)$
- $(f \times g)(x) = f(x) \times g(x)$
- $\left(\frac{f}{g}\right)(x) = \frac{f(x)}{g(x)}$

Function Composition

The composite function written as $(g \circ f)$ of two functions f and g is defined as follow: $(g \circ f)(x) = g(f(x))$

The definition set of $g \circ f$ is the set of all x in the definition set of f such that f(x) is in the definition set of g.

Visual



Example

1. f (x) = x^2 and g(x) = 2x + 1 then
   a. (f ◦ g)(x) = …
   b. (g ◦ f)(x) = ...
2. f (x) = x^2 + 1 et g(x) = $\frac{1}{x}$ then
   a. g (f (x)) = …
   b. f (g(x)) = …
3. f (x) = $\sqrt{x}$ et g(x) = 2x – 6 then
   a. (f ◦ g)(x) = …
   b. (g ◦ f)(x) =

Multivariate Functions

Let f be a function from a set A to a set B. f is a multivariate function if A is a Cartesian product of at least two sets. That is to say there exists k greater than or equal to 2 such that $A = A_1 \times \cdots \times A_k = \{(a_1, \ldots, a_k)$ where $a_i \in A_i$ for all $i \in \{1, \ldots, k\}\}$.

$$f: A \to B$$
$$(x_1, x_2, \ldots, x_k) \to f(x_1, x_2, \ldots, x_k)$$

Example

$$f: IR^2 \to IR$$
$$(x, y) \to \sqrt{x^2 + y^2}$$
$$distance: IR^2 \times IR^2 \to IR$$
$$((x, y), (a, b)) \to \sqrt{(x - a)^2 + (y - b)^2}$$

Parametric Functions

Let m be an element of a set denoted by M. Let f be a function from A to B. The function f is parametrized by m if $\forall x \in A, f(x)$ *is expressed in terms of* m. We denote f_m.

Example

Let m be a real number.

$$f: IR \to IR$$
$$x \to f(x) = 2x + m$$

f is a function parameterized by m.

Let k be a real number.

$$f: IR \to IR$$
$$x \to \sqrt{x^2 + k^2}$$ is a function parameterized by k

Note

if f_m is a function parameterized in m from A to B (with m belonging to M), then f_m comes from the function g defined as:

$g : A \times M \rightarrow B$

$(x, m) \rightarrow f_m(x)$

f_m (x) is obtained by fixing the variable m in the function g.

Let denote by F the set of functions from A to B. The function

$\Gamma : M \rightarrow F$

$m \rightarrow f_m \,/\, \forall x \in A, f_m(x) = f(x)$

Application to Haskell

Overview of Haskell

Expressions (Value, Functions, Declaration)

- Everything in Haskell is an expression, value or declaration.
- Functions are special type of expressions
- Values are results of Expression evaluation in Haskell, they are immutable

Purely functional

- Every function in Haskell is a function in the mathematical sense (i.e., "pure").
- Functions can only work with their inputs to return their results

Statically Typed

- Every expression in Haskell has a type which is determined at compile time.
- All the types composed together by function application have to match up.
- Types are not only a form of guarantee, but also a language for expressing the construction of programs

Type inference

- You don't have to explicitly write out every type in a Haskell program. For best practices we recommend to always write your types explicitly.
- Types will be inferred by unifying every type bidirectionally.
- Inferred types are generally more generic.

Concurrent & Parallel

The above characteristics set the ground for Haskell to be very well suited for concurrent and parallel programming.

Types

Types in Haskell relate to the concept of Sets in Mathematics.

- Haskell has a robust and expressive type system. Types play 3 fundamental key roles in Haskell programs / code:
  - Readability
  - Safety
  - Maintainability
- Expression in haskell reduces to values when evaluated. Every value is Haskell has a type
- Types are a great tool to group, classify, organise and delimit data that share common characteristics

Type definition Anatomy

```
data     <...[1]...>  =   < ...[2]...>

type     <...[1]...>  =   < ...[2]...>

newtype  <...[1]...>  =   < ...[2]...>
```

[1] *Type constructor* for the datatype being defined. The type constructor is also the name of the type being defined. It is used to declare inhabitant value of the type and starts with a Capital letter.

- Type constructor (type names) show up at type level (type signatures)

[2] *Data or value constructor* for the type defined. It indicated how value that inhabit the type being defined are constructed.

- This is use at term level

Basics types

Bool

The type Bool is part of the standard prelude. It is a sum type as opposed to product types. It is often used to create conditions and control how the program behaves when certain things happen.

data Bool = False | True

```
isTheHasketonHappeningNow = True

:type iSTheHasketonHappeningNow
isTheHasketonHappeningNow :: Bool

:t iSTheHasketonHappeningNow
isTheHasketonHappeningNow :: Bool

x = 2
:t (x - 2) == (x / 2)
(x - 2) == (x / 2) :: Bool

(x - 2) == (x / 2)
False

:info Bool
type Bool :: *
data Bool = False | True
      -- Defined in ‘GHC.Types’
instance Eq Bool -- Defined in ‘GHC.Classes’
instance Ord Bool -- Defined in ‘GHC.Classes’
instance Enum Bool -- Defined in ‘GHC.Enum’
instance Show Bool -- Defined in ‘GHC.Show’
instance Read Bool -- Defined in ‘GHC.Read’
instance Bounded Bool -- Defined in ‘GHC.Enum’
```

Char

The character type Char is an enumeration whose values represent Unicode (or equivalently ISO/IEC 10646) code points (i.e. characters, see http://www.unicode.org/ for details). This set extends the ISO 8859-1 (Latin-1) character set (the first 256 characters), which is itself

an extension of the ASCII character set (the first 128 characters). A character literal in Haskell has type Char.

data Char = GHC.Types.C# GHC.Prim.Char#

```
hisNamesFirstCharacterIs = 'F'
hisNamesLastCharacterIs = 'n'

hisNamesFirstCharacterIs
'F'

hisNamesLastCharacterIs
'n'

areTheyEquals = hisNamesFirstCharacterIs == hisNamesLastCharacterIs
False
```

Numbers

In Haskell numeric types are organised in 2 core groups: ***Integral*** and ***fractional*** numbers under a more generic container / type called **Num**.

```
numberOfParticipant = 30

:t numberOfParticipant
numberOfParticipant :: Num p => p
```

Integral

Integral numbers are whole numbers with no fractional component or decimal part. They can be positive or negative. Under this group exits the following types:

- **Int**: This type has a well defined range that comes with a minimum and a maximum value.
- **Integer**: It is used to define arbitrary large or small numbers. It does not have minimum and maximum values like Int
- **Word:** A Word is an unsigned integral type, with the same size as Int.


```
numberOfTiredPartcipant = 0 :: Int

numberOfTiredPartcipant
0

numberOfBraveWomen    = 3 :: Word

numberOfBraveWomen
3

countToOurHackathonsPromise = -3 :: Integer

countToOurHackathonsPromise
-3
```

Fractional

Fractional numbers are used to represent a part of a whole. They come with a whole and a decimal part. This group comprises:

- **Float**: The type is used to represent number with decimal place
- **Double**: The type is used to represent number with decimal place with a need to have move precision
- **Rational**: This is a fractional number that represents a ratio of two integers. The value 1 / 2
- **Scientific**: This is a space efficient and almost arbitrary precision scientific number type. Scientific numbers are represented using scientific notation

```
lessPrecise   =  1/3 :: Float

lessPrecise
0.33333334

morePrecise =  1/3 :: Double
morePrecise
0.3333333333333333
```

Some Operations on numbers

- Addition: +
- Multiplication: *
- Division: /
- Subtraction: -
- mod

```
2 + 5
7

(+) 2 5
7

5 - 3
2

(-) 5 3
2

:t (+)
(+) :: Num a => a -> a -> a

:t (-)
(-) :: Num a => a -> a -> a
```

Comparison operators

- Inferior or equal: <, <=
- Superior or equal: >, >=
- Equality: ==

Boolean operators

- And: &&
- Or: ||

```
2 <= 5
True

2 > 5
False

2 <= 2
True

(2 <=2) && (3 > 5)
False
```

Constants and Polymorphism

Instead of making numeric constant values of a specific type, Haskell has a clever solution for supporting values for different types: they are called polymorphic.

```
averageGroupChalengeGrade = 19.99

:t averageGroupChalengeGrade
averageGroupChalengeGrade :: Fractional p => p
```

Type variables

Types variable as they name state is a way to specify types that can vary. The type of the function head is: head :: [a] -> a. Remember that we previously stated that types (types constructors or type names) are written starting with a capital letter, so it can't exactly be a type because it's not capitalised. It's actually a type variable. That means that a can be of any type.

```
:t head
head :: [a] -> a

:t length
length :: Foldable t => t a -> Intcase
```

List

Lists are types used to contain multiple values within one. Unlike tuples, lists are homogenous in type, meaning the values they contain have the same type

data [] a = [] | a : [a]

```
allHackathonGrades = [20, 19.9, 18, 18, 20, 18.5] :: [Float]

:t allHackathonGrades
allHackathonGrades :: [Float]

allHackathonGrades' = [20, 19.9, 18, 18, 20, 18.5]

:t allHackathonGrades'
allHackathonGrades :: Fractional a => [a]
```

String

The type String in Haskell is syntactic sugar for a list of characters.

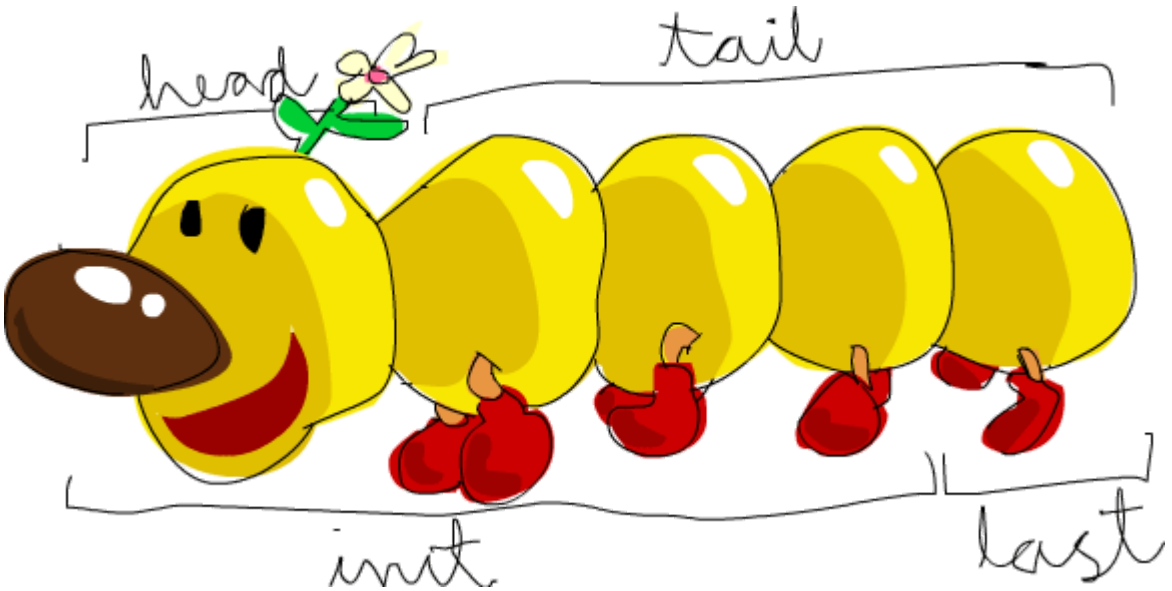
type String = [Char]

```
:info String

type String :: *
type String = [Char]
      -- Defined in 'GHC.Base'

usaGreeting = "How are you doing?"
kinkongoGreeting = "Mbote Nzola Mpangi"
mboteGretingMeaning = "recevez (Te) le feu sacré qui guérit et permet le
Kunaka  (Mbo)"
```

Some functions on list

Lists are one of the most common data structures in programming. Haskell ships a Data.List library that include predefined functions that are useful to manipulate Lists:

- head, tail, last, init, length
- null, elem, (!!), take, drop
- sum, product, maximum, minimum
- (++), concat, reverse



List comprehension

We've seen two ways of defining a set: by extension, and by comprehension. The same notion is present in Haskell and it is very handy for building more specific sets out of general sets.

A basic comprehension for a set that contains the first ten even natural numbers is `S = {2. x | x ∈N, x≤10}.` The part before the pipe is called the output function, x is the variable, N is the input set and `x≤10 is the predicate.` That means that the set contains the doubles of all natural numbers that satisfy the predicate. If we wanted to write that in Haskell, we could do something like take 10 [2,4..] . But what if we didn't want doubles of the first 10 natural numbers but some kind of more complex function applied on them? We could use a list comprehension for that. ***List comprehensions are very similar to set comprehensions.*** We'll stick to getting the first 10 even numbers for now. The list comprehension we could use is `[x*2 | x <- [1..10]]` . x is drawn from [1..10] and for every element in [1..10] (which we have bound to x), we get that element, only doubled

```
allKentos = ["Mercy", "Emilia", "Awura"]
allBakalas = ["Sidney", "Josh", "Paul"]

workGroups = [(kento, bakala) | kento <- allKentos, bakala <- allBakalas
]
workGroupsWithoutPaul = [(kento, bakala) | kento <- allKentos, bakala <-
allBakalas , bakala /= "Paul"]
```

Functions

Tuples

Tuples are types that allow storing and passing of values with different types within a single value namely tuple. Tuple arity refers to the number of values a tuple can hold. Tuples are said to be heterogeneous.

data (,) a b = (,) a b

```
--Dr Sebi recommended Item list prices by kilogram in MaatCoins and
--location where to find them

sesamePrice = ("Sesame", 3, "Ethiopia") :: (String, Int, String)
wildRicePrice = ("Wild Rice", 30, "USA") :: (String, Int, String)
chickPeasPrice = ("Chickpeas", 21, "USA")
kamutPrice = ("Kamut", 144, "Ethiopia") :: (String, Int, String)

wrongKamutPrice = ("Kamut", "Ethiopia", 144) :: (String, String, Int)

allItems = [sesamePrice, wildRicePrice, chickPeasPrice, kamutPrice]

:t chickPeasPrice
...
:t wildRicePrice
...
:t allItems
...
chickPeasPrice == wildRicePrice
...
kamutPrice == wrongKamutPrice
...
```

Declaration and definition

Functions in Haskell are functions in the mathematical sense although Haskell does have its own functional definition and declaration.

- Function declaration consists of the function name (First letter in lower case) and its argument list along with its output.
- Function definition is where you actually define a function by implementing the relation between the inputs and output. .

```
functionName :: [(Constraints)] => Input1Tpe -> Input2Type -> ... ->
InputnType -> OuputType
functionName arg1 arg2 ... argn  = functionBody
```

- A name, which in Haskell always starts with a lowercase letter
- The list of parameters, each of which must also begin with à lowercase letter, separated from the rest by spaces (not by commas, like in most languages) and not surrounded by parentheses
- An **=** sign
- The body of the function at the right of the equal (**=**) sign

Creating a simple Function

Let's consider the following function to understand the declaration and definition syntaxes. $addSqrt(x, y) = x^2 + y^2$ is a mathematical function, its declaration and definition in Haskell is:

```
addSqrt x y = x^2 + y^2
```

Specifying the Function's Type

if we check the type of addSqrt by typing `:t addSqrt` we will get: `addSqrt :: Num a => a -> a -> a` Haskell has used inference to assigned a type to the function making it the most generic possible. ***Haskell, if let to decide, would put a bean in a 300kg bag.***

Calling a function / evaluating a function / applying a function to arguments



We call a function (or apply a function to its arguments) when we want to get the output for a given input. Calling the function is also evaluating it as an expression.

```
addSqrt 2 3
```

Some useful constructs

Conditional Expression

Here is the general syntax of using the if-else conditional statement in Haskell.

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

In the above 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 returned when the Condition satisfies
- False-***Expression*** − It refers to the output that is returned when the condition does not satisfy.

As Haskell codes are interpreted as mathematical expressions, the above statement will throw an error without the else block.

```
a = 4
x = if (mod a 5) then 0 else 1
```

Guard

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.

```
myMaxInt :: Int -> Int
myMaxInt a b
  | a > b = a
  | otherwise = b
```

let and in

In Haskell let, ***binding is used to bind the names to values, which are very local***. In Haskell, we can define any type of value using the let keyword before the value name; their scope is local

```
cylinderTotalArea :: Double -> Double -> Double
cylinderTotalArea r h =
    let sideArea = 2 * pi * r * h
        topArea = pi * r ^2
    in sideArea + 2 * topArea
```

Case … of

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 general syntax is as follows:

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

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

Declaration and Binding Expression

In Haskell normally we should not talk about variables. We have values that are bound to names and are immutable.

where

Where is used to express declaration. Where bindings are a syntactic construct that let you bind expressions to variables at the end of a function and the whole function can see them.

```
cylinderTotalArea :: Double -> Double -> Double
cylinderTotalArea r h = sideArea + 2 * topArea
    where
      sideArea = 2 * pi * r * h
      topArea = pi * r^2
```

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.

```
boolNegate :: Bool -> Bool
boolNegate True = False
booNegate False = True


toletters:: Integer -> Int
toletters 1 = "One!"
toletters 2 = "Two!"
toletters 3 = "Three!"
toletters _ = "Not between 1 and 3"
```

Higher order function

Haskell functions can take functions as parameters and return other functions or values as return values. A function that does either or both of those is called a higher order function.

Curried functions

Every function in Haskell officially only takes one parameter. The multiparameter syntax is just syntactic sugar. A function taking multiple parameters and returning a result is actually just a function taking one parameter and returning another function.

Some examples

Advanced function on List

Map

map takes a function and a list and applies that function to every element in the list, producing a new list.

```
:t map
map :: (a -> b) -> [a] -> [b]

doubleGrade :: Int -> Int
doubleGrade grade = 2 * grade

participantGrade = [20, 18, 15]

map doubleGrade participantGrade
```

```
[40,36,30]
```

Filter

filter is a function that takes a predicate (a predicate is a function that tells whether something is true or not (True, False), so in our case, a function that returns a boolean value) and a list and then returns the list of elements that satisfy the predicate.

```
:t filter
filter :: (a -> Bool) -> [a] -> [a]

isDivisibleByTwo :: Int -> Bool
isDivisibleByTwo n = mod n 2 == 0

numbers = [20, 18, 15]

filter isDivisibleByTwo numbers
[20,18]
```

Function composition

In mathematics, function composition is defined like this: (f ◦ g)(x) = f(g(x)), meaning that composing two functions produces a new function that, when called with a parameter, say, x is the equivalent of evaluating g with the parameter x and then evaluating the f with that result.

Let's consider the functions that do the following:
Calculate figures out the total points of a student for a total of nine classe define as follows:

Parameterized Functions

A function taking multiple parameters and returning a result is actually just a function taking one parameter and returning another function.The returned function is parameterized by the input argument.

```
parameterizedGreeting :: String -> (String -> String)
parameterizedGreeting language = localGreeting language
    where
        localGreeting :: String -> String -> String
        localGreeting la name = la ++ name ++ "!"
```

```
kikongoGreetingTo = parameterizedGreeting "Mbote nzola "
frenchGreetingTo = parameterizedGreeting "Bonjour "
englishGreetingTo = parameterizedGreeting "Hello "
```

Final Activity

1. Copy and paste the above code into a file named Greeting.hs
2. Launch the REPL using the command ***ghci*** from your terminal
3. Load the file by running the command :load Greeting.hs
4. Enter the following command function calls and make sense of them

```
:t frenchGreetingTo
:t englishGreetingTo
:t kikongoGreetingTo

frenchGreeting "Paul"
englishGreetingTo "Mercy"
kikongoGreetingTo "Megan"
```