Discussion 16

Monads

Kenneth Fang (kwf37), Newton Ni (cn279)

March 25, 2019

Agenda

1. Review of Monads
2. Examples of Monads
   - Promises
   - Options
   - Error
   - Lazy
   - Non-Deterministic?
3. More Exercises

What is a Monad?

- Monads are a *design pattern* that pops up frequently in functional programming

What is a Monad?

- Monads are a *design pattern* that pops up frequently in functional programming
- A Monad is any module that satisfies the following interface:

```
module type Monad = sig
  type 'a t
  val return: 'a -> 'a t
  val bind: 'a t -> ('a -> 'b t) -> 'b t
end
```

What is a Monad?

- Monads are a *design pattern* that pops up frequently in functional programming
- A Monad is any module that satisfies the following interface:

```
module type Monad = sig
  type 'a t
  val return: 'a -> 'a t
  val (>>=): 'a t -> ('a -> 'b t) -> 'b t
end
```

- We often use the infix operator `(>>=)` for `bind`

What is a Monad?

```
module type Monad = sig
  type 'a t
  val return: 'a -> 'a t
  val (>>=): 'a t -> ('a -> 'b t) -> 'b t
end
```

- The `return` function lets a user lift a value into the Monad type

What is a Monad?

```
module type Monad = sig
  type 'a t
  val return: 'a -> 'a t
  val (>>=): 'a t -> ('a -> 'b t) -> 'b t
end
```

- The `return` function lets a user lift a value into the Monad type
- The `(>>=)` function does computation on values that are wrapped in the Monad type

Some Intuition

- The type `t` adds some extra computation to your functions

Some Intuition

- The type `t` adds some extra computation to your functions
- It wraps around a piece of data and gives it the ability to do more

Some Intuition

- The type `t` adds some extra computation to your functions
- It wraps around a piece of data and gives it the ability to do more
- The extra computation is done *implicitly*

Some Intuition

- The type `t` adds some extra computation to your functions
- It wraps around a piece of data and gives it the ability to do more
- The extra computation is done *implicitly*
- Compare `(>>=)` (bind) to `(|>)` (pipeline):

```
(|>): 'a -> ('a -> 'b) -> 'b
(>>=): 'a t -> ('a -> 'b t) -> 'b t
```

Example: Optional Monad

Open up today's exercises and try to implement this with your groupmates:

```
module Optional = struct
    type 'a t = 'a option
    let return a = failwith "Unimplemented"
    let (>>=) a f = failwith "Unimplemented"
end
```

Example: Optional Monad

Open up today's exercises and try to implement this with your groupmates:

```
module Optional = struct
    type 'a t = 'a option
    let return a = Some a
    let (>>=) a f =
      match a with
      | Some a -> f a
      | None -> None
end
```

Example: Error Monad

Open up today's exercises and try to implement this with your groupmates:

```
module Error = struct
    type 'a t = Error of string | Val of 'a
    let return a = failwith "Unimplemented"
    let (>>=) a f = failwith "Unimplemented"
end
```

Example: Error Monad

Open up today's exercises and try to implement this with your groupmates:

```
module Error = struct
    type 'a t = Error of string | Val of 'a
    let return a = Val a
    let (>>=) a f =
      match a with
      | Val a -> f a
      | Error s -> print_endline s; Error s
end
```

Example: Error Monad

Open up today's exercises and try to implement this with your groupmates:

```
module Error = struct
    type 'a t = Error | Val of 'a
    let return a = failwith "Unimplemented"
    let (>>=) a f = failwith "Unimplemented"
end
```

Example: Lazy Monad

Open up today's exercises and try to implement this with your groupmates:

```
type 'a t = unit -> 'a
let return a = failwith "Unimplemented"
let (>>=) a f = failwith "Unimplemented"

(** An extra function specific to Lazy
    [force l] forces the computation on l,
    immediately evaluating it to a value *)
let force (l: 'a t): 'a = failwith "Unimplemented"
```

Example: Lazy Monad

Open up today's exercises and try to implement this with your groupmates:

```
type 'a t = unit -> 'a
let return a = failwith "Unimplemented"
let (>>=) a f = failwith "Unimplemented"

(** An extra function specific to Lazy
    [force l] forces the computation on l,
    immediately evaluating it to a value *)
let force (l: 'a t): 'a = failwith "Unimplemented"
```

Example: Lazy Monad

Open up today's exercises and try to implement this with your groupmates:

```
type 'a t = unit -> 'a
let return a = fun () -> a
let (>>=) a f = f (a ())

(** An extra function specific to Lazy
    [force l] forces the computation on l,
    immediately evaluating it to a value *)
let force (l: 'a t): 'a = l ()
```

Example: Nondeterministic Monad???

Open up today's exercises and try to implement this with your groupmates:

```
module NonDeterministic = struct
    type 'a t = 'a list
    let return a = failwith "Unimplemented"
    let (>>=) a f = failwith "Unimplemented"
end
```

Example: Nondeterministic Monad???

Open up today's exercises and try to implement this with your groupmates:

```
module NonDeterministic = struct
    type 'a t = 'a list
    let return a = [a]
    let (>>=) a f = f (choose_random a)
end
```