Haskell and functional programming: a love letter

Philippe Pittoli

ABSTRACT

Haskell is a well-known functional programming language combining a unique set of features. This document will present the core concepts of functional programming through the Haskell syntax and modules. Some tips will be given on PureScript, an Haskell-like language aiming to replace JavaScript in web development.

This document is a collection of notes (on Haskell, FP, some APIs). This is not intended for a large public. However, a large part of it should be relevent for any FP and non-FP developer alike: take whatever you can! Also, I think mathematical-ish explanations of the language are a massive obstacle to its actual comprehension for a developer, so this document will be almost exempt of it. **You're welcome.**

Check out for newer versions: `https://t.karchnu.fr/doc/haskelltut.pdf`
And if you have questions: `karchnu@karchnu.fr`

Lastly compiled the **9/10/2022** (day/month/year, you know, like in any sane civilization).
Status: sections 1, 2 and 3 are about done. Others: WIP.

1. Functional programming

FP can be seen as an *"everything is a function"* paradigm, an attempt to create programs as single mathematical expressions. In that regard, each line of code represents the intent of the developer more than *how to perform stuff*.

1.1 Why this document?

- FP is great, let's talk about it.
- Haskell is great because it helps express the code with more nuances than other languages, but is a bit overwhelming on first try, so let's summary stuff.
- Everyone can learn something in this paper, even reading only the first pages and still writing in imperative languages.
- Explain Haskell, its main API and a few useful modules.
- Provide a few real-life examples.

1.2 Why FP and Haskell?

- Declarative: the language requires to write *what is* instead of *how to perform an action*. This simplifies the code tremendously, in most cases.
- Simple: the language actually is easy to grasp. There are many libraries and different ways to code, and this is what's actually hard to handle at first.
- Great type system: Haskell has one of the easiest ways to create types and constraints on types (equality, order, non empty collection, etc.). Many types are available in the standard distribution of Haskell, and are basic enough to be widespread in libraries.

  While strict, the type system also allows a very generic coding style **by default**. Code reuse is (near?) optimal.
- Concise: functions can be *composed* together, meaning that the result of a function is the parameter of another. In practice, this allows to write very short functions by removing unnecessary code[1].
- Purity (or referential transparency): a function only needs its parameters to work, no global variables. A pure function called twice with the same parameters will always provide the same result. Other functions (called *unpure*), managing events outside the application (such as IO in general), have a distinctive type. Consequently, investigation is easy when an error occurs: only a fraction of the functions handles hazardous operations. Furthermore, pure functions are great candidates for *memoization*, see the section on

1. Also, parenthesis aren't all over the place like in other FP languages (personal preference).

performances.

> *The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise.*

— Edsger W. Dijkstra

1.3 Sections

First, we'll talk about base concepts in FP, which will provide everything one need to start with both FP and Haskell. This will be quick, there is very little theory in this document.

Then the Haskell syntax: an introduction to the type system, basic types available in any Haskell distribution, how to write a function, etc.

Usual type classes will be presented to get some sense of the default Haskell library. This includes the main classes such as Functor, Applicative and Monad, and many others. The understanding of them is crutial: this is required to know how to use most of the third party libraries.

Some of the most basic functions in the Haskell distribution will be presented. This includes functions around lists (such as *fold*, *head* and *tail*, *map* and *filter*) a few functions around the `Text` module and how to manage user interaction in general, etc.

Then, a few ways to create complex data structures will be presented. This includes a clever hack (phantom type), some conventions to write data structures (tagless final, generalized algebraic data type) and the reuse of a generic data structure (Free monad).

After these sections, a few advanced topics are very briefly introduced:

- Lenses, to handle complex or nested data structures. This was one of the worst aspects of Haskell, until Lenses made nested structures so simple to manage that it became an asset.
- Networking, to exchange packets with widespread protocols.
- Profiling applications, to know what function actually takes time or memory to compute. As we will see, this is very empirical.
- Performances, to gain massive computation boosts with some tips. This includes rewriting functions in an optimizable way or use proper types, for example.
- Parallelism and concurrency, to either make several computations at the same time on different cores, or switch between two computations that require pauses[2].

Then, a *misc* section, for everything that is useful but doesn't belong anywhere else.

Finally, this document provides a few pointers on useful documentations to dig more advanced concepts.

How to read this document

The following presents a typical function declaration.

```
multBy2 :: Int -> Int  Function's type
multBy2 x = x * 2      its name, param and body
```

Function *multBy2* takes an integer and returns an integer.

Another function declaration, with a slightly different syntax.

```
add :: Int -> Int -> Int  Function's type
add x y                   its name and parameters
  = x + y                 its body
```

The first line is the function's type: *add* takes two integers and returns another one. The second line is the function name and named parameters (x and y). The last line is the body of the function. In Haskell, indentation is important to know what is the context of the function or not, but it isn't strict on the number of spaces.

This document will sometimes present function results, example:

```
1 + 2   some function call
> 3     and its result
```

2. Core concepts

Functional programming brings some concepts that aren't present (or widespread) in other paradigms. This section presents some of these concepts: *currying*, *function composition*, *High Order Functions*, *referential transparency*(purity) and *laziness*.

2.1 Currying

Or, *why do functions take a single argument*?

In Haskell and other functional programming languages, functions only take a single argument. Let's take an example with a function **add**, which is the sum of two integers.

2. For example, receiving network packets takes time, and a packet can be received only partially and requires to wait to retrieve the rest, and with concurrency the application can perform other actions while waiting for the packet to arrive. This improves performances by a large margin without even requiring parallelism.

```
add :: Int -> Int -> Int
add x y = x + y
```

The first line is the type of the function. One way to understand the type `Int -> Int -> Int` is that the function takes two integers and returns another one. *But why is this written without a clear difference between parameters and the returned value*? Because we can partially apply the function.

```
add1 :: Int -> Int
add1 = add 1
```

`add1` is `add` but with a default parameter.

How to read this? Function `add1` uses the `add` function with a default parameter set to 1. `(add 1)` is a function. It returns a function taking a single integer and returning an integer. `add1` could have been written with an explicit parameter this way `add1 x = add 1 x` but since it is just the partial application of another function, there is no need.

Currying: each time a parameter is provided to a function, another function is returned. This goes until all parameters are provided and the function actually is performed.

In Haskell and in functional programming in general, the concept of *currying* functions is widespead, and brings conciseness and code reusability. Plenty of examples will be presented later.

2.2 Function composition

Function composition is the act of pipelining the result of one function, to the input of another. This is almost like in shell but the order is reversed, and typed[3].

```
the '.' operator is used to compose functions

result of 'sort' is pipelined to 'reverse'
sort_and_reverse = reverse . sort

the result is a descending sort (from 10 to 1)
countdown = sort_and_reverse [2,8,7,10,1,9,5,3,4,6]

shorter
countdown = (reverse . sort) [2,8,7,10,1,9,5,3,4,6]
```

3. Data isn't just serialized in strings like in shell (for the most part). And each output type of a function must be the input type of the following function in the composition.

2.3 High Order Functions

Functions in functional programming languages are simple types, they are treated as any other type. Functions can be given as parameters for other functions.

```
(* 2) is a function multiplying by two
a value given in parameter
map (* 2) [1,2,3,4,5]
> [2,4,6,8,10]
```

`map` is a function taking another function as its first parameter, and applies it to each element of a list.

2.4 Referential transparency (Purity)

Referential transparency (or *purity*) is the property of an expression when it only uses its parameters to compute a value. In this context, two computations of the same expression with the same parameters will produce the same result. Example: `1 + 2` will always produce *3*. Functions aren't pure when they rely on input and output, like networking. A function can be recognized as non pure given its type, when the function relies on the IO monad for example (see later for details).

Why is purity a big deal? Two examples.

- Memoization: purity ensures that two computations produce the same result. Memory can be traded for computation speed. Pure functions can be called once for a given set of parameters, and their result can be stored in memory. There is a gain when the computation is longer to execute than a lookup in a table.

  Memoization can transform naive implementations of some recursive algorithms into legitimate solutions.

- Common subexpression elimination: the compiler can optimize the code by rewritting some expressions.

```
these two expressions share a common computation:
a = b * c + g
d = b * c * e

and can be rewritten this way:
tmp = b * c
a = tmp + g
d = tmp * e
```

In this example, `b * c` is factored, but it works with any pure function.

2.5 Laziness

Laziness means to compute a value only when necessary. Why calculate every element of a list when you only use the first one?

```
take 5 elements of an infinite list
take 5 $ [1..]
> [1,2,3,4,5]
```

Laziness is great for creating simpler programs. Algorithms can sometimes be expressed in a simpler way when infinite lists aren't a problem.

In practice, we all encountered an example of laziness that makes programming easier: shells.

```
find / |
  grep -E "(foo|bar)" |
  head -n 5 | # take only 5 elements
  mail -s "5 foo bar stuff" somebody@example.com
```

In this example, once 5 elements get to pass the *grep* filter, this script stops. Millions of files could be on this system, they won't be explored. Shell is lazy: a program stops when the next program in the pipeline closes its input, not only when it finishes its task.

However, laziness could be detrimental to performances, too. As with the shell, laziness has a cost. See the section on performances.

3. Introduction to Haskell

Functional and imperative programming languages are differents on many levels. This section provides an overview on those differences.

This section presents:

- an implementation of the fibonacci sequence to get a taste of those differences with a concrete yet simple example;
- a few Haskell basic types;
- the Haskell syntax for functions;
- the Haskell type system, which includes basic types such as integers and characters, but also more advanced types such as sum, product and algebraic types;
- a gentle introduction to Haskell type classes.

And finally, a brief discussion on the Haskell language, functional programming in general and the rest of the document.

3.1 A first example

Just to get started on how to write a program with a functional programming language, here is an example: the fibonacci sequence.

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib x = fib (x-1) + fib (x-2)
```

A few notes on what is happening:

- The first line is the type of the function: it accepts an *Int* as its first (and only) argument and returns another *Int*.
- The body of the function changes depending on the argument value.
- Finally, if the parameter isn't 0 or 1, it performs `fib (x-1) + fib (x-2)`, which can be read as the mathematical expression of the Fibonacci sequence.

Compared to an imperative programming language, the code for this function is considerably smaller and more readable. This can be explained in several points:

- Recursion: the function calls itself. Some algorithms are easier to write this way, there is no need for loops and maintaining a state (an index and accumulators in this case).
- Pattern matching: while there is no explicit conditions, the function actually provides a different implementation depending on the value of the parameter. The behavior of the function is trivial to read, less error-prone than with explicit conditions.
- No return statement: a function in Haskell is a declaration, it says what *is*. This is not a set of instructions to execute, this is a single expression representing what the function mean. Consequently, the whole expression is to be executed and its single produced value is to be returned by the function.

3.2 A few Haskell basic types

Before introducing syntax for functions, a quick overview of very simple types in Haskell.

```
5                        :: Int
5.5                      :: Float
[1,2,3]                  :: [Int] list of integers
'H'                      :: Char
True                     :: Bool
"hello"                  :: String
['w','o','r','l','d']    :: String (or [Char])
("smth", 8)              :: (String, Int)  tuple
```

Some function types (details later).

```
(+) :: Num a => a -> a -> a
      'a' = Number type (Int, Float, ...)
(==) :: Eq a => a -> a -> Bool
       'a' = type that can be tested for equality
```

Both functions have a *constraint* on their parameters. Function `+` requires both its parameters to be numbers, `(==)` requires its parameters to be any type that can be tested for equality (like integers). More on constraints later.

Just a taste of how we can define data structures.

```
real definition of Bool in the standard library
data Bool = False | True
```

A `Bool` is either a *True* or a *False*. True and False are called *constructors*.

```
real definition of String in the standard library
type String = [Char]  list of Char
```

A `String` is a list of `Char` (characters), they are synonyms. Also, writing a string with `"hello"` is syntactic sugar for `['h', 'e', 'l', 'l', 'o' ]`

3.3 Haskell's syntax for functions

Functions in imperative and functional languages are different concepts.

In an imperative language, a function is a set of instructions to execute and are gathered and named to be called later, possibly several times or just to cut the code in more readable pieces. This is more efficient than having to copy the same instructions each time they are needed: fewer lines of code, fewer potential errors, and the code is more readable.

In a functional language, the whole function is a declaration, a single expression, the content is what the function means[4]. The entire body of the function is the returned value. Functions can still be complex, and composed of several function calls. However, all these function calls are bounded together, by an operator for example[5].

The syntax for functions is rather extended in Haskell compared to an imperative language. Following sub-sections present different ways to create functions.

4. And since the function carries a meaning, there is little to no place to add unrelated instructions in a function in FP. This may seem like a curse, but this is actually a bless in disguise. One can write any debugging code without interfering with the actual useful code, debug has to be separated from the rest. Also, functions in the standard library are almost always one-liners.

5. This will be explained in details later (with *monads*).

3.3.1 Pattern Matching

A function can take parameters, and the function body can change according to their value. The fibonacci sequence included pattern matching on a number.

```
fib :: Int -> Int
fib 0 = 0    in case param == 0
fib 1 = 1    in case param == 1
fib x = fib (x-1) + fib (x-2) otherwise
```

In practice, pattern matching is often used on data structures.

```
not :: Bool -> Bool
not True  = False
not False = True
```

Constructors, such as *True* and *False* for the `Bool` data structure, can be used in pattern matching.

When the actual value of a parameter isn't necessary, there is no point to even name it; it can be replaced by an underscore.

```
not :: Bool -> Bool
not True  = False  case where the parameter is True
not _     = True   any other case
```

Pattern matching can destructure lists.

```
Quick introduction to lists:
[1,2,3] list of integers
1:[2,3] add 1 to the head of the list [2,3]
        ":" is an infix constructor taking
        a value and a list

len: computes the number of elements in a list
len :: [a] -> Int
len []     = 0          empty list
len (x:xs) = 1 + len xs at least a value (x)
```

Function *len* takes a list (of any type) and returns a number. First case, the function takes an empty list, its value is 0 (no element in the list). In case the list isn't empty, it can be destructured: a list can be seen as the infix constructor `:` and a first value *x* followed by the rest of the list *xs*. So, once destructured, two informations are available: *x* (the head value of the list) and *xs* (the rest of list, potentially empty).

Since the function *len* only has to compute the number of elements, the actual value of *x* isn't important, let's rewrite:

```
len :: [a] -> Int
len []     = 0
len (_:xs) = 1 + len xs   x became '_'
```

This time, the function doesn't name the list's head: it is explicitly ignored.

Also, *{}* can be used to pattern match on the constructor regardless of the content of the type.

```
data Foo = Bar | Baz Int
g :: Foo -> Bool
g Bar {} = True
g Baz {} = False
```

Pattern matching on more complex types will be presented later.

3.3.2 Guards

Pattern matching provides a different function body according to the value of a parameter. Sometimes, this is not enough, and the parameter has to be tested more thoroughly, by calling a function for example. Guards provide a different function body according to tests on values.

```
not :: Bool -> Bool
not v
  | v == True = False
  | otherwise = True
```

Guards elegantly replace some conditional instructions (predicates) at the start of imperative functions.

Predicates and function's body are clearly identified.

3.3.3 Case ... of

A value can be tested through *case ... of* which is like a switch in C, for example.

```
not :: Bool -> Bool
not v = case v of
  True  -> False    in case v is True
  _     -> True     in case v is any other value
```

3.3.4 Anonymous functions: lambdas

An anonymous function is created with the backslash character \ followed by the parameters, then an arrow (->) and finally the body of the function. This anonymous function is called a *lambda*[6].

6. Since the mathematical explanation of lambdas is completely overkill to understand how to use them, it is discarded in this document. You're welcome.

```
add 5 to each element of a list
map (\x -> x + 5) [1,2,3,4,5]
> [6,7,8,9,10]

sum both elements of each tuple
map (\(x,y) -> x + y) [(1,2),(3,4),(5,6)]
> [3,7,11]
```

Lambdas are widespread in Haskell and in FP in general since this makes the code very concise. However, when possible, use partial function application (even more concise), by example:

```
map (\x -> x + 5) [1,2,3,4,5]
    could be written this way:
map (+ 5) [1,2,3,4,5]
```

3.3.5 Where and let

Within the scope of a function, one can declare functions or constant values.

```
health :: Float -> Float -> String
health height weight
  | bmi < 18.5                = "underweight"
  | bmi >= 18.5 && bmi < 25.0 = "normal weight"
  | bmi >= 25.0 && bmi < 30.0 = "overweight"
  | bmi >= 30.0               = "obesity"
  where bmi = weight / (height * height)

health 1.62 70
> "overweight"
```

Function *health* uses the value *bmi* computed within the function, after the *where* keyword. The value *bmi* uses any available value within the context of the function *health*. In this case, *bmi* uses both *height* and *weight*.

Besides indentation, functions within the context of a function aren't different from what the document shown before. They also can have an explicit type.

```
health height weight
   [...]
  where
    bmi :: Float
    bmi = weight / (height * height)
```

Function *bmi* doesn't need parameters since it already has access to the relevant values (in the scope of the *health* function).

The *let* notation can be put in any place where a statement is expected. That is the main difference with *where*. Example:

```
f :: s -> (a,s)
f x =
  let y = ... x ...
      z = ... x ...
  in  y/z
```

Let or where?

Chosing either *let* or *where* is mostly a matter of taste. Though, one could be prefered in some cases. Refactoring is easier with *let* when the declarations have to be put inside a lambda expression, for example. However, *where* is prefered when the same declaration should be shared between several expressions, which would imply some boilerplate with *let*.

```
Refactoring this
f x =
  let y = ... x ...
  in  y
into this
f = State $ \x ->
  let y = ... x ...
  in  y
wouldn't have been possible with 'where'.


However, writing this with 'let' would be painful
f x
  | cond1 x   = a
  | cond2 x   = g a
  | otherwise = f (h x a)
  where
    a = w x
(it could been mixed with 'case' to make it work,
but ultimately make it harder to write and to read)
```

Chosing the right one comes with experience, nothing to worry about.

3.4 Haskell's type system

This document already introduced primitive types (integer, float and character) and a few others: Bool, tuples and lists. Functions also have their own type, and can be passed as function parameter as any other type of value.

This section introduces a few aspects of the Haskell type system. First, holes to ask the compiler what type is required at some point. Second, the multiple ways to create new structures with the *data* keyword. Finally, type synonyms, with the *type* keyword, to make the code more understandable to other developers[7].

7. Documentation through type names is both elegant and effective, even if this isn't sufficient by any mean.

3.4.1 What type should I use? Holes!

Haskell has a great type inference. When writing a function, the actual type of the missing code can be asked to the compiler by writing a *hole* in the code, which is any name starting with an interogation (**?**) character.

```
foo :: Int -> Int -> Int
foo x y = x + ?a          the hole is named 'a'
```

For an unnamed hole, write an underscore.
Holes also work in function types.

3.4.2 Data structures

One of the big challenges of a developer is to create data structures. Once this part is done, related code almost writes itself. Following sub-sections present different ways to create structures with the `data` keyword.

0.0.0.1. Sum

A sum type is a simple enumeration.

```
data Bool = False | True

how to create a Bool value
isItTrue = True

not :: Bool -> Bool
not True = False
not False = True
```

A boolean value is either true or false, which is a sum type. Both `True` and `False` are constructors for the type `Bool`. Pattern matching works on constructors.

Check for non exhaustive patterns with *-fwarn-incomplete-patterns*.

0.0.0.2. Product

A product type is a type containing data.

```
data Figure = Rectangle Double Double

how to create a Figure
myRectangle = Rectangle 10.0 30.0

pattern matching on Figure
area :: Figure -> Double
area (Rectangle height width) = height * width
```

In this example, `Rectangle` is a *constructor* to create a value of type `Figure` and it contains two floating point numbers. Pattern matching works on constructors, and their parameters are named to be used in the function.

0.0.0.3. Record

Record type is a product type with named parameters.

```
data Figure = Rectangle { height :: Double
                        , width  :: Double }

works as before
myRectangle = Rectangle 10.0 30.0

works as before
area :: Figure -> Double
area (Rectangle height width) = height * width
```

This time, `Rectangle` has two named parameters: *height* and *width*. Creating a figure works as before, and pattern matching too.

Naming parameters automatically creates functions with the same names to get their value from a figure[8].

```
compute area without pattern matching
area :: Figure -> Double
area f = height f * width f
```

In this example, functions *height* and *width* were used instead of the pattern matching.

Since constructor parameters are named, this is possible:

```
Naming constructor values on declaration.
myRectangle = Rectangle { height = 5.0
                        , width = 10.0 }

Getting only a subset of constructor values.
h (Rectangle {height=v}) = v
```

Naming constructor parameters is ∞ better than only passing arguments by value. The function *h* takes a `Rectangle` as parameter, but the parameters of the constructor aren't mentionned. The actual useful value (height) can be obtained directly. This way, data structure can evolve without breaking anything.

0.0.0.4. Algebraic

Algebraic type is both sum and product types.

```
data Figure
  = Rectangle Double Double
  | Disc Double

myDisc = Disc 5.0
myRectangle = Rectangle 5.0 10.0

area :: Figure -> Double
area (Rectangle h w) = h * w
area (Disc r)        = pi * r ** 2
```

Convenient: each figure has its own statement, any error become obvious and a missing case would be automatically detected. The equivalent in imperative programming is less readable.

8. This forces developers to think about names, not to overlap with preexisting functions.

0.0.0.5. Recursive

An algebraic data type is recursive if its declaration involves itself. This is common to describe lists, trees, etc.

```
data List
  = Element Int List
  | End

someList :: List
someList = Element 1 (Element 2 End)

mult2 :: List -> List
mult2 End                = End
mult2 (Element x rest) = Element (x*2) (mult2 rest)
```

mult2 takes a list and returns a list. When the list is empty, the return is an empty list. When the list isn't empty, it is destructured to see the current element and the *rest* of the elements (which is a list in our definition). The new list is created with the `Element` constructor, with our current element *x* multiplied by 2 as our first parameter, and `mult2 rest` as the rest of the list (the second parameter of the `Element` constructor).

Working with recursive types is a bit complicated. Any function working on all the elements of the list needs to be recursive, too. Well, for now at least. The `Functor` type class will make it trivial.

0.0.0.6. Polymorphic

Types are *polymorphic* when they have a type parameter, meaning that the type of the values they contain isn't fixed. For example, a list may contain integers, strings or anything else, and that's still a list. Fixing the type of the values it contains would be arbitrary and very limiting.

```
data List a
  = Element a (List a)
  | End

listInt = Element 1 (Element 2 End)
listString = Element "Hello" (Element "world" End)

mult2, as before BUT with a constraint on 'a'
mult2 :: Num a => List a -> List a
mult2 End                = End
mult2 (Element x rest) = Element (x*2) (mult2 rest)
```

`List` *a* is a list of values of any type. However, its declaration implies that a list is composed of values of the same type, a list cannot contain both an integer and a string. Working with types like this may require to constrain the inner value types, as it is done in *mult2* with the `Num` constraint (inner values have to be numbers).

0.0.0.7. Summary on data types

Let's recap the available data types in Haskell.

- sum type: simple enumerations.
- product type: data structure needs to store a value (not just the constructor).

- record type: product type with names for the stored values.
- algebraic type: sum of product values. This can be combined with recursive and polymorphic types.
- recursive type: data structure includes itself in its definition.
- polymorphic type: data structure needs to store a value without imposing the type.

Haskell is built on these types, they all have a purpose and help describe different data structures.

3.4.3 Type synonyms

A floating point number can be a height, a length, a random number, or the average size of guinea pigs in a pet store.

```
What does the surface function compute?
What are the parameters? Its returned value?
surface :: Float -> Float -> Float
```

Writing `Float` as a parameter doesn't provide any meaning. To that end, type synonyms help writing more meaningful function types.

```
type Height = Float
type Width = Float
type Area = Float
surface :: Height -> Width -> Area
```

There are probably better ways to name this function, but still, now its parameters and the return value are explicit.

Type synonyms provide the semantic behind the types.

3.5 Haskell's type classes

Types may be related to each other. An integer and a float, wheither their size, are both numbers, for example. A class of types is defined by the functions they implement. The class `Num` (numbers) is defined by the functions related to numerical operations, such as `+, -, *, /` and so on.

In Haskell, many type classes are provided by default, and some will be introduced later.

Syntax

Let's see some parts of the Haskell standard library: the `Semigroup` class. *Semigroup* is just a fancy word to say something really simple. It represents types with values that can be appended, joined, concatened to each other[9]. The list type is part of the semigroup class: `[1,2]` can be concatened to `[3,4]` and produces `[1,2,3,4]` (in this order).

The following example shows the definition of *Semigroup* in the standard library, then implements an instance for the recursive and polymorphic data type `List` *a*, defined earlier (in the "polymorphic type" section).

```
Type class definition: function(s) to implement
to be part of it.
class Semigroup a where
    <> is a concatenation operator.
   (<>) :: a -> a -> a

instance for the "List a" type
instance Semigroup (List a)
  Implementation time!
  '<>' operator: appending two lists.
  End            <> End              = End
  (Element x xs) <> End              = Element x xs
  End            <> (Element y ys)   = Element y ys
  (Element x xs) <> (Element y ys)
    = Element x (xs <> (Element y ys))
```

To implement the `<>` operator is very similar to create a new list. We should always start with the simpler case: both lists are empty, so the result is an empty list. In case one of the lists is empty, the result is the content of the other one. Finally, in case both lists have values, the result is a construction of a list with the values of the first list first. The concatenation of two lists, let's say `[1, 2]` and `[3, 4]` will result in `[1, 2, 3, 4]` (in that order).

Plenty of examples are provided in the section on usual type classes.

Laws

Sometimes, in order to have a meaningful type class, the behavior of the structure, given a function, has to be imposed. For example, the `<>` function from the `Semigroup` type class requires the data structure to be associative.

```
associtivity is required for the operator '<>'
(a <> b) <> c == a <> (b <> c)
```

a associated with *b* THEN associated with *c* has to provide the same result as *a* associated with the result of *b* associated with *c*.

Always verify that your structure satisfies the laws required by the type classes you implement. Otherwise the semantic of the type class will be broken and the behavior won't make sense[10].

Summary

A type class regroups similar types, related to each other by the functions they can perform. Sometimes, they have to obey laws, such as *associativity*, in order to ensure an expected

9. There are plenty of other terms like "Semigroup" used in Haskell that actually aren't complicated. They will be translated for the mere mortals in due time, don't worry.

10. Furthermore, it could be completely legitimate for the compiler to implement code optimizations to cut a few function calls, or rewrite some functions, based on these laws.

behavior for all these types.

Type classes maximize code reusability since functions are very generic, and can work not with types, but with classes of types.

Type inference is simple, too. When writing a function, finding the required type classes only is searching for used functions in available type classes. Example: in the function `blah x y = x + y` since `+` is used on both x and y, they both need to be in the type class `Num`.

3.6 Modules

Any non trivial program needs to split its code base into managable pieces. Each file will represent a *module* which can be imported (even partially) in other modules.

Module import

```
somewhere on your system there is a file named
data/bytestring.hs
import Data.ByteString
```

Import all functions and types from the module.

```
import qualified Data.ByteString as B
```

Import all functions and types from the module, but they all have to be prefixed by *B*.

```
import Data.ByteString (pack, unpack)
```

Import only *pack* and *unpack* functions.

```
import Data.ByteString (ByteString)
```

Import only the *ByteString* type (not its constructors).

```
import Data.ByteString (ByteString(..))
```

Import the *ByteString* type and its constructors.

```
import Data.ByteString hiding (head)
```

Import all functions and types except the function *head*.

Module declaration

```
File: some/simple/module.hs
module Some.Simple.Module where
followed by the module's code
```

All functions and types in the module are exported by default.

```
with explicit exports
module Some.Simple.Module (
    some, functions, or, Types(..), to, export
  ) where
```

3.7 Discussion on Haskell and common concepts

This section shown most of the common ways to create functions and data structures in Haskell. This is a boring but non avoidable part of the journey to learn the language, and this only scratched the surface.

Haskell is an evolving language, more than most other languages. It already has many extensions and more will come since Haskell is made by researchers constantly playing with the language. Fortunately, there is no point trying to document every extension: the core of the language actually is robust and wasn't touched **in decades**.

To understand idiomatic Haskell code, to understand functionnal programming and to be able to write any non trivial program, the next three sections are necessary. The first presents some very widespread data structures. The second presents the usual type classes, found in almost every non trivial code. This includes type classes used to structure the code (chaining function calls for example) and an introduction to unpure functions. And the third section presents the usual functions used in Haskell code.

4. Usual data structures

Haskell provides some very simple and ubiquitous data structures. This very brief section covers some of them.

4.1 Bool

A boolean value is either true or false. Haskell has probably one of the simplest way to express this.

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

This is a sum data type.

4.2 List

A list in Haskell is as simple as one can expect.

```
data List a = Nil | Cons a (List a)
```

This is a polymorphic, recursive data type.

However, lists do have special notations in Haskell. *List a* is written as *[a]*, the empty list (*Nil*) as *[]* and the constructor *Cons* is found as an infix operator *(:)* about everywhere in Haskell code.

4.3 String

A `String` is a list of characters.

```
type String = [Char]
```

This isn't a data structure but a type synonym. Still, it made sense to show this here.

4.4 Maybe

The `Maybe` data structure is simple: either there is a value (stored with the `Just` constructor) or there is nothing.

```
data Maybe a = Just a | Nothing
```

This is a polymorphic algebraic data type. Those words aren't so scary now, heh?

4.5 Either

A computation can work or fail, which already can be represented by a `Maybe` value. However, uppon failure, an error value could be interesting to get instead of `Nothing`. The `Either` data type is exactly that, a way to convey a potential error value.

```
data Either a b = Left a | Right b
```

This is a polymorphic algebraic data type, as Maybe.

5. Usual Haskell type classes

A type class represents a property of a data type. For example, a type that isn't empty (NonEmpty), or that can be serialized in a printable string format on the terminal (Show). Type classes allow developers to focus on an abstraction of the types they work with; a function can express constraints on its parameters instead of actual types[11].

The section introducing Haskell summarized type classes as a set of functions. A type needs to implement them to be part of the type class. Implementation also needs to respect a set of laws depending on the semantic of the type class, such as *associativity* or *transitivity* for example.

Many type classes exist in the standard distribution of Haskell. Most of them are widespread in libraries since they are useful in many contexts. Here is a first sample: `Num` (numbers), `Eq` (types that can be tested for equality), `Ord` (types that can be sorted), etc.

Knowing all classes and their little implementation details isn't necessary. However, a few of them are really interesting from an educational perspective or for composing bigger programs.

In this section, *Eq* and *Ord* type classes are first introduced. They provide simple examples of actual type classes and how to create instances. Then, the three usual type classes *Functor*, *Applicative* and *Monad* since they are related to each other and widespread in Haskell. Each of them allows to abstract some parts of usual code structure, such as loops, some conditions and a bit of error management. Some widespread monads will be introduced, such as the IO monad, allowing us to write our first application: a magnificent hello world! Finally, a brief conclusion on type classes and their usefulness.

5.1 Eq: types that can be tested for equality

The *Eq* type class represents all types that can be tested for equality. It is based on the `Bool` data type, and requires only to define a single function to test for equality. Here is the definition:

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

  x /= y     = not (x == y)
  x == y     = not (x /= y)
```

Either the operator `(==)` is defined or its opposite `(/=)` The other will be derived from the implemented one, see the default implementations.

An example of data structure, part of the Eq type class:

11. And *abstract* code isn't in any way a synonym of *slowness*. This idea will be destroyed in the section on performances.

```
data Room = Room Int a room has a number

instance Eq Room where
  Rooms are "equal" if they have the same number.
  (Room x) == (Room y) = x == y
```

The Room data structure represents a room with a number in an hostel, for example. This example shows how to implement an instance of the *Eq* type class.

Then, the code can be tested in GHCi:

```
r1 = Room 4
r2 = Room 4
r3 = Room 8
r1 == r2
> True
r1 == r3
> False
```

Before switching to another type class, here is the implementation of the *Eq* type class for the `Maybe` data structure.

```
data  Maybe a  =  Nothing | Just a
  deriving (Eq, Ord)
```

The actual implementation doesn't even exist: it is derived automatically. This way of deriving implementations of type classes from data structures won't be covered right now. See later sections.

5.2 Ord: types that can be sorted

The *Ord* type class represents all types that can be sorted (is a value smaller or equal than another). The type class introduces the following data structure and function.

```
data structure to compare values
data Ordering = LT | EQ | GT

Ord depends on the class Eq: any sortable
value has to be part of the Eq type class
class Eq a => Ord a where
  only function to implement
  compare :: a -> a -> Ordering
```

Here is an implementation of the type class for the previous `Room` data structure.

```
instance Ord Room where
  'compare' already exists for integer values
  compare (Room x) (Room y) = compare x y
```

Now the `Room` data structure is part of the *Ord* type class, a list of rooms can be sorted.

```
import Data.List (sort)
sort [Room 2, Room 3, Room 1]
> [Room 1, Room 2, Room 3]
```

As you can expect, some functions require their parameter to allow sorting. Sorting functions, sure, but also `min` and `max` functions for example.

Some widespread types and data structures are part of the Ord typeclass, such as `Int, Maybe` and `Either` for example.

```
sort [Right 2, Right 1]
> [Right 1, Right 2]
```

Nothing fancy.

This also works with nested structures.

```
sort [Right (Just 2), Right (Just 2)]
> [Right (Just 1), Right (Just 2)]

like, for real
sort [ Right (Just (Right (Room 2)))
     , Right (Just (Right (Room 1)))
     ]
> [ Right (Just (Right (Room 1)))
  , Right (Just (Right (Room 2)))
  ]
```

Let's break the last example. The `sort` function allows to sort a very nested structure. *How*? The `Room` structure is part of *Ord* so it can be sorted. `Either` too, so `Right (Room 1)` is sortable. Since this can be sorted and that `Maybe` is part of the *Ord* typeclass, then `Just (Right (Room 1))` also can be sorted, etc.

And we just touched something really interesting with Haskell. Despite its simplity, code is generic and reusability is great!

5.3 Functor: applying a function in a structure

A functor is a type allowing to apply a function to its inner value(s), through a `fmap` function. The following code represents the `Functor` type class and the only required function.

```
class Functor a where
  fmap :: (a -> b) -> f a -> f b
```

fmap takes a function (from *a* to *b*) and a functor (such as a list) containing values of type *a*. The fmap function produces a functor with values of type *b*.

`fmap` and `map` functions are synonyms for a list. Here is an example.

```
fmap (+3) [1,2,3]
> [4,5,6]
```

The fmap function for lists applies a function to each element of that list, and returns a new list.

And as we saw with the type of the fmap function, the returned functor can have a different inner type. Example:

```
fmap show [1,2,3]
> ["1","2","3"]
```

The show function takes a value and provides a string representation.

Lists aren't the only functors. Any type containing data (*product data type*) can be a functor. Maybe and Either are functors, too. So it is possible to apply a function to their inner values.

```
fmap (+3) (Just 3)
> Just 6
fmap (+3) (Nothing)
> Nothing
```

In case the Maybe functor contains a value, the function is applied. Otherwise, it just returns Nothing.

```
fmap (+3) (Right 3)
> Right 6
fmap (+3) (Left "error")
> Left "error"
```

The behavior of the Either functor is similar to Maybe. In case there is a value (read: a Right value), the function is applied to it. In case there is an error (read: a Left value), the function is ignored.

The general idea is the same behind both the Maybe and the Either functor implementations: either the value is valid and the function is applied to it, otherwise the error value is returned. This isn't exactly the same as the list functor, where the data type doesn't represent any possible "error". The implementation of the *Functor* type class actually depends on the semantic of the data type.

Maybe implementation

To further demystify the *Functor* type class, here is the implementation for the Maybe data type.

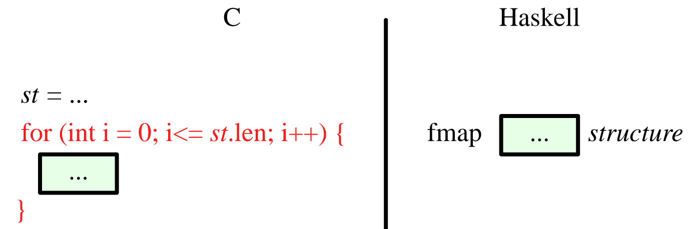
```
instance Functor Maybe where
  fmap _ Nothing       = Nothing
  fmap f (Just a)      = Just (f a)
```

In case the maybe has no value, this returns nothing. Otherwise, this creates a new Maybe with `f a` as the value.

Imperative vs functional structure manipulation

Functors are a way to manipulate data structures.

The following figure compares a list manipulation from C and Haskell perspectives.



Functor

manual structure manipulation vs fmap

(Bloat) code required in C to loop over the data in red. Green boxes represent code for data manipulation. Manipulated structure is in italics.

Why is this interesting? First, mindless code should be removed from the source. In this example, writing code to browse a list is a **very common** pattern, it should be abstracted[12]. Second, having an abstraction on "browsing a structure" allows to change the structure at no cost. What if the structure wasn't a simple list anymore? The C code should be rewritten completely.

Synonym: infix <$> operator

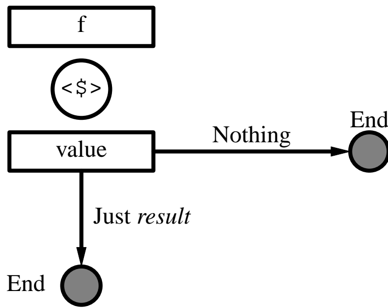
Finally, the fmap function has a synonym: the infix operator `<$>`

```
(+1) <$> Just 1
> Just 2
```

This may not seem much right now, but Haskell is often all about chaining expressions. Operators are very common, and often add readability with complex code. This operator <$> actually is relevant for the next sections and to write idiomatic Haskell code.

The following figure shows a diagram of the Maybe functor.

12. Less code means less ways to screw everything, and more focus on the actual problem to solve. That's a rule of thumb everybody should try to follow: write less code unless it makes the code cryptic.



Maybe functor
f <$> value

Conclusion on functors

The functor type class allows a great code abstraction. First, it applies a function to a whole data structure with a single and simple function call. There is no need for loops, and it is generic (works the same way for many types). Second, it follows a general and expected behavior: the function only applies when relevent, depending on the semantic of the data structure. The function isn't propagated when the data type conveys an error (such as a `Left` value in an Either data type).

And keep in mind the `<$>` operator!

5.4 Applicative: applying parameters to a functor

A functor is a type containing data that can be changed through the `fmap` function. The applied function may require several arguments, and applying it to a single argument creates another function. Example:

```
remember, <$> is an infix 'fmap'
(+) <$> Just 3
> Just (3+)
```

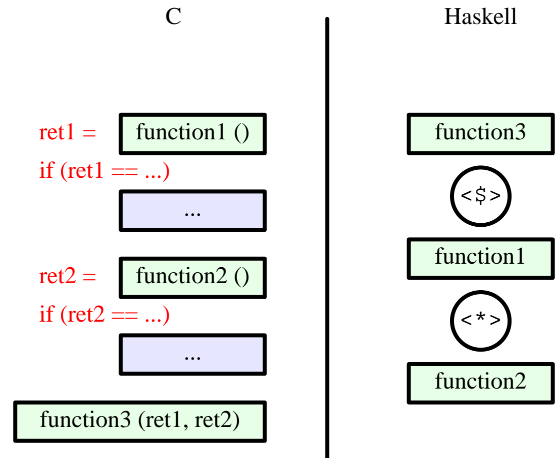
Unfortunatelly, GHCi cannot print functions this way, but this is the right result.
In functional programming, a data can be a function, as any other value. The Maybe functor now contains a function, lacking an argument.

An applicative functor is a way to provide a parameter to a functor. The Applicative type class introduces the infix operator `<*>` to *apply* a new parameter to a functor.

```
(+) <$> Just 3 <*> Just 2
> Just 5
```

The (+) function gains a first argument with `fmap` (`<$>`) and creates the Just (+3) value (which is (3+) in the `Maybe` functor). Then, the function *(3+)* is passed to the Just 2 value via the `<*>` operator. And this completes the function call and the result is Just 5

Why is this a thing? Applicative functors present a way to handle a common code structure. Let's see the following example:



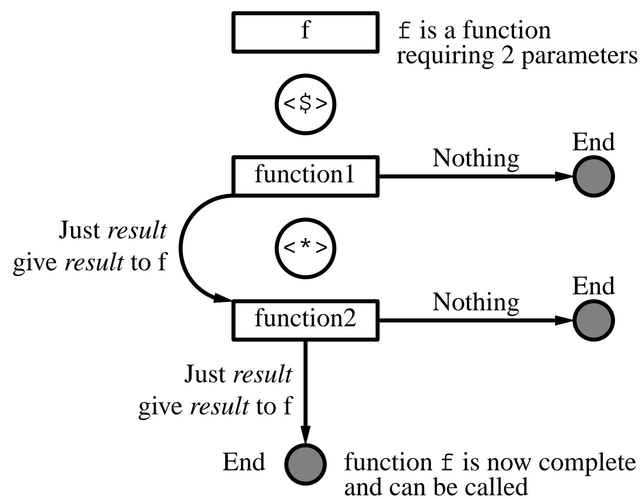
Applicative functor:
getting *function3* parameters ready, then call it

Error checking is in red, error management is in blue, function calls are in green.

In this example, in both C and Haskell, we try to call `f` with the results of `function1` and `function2` as parameters. Both handle errors: in case either the first or the second function fails, the rest isn't called and the function returns an error value. This error value is either a default error value, such as `null` in C or `Nothing` in Haskell, or the value provided by the defective function.

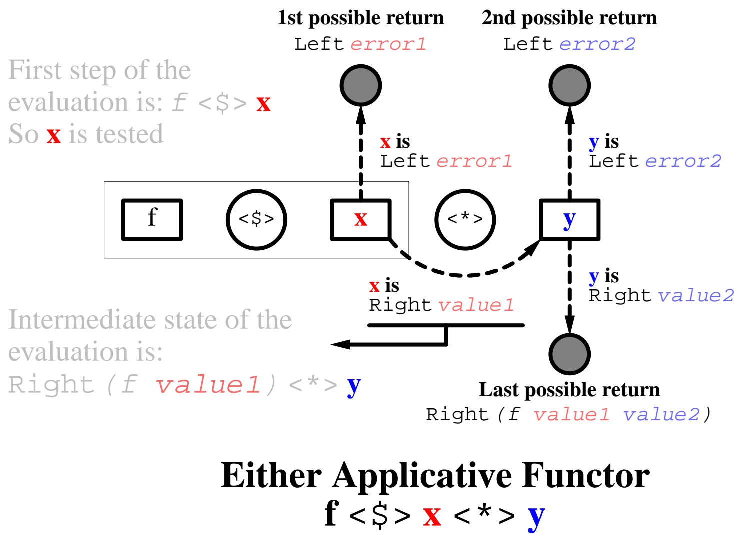
In C, the current function can do anything at any time, even returning unrelated values for example. In Haskell, both three functions have the same return type (implementing operators `<$>` and `<*>`). Thus, errors are handled in a standardized way: error management is written in the data structure code, once.

Implementation example: Maybe



Maybe Applicative functor
f <$> function1 <*> function2

Implementation example: Either



Conclusion on applicative functors

As we saw earlier, data structures implementing `fmap` (to apply a function to the data) are called `functors`. If the applied function requires more than one parameter, it is curried: the data structure now contains a partially applied function. Applicative functors are about providing a new parameter to this function.

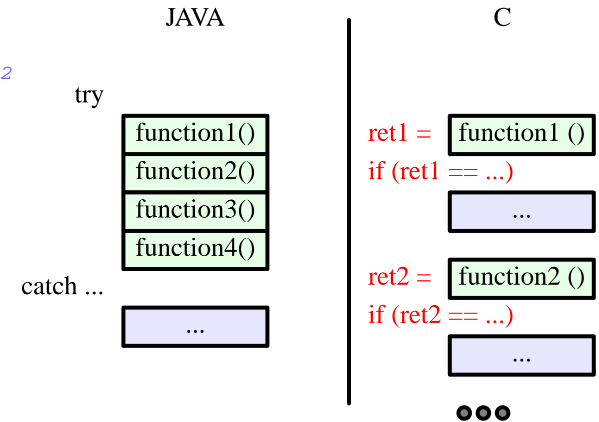
Functors and applicative functors help the developer: both were created because of recurring patterns in applications. Functors allow to manipulate data in data structures, no matter how complex the data structure is. Applicative functors gather all parameters to a function call, allowing to write code almost as simple as `f3(f1(), f2())` (in imperative languages) while still handling errors[13].

Generic and expressive data structures such as `Maybe` and `Either` implement both Functor and Applicative type classes. This allows developers to manipulate elements of these data structures and to chain function calls in a standardized way.

5.5 Monad: binding functions

In functional programming, and particularly in Haskell, a function is a single expression. However, one may want to perform multiple function calls in this single expression. A `Monad` is about binding these function calls together[14].

Binding function calls is like a *try and catch* in other languages. In Java for example, when a function returns an exception, the remaining function calls are ignored and the exception is *catched*. The *try and catch* mechanism allows to write less conditions on the return values of the functions; there is no need to test if they failed and error management is separated from regular instructions.
The following figure shows the difference between C error management and the Java's *try and catch* mechanism.



Try & Catch vs fully imperative error management

Function calls are green boxes, error management are blue boxes. (Bloat) code required in C to check for errors is in red.
The *try and catch* mechanism groups all function calls, error management is elsewhere later in the code. This greatly improves readability with multiple function calls.

Monads are simpler than *try and catch*: they are operators (simple functions) binding function calls[15]. In practice, a Monad is defined by three functions:

- *(>>=)*, also known as the *bind* operator, which computes the first function and give the result to the second (as its last argument);
- *(>>)*, also known as the *then* operator, which computes the first function and drop the result then computes the second;
- *return*, which takes a value and puts it in the context of the monad (as the *pure* function in the `Applicative` type class).

The implementation of these operators depends on the monad.
The following examples, with `Maybe` and `Either` monads, provide the general idea behind monads.

The Maybe Monad

A monad was created from the Maybe data structure. And this can be summarized this way: either there is a value and the next function is called, or the computation stops and returns Nothing.

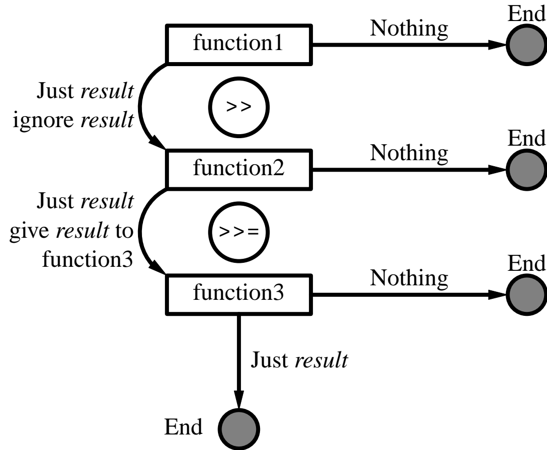
Let's take an example: three functions in the Maybe monad (returning a Maybe value).

13. In Haskell, something like `f3(f1(), f2())` can be written as `f3 <$> f1 <*> f2` which is both safe (errors are handled) and standardized.
14. It is frightening that nobody just says this to explain monads. The mathematical explaination is incredibly useless when talking to developers, stop even trying.
15. Also, monads are more generic than *try and catch* since the behavior can be changed (it depends on the monad) and do not require a compiler-supported mechanism.

```
function1, function2 :: Maybe Int
function3 :: Int -> Maybe Int
expression = function1 >> function2 >>= function3
```

function1 and function2 provide a Maybe Int (they have no parameters) and function3 has a single Int parameter.

The following figure represents the expression function.



Maybe monad: function1 >> function2 >>= function3

First, function1 is called. In case its result is Nothing, the expression stops and returns Nothing. Otherwise, function2 is called. Again, in case its result is Nothing, the expression stops and returns Nothing. Otherwise, the *result* of function2 is provided as argument to the next function. This value isn't in a Maybe structure.

Let's take a few examples with concrete values.

```
function3 x = Just (x+3)
Just 1 >> Just 2 >>= function3
> Just 5
Nothing >> Just 2 >>= function3
> Nothing
```

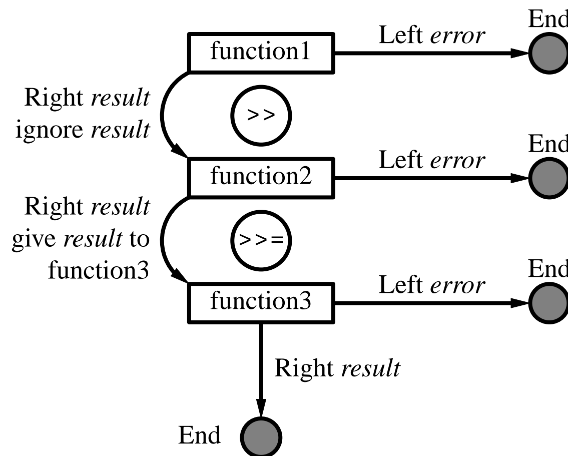
The implementation is fairly simple to guess.

```
instance Monad Maybe where
  (Just x) >>= k     = k x
  Nothing  >>= _     = Nothing
```

The monad type class only requires to implement the *(>>=)* operator. The *(>>)* operator is derived from the previous one, and the *return* function is defined by default as a synonym of the *pure* function in the Applicative type class.

The Maybe monad is very simple and works well to chain filter functions, similar to the shell programs: `cat file | grep value | grep othervalue > result`. However, the Maybe structure is fairly limited: functions cannot indicate an error, which will be fixed with the next monad.

The Either Monad



Either monad: function1 >> function2 >>= function3

Conclusion on monads

5.6 The IO monad

5.6.1 To sort

- Num: numbers.
  Required functions: + * abs signum fromInteger negate
- Eq: types that can be tested for equality.
  Required function: (==)
- Ord: types that can be ordered.
  Required function: compare
- Semigroup: types that can be concatened together (such as lists).
  Required function: (<>)
- Monoid: semigroup with an identity value. An identity value can be an empty list for a list type.
  Required function: mempty

Monad Foldable Read Alternative Show

5.7 Alternative

The Alternative class helps chaining function calls and takes the first valid value returned by these functions.

The definition of the Alternative class:

```
class Applicative f => Alternative f where
  The identity of '<|>'
  empty :: f a

  An associative binary operation
  (<|>) :: f a -> f a -> f a
```

In the Alternative type class, two functions are defined: *empty* and <|>

The instance for the "Maybe" type:

```
instance Alternative Maybe where
  empty = Nothing

  Nothing <|> r = r
  l       <|> _ = l
```

First, our value if nothing is matched: Nothing. Then, either the first parameter is invalid so the second is given, or the first parameter is valid and it is used.

In practice: let's use the module Parsec to parse an URL scheme with this logic:

```
pScheme :: Parser Text
pScheme
  =    string "file"
  <|> string "ftp"
  <|> string "https"
  <|> string "irc"
  <|> string "mailto"
```

In case you were wondering: *string* is a Parsec function testing a string at the current index location of the parsed data. In this case, the function tests for the scheme in the URL.

The main difference between Alternative and a simple *or* statement in an imperative or object-oriented programming language is the type: the semantic[16] is preserved. In an imperative language, this is valid:

```
if True or 1 or some_structure:
    serious(fuckingsly)

# or, in some languages:

val = True or 1 or some_structure
```

Both examples are valid in Python.

5.8 Arrows

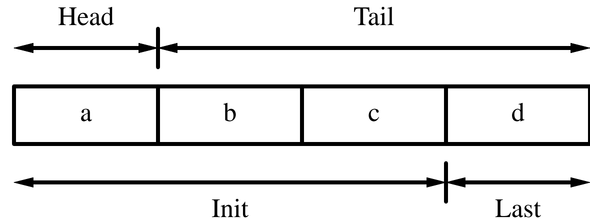
Arrows are another way than monads to express a logical implication between function calls.

16. Ok, maybe not exactly the "semantic" but mostly the type, which is already a big step forward.

6. Useful functions

6.1 show and read

6.2 list functions



6.2.1 Const and (:)

6.2.2 length and null

6.2.3 (!!)

6.2.4 elem

6.2.5 (++)

6.2.6 take and drop

6.2.7 takeWhile and dropWhile

6.2.8 reverse

6.2.9 cycle

6.2.10 repeat

6.2.11 replicate

6.2.12 sum product maximum minimum

6.2.13 map

6.2.14 fold

```
the following produces [], the list identity
foldr (:) []
```

catamorphism is when a function produces an identity given a constructor for a data type. **foldr** is the list catamorphism.

- foldl is an imperative loop
- foldr is a constructor replacement

6.2.15 scan

6.3 Functions on tuples

6.3.1 zip and zipWith

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

6.4 String manipulation

6.4.1 chr and ord

```
chr :: Int  -> Char
ord :: Char -> Int
```

7. Complex data structures

7.1 Generalized Algebraic Data Types

7.2 Phantom type

7.3 Tagless final

7.4 Free monad

8. Lenses: getters and setters on steroids

Lenses help manipulate complex, nested data structures. Let's consider the following types.

```
data Task = Task
  { taskName :: String
  , taskExpectedMinutes :: Int
  , taskCompleteTime :: UTCTime }

data Project = Project
  { projectName :: String
  , projectCurrentTask :: Task
  , projectRemainingTasks :: [Task] }
```

8.1 The problem

How to edit the content of the task (let's say, its name)? Pattern matching is great for simple tasks, but becomes really verbose in nested structures.

To be defined or to finish.

8.2 A first attempt to tackle the problem

Let's define a modifying function.

```
truncateName :: Task -> Task
truncateName task
  = task { taskName = take 15 originalName }
  where
    originalName = taskName task
```

This code is near the verbosity of a imperative (or Object-Oriented) language. This is near JavaScript code. And the code isn't even nested yet: the function becomes even harder to write when the parameter is a `Project` instead of directly getting the `Task`.

8.3 Lenses to the rescue!

Another way to proceed is to create `lenses` functions, helping us to easily reach inner structure values. Lenses come in different flavors:

- lenses for record structures.
  Example: Foo = Foo { x :: Int, y :: Int }
- prisms for sum structures.
  Example: X = A | B | C
- transversal for collections types.
  Examples: Lists, Array, Map, etc.

8.3.1 Actual lenses

Now, given the lens package, let's rewrite our structures.

```
data Task = Task
  { _taskName :: String
  , _taskExpectedMinutes :: Int
  , _taskCompleteTime :: UTCTime }

generate lenses for Task
makeLenses ''Task

data Project = Project
  { _projectName :: String
  , _projectCurrentTask :: Task
  , _projectRemainingTasks :: [Task] }

generate lenses for Project
makeLenses ''Project
```

As shown, Haskell can generate lenses for our records: it only requires each element to start with an underscore. This really is helpful not to have to write many mindless functions.

Our code now has new functions, such as taskName and projectRemainingTasks for example. These functions allow to get and set values. To illustrate, some examples:

```
Let's rewrite our last function.
truncateName :: Task -> Task
truncateName task
  = task { _taskName = take 15 originalName }
  getting originalName is now simpler with
  where originalName = view taskName task
```

view is a function to retrieve a value from a structure through a lens and taskName is the lens function to get (or set) _taskName from a Task type.

In practice, chaining operations will be common, such as getting a value from an inner structure and apply a function to it. To that end, operators are more idiomatic to use than functions. So let's see a few operators.

```
view taskName task
is equivalent to
task ^. taskName
```

And this is convenient for unnamed structures as well:

```
(1,3) ^. _2
> 3
```

_2 is a generic lens to get the second element of a tuple.

Changing a value inside a structure is easy, too. The right value has to be selected, then a function can be applied to it.

```
(1,3) & _2 %~ (+3) modify second element
> (1,6)
```

The & operator allows to prefix the value, such as ^. did for the view function, but this time the value is changed (or only some parts). In this example, & combined to the operator %~ allowed to easily modify part of the value (the second element of the tuple). The operator .~ takes the new value, %~ takes a function to modify the selected value.

Handling complex structures is one of the few parts of FP where introducing operators is essential to keep the code clean. Lenses introduce an overwhelming amount of operators[17], only a very few are necessary for a start and for most operations. Chaining operations is easy, and the more operations there is, the simpler it is compared to other programmation paradigms.

```
(1,3)
  & _1 %~ (+2) modify first element
  & _2 %~ (+3) modify second element
> (3,6)
```

Operations can be chained, easily.

Now, let's try nested structures. Important note: lenses have to be composed in reverse order from the nesting.

```
("hello", (1,3))
  & (_2._1) %~ (+2)     1 -> 3
  & (_2._2) %~ (+3)     3 -> 6
  &  _1     %~ (++ " world!")
> ("hello world!", (3,6))
```

Again, operations can be easily chained.

17. The lens module currently has 109 operators. Apparently there is never enough! Again, no need to feel overwhelm, most of it is very specialized and you won't need it. Also, a lot of them are related to each other, there is a logic behind it.

8.3.2 Prisms: lenses for sum types

8.3.3 Transversals: lenses for collections

Transversals help browse all elements of a collection, such as lists, arrays and maps. It sounds a lot like a `fmap` function, but there are a couple of differences.

0.0.0.8. At

Get and set values at a given index.

```
Map.fromList [(1,"world")] ^.at 1
> Just "world"

iat: 'at' with an index
Map.fromList [(1,"world")] ^@. iat 1
> (1,Just "world")

at 1 ?~ "hello" $ Map.empty
> fromList [(1,"hello")]

reverse order, easier to chain stuff! ;)
Map.empty
  & at 1 ?~ "hello"
  & at 2 ?~ "world!"
> fromList [(1,"hello"), (2,"world!")]
```

0.0.0.9. Contains

Lens to test for an index in a container (such as a set).

```
IntSet.fromList [1,2,3,4] ^. contains 3
> True

IntSet.fromList [1,2,3,4] & contains 3 .~ False
> fromList [1,2,4]

icontains: 'contains' with an index
IntSet.fromList [1,2,3,4] ^@. icontains 3
> (3,True)
```

8.3.4 Logic behind operators

This document only shown a few operators out of more than a hundred in the `lens` package. This may seem like this completely went out of control, but there is a logic behind all that. Here is a little recap.

Operator	Meaning
-	
^.	infix view
containing %	usually take a function
ending with ~	over (%~) and set (.~)
containing =	like operators ending with ~ but working with a `State` monad
containing @	result contains a value and an index

Vague categorization of lens operators.

Function	Operator	Meaning	Use
view	^.	Getter	view `lens structure` view `_1 (1,2,3)` `(1,2,3) ^. _1`
set	.~	Setter	set `lens` value `structure` set `_1` 1 `(0,2,3)` `(0,2,3) & _1 .~ 1`
over		Get or Set on collections	over `lens` f `structure` over `mapped` (+3) `[1..5]` over `(traverse._1)` (+1) `[(0,2),(3,4)]`

Basic functions on lenses (and their friends: prisms, transversals, etc.).

9. Networking

10. Profiling

11. Performances

Haskell can be in the same ballpark than C regarding computational speed. Good performances mostly come from:

- good algorithms;
- the *-O2* option to the ghc compiler;
- efficient data structures and types;
- strictness and laziness where they make sense;
  (the compiler may do that for you sometimes, with *strictness analysis*)
- tail recursion elimination;
- careful function inlining;
- (once everything else is done) parallelism or concurrency.

Since refactoring Haskell code is considerably easier than most languages, one could write a naive but valid implementation as a start then make incremental changes to

make it efficient. A naive implementation can be 100 times slower than an optimized one. However, writing this valid-but-slow solution is really easy given laziness, very generic functions, etc. There is a trade-off between code optimization and the time you have to write the implementation. Optimal code is hard to reach, but *good enough* is easy.

11.1 newtype

newtype, type and data

To be defined or to finish.

11.2 Memoization

Memoization is a trade-off between memory and computation, and may transform naive implementations of some recursive algorithms into legitimate solutions. The idea is simple: keep the result of pure function calls, so the computation only once for a given set of parameters. This works with all pure functions. Memoization can offer recursive algorithms a massive performance boost. **To be defined or to finish.**

11.3 Laziness

Laziness

- can make qualitative improvements to performance
- can hurt performance in some cases.
  It implies to keep track of what should or shouldn't be executed.
- Makes code simpler.
- Makes hard problems conceivable
- Allows for separation of concerns with regard to generating and processing data.

11.4 Tail recursion elimination

In the general case, a function call is stacked in memory. The *stack* is very limited in space, trying to put too much function calls and there will be a stack overflow. Furthermore, putting and removing functions calls from the stack has a time cost. This is enough to be noticed when a recursive function call itself thousands of times.

The following example shows a naive implementation of computing the length of a list.

```
len [] = 0
len (x:xs) = 1 + len xs
```

This function produces the following stack for a list with three elements.

len [] = 0
len (x:xs) = 1 + *len* xs
len (x:xs) = 1 + *len* xs
len (x:xs) = 1 + *len* xs

call stack for a 3-element list

There are several steps to optimize this function.

- First, we need to remove the need for multiple stack frames of the same function. This is done by using an accumulator parameter to remove the tail recursion.
- Then, the accumulator should be strict. Otherwise it creates *thunks*, and this also builds a stack that will end-up in overflow.

First step

Tail recursion elimination (or *tail recursion optimization*) is removing the need to stack a new function call each time. This reduces both time and space costs of putting and removing a function call in the stack.

```
len' [] acc = acc
len' (x:xs) acc = len' xs (1 + acc)
```

This time, the increment is a function passed to the next recursion. This function is lazy and won't be computed right away, this isn't efficient and may create a stack overflow. The compiler should be noticed that the accumulator is *strict*.

Rewrite the function with a strict accumulator

One way to make sure the accumulator is strict, is to use the `$!` operator, which forces a strict evaluation to its right component.

```
len' [] acc = acc
len' (x:xs) acc = len' xs $! (1 + acc)
```

The accumulator is strict, no thunk will be generated and managed, this removes the stack overflow.

Another way, is to use the `BangPattern` extension to explicitely say the accumulator is strict.

```
{-# LANGUAGE BangPatterns #-}
len' [] acc = acc
len' (x:xs) !acc = len' xs (1 + acc)
```

The extension `BangPattern` has to be enabled, then the only change in the code is the exclamation point on the accumulator. Flag parameters that are strict is a good practice. When compiling with *-O2* the compiler can find out some of the actual strict parameters.

A better way

Our *len* function follows the *fold* pattern, walking a full list and returning a single value. So, let's use it, it even has a strict variant[18]!

```
import Data.List
len = foldl' (\acc _ -> 1 + acc) 0
```

Reusing existing functions is simpler, less error-prone and actually more efficient.

Experiment

Several implementations of the *len* function were described: the *naive* without any kind of optimization, the one introducing an *accumulator* but without explicit strictness (neither the $! operator nor the bang pattern), the implementation with *bang* patterns, and finally the one with *foldl'*. Let's see how they compare!

optimization	naive	accumulator	bang	foldl'
nothing	4358	5286	709	733
-O2	878	396	395	254

Values in milliseconds, average over 100 runs. Standard deviation for these values varies from 0.1 to 0.4 %. The list has 10 millions entries. This was performed on an Alpine Linux server. On the same server, a C program only incrementing a value 10 million times spends 42 ms on the computation.

The table above shows several interesting things:

- one of the optimizations of the *-O2* compiler option is *strictness analysis* which gave the same speed to the accumulator function and the explicitely strict version (*bang*);
- (follow-up) since strictness analysis works, there is no point trying to spot every parameter that should be strict, (almost) naive implementations can still have great performances;
- the program is more than twice faster without tail recursion;
- massive performance gain with the *-O2* option (computation can be 13 times quicker), even using optimized versions with strictness (almost 3 times);
- standard functions are so darn fast, use them.

18. Remember, conventionaly a function with an apostrophe is a strict variant in the standard Haskell API.

11.5 Inlining

To be defined or to finish.

Inlining a function

To be defined or to finish.

Prevent inlining

To be defined or to finish.

11.6 Avoiding space leaks

12. Parallelism and Concurrency

13. Advanced concepts

Some concepts are derived from previous properties of the language. The following explores a few concepts that are useful in some contexts.

13.1 Continuation: a nice side-effect of composition

`WARNING: the following is a draft. NOTHING IS FINISHED HERE.`

You say what comes next, specially when there is a fail. The side-effect is that no exception system is required[19].

```
safeDivide :: Num a => a -> a -> Maybe a
safeDivide _ 0 = Nothing
safeDivide x y = Just $ divide x y
```

Continuation Passing Style vs Monads

`WARNING: the following is a draft. NOTHING IS FINISHED HERE.`

Continuation Passing Style and Monads are actually similar. In both cases, the idea is to control the behavior of a function, whether it works or fails.

Continuation Passing Style is more direct, it is a by-product of high order functions. Monads are a way to ensure the same behavior for multiple function calls, that's a shared, recognizable pattern.

19. Conferences about continuation by Scott Wlaschin are great learning resources.

13.2 Arrow operator

13.3 Closure

WARNING: the following is a draft. NOTHING IS FINISHED HERE.

13.4 Free monads

WARNING: the following is a draft. NOTHING IS FINISHED HERE.
First, a *Free Foo* happens to be the simplest thing that satisfies all of the *Foo* laws. It satisfies exactly the laws necessary to be a Foo and nothing extra.

A Free Monad is a way to create a Monad from any Functor.

Let's say we have a functor f.

```
liftFree :: Functor f => f a -> Free f a
foldFree :: Functor f => (f r -> r) -> Free f r -> r
```

The first function put your functor f into a Free monad. The second function gets the value from your functor inside the Free monad.

13.5 Dependency injection (OOP) vs Continuation Passing Style

Dependency injection is an OOP concept. The general idea: you have two classes, A and B, and B needs an instance of A to work. B depends on A. There are two ways to handle this: either B creates its own instance of A, or it is provided to B in some way (during the construction of B or later).

Dependency injection mostly allows to provide an encapsulated state to B. The behavior is changed if the provided instance is a subclass of A with a different implementation.

In any case, dependency injection offers a limited control over the behavior of the functions (methods).

To be defined or to finish.

Stolen from https://danidiaz.medium.com/free-monads-and-effect-handlers-vs-dependency-injection-bca2eb95e580

Suppose you have a piece of business logic which uses various high-level interfaces. You don't want the business logic to care about how the interfaces are implemented, or where to get hold of the implementations.

If you are a functional programmer, you might resort to a free monad, or to some form of effect handlers. If you are an object-oriented programmer, you turn to dependency injection.

So, are these aproaches basically the same in the end?

I see a couple of differences:

- With free monads/effect handlers, the computation cedes more control to the interpreter than it would cede to the dependency injector. Consider errors for example. If you have an interface for database access injected into your object, and you call some operation on it, you can catch any exception the operation throws. With a free monad, you are at the mercy of the interpreter, which may choose to terminate the computation right away, force a retry, etc. (Quote from Reddit user: "DI doesn't allow controlling the continuation".)
- With free monads/effect handlers, you can pass around and manipulate at runtime values representing abstract computations not yet tied to any interpreter. This doesn't seem to be the case with dependency injection. With dependency injection, you must provide the implementations as you construct your enterprise beans; only afterwards you can pass those beans around.
- With free monads/effect handlers, the interfaces required by the business logic are reflected in the type signature. With dependency injection, sometimes you have to stoop down to inspect a bean's internal attributes.

13.6 S-Expression (symbolic expression)

WARNING: the following is a draft. NOTHING IS FINISHED HERE.

A symbolic expression is a convention to represent data. The core property is the prefixed notation with parenthesis. Example:

```
(+ 1 2)

(defun hello-function ()
  (print "hello world!"))
```

13.7 Homoiconicity

WARNING: the following is a draft. NOTHING IS FINISHED HERE.

Homoiconicity is "code as data".

Said otherwise, code is represented as a series of primitive structures of the language. For example, in Lisp, the code is a series of nested lists. This allows very powerful metaprogrammation: developers can write code that modifies the code (maybe even at runtime) in a very concise way. All thanks to the use of primitive structures to represent the code itself.

Example in LISP:

```
(defmacro print-parameters (f)
    `(print (cdr '(,@f))))
(print-parameters (+ 1 2))
; (1 2)
```

13.8 Futures, Monads, Reactive Programming and Functors

14. To the point

Pure and simple functions (such as addition) are great: they are independant from the rest of the code. They always have the same set of parameters, and work always as expected. But, our program is way more complicated than that. Sometimes, an error occurs and we have to deal with it. We still want our application to be seen as a simple and beautiful mathematical expression. To that end, we need to deal with errors and conditional computations in a way that can be semantically understood by a human. We need to communicate the intent, not just say what must be done.

I'll describe a few cases and a good way to handle them in Haskell.

14.1 Error cases

14.1.1 Pure function, single parameter and potential erroneous parameter.

Let's see a first example of error management with the `Maybe` module. We create a function *foo* which takes a parameter. The parameter is a `Maybe Int`, it can be either a `Nothing` (there is no value) or a `Just Int` We want to perform a computation our single parameter only in case it is valid. Otherwise, we should return `Nothing.`

Here is a first way (but too verbose):

```
foo :: Maybe Int -> Maybe Int
foo Nothing = Nothing
foo (Maybe x) = Just (x + 2)
```

We use pattern matching on the parameter, then apply the function (+2) on the Int contained in the Maybe Int.

A better way:

```
foo :: Maybe Int -> Maybe Int
foo f = (+2) <$> f
```

No more pattern matching, the function (+2) is applied to our functor *f* only when it makes sense: when it is not *Nothing*. The operator `<$>` is an infix version of *fmap*.

Our second example is based on the `<$>` operator (which is an infix version of *fmap*). This operator is well-known and designed for this purpose: apply when semantically valid[20].

The semantic of the `<$>` operator depends on the functor. The `Maybe` functor applies the function to the content only when there is one. The `Either` functor applies the function to the content if it is a valid value, not an error. Either can be a `Right a` or a `Left b` (let's say for example a `Left String` with the string being an error message), in which case the function isn't applied.

Our last function is less verbose than the first version. And since the `<$>` operator is generic and works for many functors, the type of our function can be improved to be more generic too. An even better version:

```
foo :: Functor f => f Int -> f Int
foo f = (+2) <$> f
```

We only changed the type of the function. Compare this to what you would have written in an imperative language.

In this last version, our function is valid for many functors[21].

15. Notes: quick and dirty

A list of the most used operators:

```
($)   ::                (a -> b) ->   a ->   b
(<$>) :: Functor f =>   (a -> b) -> f a -> f b
(<*>) :: Functor f => f (a -> b) -> f a -> f b
```

And how to use them[22]:

20. One may refer to the *fmap* function is a *lift* function. It takes a pure function `(a -> b)` and lift it to be valid in a functor context `(f a -> f b)`
21. This is a bit like Java interfaces on steroids.
22. Remember: *Maybe* is a functor, and parameters *a* and *b* are simple types, such as *Integer*.

```
fun :: Maybe (Int -> Int)
fun = (*) <$> Just 3

app :: Maybe Int
app = fun <*> Just 3
```

The function *fun* provides the value *3* to the function `(*)` and puts it in a `Maybe`. The result is `Just (2*)` and the function in the Maybe functor isn't complete. The function *app* takes `Just (2*)` and provides another parameter to this function contained in a functor. Operator `<*>` is from the `Applicative` functor, and helps chaining (potentially faulty) parameters to an "unfinished" function in a functor (Maybe, Either, etc.). As with the operator If the functor is a Maybe and In case of a problem with one of the parameters, the whole expression will be replaced by `Nothing` in a Maybe context for example.

In an imperative programming language, avoiding a function call in case of a previous faulty computation can be done in (mainly) two ways. Either the language implements some sort of "try and catch" mechanism (Java, Python, Zig, C++, etc.), or the developer has to check each value by hand (C).

16. Pure theory

16.1 Covariant functor

A **covariant functor** is a functor with its first argument being a function like this: `(a -> b)` followed by a functor `f a` and returning a functor `f b` We can see the types of the function (from *a* to *b*) and the second parameter is *f a* meaning that we will apply the function to the content of *f* (a functor containing a value of type *a*) and this will result in another functor containing a value of type *b*. This is covariant since the function given in parameter is from *a* to *b* and we have to apply it to the content of the second parameter (a functor of type *a*) to get our result. Types are in the same direction.

In Haskell, the definition of a functor is as follow:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

The *fmap* function is covariant, as we described. The function `a -> b` is transformed into a function `f a -> f b` , and we say that the function is lifted into *f*[23].

16.2 Contravariant functor

A contravariant functor is defined as a covariant functor, but the function it takes as a first argument is from *b* to *a*.

Whereas in Haskell, one can think of a `Functor` as containing or producing values, a contravariant functor is a functor that can be thought of as consuming values[24].

In Haskell, the `Data.Functor.Contravariant` module includes this definition of a Contravariant functor:

```
class Contravariant f where
  contramap :: (b -> a) -> f a -> f b
```

As we can see, the order of the types for the first parameter are inversed compared to a covariant functor.

16.3 Bifunctor

A Bifunctor is a functor containing two functors. **To be defined or to finish.**

16.4 Profunctor

Profunctor is a bifunctor with a **contravariant** as its first argument and a **covariant** as its second.

17. Lenses: manipulate nested structures easily

This section is roughly a summary of some books and web pages (Marick, 2018). **To be defined or to finish.**

23. There are a lot of functors classes: `Applicative`, `Monad`, `Bifunctor`, etc. **To be defined or to finish.**

24. This is a citation from the Haskell documentation on `Data.Functor.Contravariant` and it deserves a read.

Optic	Meaning	Where it is used
Lens' s a	The type s contains a value of type a.	Product types like records and tuples.
Prism' s a	The type s contains zero or one value of type a, and a is sufficient to produce an s.	Sum types like Maybe and Either.
Traversal' s a	The type s contains zero, one, or many values of type a.	Collections like arrays, maps, and any other member of the Traversable type class. They are also a more general form of lenses and prisms; traversals which focus on at most one element (like lenses, prisms, and their composition) are called affine traversals.
Iso' s a	The types s and a are isomorphic if an s is sufficient to produce an a and vice versa.	Newtypes, interchangeable structures like Array and List, and any other pair of types which can be converted back and forth without losing information.

18. See also

Here is a list of things that got my attention and may be useful to you, too.

18.1 A few FP languages worth mentionning

- Haskell: FP without annoying parenthesis, a lot of modules available, purity, conciseness, laziness, great compiler debugging capabilities, etc.
- PureScript: (mostly) Haskell for the web, without the laziness. Syntax and concepts are the same, base modules are mostly identical, etc. PureScript also provides a very simple way of interacting with JavaScript.
- carp: statically typed lisp, no garbage collector and focused on performances. Great for games, video and audio applications, etc.

18.2 Modules

Here are some interesting modules, either for wide adoption in any kind of applications or in specific contexts. Some already are included in base Haskell distributions.

What you need to know to use most of these modules: Functors, Applicatives, Monads.

- Data.ByteString: a replacement for `String` focused on performances and binary representations. It comes in two varieties: strict and lazy. **To be defined or to finish.**
- Data.Text: another replacement for `String` but focused on Unicode text (contrary to `String` which accepts any `Char` input). It comes in two varieties: strict and lazy, such as `ByteString.`
- prettyprinter: a simple way to create pretty outputs for your types. **To be defined or to finish.**
- MegaParsec: a library to create compilers. There are many other libraries like this one, but this is a nice balance between functionalities, performances and simplicity.
- PyF: a library to format strings, as the f operator in Python.

18.3 Books, website and tutorials

For absolute beginners:

- Learn you a Haskell for Greater Good (Lipovača, 2011)! Good book about Haskell, for beginners. There are a few examples to easily understand functions like zip, zipWith, sort, etc. And the book presents a good part of what's actually in this document in a little more verbose way.

Once you understand concepts presented in this document:

- Haskell wiki and its TypeClassOPedia which helps understand type classes and how to use them. There are many examples, great source to learn.
- Lenses for the mere Mortal (Marick, 2018) Great learning resource on lenses, with many explanations and examples.

- Nokomprendo: nokomprendo.gitlab.io, great tutorials on Haskell, in French. The author also has a youtube channel I recommand.

19. Misc

This section is a way to provide some unordered informations.

19.1 Kinds

A simple value (such as a number) and a polymorphic type without specified type parameters are not the same kind of data. In the first case, a value is directly usable, in the second this is a way to create a value. Kinds were introduced to mark this difference, and a *kind* can be seen as a type of types.

The Kind `*` means it is an actual, directly usable type. The Kind `* -> *` means the type requires a parameter. Here is a list of examples:

```
Int :: *                    directly usable
Maybe :: * -> *             needs an argument
Maybe Bool :: *             directly usable
Either :: * -> * -> *       takes 2 arguments
a -> a :: *                 function: directly usable
[] :: * -> *                empty list: lacks a type
[Int] :: *                  list of integers: usable
(->) :: * -> * -> *         arrow operator needs 2 types
```

Kinds of types can be verified with ghci by typing `:kind` then the type to check. Example: `:kind Maybe`

To be defined or to finish.

A simple note...

Most of what was presented here isn't part of the language, but only the standard library. Function composition rests on the operator `(.)` which only is a simple function in the standard library. Same thing for (->) operator. **To be defined or to finish.**

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

20. Annex: code deduplication (OOP < type classes)

Avoiding duplication is a noble but difficult exercise. Let's see how this works in C, C++ and Haskell with some examples.

20.1 Trivial example

Let's say we have a function with a structure that can satisfy several types of arguments. Example, in pseudo code:

```
function sum (a, b) {
  return a + b
}
```

In C, functions have arguments with a defined type: no abstraction, whatsoever. A function taking an integer won't work with anything else. To make this a bit more generic, macros can generate functions replacing the types. This is not, by any mean, a good abstraction, but a quick and dirty hack to avoid to write several times the same function.

Why is this a problem? Since there isn't abstraction, functions have to be duplicated in the code. At the end of the day, the function will be duplicated in the final executable binary. But C compilers cannot figure out themselves how to create this function for other types, the developer has to take care.

In C++, *classes* and *inheritance* allow some deduplication. The function takes a parameter with a class implementing the right *method* (in our case, the *plus* function)[25].

```
class Sumable {
public:
  int value;
  int sum (Sumable x) {
    return value + x.value;
  }
};
```

In this example, the function *sum* doesn't require a particular type. Actual object passed to the *sum* function is an instance of the class `Sumable` or from a class that inherits from it[26].

In Haskell, this really is trivial.

25. The class can be *abstract* or an *interface*, in which case the actual implementation comes from another class that inherits from this one. Naming conventions vary depending on the language, but the idea is more or less the same.

26. OOP requires *multiple inheritance* (C++) or *interfaces* (Java), otherwise classes would need to duplicate a lot of code.

```
type Value = Float

class Sumable a where
  sum :: a -> a -> a

instance Sumable Value where
  sum = +
```

The actual value is just a type synonym for a floating-point number. The type isn't repeated anywhere in the code, refactoring is easier this way. Then the type class `Sumable` is declared and is composed of a single function *sum*. This function takes two parameters and returns a value, all of the same type (whatever the actual type is). And finally, the implementation is a synonym of the + function.

20.2 Less trivial code

The problem: represent both a `car` and a `boat`. Both vehicles can start and stop their engine, and have a `radio` to play some music.

First, let's forget about C, this would be very long and uninteresting to write the actual code. Code deduplication is a massive failure in C, no need to spend more time on it.

In C++, the code can be written this way.

```
class Vehicle {
  int running = 0; // the engine's state
public:
  void startEngine() { running=1; }
  void stopEngine()  { running=0; }
};

class Radio {
  int running = 0; // the radio's state
public:
  void playRadio() { running=1; }
  void stopRadio() { running=0; }
};

class Boat: public Vehicle, public Radio {
};

class Car: public Vehicle, public Radio {
};
```

Both a car and a boat can start their engine and play radio. But this isn't possible to create a function that take either a boat or a car and start their engine and play radio. To do that, either the function has to be duplicated in both `Car` and `Boat` classes, or a new class has to be implemented. Since functions cannot freely ask for multiple criteria, a lot of code has to be in a class and thought in everything has to be an object, even simple things.

In Haskell, again, this is trivial code.

```
data State = On | Off
type Engine = State
type Radio  = State
data Car  = Car  Engine Radio
data Boat = Boat Engine Radio
data Vehicle = Car | Boat

class OwnEngine where
  startEngine :: a -> a
  stopEngine :: a -> a

class OwnRadio where
  startRadio :: a -> a
  stopRadio :: a -> a

instance OwnEngine Car where
  startEngine (Car _ r) = Car On r
  stopEngine  (Car _ r) = Car Off r

instance OwnEngine Boat where
  startEngine (Boat _ r) = Boat On r
  stopEngine  (Boat _ r) = Boat Off r

instance OwnRadio Car where
  startRadio (Car e _) = Car e On
  stopRadio  (Car e _) = Car e Off

instance OwnRadio Boat where
  startRadio (Boat e _) = Boat e On
  stopRadio  (Boat e _) = Boat e Off

goToWork :: OwnRadio a, OwnEngine a => a -> a
goToWork = startRadio . startEngine
```

What a verbose way of doing things.

To be defined or to finish.

```
data State = On | Off
type Engine = State
type Radio  = State
data Car  = Car  { _carEngine  :: Engine, _carRadio  :: Rad
data Boat = Boat { _boatEngine :: Engine, _boatRadio :: Rad
data Vehicle = Car | Boat

class OwnEngine where
```

Lang	Code deduplication	Problem
C	macros	no abstraction
C++	Objects, multiple inheritence	single criterion in functions
Java	Objects, interfaces	single criterion in functions
Haskell	type classes	about none

20.3 Conclusion

C is an abstraction over assembly, nothing more. This has its use: C developers can make an educated guess how the code will be compiled into assembly. However, code is constantly duplicated, and the tools at our disposal to make it more generic are flawed on many points (static code analysis, type verifications, etc.).

OOP allows some code deduplication. A function can ask for its parameter to comply with a list of methods (a class). However, a function cannot ask for many more criteria, which forces to create new classes or interfaces. The code is more complex than it has to be[27].

FP The language allows to write code without caring for the

27. Actually, OOP is broken beyond repair, never have been a great solution and never will be, but this deserves its own document.

21. Annex: a bunch of type classes

Here is a list of the most used type classes.

Type class	Meaning	Provided methods
Num	Numbers	+ - * / etc.
Eq	Types that can be tested for equality	eq
Ord	Types that can be ordered	sort, max, etc
Semigroup	Types that can be concatened	(<>)
Monoid	Types with a default value on a an operation Examples: (Num, 0, +), or (Num, 1, *)	mempty, mconcat
Functor	Types containing data on which we can apply functions Examples: Array, List, Maybe, Either	*fmap* (<$>)
Applicative	Add a parameter to a function in a Functor	(<*>)
Alternative	Can be different values Example: `getUser x = searchLocalUser x <|> searchRemoteUser x`	(<|>)

Monad	Chaining function calls
	Example:
	`putStrLn "Your name?" >> getLine >>= (\name -> putStrLn ("Your name: " <> name))`
	Same example but with the *do* notation:
	`do putStrLn "Your name?"`
	`   name <- getLine`
	`   putStrLn ("Your name: " <> name)`
		bind (>>=) and *then* (>>)

22. TODO

```
(+) <$ Just 5 <* Nothing <*> Just 4 <*> Just 3
```

23. Annex: vocabulary

Browsing through different documentations of the Haskell language (and FP in general) can be really hard. The technical corpus one must know prior to the reading of some of the explanations is almost preposterous[28]. So, in order to tackle this problem, here is a list of technical terms explained in a simple way.

- Class: category of types.
  Example: the class `Show` represents all types that can be printed in the terminal.

- Constructor: keyword to create a structure.
  Example: `True` for a `Bool.`

- Instance: function implementation for a real type.
  Example:

```
instance  Functor Maybe  where
   fmap _ Nothing    = Nothing
   fmap f (Just a)   = Just (f a)
```

28. And sometimes you get pure garbage descriptions. What's an applicative functor? According to the Haskell wiki: "An applicative functor has more structure than a functor but less than a monad.". Seriously, is this trolling at an academic level?

- Functor: structure implementing the `fmap` function, to change its inner value(s).
  Example:

```
fmap :: Functor f => (a -> b) -> f a -> f b

Previous code snippet shows an instance of fmap.
Usage example (Maybe functor):
fmap (+2) Just 2
> Just 4

Usage example (Either functor):
fmap (+2) Right 2
> Right 4
```

- Applicative Functor: structure implementing the `<*>` operator, allowing to pass arguments to a function in a functor.
  Example:

```
<*> :: Applicative f => f (a -> b) -> f a -> f b

Usage example (Maybe applicative functor):
Just (+2) <*> Just 2
> Just 4

Usage example (Either applicative functor):
Right (+2) <*> Right 2
> Right 4
```

- Monad: structure implementing the `>>=` operator, allowing to bind function calls together. Said otherwise: the infix operator `>>=` takes a monad (containing a value of type `a`) and a function taking a value of type `a` and returning another monad (containing a value of type `b`).
  Example:

```
>>= :: Monad m => m a -> (a -> m b) -> m b

Usage example (Maybe monad):
Just 2 >>= \x -> Just (2+x)
> Just 4

Usage example (Either monad):
Right 2 >>= \x -> Right (2+x)
> Right 4
```

- Variadic: undefined number of something (mostly used to describe undefined number of parameters for a function). Functions can have an undefined number of parameters in C, LISP and many other languages, not in Haskell. **To be defined or to finish.** (TODO: maybe reword a bit)

- Prelude: standard library, available by default in every distribution of Haskell. Even included by default in any Haskell code without any *import* statement.

- Recursivity: the definition of something (a function or a structure) includes itself.
  Example:

```
data List = Element Int List | Void
```

- Referential transparency and purity: same parameters leads to same result. A function will always provide the same result given a set of parameters.
  Example: `1 + 1` always returns 2.
  These functions only work on their parameters and have no side effects, they are called *pure*. On the contrary, when a function requires side effects (through networking, printing something in the terminal or getting an input from somewhere), the function isn't pure and its result cannot be known from a previous call.

- Laziness: compute a value only when necessary. As a side effect, infinite lists are valid in Haskell. An infinite list can be declared and used, unless the code tries to get all its values, they won't be computed.
  Example:

```
taking 5 elements of an infinite list
take 5 $ [1..]
provides: 1, 2, 3, 4, 5
```

- High order function: treat functions as values.
  Example:

```
apply :: (a -> b) -> a -> b
apply function value = function value
```

- Predicate: function (or expression) whose range consists of truth values. See guards for example, they are used to chose the right function body.

- Coroutine: function that can be interrupted by *yielding* (providing a value or not) then resumed where it stopped.

- Closure: **To be defined or to finish.**

- Continuation Passing Style (CPS): to add to each function an extra *continuation* argument, a function to call to continue the program instead of simply returning a value. **To be defined or to finish.**

- Continuation: functions take an extra parameter corresponding to the following function to execute, the rest of the application. Thus, functions do not really end, they jump to the next function, it is a goto. **To be defined or to finish.** (why this can be useful)

- Applicative order: execute code within inner list first. Example:

```
x = f (g)
```

In an *Applicative* order, g is executed first. This is the case in most programming languages, and in LISP by default.

- Normal order: evaluate an expression only when needed. Example:

```
x = f (g)
```

In a *Normal* order, g is executed only if required by f. It is *lazy* and it is the way Haskell works.

- Comprehension list: a way to create lists. Example:

```
create a list from 1 to infinity
[1..]

create unique pairs of values
from (1,1) to (10,10)
[(x,y) | x <- [1..10], y <- [x..10]]
```

- Point free: writing a function without explicit parameters. Example:

```
add1 = + 1              no explicit integer param
h = reverse . sort      no explicit list param
```

- Eta conversion: making a function either more abstract (eta expansion) or less abstract (eta reduction). Example:

```
\x -> abs x   more abstract
abs           less abstract
```

From the first to the second notation: *eta reduction*. From the second to the first notation: *eta expansion*.

- Lambda lifting: taking an inner function (in the *where* part of another function) and making it top-level. Example:

```
from
f x y = g
  where g = x + y
to
f x y = g x y
g x y = x + y
```

- Free variable: when a value doesn't come from the context of the function. A free variable is neither passed as parameter or computed within the function. Example:

```
f x y = g
  where g = x + y
```

Function *g* contains two free variables: *x* and *y*. These values aren't parameters of the *g* function, they're from the function *f*.

- Free expression: an expression in which every variable is a Free variable.

- Thunk: unevaluated code in a non-strict environment.

```
snd (undefined, 2)
> 2
```

First element isn't evaluated since it isn't required. The value could have been the result of a very long and complex code, but since it isn't used, the code behind this value isn't evaluated. When the expression `snd (undefined, 2)` is evaluated, the *snd* function takes the second element of the tuple, so the tuple itself is being evaluated, and the first element is dropped since it won't be useful later in the code.

- Normal form: fully evaluated value.

- Weak head normal form: partially evaluated expression. Any intermediate evaluation between a thunk and a normal form.

- Persistence: when a value is updated, older versions are still there, the update isn't *in place*.

- Amortization: distribute unequal running times across a sequence of operations. **To be defined or to finish.**

- Bottom (⊥): a computation which never completes successfully. This notation isn't used in Haskell code, but rather in documentation to explain the behavior of a function. The code equivalent of ⊥ is **undefined**.
  Example: the function *zip* takes two lists as parameters. When its left list is empty, the function works even if the right list can't be computed. However, *zip* crashes when its left list can't be computed. This can be said in the documentation this way:

```
remember: this is documentation, not code
zip [] ⊥ = []
zip ⊥ [] = error
```

References

Brian Marick, *Lenses for the Mere Mortal,* Lean Publishing (2018).

Miran Lipovača, *Learn You a Haskell for Great Good!* No Starch Press (2011).