# Paradigmas de Programação

## Week 8
## Monads[1]

Alexandra Mendes

[1] These slides use the information presented in `https://wiki.haskell.org/All_About_Monads`.

## What is a Monad?

- A way to structure computations in terms of values and sequences of computations using those values.
- Allow the programmer to build up computations using sequential building blocks, which can themselves be sequences of computations, without having to code the combination manually each time it is needed.
- It is useful to think of a monad as a strategy for combining computations into more complex computations.

## What is a Monad?

### Maybe Type

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

represents computations which may fail to return a result.

Suggests a strategy for combining computations:

- Combining two computations $A$ and $B$, with $B$ depending on the result of computation $A$;
- The combined computation should yield `Nothing` whenever either $A$ or $B$ yield Nothing;
- The combined computation should yield the result of $B$ applied to the result of $A$ when both computations succeed.

## Why should I make the effort to understand monads?

Useful tools for structuring functional programs. Three properties:

- **Modularity:** Allow computations to be composed from simpler computations and separate the combination strategy from the actual computations being performed.
- **Flexibility:** Allow functional programs to be much more adaptable than equivalent programs written without monads. This is because the monad distills the computational strategy into a single place instead of requiring it be distributed throughout the entire program.
- **Isolation:** Can be used to create imperative-style computational structures which remain safely isolated from the main body of the functional program. This is useful for incorporating side-effects and state into a pure functional language.

## Physical Analogy: Assembly Line

- Think of a Haskell program as a conveyor belt: input goes on one end of the conveyor belt and is carried to a succession of work areas.
- At each work area, some operation is performed on the item on the conveyor belt and the result is carried by the conveyor belt to the next work area.
- Finally, the conveyor belt carries the final product to the end of the assembly line to be output.

In this assembly line model, our physical monad is a system of machines that controls how successive work areas on the assembly line combine their functionality to create the overall product.

## Physical Analogy: Assembly Line

Our monad consists of three parts:

- **Trays** that hold work products as they move along the conveyor belt.
- **Loader** machines that can put any object into a tray.
- **Combiner** machines that can take a tray with an object and produce a tray with a new object.
- Combiner machines are attached to **worker** machines that actualy produce the new objects.

## Physical Analogy: Assembly Line

How we use the monad to set up the assembly line:

- Loader machine put materials into trays at the beginning of the assembly line.
- The conveyor belt then carries these trays to each work area, where a combiner machine takes the tray and may decide based on its contents whether to run them through a worker machine.

Important: The monadic assembly line **separates out the work of combining the output of the worker machines from the actual work done by the worker machines**. So, the combiners and workers can be used in different contexts.

## Assembly Line of Chopsticks: Worker Machines

```haskell
-- the basic types we are dealing with
type Wood = ...
type Chopsticks = ...
data Wrapper x = Wrapper x

-- NOTE: the Tray type comes from the Tray monad

-- worker function 1: makes roughly shaped chopsticks
makeChopsticks :: Wood -> Tray Chopsticks
makeChopsticks w = ...

-- worker function 2: polishes chopsticks
polishChopsticks :: Chopsticks -> Tray Chopsticks
polishChopsticks c = ...

-- worker function 3: wraps chopsticks
wrapChopsticks :: Chopsticks -> Tray Wrapper Chopsticks
wrapChopsticks c = ...
```

## Example: Assembly Line of Chopsticks

- Our trays should either be empty or contain a single item.
- Our loader machine would simply take an item and place it in a tray on the conveyor belt.
- The combiner machine would take each input tray and pass along empty trays while feeding the contents of non-empty trays to its worker machine.

## Example: Assembly Line of Chopsticks

In Haskell, we would define the Tray monad as:

```
-- trays are either empty or contain a single item
data Tray x = Empty | Contains x

-- Tray is a monad
instance Monad Tray where
    -- pass along
    Empty          >>= _      = Empty
    -- feed to worker machine
    (Contains x) >>= worker = worker x
    -- place item in a tray
    return                    = Contains
    fail _                    = Empty
```

## Example: Assembly Line of Chopsticks

All that remains is to sequence the worker machines together using the loader and combiner machines to make a complete assembly line.
In Haskell, the sequencing can be done using the standard monadic functions:

```
assemblyLine :: Wood -> Tray Wrapped Chopsticks
assemblyLine w = (return w) >>= makeChopsticks
                 >>= polishChopsticks >>= wrapChopsticks
```

or using the built in Haskell "do" notation for monads:

```
assemblyLine :: Wood -> Tray Wrapped Chopsticks
assemblyLine w = do c   <- makeChopsticks w
                    c'  <- polishChopsticks c
                    c'' <- wrapChopsticks c'
                    return c''
```

## Why build assembly line using a monadic approach?

What if we wanted to change the manufacturing process to have an error message instead of having an empty tray?

```haskell
-- tray2s either contain a single item
-- or contain a failure report
data Tray2 x = Contains x | Failed String

-- Tray2 is a monad
instance Monad Tray2 where
    (Failed reason) >>= _      = Failed reason
    (Contains x)    >>= worker = worker x
    return                     = Contains
    fail reason                = Failed reason
```

Replacing the Tray monad with the Tray2 monad instantly upgrades your assembly line.

## Meet the Monads

### Type Constructors

A `type constructor` is a parameterized type definition used with polymorphic types. Example:

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

`Maybe` is a type constructor and `Nothing` and `Just` are data constructors.

You can construct a **data value** by applying the `Just` data constructor to a value:

```haskell
country = Just "Portugal"
```

You can construct a **type** by applying the `Maybe` type constructor to a type:

```haskell
lookupAge :: DB -> String -> Maybe Int
```

## Meet the Monads

### Type Constructors

- Polymorphic types are like containers that are capable of holding values of many different types.
- `Maybe Int` can be thought of as a `Maybe` container holding an `Int` value (or Nothing) and `Maybe String` would be a `Maybe` container holding a `String value` (or Nothing).
- In Haskell, we can also make the **type of the container polymorphic**, so we could write ”m a” to represent a container of some type holding a value of some type!

## Meet the Monads

In Haskell a monad is represented as:

- A type constructor (call it m)
- A function that builds values of that type ($a \to m\ a$)
- And a function that combines values of that type with computations that produce values of that type to produce a new computation for values of that type ($m\ a \to (a \to m\ b) \to m\ b$).

Note that the container is the same, but the type of the contents of the container can change.

## Meet the Monads

The function that builds values of type `m` is traditionally called `return` and the third function is known as `bind` but is written `>>=`. The signatures of the functions are:

```haskell
-- the type of monad m
data m a = ...

-- return takes a value and embeds it in the monad.
return :: a -> m a

-- bind is a function that combines a monad instance m a
-- with a computation that produces another monad
-- instance m b from a's to produce a new monad
-- instance m b
(>>=) :: m a -> (a -> m b) -> m b
```

## Meet the Monads

Put simply:

- The monad type constructor defines a type of computation.
- The return function creates primitive values of that computation type.
- >>= combines computations of that type together to make more complex computations of that type.

## Meet the Monads

Using the container analogy:

- The type constructor `m` is a container that can hold different values and `m a` is a container holding a value of type a.
- The return function puts a value into a monad container.
- The >>= function takes the value from a monad container and passes it to a function to produce a monad container containing a new value, possibly of a different type. The >>= function is known as "bind" because it binds the value in a monad container to the first argument of a function.

## An Example

Suppose that we are writing a program to keep track of sheep cloning experiments. We want to keep the genetic history of all of our sheep, so we would need mother and father functions. But since these are cloned sheep, they may not always have both a mother and a father!

We would represent the possibility of not having a mother or father using the Maybe type constructor in our Haskell code:

```haskell
type Sheep = ...

father :: Sheep -> Maybe Sheep
father = ...

mother :: Sheep -> Maybe Sheep
mother = ...
```

## An Example

Then, defining functions to find grandparents is a little more complicated, because we have to handle the possibility of not having a parent:

```
maternalGrandfather :: Sheep -> Maybe Sheep
maternalGrandfather s = case (mother s) of
                            Nothing -> Nothing
                            Just m  -> father m
```

and so on for the other grandparent combinations.

## An Example

It gets even worse if we want to find great grandparents:

```
mPGrandfather :: Sheep -> Maybe Sheep
mPGrandfather s = case (mother s) of
                    Nothing -> Nothing
                    Just m  -> case (father m) of
                                 Nothing -> Nothing
                                 Just gf -> father gf
```

It is clear that a Nothing value at any point in the computation will cause Nothing to be the final result, and it would be much nicer to implement this notion once in a single place and remove all of the explicit case testing scattered all over the code.

## An Example

So good programming style would have us create a combinator that captures the behavior we want:

```
-- comb is a combinator for sequencing
-- operations that return Maybe
comb :: Maybe a -> (a -> Maybe b) -> Maybe b
comb Nothing  _ = Nothing
comb (Just x) f = f x

-- now we can use `comb` to build
-- complicated sequences
mPGrandfather :: Sheep -> Maybe Sheep
mPGrandfather s = (Just s) `comb` mother `comb`
                              father `comb` father
```

The code is much cleaner and easier to write, understand and modify.

The comb function is entirely polymorphic – it is not specialized for Sheep.

## An Example

**The happy outcome is that common sense programming practice has led us to create a monad without even realizing it.**

The Maybe type constructor along with the Just function (acts like return) and our combinator (acts like >>=) together form a simple monad for building computations which may not return a value.

All that remains to make this monad truly useful is to make it conform to the monad framework built into the Haskell language.

**That will be next week's task!**
**Watch the videos available on Moodle to prepare further for the next steps.**