January 2022 Hackathon

Table of content

What is Haskel?

Haskell is an advanced, purely functional programming language. Haskell has **first-class functions**. It treats functions as first-class citizens. (Functions can be treated as values, they can be passed around as arguments and returned from functions).

What you need to dive in

A text editor and a Haskell compiler. You probably already have your favorite text editor installed so we won't waste time on that. For the purposes of this course we'll be using GHC, the most widely used Haskell compiler. The best way to get started is to download the Haskell Platform, which is basically Haskell with batteries included at this link https://www.haskell.org/downloads/. GHC can take a Haskell script (they usually have a .hs extension) and compile it but it also has an interactive mode which allows you to interactively interact with scripts. You can call functions from scripts that you load and the results are displayed immediately. For learning it's a lot easier and faster than compiling every time you make a change and then running the program from the prompt. The interactive mode is invoked by typing in ghci at your prompt. If you have defined some functions in a file called, myfunctions.hs , you load up those functions by typing in :l myfunctions and then you can play with them, provided myfunctions.hs is in the same folder from which ghci was invoked. If you change the .hs script, just run :l myfunctions again or do :r , which is equivalent because it reloads the current script.
Here is an example of how to use the ghc in compile mode:

```
$ ghc --make MatrixProduct.hs
[1 of 1] Compiling Main ( MatrixProduct.hs, MatrixProduct.o )
Linking MatrixProduct ...

$ ./MatrixProduct
[[26,16,28],[56,40,64],[86,64,100]]
```

To be more efficient you can use an IDE like Visual Studio Code which provide à gread framework.

Types

Types are ways to categorize data/value/expression/function in haskell.
in iteractive mode, the command *:t* followed by a value gives the type of the value.

:: is read as "has type of"

Basic Types

Bool :

Type to group data that can get only two values: True and False.

```
:t True
:t False
```

Char

Haskell has a special type called Char for representing character data. To prevent problems with locales and languages, a Char value contains one Unicode character. These values can be created in two ways.
• Writing the character itself between single quotes, like 'a'.
• Writing the code point, that is, the numeric value which represents the character as defined in the Unicode standard, in decimal between '\ and ' or in hexadecimal between '\x and '. For example, the same 'a' character can be written as '\97' or '\x61'. Using GHCi, you can check the actual type of each expression you introduce in the system. To do so, you use the :t command, followed by the expression. Let’s check that characters indeed are characters.

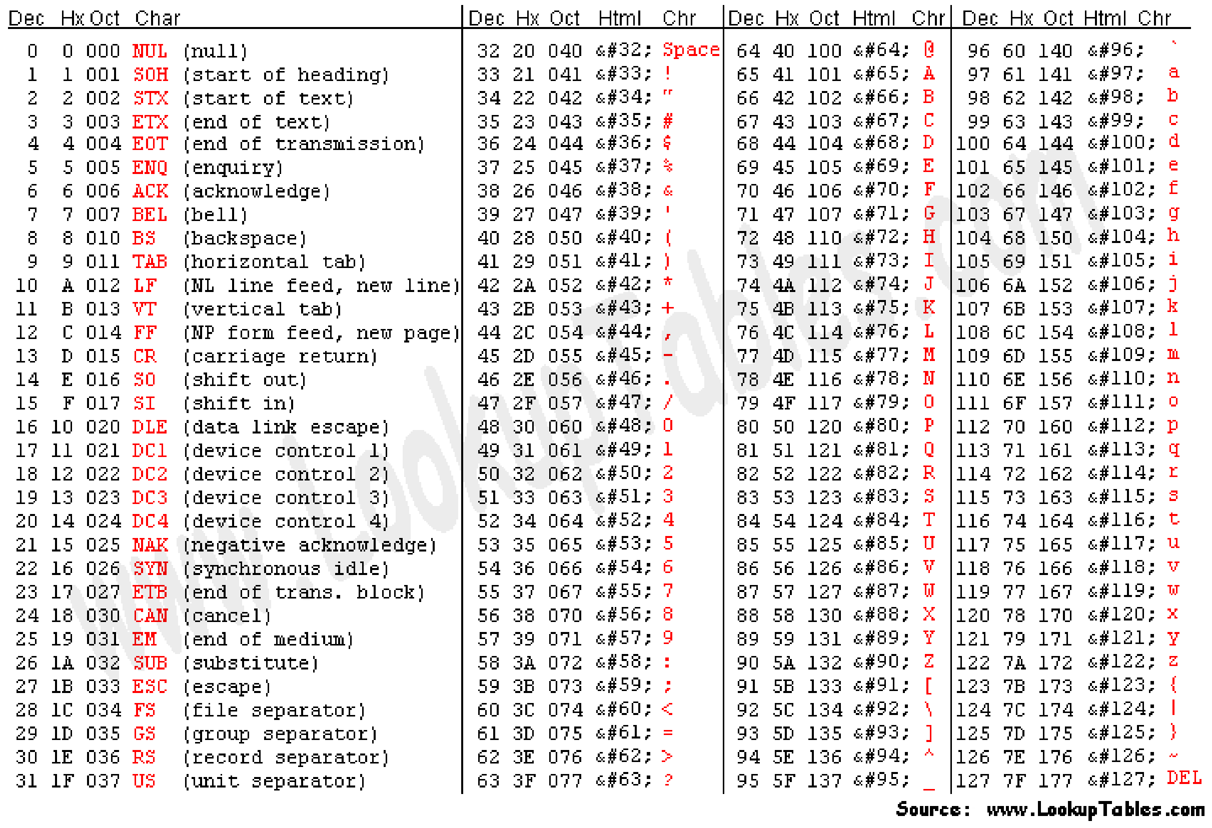
```
Prelude> :t 'a'
'a' :: Char
```

Let’s now explore some of the functionality that Haskell provides for Chars. Only a few functions are loaded by default, so let’s import a module with a lot more functions, in this case Data.Char

```
Prelude> import Data.Char
Prelude Data.Char> :t toUpper
toUpper :: Char -> Char
```

(This is red as toUpper has type of Char -> Char ei toUpper is à function that take as input a character and return a character)

ASCII Table



Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH	(start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX	(start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX	(end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT	(end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ	(enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK	(acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL	(bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS	(backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB	(horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT	(vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR	(carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO	(shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI	(shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE	(data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1	(device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2	(device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3	(device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4	(device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK	(negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN	(synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB	(end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN	(cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM	(end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB	(substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC	(escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS	(file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS	(group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS	(record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US	(unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Numbers

Like most programming languages, Haskell supports a great variety of number types, depending on the width, precision, and support for decimal parts.

- **Int** is the bounded integer type. It supports values between at least ±536870911, which corresponds to 2^{29} -1 (even though GHC uses a much wider range). Usually, values of the Int type have the native width of the architecture, which makes them the fastest.
- **Integer** is an unbounded integral type. It can represent any value without a decimal part without underflow or overflow. This property makes it useful for writing code without caring about bounds, but it comes at the price of speed.
- The Haskell base library also bundles exact rational numbers using the **Ratio** type. Rational values are created using n % m.
- Float and Double are floating-point types of single and double precision, respectively.

Constants and polymorphism

```
Prelude> :t 5
5 :: Num a => a
Prelude> :t 3.4
3.4 :: Fractional a => a
```

Instead of making a numeric constant of a specific type, Haskell has a clever solution for supporting constants for different types: they are called polymorphic. For example, 5 is a constant that can be used for creating values of every type supporting the Num type class (which includes all types introduced before). On the other hand, 3.4 can be used for creating values of any type that is Fractional (which includes Float and Double but not Int or Integer). You will read in detail about type classes in next section, but right now you can think of **a type class as a way to group sets of types that support the same operations**.

Some Operations on number

- +
- *
- /
- -
- mod (Modulo)
- etc …

Comparison operators

- <=
- >=
- ==
- /=
- <
- >
- etc.

Inference

```
prelude> x = 5.2 + 8
prelude> :t x
x :: Fractional a => a
```

Type inference is the automatic determination of the type of each expression based on the functions and syntax construct being used.

Lists

Strings

After playing for some time with characters, you may wonder whether you can have a bunch of them together, forming what is commonly known as a string. The syntax for strings in Haskell are similar to C: you wrap letters in double quotes. The following code creates a string. If you ask the interpreter its type, what do you expect to get back?
Prelude Data.Char> :t "Hello world!"
"Hello world!" :: [Char]
Instead of some new type, like String, you see your old friend Char but wrapped in square brackets. Those brackets indicate that "Hello world!" is not a character but a list of characters.
In general, given a type T, the notation [T] refers to the type of all lists whose elements are of that type T. Lists are the most used data structure in functional programming. The fact that a type like a list depends on other types is known as parametric polymorphism, and you will delve into the details of it in the next section.

generalization

List literals (i.e., lists whose values are explicitly set into the program code) are written with commas separating each of the elements, while wrapping everything between square brackets. As I have said, there's also special string syntax for a list of characters. Let's look at the types of some of these literals and the functions reverse, which gives a list in reverse order, and (++), which concatenates two lists.

```
Prelude> :t [1,2,3]
[1, 2, 3] :: Num t => [t]
Prelude> :t reverse
reverse :: [a] -> [a]
Prelude> :t (++)
(++) :: [a] -> [a] -> [a]
Prelude> reverse [1,2,3]
```

```
[3,2,1]
Prelude> reverse "abc"
"cba"
Prelude> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]
```

Some function on list

head takes a list and returns its head. The head of a list is basically its first element.

```
ghci> head [5,4,3,2,1]
5
```

tail takes a list and returns its tail. In other words, it chops off a list's head.

```
ghci> tail [5,4,3,2,1]
[4,3,2,1]
```

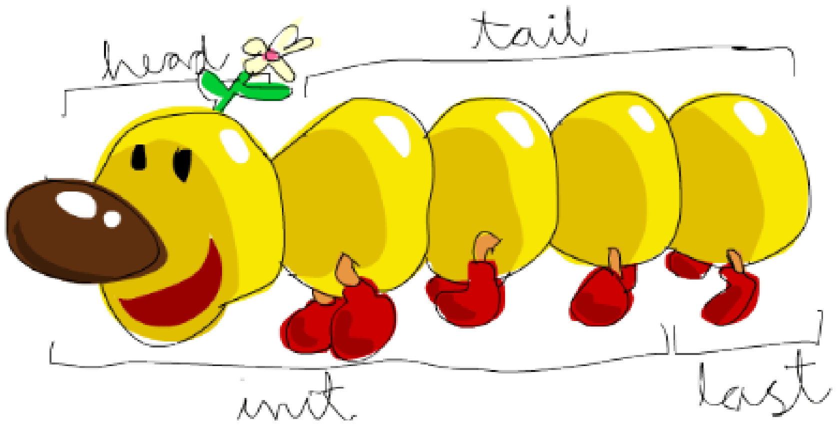
last takes a list and returns its last element.

```
ghci> last [5,4,3,2,1]
1
```

init takes a list and returns everything except its last element.

```
ghci> init [5,4,3,2,1]
[5,4,3,2]
```

If we think of a list as a monster, here's what's what.



But what happens if we try to get the head of an empty list?

```
ghci> head []
*** Exception: Prelude.head: empty list
```

Oh my! It all blows up in our face! If there's no monster, it doesn't have a head. When using

head , tail , last and init , be careful not to use them on empty lists. This error cannot be caught
at compile time so it's always good practice to take precautions against accidentally telling Haskell
to give you some elements from an empty list.

length takes a list and returns its length, obviously.

```
ghci> length [5,4,3,2,1]
5
```

null checks if a list is empty. If it is, it returns True , otherwise it returns False . Use this function instead of xs == [] (if you have a list called xs)

```
ghci> null [1,2,3]
False
ghci> null []
True
```

reverse reverses a list.

```
ghci> reverse [5,4,3,2,1]
[1,2,3,4,5]
```

take takes number and a list. It extracts that many elements from the beginning of the list. Watch.

```
ghci> take 3 [5,4,3,2,1]
[5,4,3]
ghci> take 1 [3,9,3]
[3]
ghci> take 5 [1,2]
[1,2]
ghci> take 0 [6,6,6]
[]
```

See how if we try to take more elements than there are in the list, it just returns the list. If we try
to take 0 elements, we get an empty list.

drop works in a similar way, only it drops the number of elements from the beginning of a list.

```
ghci> drop 3 [8,4,2,1,5,6]
[1,5,6]
ghci> drop 0 [1,2,3,4]
[1,2,3,4]
ghci> drop 100 [1,2,3,4]
```

```
[]
```

maximum takes a list of stuff that can be put in some kind of order and returns the biggest element.

minimum returns the smallest.

```
ghci> minimum [8,4,2,1,5,6]
1
ghci> maximum [1,9,2,3,4]
9
```

sum takes a list of numbers and returns their sum.

product takes a list of numbers and returns their product.

```
ghci> sum [5,2,1,6,3,2,5,7]
31
ghci> product [6,2,1,2]
24
ghci> product [1,2,5,6,7,9,2,0]
0
```

elem takes a thing and a list of things and tells us if that thing is an element of the list. It's usually called as an infix function because it's easier to read that way.

```
ghci> 4 `elem` [3,4,5,6]
True
ghci> 10 `elem` [3,4,5,6]
False
```

(!!) takes a list of things and a position and returns the element at that position in the list. If the position is greater than the length of the list an error is thrown. It's usually called an infix function because it's easier to read that way.

```
Prelude> :t (!!)
(!!) :: [a] -> Int -> a
Prelude> [1, 2, 3, 4, 5] !! 2
3
Prelude> (!!) [1, 2, 3, 4, 5]  2
3
```

Those were a few basic functions that operate on lists. We'll take a look at more list functions later.

Texas range

What if we want a list of all numbers between 1 and 20? Sure, we could just type them all out but obviously that's not a solution for gentlemen who demand excellence from their programming languages. Instead, we'll use ranges. Ranges are a way of making lists that are arithmetic sequences of elements that can be enumerated. Numbers can be enumerated. One, two, three, four, etc. Characters can also be enumerated. The alphabet is an enumeration of characters from A to Z. Names can't be enumerated. What comes after“John” ? I don't know.

To make a list containing all the natural numbers from 1 to 20, you just write [1..20] . That is the equivalent of writing [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20] and there's no difference between writing one or the other except that writing out long enumeration sequences manually is stupid.

```
ghci> [1..20]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
ghci> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
ghci> ['K'..'Z']
"KLMNOPQRSTUVWXYZ"
```

Ranges are cool because you can also specify a step. What if we want all even numbers between 1and 20? Or every third number between 1 and 20?

```
ghci> [2,4..20]
[2,4,6,8,10,12,14,16,18,20]
ghci> [3,6..20]
[3,6,9,12,15,18]
```

It's simply a matter of separating the first two elements with a comma and then specifying what the upper limit is. While pretty smart, ranges with steps aren't as smart as some people expect them to be. You can't do [1,2,4,8,16..100] and expect to get all the powers of 2. Firstly because you can only specify one step. And secondly because some sequences that aren't arithmetic are ambiguous if given only by a few of their first terms. To make a list with all the numbers from 20 to 1, you can't just do [20..1] , you have to do [20,19..1] .

Watch out when using floating point numbers in ranges! Because they are not completely precise (by definition), their use in ranges can yield some pretty funky results.

```
ghci> [0.1, 0.3 .. 1]
[0.1,0.3,0.5,0.7,0.8999999999999999,1.0999999999999999]
```

It is not recommended to use them in list ranges. You can also use ranges to make infinite lists by just not specifying an upper limit. Later we'll go into more detail on infinite lists. For now, let's examine how you would get the first 24 multiples of 13. Sure, you could do *[13,26..24*13]* . But there's a better way: *take 24 [13,26..]* . Because Haskell is lazy, it won't try to evaluate the infinite list immediately because it would never finish. It'll wait to see what you want to get out of that infinite lists. And here it sees you just want the first 24 elements and it gladly obliges. A handful of functions that produce infinite lists:

cycle takes a list and cycles it into an infinite list. If you just try to display the result, it will go on forever so you have to slice it off somewhere.

```
ghci> take 10 (cycle [1,2,3])
[1,2,3,1,2,3,1,2,3,1]
ghci> take 12 (cycle "LOL ")
"LOL LOL LOL "
```

repeat takes an element and produces an infinite list of just that element. It's like cycling a list with only one element.

```
ghci> take 10 (repeat 5)
[5,5,5,5,5,5,5,5,5,5]
```

Although it's simpler to just use the replicate function if you want some number of the same element in a list. replicate 3 10 returns [10,10,10] .

List comprehension

If you've ever taken a course in mathematics, you've probably run into set comprehensions. They're normally used for building more specific sets out of general sets. A basic comprehension for a set that contains the first ten even natural numbers is $S = \{2.x \mid x \in N,\ x \leq 10\}$. The part before the pipe is called the output function, x is the variable, N is the input set and $x \leq 10$ 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. Here's that comprehension in action.

```
ghci> [x*2 | x <- [1..10]]
[2,4,6,8,10,12,14,16,18,20]
```

listComprehension = [exp in x, y … | domain of x, y, …, predicate]

Exercise: Write a function that takes a string and removes everything except uppercase letters from it.

```
removeNonUppercase str = [ c | c <- str, c `elem` ['A'..'Z']]
```

Tuples

Tuples are used when you know exactly how many values you want to combine and its type depends on how many components it has and the types of the components. They are denoted with parentheses and their components are separated by commas. They don't have

to be homogenous.Tuples can be used to represent a wide variety of data. For instance, if we wanted to represent someone's name and age in Haskell, we could use a triple: ("Christopher", "Walken", 55) .
Use tuples when you know in advance how many components some piece of data should have.Tuples are much more rigid because each different size of tuple is its own type, so you can't write a general function to append an element to a tuple.
Two useful functions that operate on pairs:

fst takes a pair and returns its first component.

```
ghci> fst (8,11)
8
ghci> fst ("Wow", False)
"Wow"
```

snd takes a pair and returns its second component. Surprise!

```
ghci> snd (8,11)
11
```

Note: these functions operate only on pairs. They won't work on triples, 4-tuples, 5-tuples, etc. We'll go over extracting data from tuples in different ways a bit later.

Type variables

What do you think is the type of the head function? Because head takes a list of any type and returns the first element, so what could it be? Let's check!

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

What is this a? Is it a type? Remember that we previously stated that types are written in capital case, so it can't exactly be a type. Because it's not in capital case it's actually a **type variable**. That means that a can be of any type. This is much like generics in other languages, only in Haskell it's much more powerful because it allows us to easily write very general functions if they don't use any specific behavior of the types in them. Functions that have type variables are called **polymorphic functions**. The type declaration of **head** states that it takes a list of any type and returns one element of that type.

Although type variables can have names longer than one character, we usually give them names of a, b, c, d …

Remember fst? It returns the first component of a pair. Let's examine its type.

```
ghci> :t fst
fst :: (a, b) -> a
```

We see that fst takes a tuple which contains two types and returns an element which is of the same type as the pair's first component. That's why we can use fst on a pair that contains any two types. Note that just because a and b are different type variables, they don't have to be different types. It just states that the first component's type and the return value's type are the same.

Functions

Inputs
f function
Outputs

Functions declaration and definition

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

- Function declaration consists of the function name and its argument list along with its output.
- Function definition is where you actually define a function.

The function declaration consists of specifying the type of the function. The general syntax is:

```
myFunction :: [(Constraint) =>] typeOfInput1 -> typeOfInput2 -> ... ->
typeOfInputn -> typeOfInput
```

this is telling the compiler that *myFunction* is a function that take n values as input (n can be zero) of type *typeOfInput1, typeOfInput2, ... typeOfInputn* respectively and returns a value of type *typeOfInput*

Constraint is a list comma separated of constraints

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

Function definition include the following:

- 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 and the body of the function

Let us take a small example of an add function to understand this concept in detail.

```
add :: Integer -> Integer -> Integer   --function declaration
add x y =  x + y                       --function definition
```

Creating a Simple Function

Given a list of strings, the function returns either the first string in the list or the string "empty" if there is nothing in the list.

```
firstOrEmpty lst = if not (null lst) then head lst else "empty"
firstOrEmpty ["hello","hola"]
"hello"
```

Specifying the Function's Type

if we check the type of *firstOrEmpty* by typing :t *firstOrEmpty* we will get:
firstOrEmpty :: [[Char]] -> [Char]
By inference, Haskell has assigned a type to the function.

Conditional Expression

Here is the general syntax of using the if-else conditional statement in Haskell.
if<Condition> then <True-Value>else <False-Value>

In the above expression,

- Condition − It is the binary condition which will be tested. it is an expression that returns True or False.
- True-Value − It refers to the output that comes when the Condition satisfies
- False-Value − It refers to the output that comes 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. The following code shows how you can use the if-else statement in Haskell.

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

In the above example, the given condition fails. Hence, the else block will be executed. This leads to really neat code that's simple and readable.

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 from 1 to 5 and says "Not between 1 and 5" for any other number?
Without pattern matching, we'd have to make a pretty convoluted if then else tree. However, with it:

```
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"
```

Factorial

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

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

Let's make a trivial function that tells us some of the first elements of the list in (in)convenient English form.

```
tell :: (Show a) => [a] -> String
tell [] = "The list is empty"
tell (x:[]) = "The list has one element: " ++ show x
tell (x:y:[]) = "The list has two elements: " ++ show x ++ " and " ++
show y
tell (x:y:_) = "This list is long. The first two elements are: " ++ show
x ++ " and " ++ show y
```

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 à 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. If your BMI is less than 18.5, you're considered underweight. If it's anywhere from 18.5 to 25 then you're 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)

```haskell
bmiTell :: (RealFloat a) => a -> String
bmiTell bmi
| bmi <= 18.5 = "You're underweight, you emo, you!"
| bmi <= 25.0 = "You're supposedly normal. Pffft, I bet you're ugly!"
| bmi <= 30.0 = "You're fat! Lose some weight, fatty!"
| otherwise = "You're a whale, congratulations!"
```

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.
other example:

```haskell
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're underweight, you emo, you!"
| bmi <= 25.0 = "You're supposedly normal. Pffft, I bet you're ugly!"
| bmi <= 30.0 = "You're fat! Lose some weight, fatty!"
| otherwise= "You're a whale, congratulations!"
where bmi = weight / height ^ 2
```

Let's see if I'm fat ...

```
ghci> bmiTell 85 1.90
"You're supposedly normal. Pffft, I bet you're ugly!"
Yay! I'm not fat! But Haskell just called me ugly. Whatever!
```

Let and In

Very similar to where bindings are let bindings. Where bindings are a syntactic construct that let you bind to variables at the end of a function and the whole function can see them, including all the guards. 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
      ...
```

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."
xs -> "a longer list."
```

Lambda expressions

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.)

```
Prompt> (\x -> x + 1) 4
5
Prompt> (\x y -> x + y) 3 5
8
```

Overview on recursion function

We mention recursion briefly in the previous section.

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

What is GCD (Greatest Common Divisor)?

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 principaly uses this 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.

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

Higher order function

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

Curried functions

Every function in Haskell officially only takes one parameter. So how is it possible that we defined and used several functions that take more than one parameter so far? Well, it's a clever trick! All the functions that accepted several parameters so far have been curried functions. What does that mean? You'll understand it best from an example. Let's take our good friend, the max function. It looks like it takes two parameters and returns the one that's bigger. Doing max 4 5 first creates a function that takes a parameter and returns either 4 or that parameter, depending on which is bigger. Then, 5 is applied to that function and that function produces our desired result. That sounds like a mouthful but it's actually a really cool concept. The following two calls are equivalent:

```
ghci> max 4 5
5
ghci> (max 4) 5
5
```

Putting a space between two things is simply a function application. The space is sort of like an operator and it has the highest precedence. Let's examine the type of max.
It's *max : : (Ord a) => a -> a -> a*. That can
also be written as *max : : (Ord a) => a -> (a -> a)* . That could be read as: max takes an a and returns (that's the ->) a function that takes an a and returns an a. That's why the return type and the parameters of functions are all simply separated with arrows. So how is that beneficial to us? Simply speaking, if we call a function with too few parameters, we get back a partially applied function, meaning a function that takes as many parameters as we left out. Using partial application (calling functions with too few parameters, if you will) is a neat way to create functions on the fly so we can pass them to another function or to seed them with some data.

Some higher-orderism is in order

Functions can take functions as parameters and also return functions. To illustrate this, we're going to make a function that takes a function and then applies it twice to something!

```
applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)
```

First of all, notice the type declaration. Before, we didn't need parentheses because -> is naturally right-associative. However, here, they're mandatory. They indicate that the first parameter is a function that takes something and returns that same thing. The second parameter is something of that type also and the return value is also of the same type. We could read this type declaration in the curried way, but to save ourselves a headache, we'll just say that this function takes two parameters and returns one thing. The first parameter is a function (of type a -> a) and the second is that same a. The function can also be Int -> Int or String -> String or whatever. But then, the second parameter also has to be of that type.

Advanced functions on List

Map

map takes a function and a list and applies that function to every element in the list, producing a new list. Let's see what its type signature is and how it's defined.

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs

ghci> map (+3) [1,5,3,1,6]
[4,8,6,4,9]
```

Filter

filter is a function that takes a predicate (a predicate is a function that tells whether something is true or not, 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. The type signature and implementation go like this:

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
    | p x       = x : filter p xs
    | otherwise = filter p xs

ghci> filter (>3) [1,5,3,2,1,6,4,3,2,1]
[5,6,4]
```

Folder

A fold takes a binary function, a starting value (I like to call it the accumulator) and a list to fold up. The binary function itself takes two parameters. The binary function is called with the accumulator and the first (or last) element and produces a new accumulator. Then, the binary function is called again with the new accumulator and the now new first (or last) element, and so on. Once we've walked over the whole list, only the accumulator remains, which is what we've reduced the list to.

First let's take a look at the *foldl* function, also called the left fold. It folds the list up from the left side. The binary function is applied between the starting value and the head of the list. That produces a new accumulator value and the binary function is called with that value and the next element, etc.

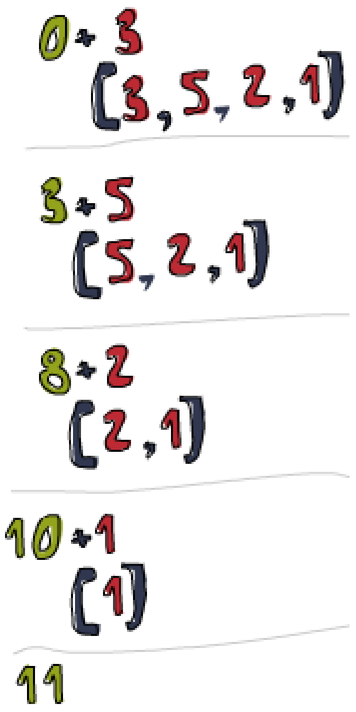
```haskell
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f acc [] = acc
foldl f acc (x:xs) = foldl f (f acc x) xs
```

This is the definition of the fold right.

```haskell
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f acc [] = acc
foldr f acc (x:xs) = f x (foldr f acc xs)
```

Let's implement sum again, only this time, we'll use a fold instead of explicit recursion.

```haskell
sum' :: (Num a) => [a] -> a
sum' xs = foldl (\acc x -> acc + x) 0 xs
Testing, one two three:
ghci> sum' [3,5,2,1]
11
```



Function application with $

Alright, next up, we'll take a look at the $ function, also called *function application*. First of all, let's check out how it's defined:

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

Whereas normal function application (putting a space between two things) has a really high precedence, the $ function has the lowest precedence. Function application with a space is left-associative (so f a b c is the same as ((f a) b) c)), function application with $ is right-associative.

That's all very well, but how does this help us? Most of the time, it's a convenience function so that we don't have to write so many parentheses. Consider the expression sum (map sqrt [1..130]). Because $ has such a low precedence, we can rewrite that expression as sum $ map sqrt [1..130], saving ourselves precious keystrokes! When a $ is encountered, the expression on its right is applied as the parameter to the function on its left. How about sqrt 3 + 4 + 9? This adds together 9, 4 and the square root of 3. If we want to get the square root of 3 + 4 + 9, we'd have to write sqrt (3 + 4 + 9) or if we use $ we can write it as sqrt $ 3 + 4 + 9 because $ has the lowest precedence of any operator. That's why you can imagine a $ being sort of the equivalent of writing an opening parentheses and then writing a closing one on the far right side of the expression.

Function composition

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

In Haskell, function composition is pretty much the same thing. We do function composition with the . function, which is defined like so:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)    or (.) f g x = f (g x)
```

Mind the type declaration. f must take as its parameter a value that has the same type as g's return value. So the resulting function takes a parameter of the same type that g takes and returns a value of the same type that f returns.

Custom type

Understanding the type system is a very important part of learning Haskell. A type is a kind of label that every expression has. It tells us in which category of things that expression fits. The expression True is a boolean,“hello”is a string, etc.
So far, we've run into a lot of data types. Bool, Int, Char, etc. But how do we make our own? Well, one way is to use the data keyword to define a type. Let's see how the Bool type is defined in the standard library.

```
data Bool = False | True
```

data means that we're defining a new data type. The part before the = denotes the type, which is Bool. The parts after the = are value constructors. They specify the different values that this type can have. The | is read as or. So we can read this as: the Bool type can have a value of True or False. Both the type name and the value constructors have to be capital cased.

We've been tasked with creating a data type that describes a person. The info that we want to store about that person is: full name, place of birth, year of birth, height, phone number. This is define as follow:

```
data Person = Person String String Int Float String deriving (Show)
let takam = Person "Takam Fridolin" "Yaounde" 1925  1.78  "691179855" .
```

This code create à person and assigns it the variable $takam$. In the creation of data assigned to $takam$, Person is used as a function and effectively it is.
The type of Person can be obtained by checking it ghci.

```
:t Person
```

```
Person :: String -> String -> Int -> Float -> String -> Person
```

As we can see, it takes as input String, String, Int, Float, String and returns à Person. *Person* is a value constructor.

What if we want to create a function to get separate info from a person? A function that gets some person's full name for instance.

```
getFullName :: Person -> String
getFullName (Person fullName placeOfBirth year heigh phone) = fullName
```

Here's how we could achieve the above functionality with record syntax.

```
data Person = Person {
      fullName     :: String
    , placeOfBirth :: String
    , yearOfBirth  :: Int
    , height       :: Float
    , phoneNumber  :: String
} deriving (Show)
```

So instead of just naming the field types one after another and separating them with spaces, we use curly brackets. First we write the name of the field, for instance, fullName and then we write a double colon : : and then we specify the type. The resulting data type is exactly the same. The main benefit of this is that it creates functions that lookup fields in the data type. By using record syntax to create this data type, Haskell automatically made these functions: *fullName*, *placeOfBirth*, *yearOfBirth*, *height*, and *phoneNumber*.

```
:t placeOfBirth
placeOfBirth :: Person -> String
```

There's another benefit to using record syntax. When we derive Show for the type, it displays it differently if we use record syntax to define and instantiate the type.

Typeclasses

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.
What's the type signature of the == function?

```
ghci> :t (==)
(==) :: (Eq a) => a -> a -> Bool
```

*Note: the equality operator, == is a function. So are +, *, -, / and pretty much all operators. If a function is comprised only of special characters,* *it's considered an infix function by default. If we want to examine its type, pass it to another function or call it as a prefix function, we have to surround it in parentheses.*

We see a new thing here, the => symbol. Everything before the => symbol is called a class constraint. We can read the previous type declaration like this: the equality function takes any two values that are of the same type and returns a Bool. The type of those two values must be a member of the Eq class (this was the class constraint).

The Eq typeclass provides an interface for testing for equality. Any type where it makes sense to test for equality between two values of that type should be a member of the Eq class. All standard Haskell types except for IO (the type for dealing with input and output) and functions are a part of the Eq typeclass.

The elem function has a type of (Eq a) => a -> [a] -> Bool because it uses == over a list to check whether some value we're looking for is in it.

how it works

This is how the Eq class is defined in the standard prelude:

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  x == y = not (x /= y)
  x /= y = not (x == y)
```

First off, when we write *class Eq a where*, this means that we're defining a new typeclass and that's called Eq. The a is the type variable and it means that a will play the role of the type that we will soon be making an instance of Eq.Then, we define several functions. It's not mandatory to implement the function bodies themselves, we just have to specify the type declarations for the functions.
we *did* implement the function bodies for the functions that Eq defines, only we defined them in terms of mutual recursion. We said that two instances of Eq are equal if they are not different and they are different if they are not equal. We didn't have to do this, really, but we did and we'll see how this helps us soon.
So once we have a class, what can we do with it? Well, not much, really. But once we start making types instances of that class, we start getting some nice functionality. So check out this type:

```
data TrafficLight = Red | Yellow | Green
```

It defines the states of a traffic light. Notice how we didn't derive any class instances for it. That's because we're going to write up some instances by hand, even though we could derive them for types like Eq and Show. Here's how we make it an instance of Eq.

```haskell
instance Eq TrafficLight where
    Red == Red = True
    Green == Green = True
    Yellow == Yellow = True
    _ == _ = False
```

We did it by using the **instance** keyword. So **class** is for defining new typeclasses and **instance** is for making our types instances of typeclasses. When we were defining Eq, we wrote *class Eq a where* and we said that a plays the role of whichever type will be made an instance later on. We can see that clearly here, because when we're making an instance, we write instance Eq TrafficLight where. We replace the a with the actual type.

Because == was defined in terms of /= and vice versa in the class declaration, we only had to overwrite one of them in the instance declaration. That's called the minimal complete definition for the typeclass — the minimum of functions that we have to implement so that our type can behave like the class advertises. To fulfill the minimal complete definition for Eq, we have to overwrite either one of == or /=. If Eq was defined simply like this:

```haskell
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
```

we'd have to implement both of these functions when making a type an instance of it, because Haskell wouldn't know how these two functions are related. The minimal complete definition would then be: both == and /=.

You can see that we implemented == simply by doing pattern matching. Since there are many more cases where two lights aren't equal, we specified the ones that are equal and then just did a catch-all pattern saying that if it's none of the previous combinations, then two lights aren't equal.

Let's make this an instance of Show by hand, too. To satisfy the minimal complete definition for Show, we just have to implement its show function, which takes a value and turns it into a string.

```haskell
instance Show TrafficLight where
    show Red = "Red light"
    show Yellow = "Yellow light"
    show Green = "Green light"
```

Once again, we used pattern matching to achieve our goals. Let's see how it works in action:

```
ghci> Red == Red
True
ghci> Red == Yellow
False
ghci> Red `elem` [Red, Yellow, Green]
```

```
True
ghci> [Red, Yellow, Green]
[Red light,Yellow light,Green light]
```

Nice. We could have just derived Eq and it would have had the same effect (but we didn't for educational purposes). However, deriving Show would have just directly translated the value constructors to strings. But if we want lights to appear like "Red light", then we have to make the instance declaration by hand.