



Haskell04

함수형 패러다임, 그 이상

차례


- 기억력 테스트
 - 훑아보기 - Review
- Haskell Upper-Intermediate
 - 잡다한 함수 기술들 - Function functions
 - 이불 덮어 씌우기 - Functor
 - 뺏었다가 줬다가 - Applicative Functor
 - 두 번 포장하기 - Monad

차례

- Haskell Upper-Intermediate
 - 참고, 어렵게 보기 - Functor, Applicative Functor, Monad
 - 재산세 내기 - Functor, Applicative, Monad Inheritance
 - 조심히 만들기 - Functor, Applicative, Monad Implementation
 - 조각조각 프로그래밍 - Module
 - 나갈 때 따로, 들어올 때 따로 - Main & I/O Type
 - 되로 주고 말로 받기 - I/O & do expression

차례

- Practices



기억력 테스트

아 기억이 안나잖아 안돼 이럴리가 없어

튜아보기

- `data Test a = Fault | Perfect a deriving (Eq, Show)`
`calcTest Fault = False`
`calcTest x = (x == (Perfect 3))`
- `calcTest`의 타입은?

튜아보기

- `data Test2 a = WrongTest a | RightTest {testData::[Test2 a]}`
- `testData`의 타입은?



Haskell Upper-Intermediate

내가 무릎을 꿇은 것은 추진력을 얻기 위함이었다!

잡다한 함수 기술들

Function functions

- 함수를 정의하는 법이 아니라, 이미 있는 함수를 잘 쓰고 싶다면 어떻게 해야할까?
- 이를테면, 이미 있는 함수 두 개를 차례대로 적용하고 싶다면 어떻게 해야할까?

잡다한 함수 기술들

Function functions

- map과 head를 차례대로 적용하는 방법
- 1. `head (map f l)`
- 2. `head . map f $ l`

잡다한 함수 기술들

Function functions

- map과 head를 차례대로 적용하는 방법
- 1. head (map f l)
 - 우리가 알고있는 일반적인 방법이다.
 - map f l의 결과값을 head에 적용한다.

잡다한 함수 기술들

Function functions

- map과 head를 차례대로 적용하는 방법
- 2. head . map f \$ l
 - (.)과 (\$)를 사용하는 방법이다.

잡다한 함수 기술들

Function functions

- (.)? (\$)?
- (.)는 함수 두 개를 합성하는 연산자이다.
- (\$)는 함수에 값을 적용하는 연산자이다.

잡다한 함수 기술들

Function functions

- 함수를 합성하는 연산자가 왜 따로 필요할까?
- `head map f`과 같이 쓰면 `head`의 인자로 `map`이 넘어가게 된다.
- 타입이 맞지 않기 때문에 오류, 맞는다 하더라도 이상한 결과값을 얻게 된다.
- 우리가 원하는 것은 `head`의 인자로 `map f` 함수의 결과 값이 넘어가는 것이다!

잡다한 함수 기술들

Function functions

- $(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$
- 함수 두 개를 받아서 한 함수의 결과값이 다음 함수에 들어갈 수 있게 해준다.

잡다한 함수 기술들

Function functions

- 그러면 (\$)는 왜 필요할까?
- `head . map f l`을 시도하면 함수 적용은 우선순위가 굉장히 높기 때문에 `map f l`이 먼저 계산되고, 그 결과값을 `head`와 합성하려고 시도하게 된다.
- 우리가 원하는 것은 `head`와 `map f`를 합성하는 것이지, `map f l`과 합성하는 것이 아니다!

잡다한 함수 기술들

Function functions

- $(\$)\ ::\ (a \rightarrow b) \rightarrow a \rightarrow b$
- 굉장히 우선순위가 낮은 함수 적용
- `map f l1 ++ l2` 와 같이 쓰면 `l1`에 `map`이 적용된 결과와 `l2`가 더해진다.
- `map f $ l1 ++ l2` 와 같이 쓰면 `l1`과 `l2`가 더해진 결과에 `map`이 적용된다.

잡다한 함수 기술들

Function functions

- `listNegSumList1 l = map (\x -> negate (sum x)) l`
- `listNegSumList2 l = map (negate . sum) l`
- `oddSquareSum1 x = sum (map (^2) (filter odd (enumFromTo 1 x)))`
- `oddSquareSum2 = sum . map (^2) . filter odd . enumFromTo 1`

이불 덮어 씌우기 Functor

- $l = [1,2,3]$ 의 내용물에 (2^*) 를 하고 싶다면 어떻게 해야할까?
- `map (2*) l`

이불 덮어 씌우기 Functor

- `l = Just 5`의 내용물에 (2^*) 를 하고 싶다면 어떻게 해야할까?
- `case l of`
 `Nothing -> Nothing`
 `Just x -> Just (2*x)`
- `Maybe`에 `map` 같은 함수를 만들어서 아래처럼 할 수는 없을까?
- `mapLike (2*) l`

이불 덮어 씌우기 Functor

- `[a]`, `Maybe a`, `Either t a`, ... 이렇게 **Type**을 인자로 받는 **Type**들, 즉 **Wrapper**들한테 `a`가 할 수 있는 연산을 할 수는 없을까?

이불 덮어 씌우기 Functor

- $\text{fmap} :: (\text{Functor } f) \Rightarrow (a \rightarrow b) \rightarrow f a \rightarrow f b$
- Functor Class가 되기 위한 Minimal Complete Definition
- $a \rightarrow b$ 인 함수를 $f a \rightarrow f b$ 인 함수로 바꿔준다.
- $\text{fmap } (2^*) :: (\text{Functor } f) \Rightarrow f a \rightarrow f b$
- $\text{fmap } (2^*) [1,2,3]$
- $\text{fmap } (2^*) (\text{Just } 5)$

이불 덮어 씌우기 Functor

- Functor의 동작을 모형으로 살펴보자

이불 덮어 씌우기 Functor

*2

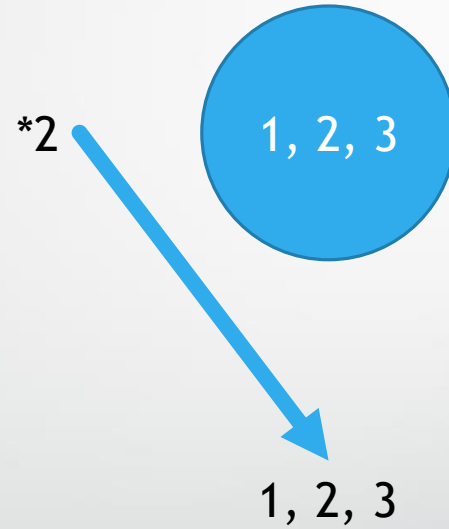


이불 덮어 씌우기 Functor

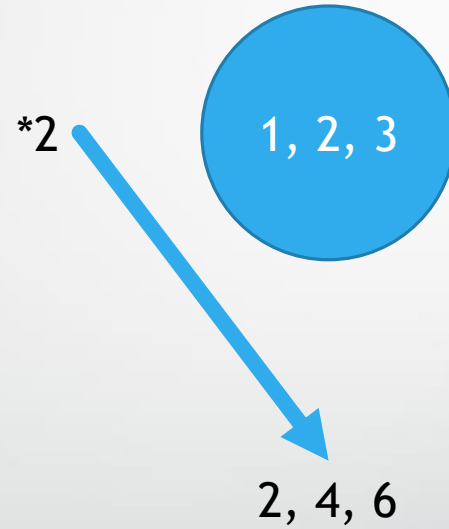
*2



이불 덮어 씌우기 Functor



이불 덮어 씌우기 Functor



이불 덮어 씌우기 Functor

*2



이불 덮어 씌우기 Functor

- 이제 **Pattern Matching**을 복잡하게 하지 않고서도 함수들을 **Wrapping**된 값에 적용할 수 있게 되었다.
- 하지만 여전히 문제는 남아있다!

뺏었다가 줬다가 Applicative Functor

- `fl = [(2*), (3*), (4*)]` 같은 함수들의 List를 `vl = [1, 2, 3, 4]`와 같은 값의 List에 적용하고 싶다고 해보자.
- `case fl of`
 `f1:f2:f3:[] ->`
 `(fmap f1 vl) ++ (fmap f2 vl) ++ (fmap f3 vl)`
- 함수의 개수가 정해져 있지 않다면? 재귀함수!

뺏었다가 줬다가 Applicative Functor

- No no no
- 모든 Wrapper마다 이런 함수를 일일이 쓰는 것은 불편하다.
- 즉, Wrapping된 함수를 Wrapping된 값들에 사용할 수 있는 간편한 방법은 없을까?

뺏었다가 줬다가 Applicative Functor

- $\text{pure} :: (\text{Applicative } f) \Rightarrow a \rightarrow f\ a$
- $(\langle * \rangle) :: (\text{Applicative } f) \Rightarrow f\ (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$
- Applicative Class가 되기 위한 Minimal Complete Definition
- pure는 임의의 값을 Wrapping 해주고,
- $(\langle * \rangle)$ 는 $f\ (a \rightarrow b)$ 인 Wrapping된 함수를 $f\ a \rightarrow f\ b$ 인 함수로 바꿔준다

뺏었다가 줬다가 Applicative Functor

- `pure (*2) <*> [1,2,3]`
- `[(*2)] <*> [1,2,3]`
- `[] <*> [1,2,3]`
- `[(*2), (*3)] <*> [1,2,3]`
- `Just (*3) <*> Just 5`

뺏었다가 줬다가 Applicative Functor

- Applicative Functor의 동작을 모형으로 살펴보자

뺏었다가 줬다가 Applicative Functor

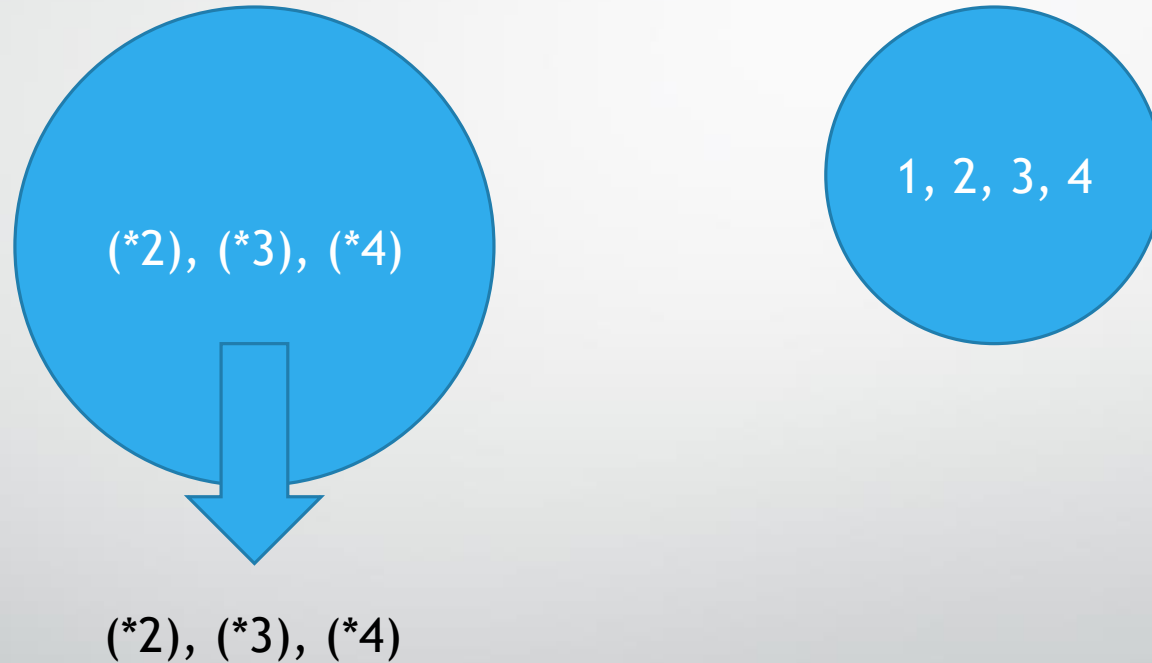


$(*2), (*3), (*4)$

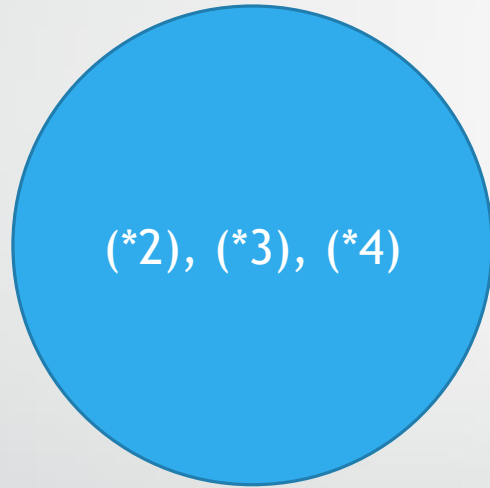


1, 2, 3, 4

뺏었다가 줬다가 Applicative Functor



뺏었다가 줬다가 Applicative Functor

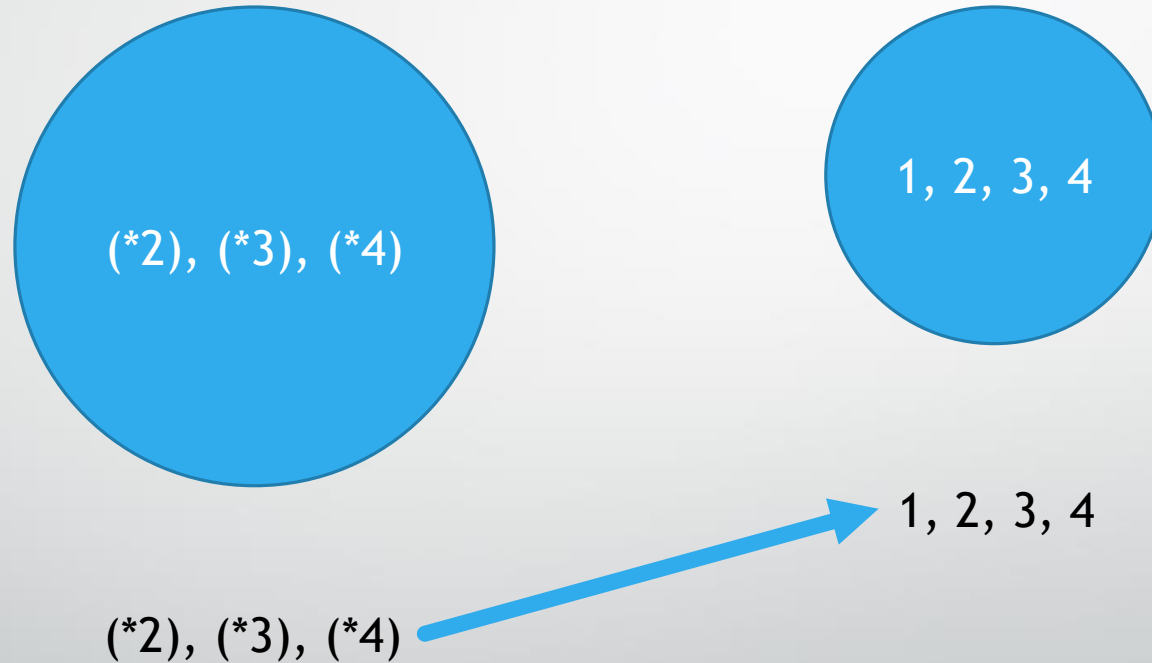


$(*2), (*3), (*4)$

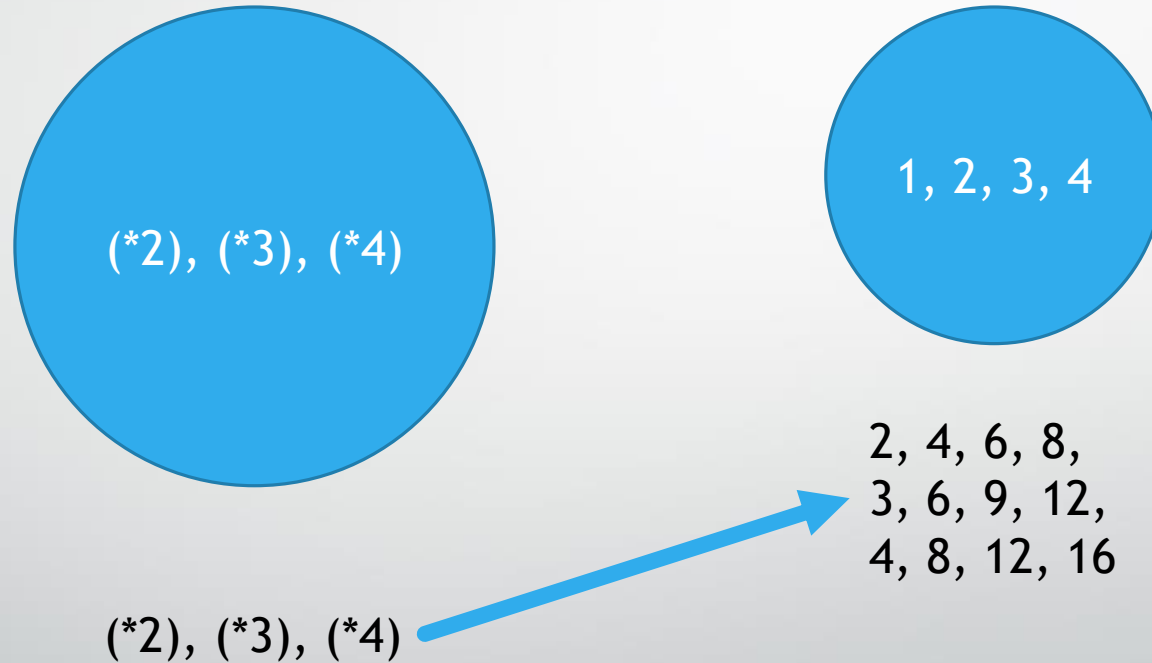


$1, 2, 3, 4$

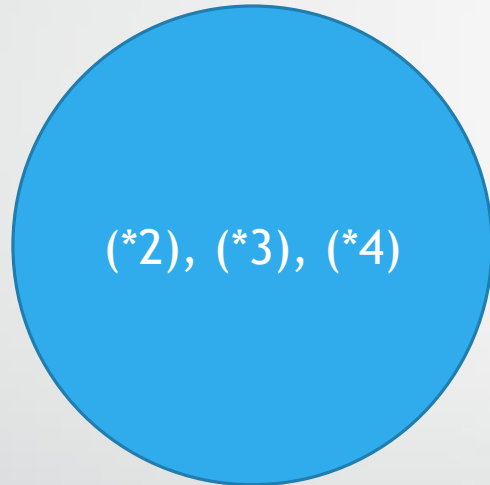
뺏었다가 줬다가 Applicative Functor



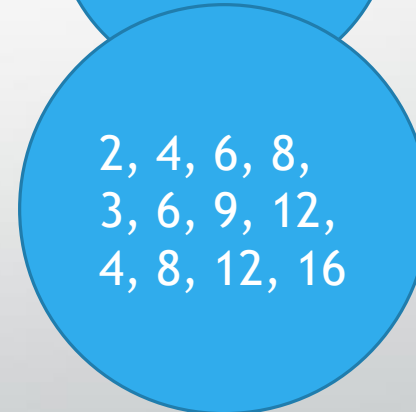
뺏었다가 줬다가 Applicative Functor



뺏었다가 줬다가 Applicative Functor



(*2), (*3), (*4)



뺏었다가 줬다가 Applicative Functor

- Pattern Matching 없이도 Wrapping된 함수를 Wrapping된 값들에게 적용할 수 있게 되었다.
- 이젠 다 해결되었을까?

두 번 포장하기

Monad

- 우리가 다음 같은 Factorial 함수를 만들었다고 하자.
- `factorial :: (Num a, Ord a) => a -> Maybe a`
`factorial n | n < 0 = Nothing`
`| n == 0 = Just 1`
`| otherwise = fmap (*n) (factorial (n-1))`

두 번 포장하기

Monad

- `mv = Just 5`에 `factorial`을 적용하고 싶다면 어떻게 하면 될까?
- 1. `fmap factorial mv`
- 2. `pure factorial <*> mv`
- 3. ???

두 번 포장하기

Monad

- `mv = Just 5`에 `factorial`을 적용하고 싶다면 어떻게 하면 될까?
- 1. `fmap factorial mv`
- `fmap :: (Functor f) => (a -> b) -> f a -> f b`
- `factorial :: (Num a, Ord a) => a -> Maybe a`
- `(fmap factorial) :: (Functor f, Num a, Ord a) => f a -> f (Maybe a)`

두 번 포장하기

Monad

- `mv = Just 5`에 `factorial`을 적용하고 싶다면 어떻게 하면 될까?
- 1. `fmap factorial mv`
- `f` Type은 뭐가 되지?
- `f (Maybe a)`가 정말로 원하는 결과인가?

두 번 포장하기

Monad

- `mv = Just 5`에 `factorial`을 적용하고 싶다면 어떻게 하면 될까?
- 2. `pure factorial <*> mv`
- `(<*>) :: (Applicative f) => f (a -> b) -> f a -> f b`
- `pure :: (Applicative f) => a -> f a`
- `(pure factorial <*>) :: (Applicative f, Num a, Ord a) => f a -> f (Maybe a)`

두 번 포장하기 Monad

- `mv = Just 5`에 `factorial`을 적용하고 싶다면 어떻게 하면 될까?
- 2. `pure factorial <*> mv`
- 1번과 똑같은 결과를 불러온다!

두 번 포장하기

Monad

- `mv = Just 5`에 `factorial`을 적용하고 싶다면 어떻게 하면 될까?
- 3. ???
- Pattern Matching을 통해 `Just 5`에서 5를 꺼내 오는 수밖에 없을까??
- No!

두 번 포장하기

Monad

- `return :: (Monad m) => a -> m a`
- `(>>=) :: (Monad m) => m a -> (a -> m b) -> m b`
- Monad Class가 되기 위한 Minimal Complete Definition
- `return`은 임의의 값을 Wrapping해 주고,
- `(>>=)` 연산(bind라고 읽는다)은 Wrapping하는 함수를 Wrapping된 값에 적용할 수 있게 해준다.

두 번 포장하기

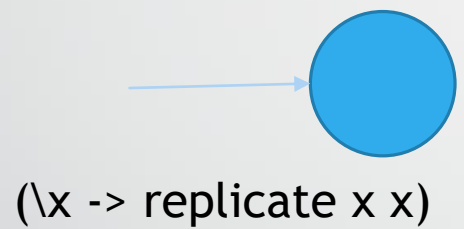
Monad

- Just 5 >>= factorial
- [1,2,3] >>= (\x -> replicate x x)
- getLine >>= (\x -> (read x) + 1)
- ...

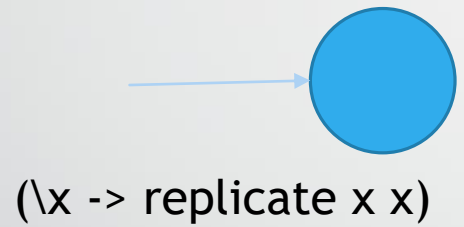
두 번 포장하기 Monad

- Monad의 동작을 모형으로 살펴보자

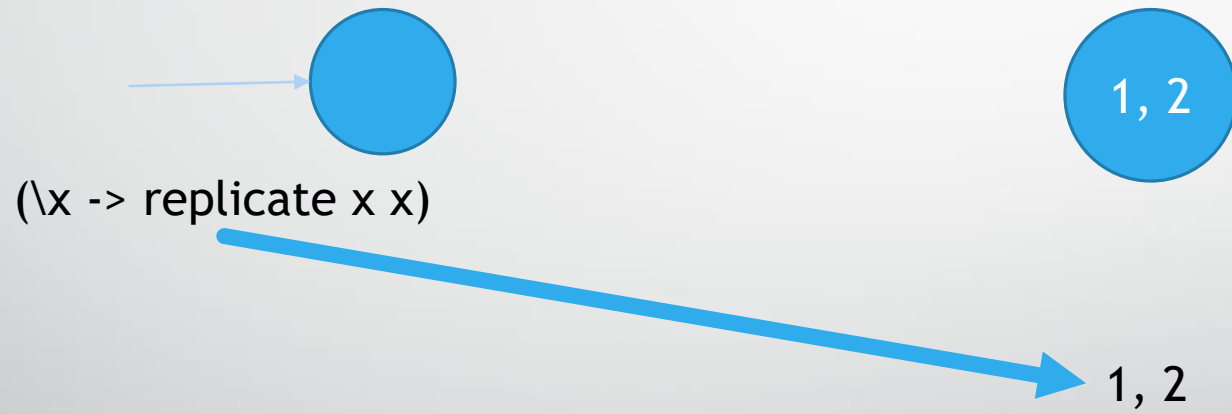
두 번 포장하기 Monad



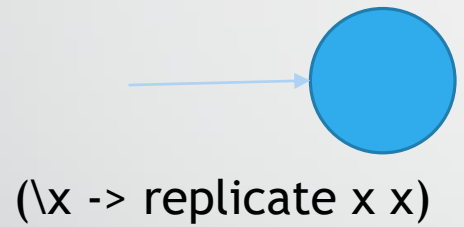
두 번 포장하기 Monad



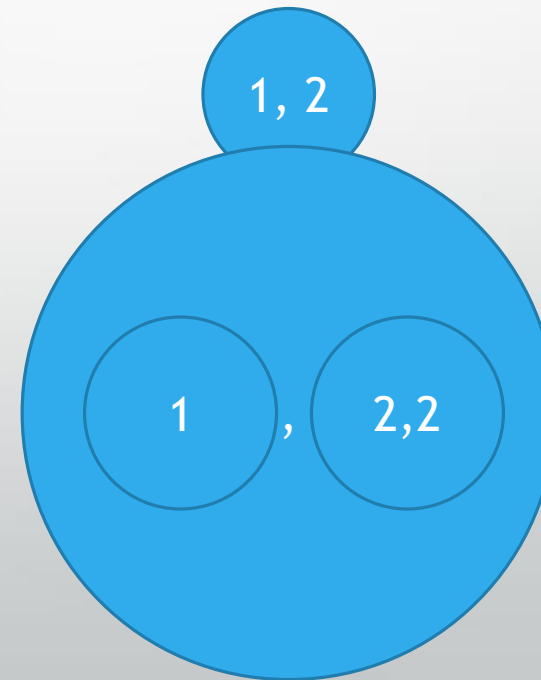
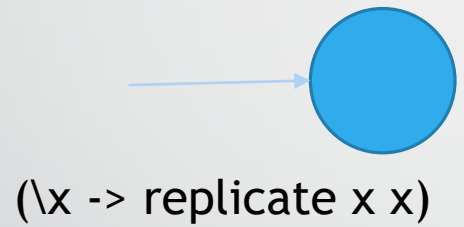
두 번 포장하기 Monad



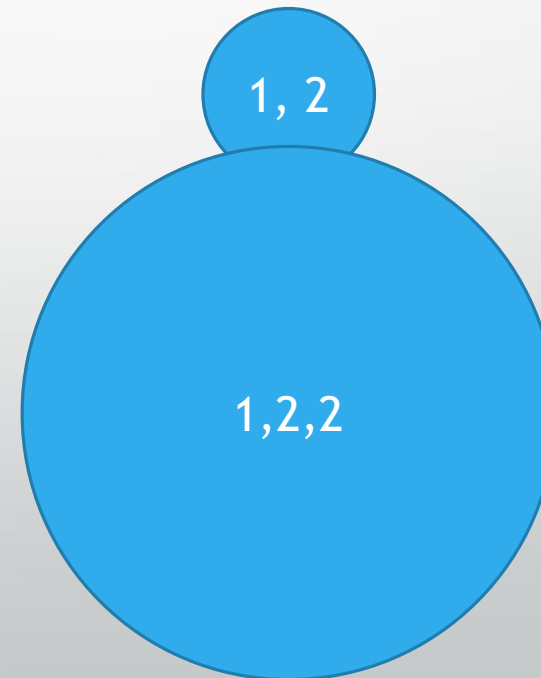
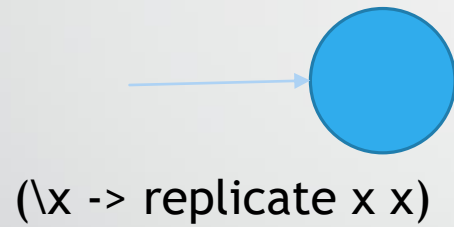
두 번 포장하기 Monad



두 번 포장하기 Monad



두 번 포장하기 Monad



뺏었다가 줬다가 Applicative Functor

- Pattern Matching 없이도 Wrapping된 함수를 Wrapping된 값들에게 적용할 수 있게 되었다.
- 여기다 더해서?
 - Foldable
 - Traversable
 - ...

뺏었다가 줬다가 Applicative Functor

- Pattern Matching 없이도 Wrapping된 함수를 Wrapping된 값들에게 적용할 수 있게 되었다.
- 여기다 더해서?
 - Foldable
 - Traversable
 - ...

참고, 어렵게 보기

Functor, Applicative Functor, Monad

- Functor와 Applicative, Monad Type Class에 대해서 다뤄보았다.
- 이 세 Type Class에는 특징적인 함수가 하나씩 있다.
- fmap
- ($\langle * \rangle$)
- ($\gg =$)

참고, 어렵게 보기

Functor, Applicative Functor, Monad

- Functor와 Applicative, Monad Type Class에 대해서 다뤄보았다.
- 이 세 Type Class에는 특징적인 함수가 하나씩 있다.
- $\text{fmap} :: (\text{Functor } f) \Rightarrow (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$
- $\langle * \rangle :: (\text{Applicative } f) \Rightarrow f\ (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$
- $\text{(>=>) } :: (\text{Monad } m) \Rightarrow m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$

참고, 어렵게 보기

Functor, Applicative Functor, Monad

- Functor와 Applicative, Monad Type Class에 대해서 다뤄보았다.
- 이 세 Type Class에는 특징적인 함수가 하나씩 있다.
- ($>>=$)가 조금 헷갈릴 수 있으므로 Control.Monad 모듈에 있는 다른 함수로 잠깐 설명하겠다
- $(=<<) :: (Monad\ m) \Rightarrow (a \rightarrow m\ b) \rightarrow m\ a \rightarrow m\ b$
 $(=<<) a\ b = (>>=) b\ a$

참고, 어렵게 보기

Functor, Applicative Functor, Monad

- Functor와 Applicative, Monad Type Class 가 가진 특징적인 함수들의 성질에 대해 살펴보자.
- `fmap`은 `(a -> b)`를 받아서 `f a` 에서 `f b`로 가는 함수를 만든다. 이때 `f a`의 `f`에 따라서 결과의 `f`가 결정된다.
- `fmap`이 `List`에 사용될 때에는 `List`의 길이가 변하지 않는다.
- `fmap`이 `Maybe`에 사용될 때에는 `Just x`를 `Nothing`으로, `Nothing`을 `Just x`로 바꾸지 않는다.

참고, 어렵게 보기

Functor, Applicative Functor, Monad

- Functor와 Applicative, Monad Type Class 가 가진 특징적인 함수들의 성질에 대해 살펴보자.
- ($\<*\>$)는 $f (a \rightarrow b)$ 를 받아서 $f\ a$ 에서 $f\ b$ 로 가는 함수를 만든다. 이때, $f (a \rightarrow b)$ 의 f 와 $f\ a$ 의 f 에 따라서 결과의 f 가 결정된다.
- ($\<*\>$)를 `List`에 적용하면, 함수 `List`의 길이에 따라 값 `List`의 길이가 변한다.
- ($\<*\>$)를 `Maybe`에 적용하면, 함수 `Maybe`에 따라 값 `Maybe`가 바뀐다.

참고, 어렵게 보기

Functor, Applicative Functor, Monad

- Functor와 Applicative, Monad Type Class 가 가진 특징적인 함수들의 성질에 대해 살펴보자.
- $(= < <)$ 는 $(a \rightarrow m\ b)$ 를 받아 $m\ a$ 에서 $m\ b$ 로 가는 함수를 만든다. 이때, a 의 값과 $m\ a$ 의 m 에 따라서 결과의 m 이 결정된다.
- $(= < <)$ 이 `List`에 적용될 때, `List`의 길이가 자유롭게 변할 수 있다.
- $(= < <)$ 이 `Maybe`에 적용될 때, 어떤 `Maybe`든지 출현할 수 있다.

참고, 어렵게 보기

Functor, Applicative Functor, Monad

- Functor, Applicative, Monad로 갈수록 Wrapper를 더 자유롭게 변경할 힘이 주어진다.
- 필요한 만큼의 자유도만 사용하는 것이 중요하다!

재산세 내기

Functor, Applicative, Monad Inheritance

- Eq Class와 Ord Class 처럼 Functor, Applicative, Monad 간에는 상속 관계가 존재한다.
- Ord이면 무조건 Eq인 것처럼
- Monad이면 무조건 Applicative, Applicative면 무조건 Functor이다.

재산세 내기

Functor, Applicative, Monad Inheritance

- 이 상속 관계를 검증해보자.
- Ord 는 $(a == b) = (a >= b) \ \&\& \ (b >= a)$ 이기 때문에 Eq 였다.

재산세 내기


Functor, Applicative, Monad Inheritance

- 마찬가지로 Functor와 Applicative의 상속 관계를 보면
- Functor의 Minimal Complete Definition은 fmap이다.
- fmap을 pure와 <*>로 구현해보면
- $\text{fmap } f \ v = \text{pure } f \ \langle * \rangle \ v$

재산세 내기

Functor, Applicative, Monad Inheritance

- 마찬가지로 Applicative와 Monad의 상속 관계를 보면
- Applicative의 Minimal Complete Definition은 `pure`, `(<*>)`이다.
- `pure`와 `(<*>)`를 `return`과 `(>>=)`로 구현해보면
- `pure = return`
- `f <*> v = v >>= return . f`



재산세 내기

Functor, Applicative, Monad Inheritance

- 따라서 상속 관계가 성립한다.

조심히 만들기

Functor, Applicative, Monad Implementation

- 이미 있는 Functor, Applicative, Monad를 쓸 때에는 상관 없으나
- 새로운 Type이 Functor, Applicative, Monad를 구현할 때에는 주의할 점이 있다.
- 바로 Functor 규칙, Applicative 규칙, Monad 규칙이라는 규칙들이다.

조심히 만들기

Functor, Applicative, Monad Implementation

- Functor 규칙
- $\text{fmap id} == \text{id}$
- $\text{fmap (f.g)} == \text{fmap f} . \text{fmap g}$

조심히 만들기

Functor, Applicative, Monad Implementation

- Applicative 규칙
- $\text{pure id} \langle * \rangle v == v$
- $\text{pure } (.) \langle * \rangle u \langle * \rangle v \langle * \rangle w == u \langle * \rangle (v \langle * \rangle w)$
- $\text{pure } f \langle * \rangle \text{pure } x == \text{pure } (f\ x)$
- $u \langle * \rangle \text{pure } y == \text{pure } (\$ y) \langle * \rangle u$

조심히 만들기

Functor, Applicative, Monad Implementation

- Monad 규칙
- $\text{return } a \gg= k == k \ a$
- $m \gg= \text{return} == m$
- $m \gg= (\lambda x \rightarrow k \ x \gg= h) == (m \gg= k) \gg= h$

조심히 만들기

Functor, Applicative, Monad Implementation

- 자연스러운 규칙이기 때문에 복잡한 타입을 만들 때가 아니면 신경 쓸 필요가 없지만, 복잡한 타입의 경우에는 조금 신경을 써 주어야 한다.

조심히 만들기

Functor, Applicative, Monad Implementation

- 자연스러운 규칙이기 때문에 복잡한 타입을 만들 때가 아니면 신경 쓸 필요가 없지만, 복잡한 타입의 경우에는 조금 신경을 써 주어야 한다.

조심히 만들기

Functor, Applicative, Monad Implementation

- Quiz!
- 다음과 같은 `study`와 `progress`함수가 있다고 해보자.
- ```
type StudySt = Int
type ProgressSt = Int
type LectureSt = (StudySt, ProgressSt)
study :: LectureSt -> StudySt -> Maybe LectureSt
progress :: LectureSt -> ProgressSt -> Maybe LectureSt
```

# 조심히 만들기

## Functor, Applicative, Monad Implementation

- Quiz!
- 학생이 공부를 할 때마다 **study** 함수를 부르고, 수업이 진행될 때마다 **progress** 함수가 불린다고 하자. 다만, **study**와 **progress**의 차이가 5 이상 벌어지면 학생이 수업을 드랍하기 때문에 두 함수는 **Nothing**을 리턴하게 된다고 하자.
- 3 수업, 2 공부, 3 수업, 4 공부, 3 수업, 1 공부, 1공부, 3 수업, 2 공부, 3 수업, 4 공부와 같은 시간표를 지켰을 때 학생이 드랍했는지 아닌지 판별하는 코드를 짜보자.

# 조심히 만들기

## Functor, Applicative, Monad Implementation

- Quiz!
- `print (return (0,0) >>= lecture 3 >>= study 2 >>= lecture 3 >>= study 4 >>= lecture 3 >>= study 1 >>= study 1 >>= lecture 3 >>= study 2 >>= lecture 3 >>= study 4)`



# 조각조각 프로그래밍 Module

- 무슨 프로그래밍이던지 간에 적당한 크기로 **Module**을 구성하는 것은 중요하다.
- 이미 구성된 표준 라이브러리 **Module**의 사용법을 아는 것 또한 매우 중요하다.

# 조각조각 프로그래밍 Module

- **Prelude**를 넘어서 모듈들을 사용하는 법을 알아보고, 직접 모듈을 만들어보자.

# 조각조각 프로그래밍 Module

- Import 문
- import 이름
  - 이름에 해당하는 Module을 불러온다.

# 조각조각 프로그래밍 Module

- Qualifier Import 문
- import qualified 이름
  - 이름.쓰고싶은정의로 모듈 안의 것들을 쓸 수 있게 해준다.
  - 함수 이름의 중복을 막기 위해 쓴다.

# 조각조각 프로그래밍 Module

- Import as 문
- import 이름 as 새이름
  - 이름에 해당하는 Module을 불러오고, Module에 새이름이라는 별명을 지어준다.

# 조각조각 프로그래밍 Module

- 부분 Import 문
- import 이름 (가져올 정의들)
  - 가져올 정의들에 해당하는 정의들만 Module에서 가져온다.
- import 이름 hiding (숨길 정의들)
  - 숨길 정의들 o에 해당하는 정의들은 Module에서 가져오지 않는다.

# 조각조각 프로그래밍 Module

- Import 문
- `import qualified 이름 as 새이름` (가져올 정의들)
  - 가장 일반적인 `import` 문 중 하나
  - 새이름.가져온정의로 정의들을 사용할 수 있다.
- `import qualified 이름 as 새이름 hiding` (숨길 정의들)
  - 가장 일반적인 `import` 문 중 하나

# 조각조각 프로그래밍 Module

- `import Data.List`  
`v = permutation [1,2,3]`
- `import qualified Data.List`  
`v = Data.List.permutation [1,2,3]`
- `import qualified Data.List as L`  
`v = L.permutation [1,2,3]`



# 조각조각 프로그래밍 Module

- Module을 불러와서 사용하는 방법에 대해서 알아보았다.
- Module을 정의하는 방법은 무엇일까?

# 조각조각 프로그래밍 Module

- 기본적인 `Module` 정의
- `module` 이름 `where`  
...
- 이름이라는 `Module`을 정의한다.
- 이름은 대문자로 시작하여야하고, 파일 명과 동일해야한다.

# 조각조각 프로그래밍 Module

- 한정적 **Module** 정의
- **module** 이름 (외부에 노출할 정의들) **where**  
...
- 이름이라는 모듈을 정의하고, 일부 정의들만 노출한다.
- 이름은 대문자로만 시작하여야하고, 파일명과 동일해야 한다.

# 조각조각 프로그래밍 Module

- 계층적 Module 정의
- module 폴더명.이름 where
- ...
- 여러 모듈들을 하나의 폴더 안에 묶어서 정의할 수 있다.
- 폴더명과 이름은 대문자로만 시작하여야하고, 실제 폴더 명, 파일 명과 동일해야 한다.

# 조각조각 프로그래밍 Module

- Module 사용 예시 (Fibonacci.hs)
- `module Fibonacci (fibonacci) where`  
`fib a b = a : (fib b (a+b))`  
`fibonacci n = (fib 1 1)!!n`

# 조각조각 프로그래밍 Module

- Module 사용 예시 (OtherModule1.hs)
- `module OtherModule1 where`  
`import Fibonacci`  
  
`useFibonacci = fibonacci`

# 조각조각 프로그래밍 Module

- Module 사용 예시 (OtherModule2.hs)
- `module OtherModule2 where`  
`import Fibonacci`

`useFibonacci n = (fib 1 1) !! n`

# 조각조각 프로그래밍 Module

- Load해보면 OtherModule2는 에러가 발생하는 것을 확인할 수 있다.



# 나갈 때 따로, 들어올 때 따로 Main & I/O Type

- 이제 컴파일해서 프로그램을 만드는 법에 대해서 알아보자.
- Main Module과 main함수에 의해서 프로그램을 만들 수 있다.

# 나갈 때 따로, 들어올 때 따로 Main & I/O Type

- 적당한 파일 (HelloWorld.hs)을 하나 만들자.

# 나갈 때 따로, 들어올 때 따로 Main & I/O Type

- 우선 Main module이 있어야한다.
- module Main where

# 나갈 때 따로, 들어올 때 따로 Main & I/O Type

- Module 안에 main이라는 녀석의 정의가 있어야 한다.
- `module Main where`  
`main = putStrLn "hello world!"`

# 조각조각 프로그래밍 Module

- 소스파일이 있는 폴더에 cmd 창 (terminal 창)을 연 뒤,
- `ghc HelloWorld.hs`
- 를 치면 `HelloWorld.exe` (또는 `HelloWorld`)이라는 프로그램이 생성된다.

# 나갈 때 따로, 들어올 때 따로 Main & I/O Type

- 마침내 Hello World를 출력해보았다!
- 그런데 대체 `putStrLn`은 무엇일까? `main`이라는 녀석은 무슨 타입인걸까?

# 나갈 때 따로, 들어올 때 따로 Main & I/O Type

- `putStrLn :: String -> IO ()`
- `main :: IO ()`

# 나갈 때 따로, 들어올 때 따로 Main & I/O Type

- IO?
  - 새로운 Wrapper!
- IO a는 Input/Output을 수행한 결과값이 a Type임을 이야기한다.
- IO는 Data Constructor를 외부로 노출하지 않는다!
  - Pattern Matching으로 IO를 사용할 수 없다!



# 나갈 때 따로, 들어올 때 따로 Main & I/O Type

- ()?
  - C의 void와 같은 Type
  - 아무것도 아닌 Type을 이야기한다.

# 나갈 때 따로, 들어올 때 따로 Main & I/O Type

- `putStrLn :: String -> IO ()`
- `String`을 받아서 `IO`를 수행하고, 그 결과값은 없다.
- `main :: IO ()`
- `IO`를 수행하고, 그 결과값은 없다.

# 나갈 때 따로, 들어올 때 따로 Main & I/O Type

- 입력은?
- `getLine :: IO String`
- IO를 수행하고, 그 결과값은 `String`이다.

# 나갈 때 따로, 들어올 때 따로 Main & I/O Type

- 결과값을 도대체 어떻게 쓸 수 있을까?
  - Pattern Matching은 할 수 없다 (생성자가 모두 숨겨져 있다.)
  - IO 또한 Wrapper Type이다.
- Monad!!

# 나갈 때 따로, 들어올 때 따로 Main & I/O Type

- `main = getLine >>= (\x -> putStrLn x)`
- 간단한 Mirroring program이다.

# 되로 주고 말로 받기 I/O & do expression

- 만일 `main` 함수가 길어진다면 어떻게 될까?
- 물론 함수를 쪼개면 `main` 함수를 짧게 만들 수 있다.
- 하지만 간단한 프로그램을 만들면서 함수를 여러 개 이름 붙이기란 쉬운 일이 아니다.

# 되로 주고 말로 받기 I/O & do expression

- 만일 `main` 함수가 길어진다면 어떻게 될까?
- `... >>= (\x -> ... >>= (\y -> ... >>= ...))`
- `>>=`가 정신없이 나열되는 모습을 보게 될 것이다.
- 더럽다!

# 되로 주고 말로 받기 I/O & do expression

- Monad를 보다 손쉽게 쓸 수 있는 방법은 없을까?
- do expression!



# 되로 주고 말로 받기 I/O & do expression

- do  
    a  
    b
- a >> b

# 되로 주고 말로 받기 I/O & do expression

- do  
    x <- a  
    b
- a >>= (\x -> b)

# 되로 주고 말로 받기 I/O & do expression

- Monad를 사용한 연산들을 보다 읽기 쉽게 바꿔준다.

# 되로 주고 말로 받기 I/O & do expression

- Fibonacci 수열을 계산해주는 프로그램
- module Main where  
import Fibonacci

```
main = do
 x <- getLine
 putStrLn . show . fibonacci . read $ x
```

# 되로 주고 말로 받기

## I/O & do expression

- 주의! `do expression` 중간에 `return`이 등장한다고 해도 거기서 수행이 끝나지 않는다.
  - `return x`는 단순히 결과값이 `x`인 `Monad`를 만드는 함수일 뿐이다.
  - `x :: a`일 때, `return x`는 `IO a`일 뿐이다.

# Practices

- 1. 다음과 같은 명령어를 지원하는 DB를 만들어보자
- `data Command = CInput {inputKey::Int, inputValue::Int}`  
    `| CFind {findKey::Int}`
- 이 Data Type에게 deriving해 줄 Class들을 생각해보자.
- main의 재귀를 생각해보자.

# Practices

- 2. 간단한 언어를 만들어보자.
- `data Expr =`  
    `EInt Int | EAdd Expr Expr | EMinus Expr Expr`  
    `| EMult Expr Expr | EDiv Expr Expr | ERem Expr Expr`
- `eval :: Expr -> Maybe Int`

# Practices

- 2. 간단한 언어를 만들어보자.
- `eval (EAdd (EInt 5) (EMult (EInt 5) (EMinus (EInt 3) (EDiv (EInt 4) (EInt 3)))))) == Just 15`
- `eval (EMult (EInt 3) (ERem (EInt 3) (EInt 0))) == Nothing`



# Practices

- 2. 간단한 언어를 만들어보자.
- deriving Read와 간단한 main 함수를 사용하면 파일을 읽어서 실행하는 것도 가능하다.