Everything you wanted to know about *monads*, but were afraid to ask

Rolandas Griškevičius Swedbank/VGTU

Chapter 1

Alien

OO and FP

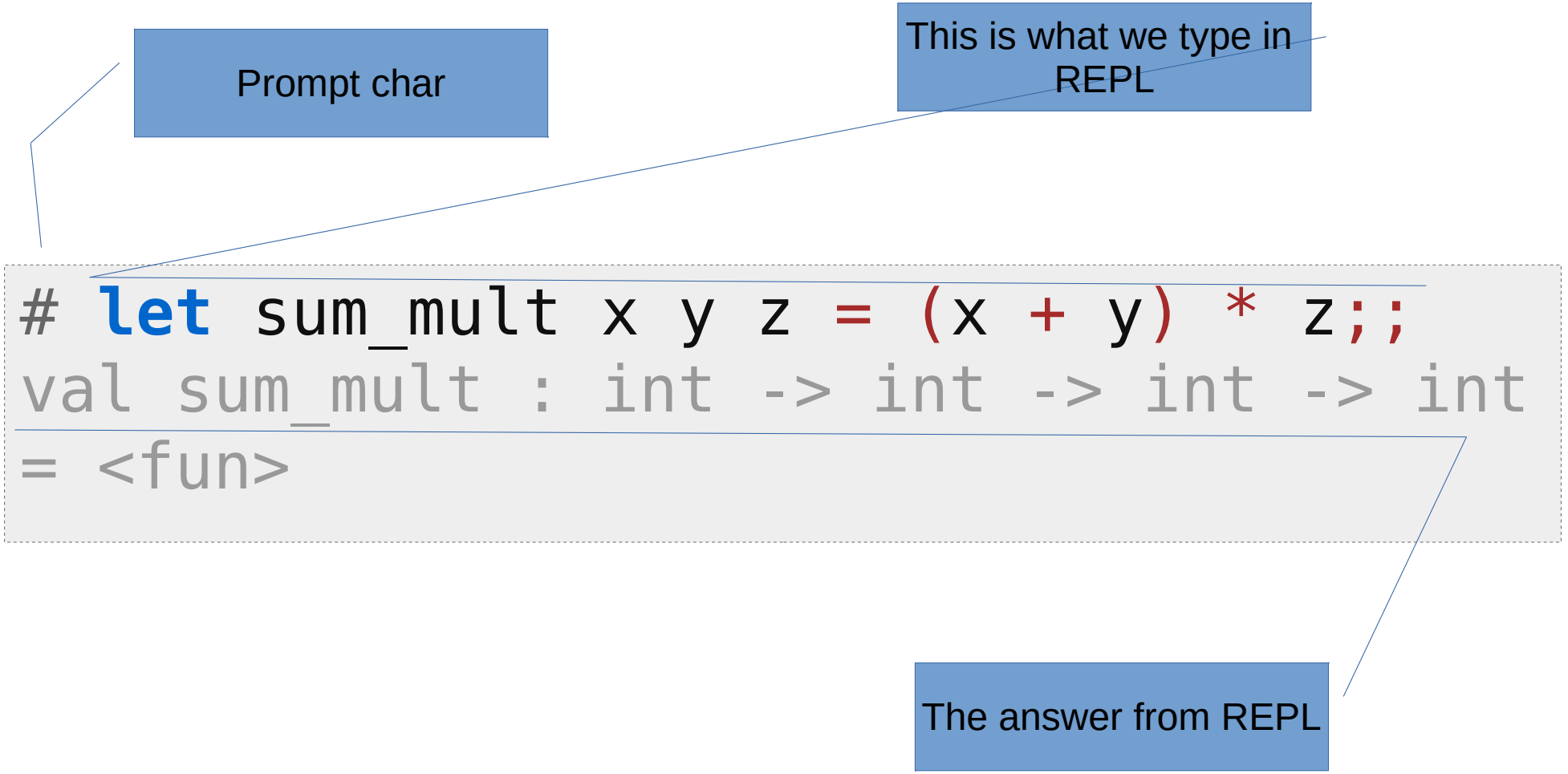
- OO world
  - Objects
  - Program is object composition via invocation
  - We use design patterns for reusability, clean and correct code
- FP world
  - Functions
  - Program is function composition via application
  - We use monads to have reusable, clean and short code

A monad is an FP design pattern

Chapter 2

Batman begins

Code format


Prompt char
This is what we type in REPL
let sum_mult x y z = (x + y) * z;;
val sum_mult : int -> int -> int -> int
= <fun>
The answer from REPL

REPL output

```
# let a = fn_with_3_params 5 5 10;;
val a : int = 100
# fn_with_3_params 5 5 10;;
- : int = 100
```

Answer is bound to variable “result”, type int, value 100

Answer was calculated, but not bound to any variable
REPL just shows it for information
Answer has type int, value 100

Some key FP concepts. Currying

- We define 3 parameter function:

```
let f3p b i s =
  if b then String.sub s 0 i
  else String.make i '*'
val f3p : bool -> int -> string -> string = <fun>
```

All but last are function parameters

Last one – return type

- We apply parameters to the function:

```
# f3p true 5 "abcdefghjikl";;
- : string = "abcde"
# f3p false 3 "abcdefghjikl";;
- : string = "***"
```

FP concepts. Currying 2

```
let f3p b i s =
   if b then String.sub s 0 i
   else String.make i '*'
val f3p : bool -> int -> string -> string = <fun>
```

We expect we have 3 parameter function:

```
f3p (b,i,s)=  . . .
```

But OCaml converts it to curried form:

```
let f3p = fun b -> (fun i -> (fun s -> .... ))
```

FP concepts. Currying 3

```
let f3p b i s =
   if b then String.sub s 0 i
   else String.make i '*'
val f3p : bool -> int -> string -> string = <fun>
```

Let's rewrite it in curried way:

```
# let f3p =
      fun b -> (
         fun i -> (
            fun s -> (
              if b then String.sub s 0 i
              else String.make i '*')));;
val f3p : bool -> int -> string -> string = <fun>
```

… and we get the same result

FP Concepts. Partial application

- If this is composed single parameter functions
  - We can apply one parameter at a time

```
# let f2p = f3p true;;
val f2p : int -> string -> string = <fun>
```

- Result is 2 parameter function
- Parameter true is taken to the closure
- No way to change it to false

FP concepts. Higher-order functions

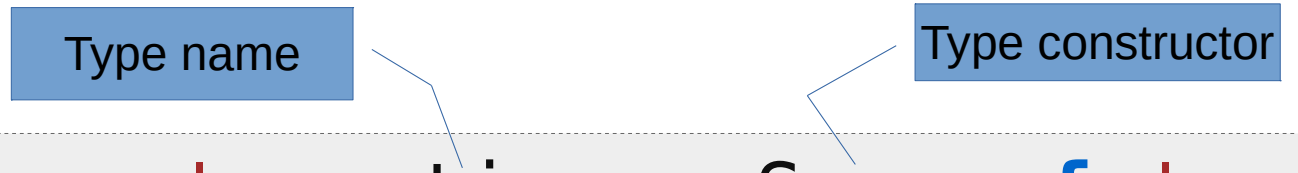
```
# List.map;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

- 'a , 'b – type variables, similar to Java generics.
- Can be instantiated with any types

- List.map takes another function 'a->'b

```
# let is_even = fun x -> x mod 2 = 0;;
val is_even : int -> bool = <fun>
# List.map is_even;;
- : int list -> bool list = <fun>
```

Type names




```
type 'a option = Some of 'a | None;;
```

- Value of type option may have Some value or None
- A way to wrap values into a “box”

- As in Java's Guava:

```
Optional<Integer> possible = Optional.of(5);
possible.isPresent();
possible.get();
```

Option type

```
type 'a option = Some of 'a | None;;
```

- Commonly used to indicate failure

- Function get_parameter reads and parses parameters from file

```
# get_parameter;;
- : string -> string -> string option = <fun>
get_parameter "option_parameters.txt"  "param1";;
- : string option = Some "parameter 1"
# get_parameter "option_parameters.txt" "pp";;
- : string option = None
```

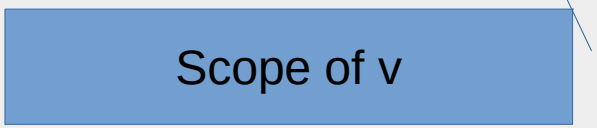
To and from option

- Make an option is easy:

```
# Some 5;;
- : int option = Some 5
# Some "abc";;
- : string option = Some "abc"
# None;;
- : 'a option = None
```

- Unwrap requires pattern matching:

```
# let unoption o = match o with
    | Some   v -> print_endline ("Unwrapped v = " ^ v)
    | None     -> failwith ("Crash");;
val unoption : string option -> unit = <fun>

# unoption (Some "abc123");;
Unwrapped v = abc123
- : unit = ()

# unoption None;;
Exception: Failure "Crash".
```



Chapter 3

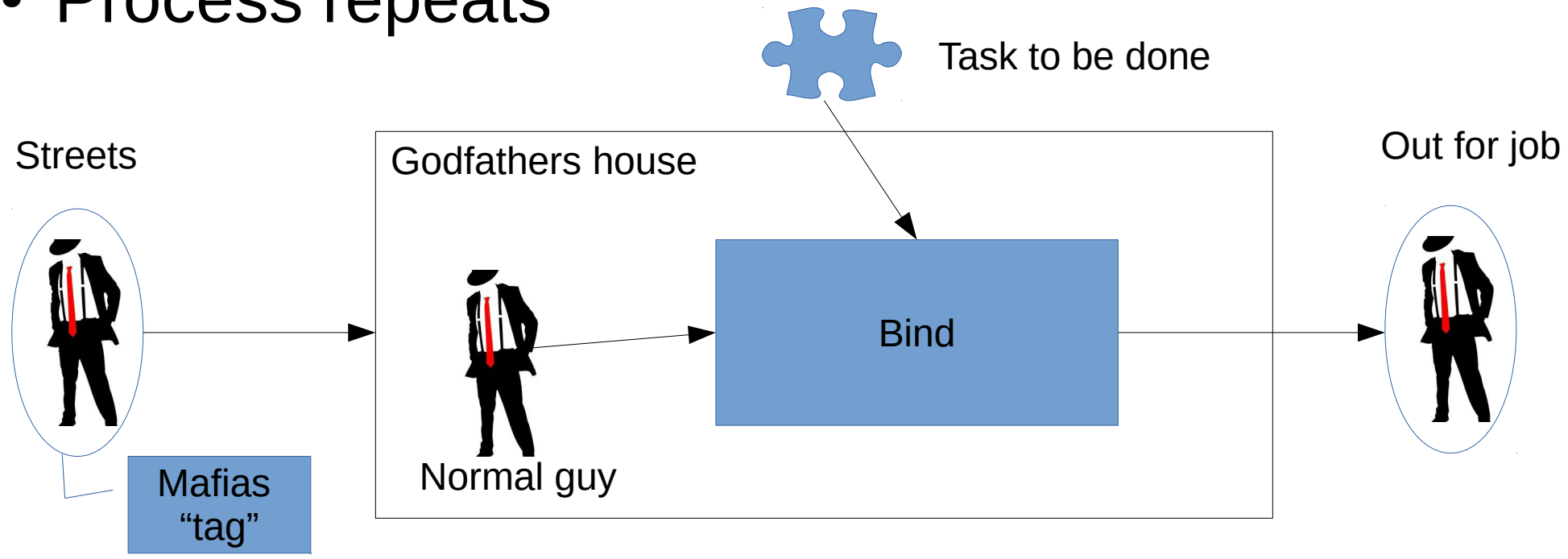
The Godfather

The mafia contract

- That's definitely for life
- Mafia is managed in Godfathers house, but jobs are done on the streets
- Function “unit” brings you in
  - You was normal type of the guy
  - And now you are not
    - People do not mess with you on the streets
- But inside Godfathers house, you are normal guy
    - Because they all are mafia guys there

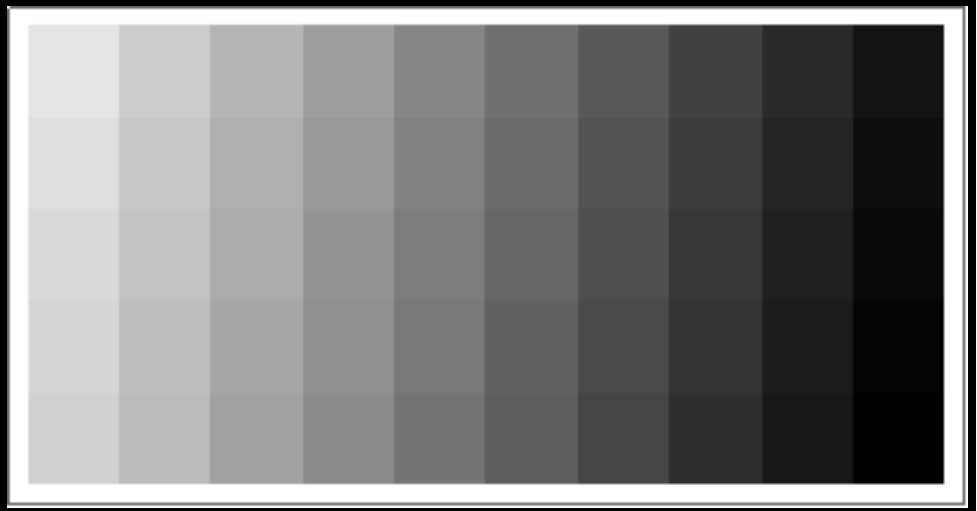
Mafia `binds` the job

- You come on request
- Various types of the guys are considered
- Some calculation is run and guy is selected
- The guy is sent to do the job
- Process repeats



Chapter 4

Fifty shades of monad



A monad

- A monad in Ocaml

```
module type MonadRequirements = sig
    type 'a m
    val bind : 'a m -> ('a -> 'b m) -> 'b m
    val return : 'a -> 'a m
end;;
```

- A structure with a "type constructor" m and 2 operations
- Ocaml module is a collection of types and functions
  - Similar to Scala traits

bind

```
module type MonadRequirements = sig
    type 'a m
    val bind : 'a m -> ('a -> 'b m) -> 'b m
    val return : 'a -> 'a m
end;;
```

- Performs “conversion” `'a m -> 'a` so you can do computation
- Accepts computation as second parameter
  - And applies “conversion” result 'a to computation function

Fifty shades

```
module type MonadRequirements = sig
    type 'a m
    val bind : 'a m -> ('a -> 'b m) -> 'b m
    val return : 'a -> 'a m
end;;
```

- bind returns monadic type
  - It will be passed to another bind
- You never leave the monadic context
  - But you may apply different functions (of different types)
- 'a and 'b may be different types

Monad vs option

```
val bind : 'a m -> ('a -> 'b m) -> 'b m
val return : 'a -> 'a m
```

mkoption = return !

```
# let mkoption a = Some a;;
val mkoption : 'a -> 'a option = <fun>
```

Chapter 5
Definitely, Maybe

Option / Choice / Maybe monad

```
type 'a maybe_monad =
   Success of 'a
   | Failure

let return x = Success x

let bind x y = match x with
    | Success a -> y a
    | Failure -> Failure

let (>>=) = bind
let fail = Failure
```

Monad definition:

'a M

return: 'a → 'a M

bind: 'a M → ('a → 'b M) → 'b M

Chapter 6

Rambo. First blood

Simple Maybe

```
type 'a maybe_monad = Success of 'a
| Failure

let return x = Success x

let bind x y = match x with
    | Success a -> y a
    | Failure -> Failure

let (>>=) = bind

…. more functions are defined

let is_odd x = if x mod 2 != 0
   then Success x
   else Failure

let is_greater_than y x = if x > y
   then Success x
   else Failure
```

REPL
DEMO

… and the use

```
# return 101 >>= is_prime >>= is_odd >>= to_string >>=
is_reversible;;
- : string maybe_monad = Success "101"

# return 2 >>= is_prime >>= is_odd >>= to_string >>=
is_reversible;;
- : string maybe_monad = Failure
```

- If answer is Success, means Success was in all cases
  - Guarantee that failure will not pass further
- Reusable
- Clean
- Many types can occur in computation
- But could be better.
  - Reason of failure ?

Maybe motivation

- Without a monad:

```
# let process filename =
    match get_parameter filename "param1" with
    |Some p1 -> (match get_parameter filename "param2" with
        |Some p2 ->
            let _ = Printf.printf
                  "Got params p1=%s, p2=%s, starting...\n" p1 p2
            in ""
        |None -> failwith "Cannot run, no parameter p2"
        )
    |None -> failwith "cannot run, no parameter p1" ;;
```

- With maybe monad:

```
# let process filename =
    get_parameter_m filename "param1" >>= fun p1 ->
    get_parameter_m filename "param2" >>= fun p2 ->
    let _ = print_endline (Printf.sprintf "Got params: p1=%s, p2=%s" p1 p2)
    in return ();;
val process : string -> unit maybe_monad = <fun>
```

```
# process "option_parameters.txt";;
Got params: p1=parameter 1, p2=5
- : unit maybe_monad = Success ()
```

Maybe better

- Failure has message

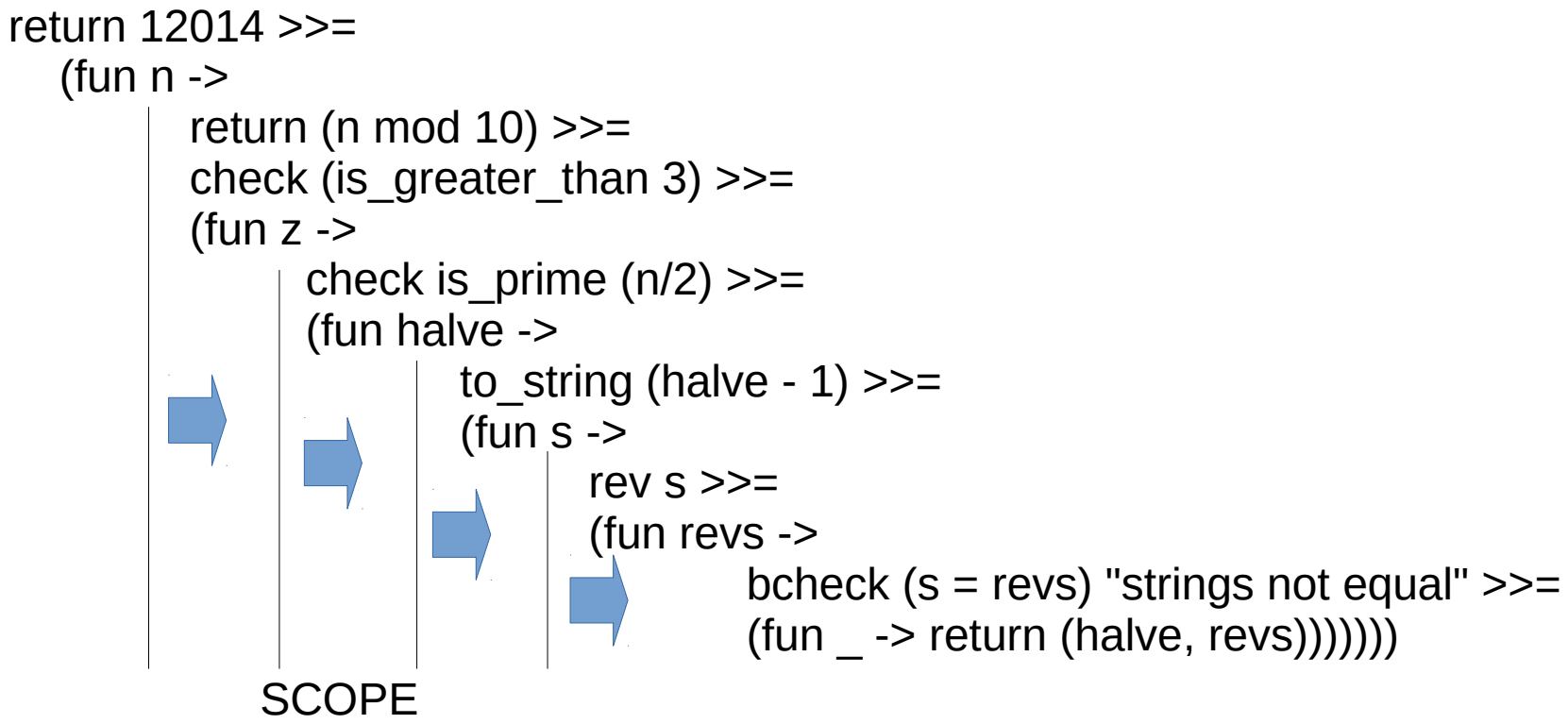
```
let bind x y = match x with
    | Success a -> y a
    | Failure s -> Failure s
```

- Some functions added

```
let check fn x = let (is_ok,e) = fn x in if is_ok
    then return x else fail e

let bcheck b e = if b then return b else fail e

let is_odd x = x mod 2 != 0, Printf.sprintf "Number %d is not odd" x

let matches_regex r s = Str.string_match (Str.regexp r) s 0,
    Printf.sprintf "%s did not match regexp %s" s r

let extract_match nth s =
    try return (Str.matched_group nth s) with
    Not_found -> fail (Printf.sprintf "regexp cannot extract %d group
    from string %s" nth s)
```

Inside the function

```
return 12014 >>= fun n ->
  return (n mod 10) >>=
  check (is_greater_than 3) >>= fun z ->
  check is_prime (n/2) >>= fun halve ->
  to_string (halve - 1) >>= fun s ->
  rev s >>= fun revs ->
  bcheck (s = revs) "strings not equal" >>= fun _ ->
  return (halve, revs) ;;
- : (int * string) maybe_monad = Success (6007, "6006")
```



With sugar or not

- Native Ocaml expression

```
return 12014 >>= fun n ->
  return (n mod 10) >>=
  check (is_greater_than 3) >>= fun z ->
  check is_prime (n/2) >>= fun halve ->
  to_string (halve - 1) >>= fun s ->
  rev s >>= fun revs ->
  bcheck (s = revs) "strings not equal" >>= fun _ ->
  return (halve, revs) ;;
- : (int * string) maybe_monad = Success (6007, "6006")
```

would look like this in Haskell with **do** sugar

```
foo = do
        n <- return 12014
        c <- return (n mod 10 )
        z <- check (is_greater_than 3)
        halve <- check is_prime (n/2)
        s <- to_string (halve - 1)
        revs <- rev s
        bcheck (s = revs) "strings not equal"
        return (halve, revs)
```

Chapter 7

The bone collector

The List monad

```
type 'a list_monad = 'a list


let return x = [x]

let bind l f =
    List.flatten (List.map f l)

let (>>=) = bind
```

for for the win

- *for* in Scala/Clojure is a monad

```
for (
     x <- List(1,2,3);
     y <- List("a","b","c")
    ) yield Pair(x,y);
```

```
List((1,a), (1,b), (1,c), (2,a), (2,b), (2,c), (3,
a), (3,b), (3,c))
```

- Using a List monad:

```
#
[1;2;3] >>= fun x ->
["a";"b";"c"] >>= fun y ->
return (x, y);;
- : (int * string) list =
[(1, "a"); (1, "b"); (1, "c"); (2, "a"); (2, "b"); (2, "c"); (3, "a");
 (3, "b"); (3, "c")]
```

Chapter 8
The enemy of the state

State monad

- Take maybe as basis and add state handling

```
type 'a maybe = Success of 'a | Failure of string
type ('a,'s) maybe_state_monad = 'a -> 'a maybe * 's

let bind f g x = match f x with
    | (Success v), s -> g v s
    | (Failure m), s -> (Failure m), s

let (>>=) = bind

let return x s = (Success x), s

let fail r s = (Failure r), s
```

```
let tick v s = return v (s+1)
```

OMG! Bind and 3 parameters

```
let bind f g x = match f x with
    | (Success v), s -> g v s
    | (Failure m), s -> (Failure m), s

let (>>=) = bind
```

- How >>= will behave with 3 parameters ?
- Answer is currying
- State monad produces state machine (function)

```
# return 5 >>= check is_odd ;;
- : 'a -> int maybe * 'a = <fun>

# ( return 5 >>= check is_odd ) 10 ;;
- : int maybe * int = (Success 5, 10)
```

Ticking state

- At every bind can examine state and calculate new one

```
let tick v s = return v (s+1)
```

- Every tick increases state counter

```
# let state_machine =
    return 5 >>= tick >>= check is_odd >>= (fun x y ->
    return x (y*10000)) >>= check is_prime;;
val state_machine : int -> int maybe * int = <fun>

# state_machine 1;;
- : int maybe * int = (Success 5, 20000)

# state_machine 10;;
- : int maybe * int = (Success 5, 110000)
```

Chapter 9

The law and order

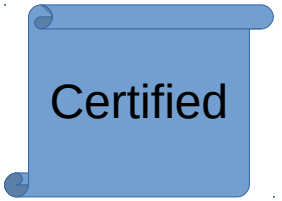
Is my monad correct ?

- 3 laws:
- Left unit
  - return a >>= k == k a
- Right unit
  - a >>= return == a
- Associativity
  - a >>= (f >>= g) == (a >>= f) >> g

Checking maybe

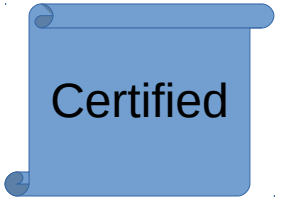
return a >>= k == k a

```
# return 5 >>= fun x -> return (x * 10);;
- : int maybe_monad = Success 50
# (fun x -> return (x * 10)) 5;;
- : int maybe_monad = Success 50
```

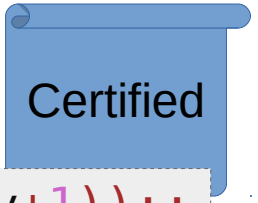


a >>= return == a

```
# return 5 >>= return ;;
- : int maybe_monad = Success 5
```



a >>= (f >>= g) == (a >>= f) >> g




```
# return 5 >>= (fun x -> return (x*10) >>= fun y -> return (y+1));;
- : int maybe_monad = Success 51
# (return 5 >>= fun x -> return (x*10)) >>= fun y -> return (y+1);;
- : int maybe_monad = Success 51
```

Chapter 10
Slumdog millionaire

A stolen parser

```
type 'a maybe_monad = Success of 'a | Failure of string
let parse p inp = p inp
let bind fi f inp =
    match fi inp with
        | Success ( a, inp1 ) -> f a inp1
        | Failure a           -> Failure a

let return x inp = Success ( x, inp )

let item inp = match inp with
    | []      ->  Failure "No more input"
    | (h::t)  ->  return h t

let or_choose p q inp = match p inp with
    | Failure _             ->  q inp
    | Success ( x , inp )   ->  Success ( x , inp )

let fail =  Failure ""
let (>>=) = bind
let (+++) = or_choose;;
```

Bind has 3 params. Parser will return a parsin function

“eats” the input

Parser combinator. If one parser fails, chooses another

A very nice FP into from Erik Meijer in Haskell

Adapted from http://channel9.msdn.com/Shows/Going+Deep/C9-Lectures-Dr-Erik-Meijer-Functional-Programming-Fundamentals-Chapter-8-of-13

Parser functions

```
let sat p = item >>= fun x -> if p x then return
x else fun _ -> fail;;

let isDigit c = c >= '0' && c <= '9';;
let isSmallLetter c = c >= 'a' && c <= 'z';;
let isCapitalLetter c = c >= 'A' && c <= 'Z';;
let isSpace c = c = ' ';;
let isOperator c = c = '*' || c = '+' || c = '-'
|| c = '/' ;;


let digit = sat isDigit;;
let smallLetter = sat isSmallLetter;;
let capitalLetter = sat isCapitalLetter;;
let space = sat isSpace;;
let operator = sat isOperator;;
```

Takes item from input and applies it to boolean function.

Parser functions 2

```
let rec many p =
    let many1 pp =
        pp >>= fun v ->
        many pp >>= fun vs ->
        return (v::vs) in
    many1 p +++ return [] ;;

let optional p =
    (p >>= fun x -> return [x] ) +++ return [] ;;
```

Applies same rule until Success and collects results into list

Parser in action

```
# let rule =
    many digit >>= fun ds ->
    many space >>= fun _ ->
    operator >>= fun op ->
    many digit >>= fun ds2 ->
    return (ds,op,ds2);;
val rule :
  char list -> ((char list * char * char list) * char list) maybe_monad =
  <fun>
```

```
# parse rule (explode "456+789");;
- : ((char list * char * char list) * char list)
maybe_monad =
Success ((['4'; '5'; '6'], '+', ['7'; '8'; '9']), [])
```

A nice parsing task

- We have thousands of receipts
- Receipts change over time
- Need to extract specific data
  - Number of lines vary
  - Receipt purpose vary
  - Need to extract dependent data

The receipt

Some Retailer, UAB
Degalinė XXXXX
******** Vilko g. *a, Vilnius
PVM kodas LT11xxxxxx, tel. 85xxxxx

95 BENZ 22.37 L 108.05 A
Kolonėlė Nr3 4.83 LT/L
Nuolaida -
95 BENZ -2.24 A

MOKĖTI 105.81
Mokėta LT 206.00
Kortelė 777666 ******* 1111

Kasininkas Jonas Jonaitis

KLIENTO KVITAS

Term. 1221 / 1 - Kvitas 9999
Transakcijos Data 2013/04/01 Laik 00:00

Pavad : UAB "KLIENTAS"
Adresas : ********** g. *-*
Kaunas

Įm.Kodas: 123xxxx99
PVM-Nr : LTxxxxxxx99

Many lines expected

Parašas

As far as this vernture code appears, Need to extract liters, venture code, etc

The parser for receipt

- Items = lines of receipt
- From char → bool to string → bool.
- Checking line with regexps
  - Extract data on full parser rule match

Check functions

```
let rexsat p nth = item >>= fun x -> if p x then return (Str.matched_group nth x)
else fun _ -> fail;;

    let line_is rx = rexsat (fun inp -> Str.string_match rx inp 0) 2;;
    let line_non rx = sat (fun inp -> not (Str.string_match rx inp 0)) ;;
    let line_is_txt txt = sat (fun inp -> txt = inp ) ;;
    let line_isnot_txt txt = sat (fun inp -> txt <> inp ) ;;

    let rxFuel           = Str.regexp "\\(.*  *\\)\\([0-9][0-9]*,[0-9][0-9]*\\)\\
( L  *[0-9][0-9]*,[0-9][0-9]* A\\)" ;;
    let rxVentureCode    = Str.regexp "\\(.*m\\.Kodas:  *\\)\\([0-9][0-9]*\\)" ;;
    let rxEndOfReceipt   = Str.regexp "20[0-9][0-9]-[0-9][0-9]-[0-9][0-9] [0-9]
[0-9]:[0-9][0-9]:[0-9][0-9] *\\(LTF \\)*QN 2[0-9][0-9][0-9][0-9][0-9][0-9]";;
    let rxComma          = Str.regexp "," ;;
    let rxStationName    = Str.regexp ".+" ;;
    let rxSpaces         = Str.regexp " *" ;;

    let fuelLine       = line_is rxFuel ;;
    let ventureCode    = line_is rxVentureCode ;;
    let endOfReceipt   = line_is rxEndOfReceipt ;;
    let comma str      = line_is rxComma ;;
    let startOfReceipt = line_is_txt "********** UAB" ;;
    let notStartOfReceipt = line_isnot_txt "********** UAB" ;;
    let stationName    = line_is rxStationName ;;
```

Main functions

- Receipt split and parse functions

```
let receipt =
        startOfReceipt >>= fun _ ->
        many (line_non rxEndOfReceipt) >>= fun lines ->
        endOfReceipt >>= fun _ -> return lines
    ;;

    let receipt_split_parser =
        many (notStartOfReceipt) >>= fun _ ->
        many receipt >>= fun receipts ->
        line_is rxSpaces >>= fun _ -> return receipts
    ;;

    let receipt_with_entcode =
        item >>= fun station_name ->
        many (line_non rxFuel)         >>= fun _ ->
        many (line_is  rxFuel)         >>= fun fuel_lines ->
        many (line_non rxVentureCode) >>= fun _ ->
        line_is  rxVentureCode         >>= fun vent_code ->
        return (station_name,fuel_lines,vent_code);;
```

Execution

- And finally, execute parser

```
# parse "fj.txt";;
- : (string * string list * string) list =
[("*** UAB", ["27,55"], "1231235xx");
 ("*** UAB", ["15,27"], "1239074xx");
 ("*** UAB", ["19,79"], "1237233xx");
 ("*** UAB", ["14,01"], "1234270xx");
 ("*** UAB", ["46,96"; "00,01"], "1236392xx")]
```

Liters from receipts

Venture code

Chapter 11

Coffee and cigarettes

Can we have monads in Java 7 ?

NO

We must !

Interfaces

```
public interface ContractApplicable<M extends Monad, T1, T2> {
    public Monad<M,T2> apply(T1 arg);
}


public interface Monad<M extends Monad,A> {
    public M returnn (A in);
    public <T> Monad<M,T> bind(ContractApplicable<M,A,T> f);
}
```

Implementation of Maybe

```
public abstract class MaybeMonad<A> implements Monad<MaybeMonad,A> {
    String message = null;
    A value = null;
    public A getValue() {
        return value;
    }
    public String getMessage() {
        return message;
    }
    .......
}
```

Some and None

```
public static class Some<A> extends MaybeMonad<A> {

        @Override
        public boolean hasFailed() {
            return false;
        }

        @Override
        public <T> Monad<MaybeMonad, T> bind(ContractApplicable<MaybeMonad, A, T> f) {
            return f.apply(value);
        }
    }

    public static MaybeMonad<?> NONE  = new MaybeMonad(null) {

        @Override
        public boolean hasFailed() {
            return true;
        }

        @Override
        public Monad bind(ContractApplicable f) {
            return this;
        }
    };
```

If Success, then apply next

If failed, then return NONE always

Usage

```
 @Test
    public void testMaybeMonad() {

        Monad<MaybeMonad,String> result =
                (new MaybeMonad.Some<>(5))
                .bind(MaybeApplicables.isOdd)
                .bind(MaybeApplicables.isGreaterThan(3))
                .bind(MaybeApplicables.toString);

        assertFalse("Maybe has failed with message " + ((MaybeMonad)
result).getMessage(),((MaybeMonad) result).hasFailed());

        System.out.println("done");
```

- Not real “functional” binding
- Done via dot operator
  - Once failed, will traverse all bind calls

Chapter 12

Burn after reading

Where to read more

- Google for
  - Wadler monads → Monads for functional programming
  - Wadler essence → “The essence of functional programming”
  - Erik Meijer channel 9 haskell

The end of affair