Paradigmas de Programação

Week 11
Functors, Applicatives, and Monads (Recap)
State Monad

Alexandra Mendes

Functors, Applicative Functors, and Monads

What have we seen so far?

Functors, Applicative Functors, and Monads

Class Functor

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

Functors, Applicative Functors, and Monads

Functor Laws

```
1. fmap id == id
-- If we map the id function over a functor,
-- the functor that we get back should be the
-- same as the original functor.

2. fmap (f . g) ==  fmap f . fmap g
-- Composing two functions and then mapping the
-- resulting function over a functor should be
-- the same as first mapping one function over the
-- functor and then mapping the other one.
```

Functors, Applicative Functors, and Monads

Class Applicative

```
class (Functor f) => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

Functors, Applicative Functors, and Monads

Applicative Functor Laws

```
1. pure id <*> v == v
-- Identity
-- Pure preserves the identity function.

2. pure (.) <*> u <*> v <*> w == u <*> (v <*> w)
-- Composition
-- Analogous to (u.v)w == u(vw)
-- The operator <*> is associative.

3. pure f <*> pure x == pure (f x)
-- Homomorphism
-- Pure preserves function application.
```

Functors, Applicative Functors, and Monads

Applicative Functor Laws

```
4. u <*> pure y == pure (\$ y) <*> u
-- Interchange
-- Same as u <*> pure y = pure (\g ->  g y) <*> u
-- When an effectful function is applied to a pure
-- argument, the order in which we evaluate the two
-- components doesn t matter.
-- Applying $u$ to a lifted an argument is the same
-- as extracting the underlying function from $u$ and
-- applying it to the unlifted argument.
```

Functors, Applicative Functors, and Monads

Class Monad

```
class Monad m where
    (>>=)  :: m a -> (a -> m b) -> m b
    return :: a -> m a
```

Monad Laws

```
1. (return x) >>= f == f x
2. m >>= return == m
3. (m >>= f) >>= g == m >>= (\x -> f x >>= g)
```

Law 3 can be re-written, for clarity, as:

```
(m >>= (\x -> f x)) >>= g == m >>= (\x -> f x >>= g)
```

Functors, Applicative Functors, and Monads

- **Functors:** allow us to apply some function over the values contained by a functor (`fmap`);

Functors, Applicative Functors, and Monads

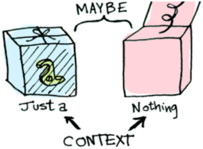
- **Functors:** allow us to apply some function over the values contained by a functor (`fmap`);
- **Applicative Functors:** allow us to apply a function which is inside a functor to a value inside a functor (operator `<*>`)

Functors, Applicative Functors, and Monads

- **Functors:** allow us to apply some function over the values contained by a functor (`fmap`);
- **Applicative Functors:** allow us to apply a function which is inside a functor to a value inside a functor (operator `<*>`)
- **Monads:** allow us to apply a function that returns a wrapped value to a wrapped value.

Functors[1]

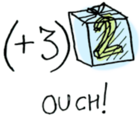
```
data Maybe a = Nothing | Just a
```



[1] Images taken from http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html

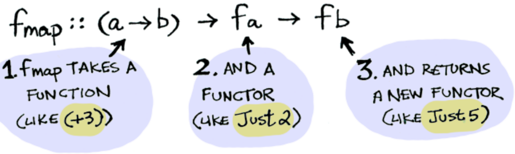
Functors[2]

If we try to apply a function to a context:



[2]Images taken from http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html

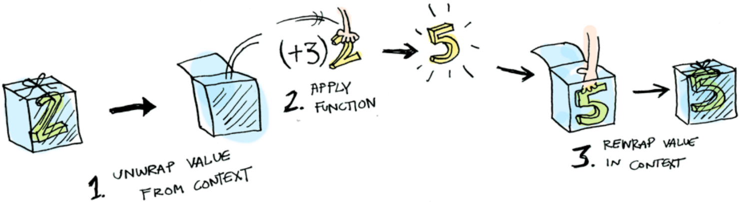
Functors[3]



[3] Images taken from `http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html`

Functors, Applicative Functors, and Monads[4]

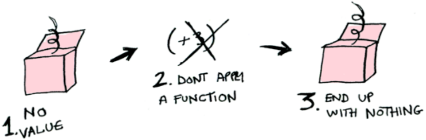
fmap (+3) (Just 2)



[4] Images taken from `http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html`

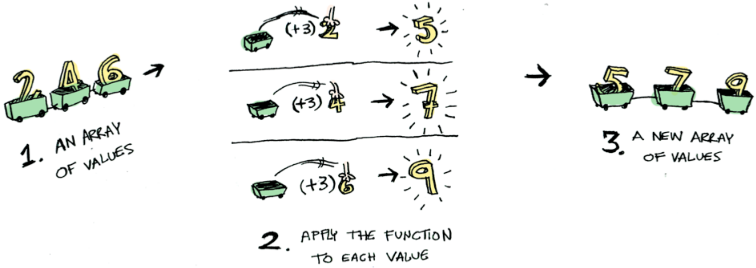
Functors, Applicative Functors, and Monads[5]

fmap (+3) Nothing



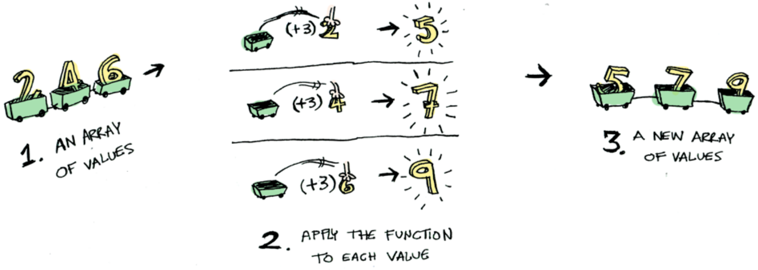
[5] Images taken from `http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html`

Functors[6]



[6] Images taken from `http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html`

Functors[6]

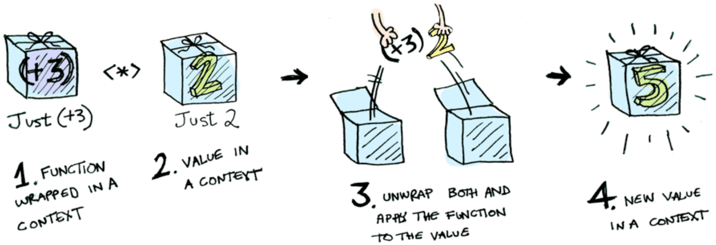


What if we want to apply a function that is inside a context to a value that is also inside a context?

[6] Images taken from `http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html`

In come the Applicative Functors[7]

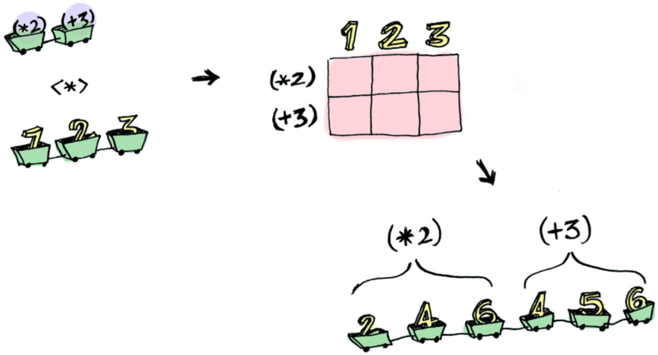
$$\text{Just } (+3) <*> \text{Just } 2 == \text{Just } 5$$



[7] Images taken from `http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html`

In come the Applicative Functors[8]

$$[(*2), (+3)] < * > [1, 2, 3]$$



[8] Images taken from `http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html`

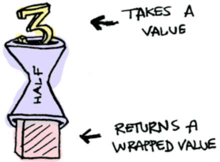
Functors[9]

What if we want to apply to a value inside a context a function that takes a value and returns a value inside a context?

[9] Images taken from http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html

In come the Monads[10]

```
half x = if even x
            then Just (x `div` 2)
            else Nothing
```



[10] Images taken from http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html

And then the Monads[11]

```
half x = if even x
           then Just (x `div` 2)
           else Nothing
```

[11]Images taken from http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html

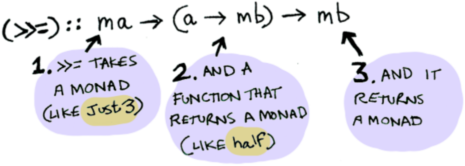
And then the Monads[11]

```
half x = if even x
           then Just (x `div` 2)
           else Nothing
```

We need to use >>= ...

[11] Images taken from http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html

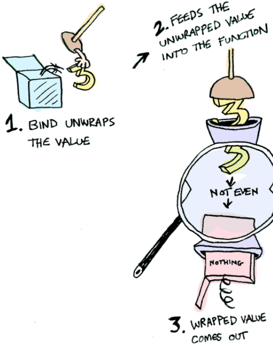
And then the Monads[12]



[12] Images taken from `http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html`

And then the Monads[13]

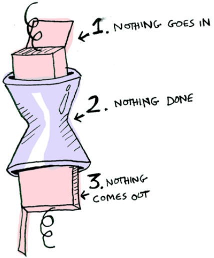
```
half x = if even x
           then Just (x `div` 2)
           else Nothing
```



[13] Images taken from http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html

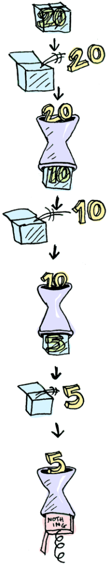
And then the Monads[14]

```
half x = if even x
           then Just (x `div` 2)
           else Nothing
```



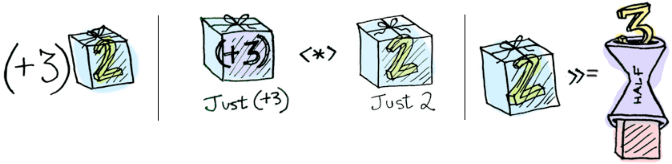
[14]Images taken from http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html

Monads[15]



[15] Images taken from http://adit.io/posts/2013-04-17-functors,

Functors, Applicative Functors, and Monads: Recap[16]



Functor Applicative Monad

[16]Images taken from `http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html`

Quizz

- With `f x = x + 1` how do you solve the error in
  `> f (Just 1)`?

Quizz

- With `f x = x + 1` how do you solve the error in `> f (Just 1)`?
- `fmap f (Just 1)`

Quizz

- With `f x = x + 1` how do you solve the error in `> f (Just 1)`?
- `fmap f (Just 1)`

- With `f x = Just (x + 1)` what's the result of `> pure f <*> [1,2,4]`?

Quizz

- With `f x = x + 1` how do you solve the error in `> f (Just 1)`?
- `fmap f (Just 1)`

- With `f x = Just (x + 1)` what's the result of `> pure f <*> [1,2,4]`?
- `[Just 2, Just 3, Just 5]`

Quizz

- With `f x = x + 1` how do you solve the error in
  `> f (Just 1)`?
- `fmap f (Just 1)`

- With `f x = Just (x + 1)` what's the result of
  `> pure f <*> [1,2,4]`?
- `[Just 2, Just 3, Just 5]`

- What's the result of
  `> pure (*) <*> [1,2,3] <*> [4,5,6]`?

Quizz

- With `f x = x + 1` how do you solve the error in `> f (Just 1)`?
- `fmap f (Just 1)`

- With `f x = Just (x + 1)` what's the result of `> pure f <*> [1,2,4]`?
- `[Just 2, Just 3, Just 5]`

- What's the result of `> pure (*) <*> [1,2,3] <*> [4,5,6]`?
- `[4,5,6,8,10,12,12,15,18]`

Quizz

- With `f x = x + 1` how do you solve the error in `> f (Just 1)`?
- `fmap f (Just 1)`
- With `f x = Just (x + 1)` what's the result of `> pure f <*> [1,2,4]`?
- `[Just 2, Just 3, Just 5]`
- What's the result of `> pure (*) <*> [1,2,3] <*> [4,5,6]`?
- `[4,5,6,8,10,12,12,15,18]`
- What's the result of `(+) <$> [1,2,3] <*> [4,5,6]`?

Quizz

- With `f x = x + 1` how do you solve the error in
  `> f (Just 1)`?
- `fmap f (Just 1)`

- With `f x = Just (x + 1)` what's the result of
  `> pure f <*> [1,2,4]`?
- `[Just 2, Just 3, Just 5]`

- What's the result of
  `> pure (*) <*> [1,2,3] <*> [4,5,6]`?
- `[4,5,6,8,10,12,12,15,18]`

- What's the result of `(+) <$> [1,2,3] <*> [4,5,6]`?
- `[5,6,7,6,7,8,7,8,9]`

Quizz

- Indicate two different ways of solving the error in `(subtract) <*> [1,2,3] <*> [1,2,3]`.

Quizz

- Indicate two different ways of solving the error in `(subtract) <*> [1,2,3] <*> [1,2,3]`.
- `(subtract) <$> [1,2,3] <*> [1,2,3]` and `pure (subtract) <*> [1,2,3] <*> [1,2,3]`

Quizz

- Indicate two different ways of solving the error in `(subtract) <*> [1,2,3] <*> [1,2,3]`.
- `(subtract) <$> [1,2,3] <*> [1,2,3]` and `pure (subtract) <*> [1,2,3] <*> [1,2,3]`
- With `f x = Just (x + 1)` how do you solve the error in `> 1 >>= f >>= f >>= f`?

Quizz

- Indicate two different ways of solving the error in `(subtract) <*> [1,2,3] <*> [1,2,3]`.
- `(subtract) <$> [1,2,3] <*> [1,2,3]` and `pure (subtract) <*> [1,2,3] <*> [1,2,3]`
- With `f x = Just (x + 1)` how do you solve the error in `> 1 >>= f >>= f >>= f`?
- `> return 1 >>= f >>= f >>= f`

Quizz

- Indicate two different ways of solving the error in `(subtract) <*> [1,2,3] <*> [1,2,3]`.
- `(subtract) <$> [1,2,3] <*> [1,2,3]` and `pure (subtract) <*> [1,2,3] <*> [1,2,3]`

- With `f x = Just (x + 1)` how do you solve the error in `> 1 >>= f >>= f >>= f`?
- `> return 1 >>= f >>= f >>= f`

- What's the result of `> return 1 >>= f >>= f >>= f`?

Quizz

- Indicate two different ways of solving the error in `(subtract) <*> [1,2,3] <*> [1,2,3]`.
- `(subtract) <$> [1,2,3] <*> [1,2,3]` and `pure (subtract) <*> [1,2,3] <*> [1,2,3]`

- With `f x = Just (x + 1)` how do you solve the error in `> 1 >>= f >>= f >>= f`?
- `> return 1 >>= f >>= f >>= f`

- What's the result of `> return 1 >>= f >>= f >>= f`?
- `Just 4`

Quizz

- What's the result of `> return 1 >>= f >> f 1 >>= f`?

Quizz

- What's the result of > `return 1 >>= f >> f 1 >>= f`?
- `Just 3`

Quizz

- What's the result of `> return 1 >>= f >> f 1 >>= f`?
- `Just 3`
- What's the result of
  `> return 1 >> Nothing >>= f >>= f`?

Quizz

- What's the result of `> return 1 >>= f >> f 1 >>= f`?
- `Just 3`
- What's the result of
  `> return 1 >> Nothing >>= f >>= f`?
- `Nothing`

State Monad

Consider the problem of writing functions that manipulate a state. For simplicity, we assume that the state is just an integer:

```
type State = Int
```

State Monad

```
type State = Int
```

The most basic form of function on this type is a state transformer (ST) which takes an input state as its argument and produces an output state as its result:

```
type ST = State -> State
```

It is called a state transformer because it transforms a state into another.

State Monad

```
type State = Int
type ST = State -> State
```

However, in general, we may want to return a result in addition to updating the state (e.g. if we have a counter as the state). Thus, we generalize the type of state transformers to:

```
type ST a = State -> (a,State)
```

with the type of the return value being a paramenter of the `ST` type.

The general idea: take a state, do something with it, and return a result and possibly a new state.

State Transformer

- Given that ST is a parameterised type, it is natural to make it into a Monad so that the do notation can be used.
- However, types declared using the type mechanism are a type synonym and not a newtype, so these cannot be made into instances of classes.
- We redefine ST using the newtype mechanism which requires the introduction of a constructor S:

```
newtype ST a = S (State -> (a,State))
```

- It is also convenient to define a special-purpose application function that simply removes the constructor from this type:

```
app :: ST a -> State -> (a,State)
app (S st) x = st x
```

State Transformer: Functor

As a first step towards making the parameterised type ST into a Monad is to make this type a Functor:

```
instance Functor ST where
 -- fmap         :: (a -> b) -> ST a -> ST b
 fmap g st = S (\s -> let (x,s') = app st s
                      in (g x, s'))
```

State Transformer: Applicative Functor

The type ST can now be made into an Applicative Functor:

```
instance Applicative ST where
 -- pure          :: a -> ST a
 pure x = S (\s -> (x,s))

 -- (<*>) :: ST (a -> b) -> ST a -> ST b
 stf <*> stx = S (\s -> let (f,s') = app stf s
                            (x,s'') = app stx s'
                        in (f x, s''))
```

- Pure transforms a value into a state transformer that returns this value without modifying the state.
- The operator $<*>$ applies a state transformer that returns a function to a state transformer that returns an argument to give a state transformer that returns the result of applying the function to the argument.

State Transformer: Monad

Finally, the Monad instance:

```
instance Monad ST where
 -- return :: a -> ST a
 return x  =  S(\s -> (x,s))
 -- same as return x s = (x,s) and as return x = pure x

 -- (>>=) :: ST a -> (a -> ST b) -> ST b
 st >>= f = S (\s -> let (x,s') = app st s
                     in app (f x) s')
```

- *return* converts a value into a state transformer that simply returns that value without modifying the state.
- We could use $returnxs = (x, s)$ but our makes explicit that *return* is a function that takes a single argument and returns a state transformer (`a -> ST a`).
- $st \quad >>= \quad f$ applies the state transformer to an initial state `s`, then applies the function `f` to the resulting value `x` to give a new state transformer `f x`, which is then applied to the new state `s'`.

Example: Relabelling Trees

Consider the following type of trees:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
              deriving Show
```

We can define the following tree:

```
tree :: Tree Char
tree = Node (Node (Leaf 'a') (Leaf 'b')) (Leaf 'c')
```

Our goal is to develop a function that relabels each leaf in these trees with a unique or fresh integer.

Example: Relabelling Trees

This can be implemented by taking the next fresh integer can be implemented by taking the fresh integer as an extra parameter and returning the next fresh integer as an additional result:

```
rlabel :: Tree a -> Int -> (Tree Int, Int)
rlabel (Leaf _) n = (Leaf n, n+1)
rlabel (Node l r) n = (Node l' r', n'')
                        where
                         (l',n') = rlabel l n
                         (r',n'') = rlabel r n'
```

This definition is complicated by the need to explicity pass an integer state through the computation.

Example: Relabelling Trees

A simpler definition can be obtained by noting that the type:

```
Tree a -> Int -> (Tree Int, Int)
```

can be rewritten using the type of state transformers by:

```
Tree a -> ST (Tree Int)
```

where the state is the next fresh integer.

The next fresh integer can be generated by defining a state transformer that returns the current state as its result and the next integer as the new state:

```
fresh :: ST Int
fresh = S (\n -> (n, n+1))
```

Example: Relabelling Trees

The fact that ST is a monad allows us to define a monadic version of the function rlable:

```
mlabel :: Tree a -> ST (Tree Int)
mlabel (Leaf _) = do n  <- fresh
                    return (Leaf n)
mlabel (Node l r) = do l' <- mlabel l
                       r' <- mlabel r
                       return (Node l' r')
```

Now run:

```
> app (mlabel tree) 0
```

Note: In the file `State.hs` we use the more general type:

```
newtype ST st a = S {runstate:: st -> (a, st)}
```

The Record Syntax

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     , height :: Float
                     , phoneNumber :: String
                     } deriving (Show)
```

```
tomas :: Person
tomas = Person "Tomas""Jeronimo" 22 1.75 "000000000"
```

This creates acessor functions and a convenient update method:

```
>age tomas
22
>> tomas {age = 23}
Person {firstName = "Tomas", lastName = "Jeronimo", age = 23,
        height = 1.75, phoneNumber = "000000000"}
```

Another Example: Counter of divisions

Consider the following type:

```
data Term = Cons Int | Div Term Term
                deriving Show

expr :: Term
expr = (Div (Cons 16) (Div (Cons 8) (Cons 4)))
```

Basic Evalution:

```
eval :: Term -> Int
eval (Cons a) = a
eval (Div t u) = (eval t) div (eval u)
```

Another Example: Counter of divisions

Consider that we wanted to count the number of divisions performed during an evaluation.
An option is to add an additional component, called state, which is an integer initialised to zero at the start.
Implement the function:

```
mEval :: Term -> ST Int Int
```

Counter of divisions: Solution

```
mEval :: Term -> ST Int Int
mEval (Cons x) = S (\s -> (x,s))
mEval (Div t u) = S (\s ->
                      let (x, s') = runstate (mEval t) s
                          (y, s'') = runstate (mEval u) s'
                      in (x `div` y, s''+1))
```

Now run:

```
> runstate (mEval expr) 0
```

Show instance

To avoid typing

```
> runstate (mEval expr) 0
```

we can create an instance of the class Show:

```haskell
instance (Show a, Show b, Num a) => Show (ST a b) where
  show f = "value:"++ show x ++ ",  count:"++ show s
                where (x,s) = runstate f 0
```

And now type only:

```
> mEval expr
value:Node (Node (Leaf 0) (Leaf 1)) (Leaf 2),  count:3
```

Exercise: Implementing a Moving Robot

Consider the following type that represents the position of a Robot:

```
type Pos = (Int, Int)
```

Create the following functions which move the Robot:

```
left, right, up, down :: ST Pos ()
```

Exercise: Implementing a Moving Robot - Solution

```
left :: ST Pos ()
left = S (\(x,y) -> ((),(x-1,y)))

right :: ST Pos ()
right = S (\(x,y) -> ((),(x+1,y)))

up :: ST Pos ()
up = S (\(x,y) -> ((),(x,y+1)))

down :: ST Pos ()
down = S (\(x,y) -> ((),(x,y+1)))

moves = do right; right; left; up; down; up
```

Run in GHCi: `runstate moves (0,0)`

Exercise: Implementing a Stack Machine

Implement a simple stack machine with four instructions:

- push an integer;
- pop an integer;
- add values on the stack;
- multiply values on the stack.

These operations `add` and `multiply` pop two values from the stack and push the result.
Consider the following type to represent the Stack:

```
type Stack = [Int]
```

Exercise: Implementing a Stack Machine – Solution

```
type Stack = [Int]

push :: Int -> ST Stack ()
push x = S (\s -> ((),(x:s)))

pop :: ST Stack Int
pop = S (\(x:xs) -> (x, xs))


add :: ST Stack ()
add = do x <- pop
         y <- pop
         push (x+y)

mult :: ST Stack ()
mult = do x <- pop
          y <- pop
          push (x*y)
```

Run in GHCi:

```
runstate (do pop; pop; push 1; push 4; add; push 2; mult)
```

Final Note

You can compile Haskell programs to produce a standalone program. Just type in the command line:

```
ghc -o Aula11 Aula11.hs
```

and then run it by typing:

```
./Aula11
```

Note that you need an action `main`, the starting point of your program, for it compile.

Practical Session

Now solve all exercises in the practical sheet for Week 11 available on Moodle.