



Haskell03


함수형 패러다임, 그 이상

차례

- 기억을 찾아서
 - 나는 네가 지난 일요일에 한 일을 알고 있다 - Review
- Haskell Intermediate
 - 모양 1반 모여라! - Type Class Basic
 - 너는 내 타입이 아닌걸 - User Define Type
 - 전학생 맞아보기 - Type Class Implementation

차례

- Haskell Intermediate
 - Type 뒤에 Class 있어요, Class 있다구요! - Type Class Definition
 - 함수를 계승 중입니다, 아버지 - Type Class Inheritance



기억을 찾아서

금붕어 기억력이 읊니다

나는 네가 지난 일요일에 한 일을 알고 있다

- Function & Constant Bindings
- Currying & Purity
- Type Basic & Complex Types
- Pattern Matching and Others
- First Class Object
- Lambda Function

나는 네가 지난 일요일에 한 일을 알고 있다

- Function & Constant Bindings
- GHCi에서
 - `let <이름> = <정의>`
 - `let <이름> <인자1> <인자2> ... = <정의>`
- 소스파일에서
 - `<이름> = <정의>`
 - `<이름> <인자1> <인자2> ... = <정의>`

나는 네가 지난 일요일에 한 일을 알고 있다

- Currying & Purity
- Currying
 - 모든 다변수 함수는 일변수 함수 여러 개랑 동일하다!
- Purity
 - 모든 함수는 인자에만 영향을 받고, 결과에만 영향을 준다!

나는 네가 지난 일요일에 한 일을 알고 있다

- Type Basic & Complex Types
- `:t`
 - 타입 확인하기
- `::`
 - 타입 나타내기
- List & Tuple

나는 네가 지난 일요일에 한 일을 알고 있다

- Pattern Matching and Others
- 조각 함수 정의 Piecewise Function Definition
 - $\text{fun } 0 = 1$
 $\text{fun } n = 3$
- 가드 Guard
 - $\text{fun } n \mid n < 0 = 1$
 $\mid n \geq 0 = 2$

나는 네가 지난 일요일에 한 일을 알고 있다

- Pattern Matching and Others
- case ~ of expression
- let ~ in expression
- where statement

나는 네가 지난 일요일에 한 일을 알고 있다

- First Class Object
- 그 언어에서 객체가 할 수 있는 모든 행동을 할 수 있는 것
 - 함수에 인자로 넘기기, 함수의 결과값으로 돌려주기, 변수로 사용하기, ...
- Haskell에서
 - 상수와 일변수 함수들!
 - map 등의 함수에서 사용하는 걸 볼 수 있다.

나는 네가 지난 일요일에 한 일을 알고 있다

- Lambda Function
- 이름 없는 함수
 - $\backslash \langle \text{인자1} \rangle \langle \text{인자2} \rangle \dots \rightarrow \langle \text{정의} \rangle$



Haskell Intermediate

이게! 모나드도 모르는 게 까불어!

모양 1반 모여라!

Type Class Basic

- 지난 번 **Type**을 확인하는 법을 다루면서 **Num a**나 **Eq a**같은 기본적인 **Type Class**들을 잠깐 보았었다.
- 구체적으로 **Type Class**가 무엇인지 살펴보도록 하자.

모양 1반 모여라!

Type Class Basic

- Type Class
 - 공통적인 함수들을 가지고 있는 **Type**들의 집합

모양 1반 모여라!

Type Class Basic

- Num Class
 - (+), (*), abs, signum, (-), ... 등이 가능한 Type들
 - Int, Word, Double, ...
- Eq Class
 - (==), (/=) 가 가능한 Type들
 - Prelude에 정의된 모든 Type들

모양 1반 모여라!

Type Class Basic

- Show Class
 - `show :: a -> String` 이 가능한 Type들
 - Bool, Char, Int, ...
- Read Class
 - `read :: String -> a` 가 가능한 Type들
 - Bool, Char, Int, ...

모양 1반 모여라!

Type Class Basic

• ... $\frac{1}{0}$ $\frac{1}{0}$

모양 1반 모여라!

Type Class Basic

- 최소 완전 정의 Minimal Complete Definitions
 - Type Class에 속하기 위한 최소한의 정의
 - Minimal Complete Definition을 하면 나머지 함수는 Haskell이 알아서 정의해준다.
 - 최적화를 위해 직접 정의할 수는 있지만 일관성을 잃어버릴 수도 있다.

모양 1반 모여라!

Type Class Basic

- Num Class의 경우
 - (+), (*), abs, signum, fromInteger, (negate 또는 (-))
- Eq Class의 경우
 - (==) 또는 (/=)

모양 1반 모여라!

Type Class Basic

- Show Class의 경우
 - showsPrec 또는 show
- Read Class의 경우
 - readsPrec 또는 read

모양 1반 모여라!

Type Class Basic

- 왜 Type Class 를 써야할까?
 - Polymorphism을 위해서
 - Interface를 주기 위해서

모양 1반 모여라!

Type Class Basic

- Polymorphism을 위해서
 - Type Class에 의해서 Ad-hoc Polymorphism을 할 수 있다.
 - 동일한 (==) 함수를 Type에 따라 다르게 정의할 수 있다.
 - 다른 언어들에서는 Overloading이라고 불리기도 한다.

모양 1반 모여라!

Type Class Basic

- Interface를 주기 위해서
 - 특정한 Type Class에 해당하는 Type 들은 가능한 연산이 무엇인지에 대한 최소한의 단서를 가지고 있다.
 - `fun :: a -> a`인 함수 `fun`의 결과 값을 가지고 무엇을 할 수 있을까?
 - `funN :: Num a => a -> a`인 함수 `funN`의 결과 값을 가지고 무엇을 할 수 있을까?

넌 내 타입이 아니야

User Define Type

- 가장 간단한 방법은 타입 동의어 Type Synonyms를 사용하는 것이다.
- ~~type String = [Char]~~
- type Name = String
- type Dictionary a b = [(a,b)]
- ...

넌 내 타입이 아니야

User Define Type

- **Type Synonyms**을 사용할 경우 **Pattern**은 이미 있는 타입과 동일하게 된다.
- 즉 이미 있는 타입과 구분할 수 없다.
- 함수에게 줄 복잡한 **Type**을 미리 정하거나, 함수의 인자에 명확하게 이름을 붙일 때 사용한다.

넌 내 타입이 아니야

User Define Type

- `type Name = String`
`type Phone = String`
`type PhoneBook = (Name, String)`
- PhoneBook Type의 내용물을 보다 명확하게 쓸 수 있다.

넌 내 타입이 아니야

User Define Type

- 좀 더 복잡한 방법으로는 타입 생성자 **Type Constructor**를 사용할 수 있다.
- ~~data Bool = True | False~~
- data TrafficSign = GreenSign | YellowSign | RedSign
- data Tree a = Empty | Node a (Tree a) (Tree a)
- data Direction = ToLeft | ToRight

넌 내 타입이 아니야

User Define Type

- 좀 더 복잡한 방법으로는 타입 생성자 **Type Constructor**를 사용할 수 있다.
- **Tree a**
 - **Type Constructor**
 - **Type** 표기에서 사용 가능한 것
 - **[a]**에서 **[]**와 같은 것

넌 내 타입이 아니야

User Define Type

- 좀 더 복잡한 방법으로는 타입 생성자 **Type Constructor**를 사용할 수 있다.
- `Node a (Tree a) (Tree a)`
 - Data Constructor
 - Pattern Matching에서 사용 가능한 것
 - `a:[a]`에서 `(:)`와 같은 것

넌 내 타입이 아니야

User Define Type

- Tree와 Direction를 사용해 보자.
- Tree와 [Direction]을 받아 해당하는 위치의 원소를 결과값으로 주는 함수를 짜보자.
- `walk :: Tree t -> [Direction] -> t`

넌 내 타입이 아니야

User Define Type

- Tree와 Direction를 사용해 보자.
- `walk :: Tree t -> [Direction] -> t`
`walk (Node x lc rc) l = case l of`
 `[] -> x`
 `ToLeft:ds -> walk lc ds`
 `ToRight:ds -> walk rc ds`

넌 내 타입이 아니야

User Define Type

- Tree와 Direction을 사용해 보자.
- 테스트 해 보자
- ```
let t1 = Node 1 Empty Empty
let t3 = Node 3 Empty Empty
let t2 = Node 2 t1 t3
let t5 = Node 5 Empty Empty
let t4 = Node 4 t2 t5
walk t4 [ToLeft, ToLeft]
```

# 넌 내 타입이 아니야 User Define Type

- Tree 안의 데이터에 접근하기 위해서 Pattern Matching만 사용할 수 있을까?
- `treeItem (Node x _ _) = x`
- 같은 함수가 있다면 좋을텐데...

# 넌 내 타입이 아니야 User Define Type

- Tree 안의 데이터에 접근하기 위해서 Pattern Matching만 사용할 수 있을까?
- 1. 직접 정의한다
- 2. Record 구문을 써서 Type을 정의한다

# 넌 내 타입이 아니야 User Define Type

- **Tree** 안의 데이터에 접근하기 위해서 **Pattern Matching**만 사용할 수 있을까?
- 1. 직접 정의한다 => 일일이 모든 **Field**에 대해 정의한다???

# 넌 내 타입이 아니야 User Define Type

- Tree 안의 데이터에 접근하기 위해서 Pattern Matching만 사용할 수 있을까?
- 2. Record 구문을 써서 Type을 정의한다 => Record 구문?

# 넌 내 타입이 아니야

## User Define Type

- Record 구문
  - Type의 각 Field에 이름을 붙여서 정의하는 구문
  - OOP 언어의 `getter/sette`와 비슷하다.
- `data Tree a = Empty | Node {treeItem::a, leftTree::(Tree a), rightTree::(Tree a)}`
- `treeItem`, `leftTree`, `rightTree`라는 함수가 자동으로 정의된다.

# 전학생 맞아보기

## Type Class Implementation

- 우리가 새로 만든 Type이 Type Class안에 들어가게 하려면 어떻게 해야할까?
- Type Class Implementation 하기

# 전학생 맞아보기

## Type Class Implementation

- 어떻게 Type Class를 Implementation 할 수 있을까?
- Type Class의 Minimal Complete Definition에 해당하는 함수들을 정의하면 된다!



# 전학생 맞아보기

## Type Class Implementation

- TrafficSign Type<sup>0</sup>이 Eq Class를 만족하도록 구현해보자.
- instance Eq TrafficSign of  
    RedSign == RedSign = True  
    YellowSign == YellowSign = True  
    GreenSign == GreenSign = True  
    \_ == \_ = True
- 위와 같이 하면 Eq a에 TrafficSign을 추가할 수 있다.

# 전학생 맞아보기

## Type Class Implementation

- 너무 당연한 정의 아닌가??? 일일이 사람 손으로 써줘야 하나?
- NO!

# 전학생 맞아보기

## Type Class Implementation

- Eq, Ord, Enum, Bounded, Show, Read 등 시시콜콜한 정의로 이루어진 Type Class 들은 Type을 정의할 때 자동으로 Implementation을 할 수 있다.

# 전학생 맞아보기

## Type Class Implementation

- `data TrafficSign = GreenSign | YellowSign | RedSign deriving (Eq, Show, Read)`
- `data Tree a = Empty | Node {treeItem::a, leftTree::(Tree a), rightTree::(Tree a)} deriving (Show)`

# Type 뒤에 Class 있어요, Class 있다구요!

## Type Class Defintion

- 기존의 Type Class로 불충분하다면 어떻게 해야할까?
- 새 Type Class를 만들자!
- 다만, 그럴 일은 거의 없기 때문에 간단하게만 다루고 넘어간다.

# Type 뒤에 Class 있어요, Class 있다가구요!

## Type Class Defintion

- class NewTypeClass a where  
    strangeFunction :: (Num b) => a -> a -> b  
    notStrangeFunction :: (Num b) => a -> a -> b -> b  
    notStrangeFunction x y z = (strangeFunction x y) + z
- NewTypeClass의 Minimal Complete Definition은?
- strangeFunction

# 함수를 계승 중입니다, 아버지

## Type Class Inheritance

- Type Class Eq와 Ord를 생각해 보자.
  - Eq는 같은지 다른지 비교가 가능한 타입들의 집합
    - `(==) :: a -> a -> Bool`
  - Ord는 크기의 비교가 가능한 타입들의 집합
    - `compare :: a -> a -> Ordering`
    - `(<=) :: a -> a -> Bool`

# 함수를 계승 중입니다, 아버지

## Type Class Inheritance

- Type Class Eq와 Ord를 생각해 보자.
- Eq와 Ord의 내용은 중복된다!
  - $\leq$ 가 정의 가능하면,  $(a \leq b) \ \&\& \ (b \leq a) == (a == b)$  이다.
  - 즉, Ord이기만 하면 반드시 Eq이다.



# 함수를 계승 중입니다, 아버지

## Type Class Inheritance

- Type Class Eq와 Ord를 생각해 보자.
- Eq와 Ord의 내용은 중복된다!
  - ( $\leq$ )가 정의 가능하면,  $(a \leq b) \ \&\& \ (b \leq a) == (a == b)$  이다.
  - 즉, Ord이기만 하면 반드시 Eq이다.

# 함수를 계승 중입니다, 아버지

## Type Class Inheritance

- Type Class Eq와 Ord를 생각해보자.
- Eq와 Ord의 내용은 중복된다!
  - 거꾸로, Ord이라면 반드시 (==)도 정의되어야 한다.
  - 즉, Ord이라면 Eq의 내용을 구현해야만 한다.

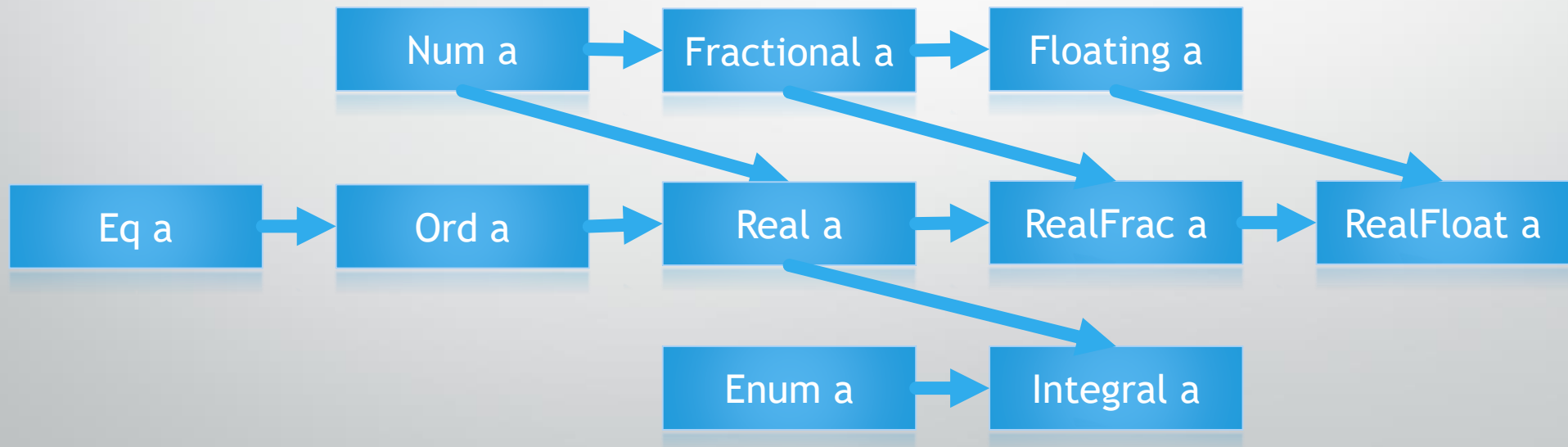
# 함수를 계승 중입니다, 아버지

## Type Class Inheritance

- B Class이기 위해서는 반드시 A Class여야 하는 관계를 가진 Type Class들이 있다.
- 이들의 관계를 상속 Inheritance라고 하고, 더 적은 수의 함수만 요구하는 것(Eq, A)을 부모, 더 많은 수의 함수를 요구하는 것(Ord, B)을 자식이라고 부른다

# 함수를 계승 중입니다, 아버지 Type Class Inheritance

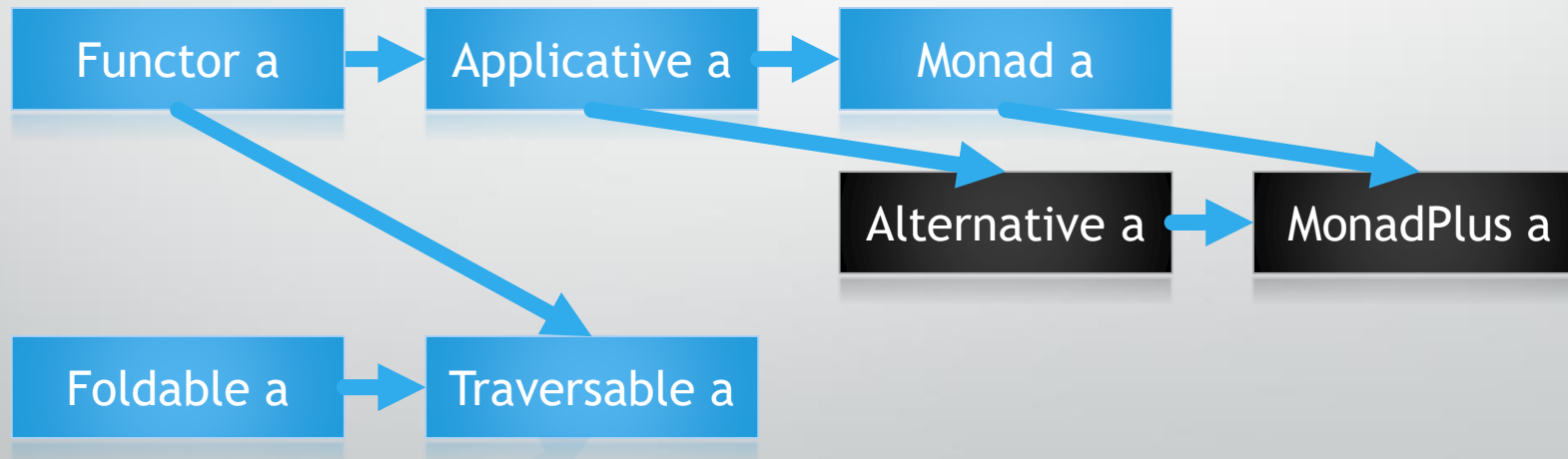
- 대표적인 상속 관계



# 함수를 계승 중입니다, 아버지

## Type Class Inheritance

- 대표적인 상속 관계



# 함수를 계승 중입니다, 아버지

## Type Class Inheritance

- 너무 많다???
- C의 표준 라이브러리처럼 쓰다보면 익히는 것들
- 한 번에 다 외울 필요 없다 (Hoogole이 있으니까!)

# 함수를 계승 중입니다, 아버지

## Type Class Inheritance

- Quiz!
- 다음 함수의 **Type**은 무엇일까?
- ```
fun x y | x == y = 2 * x
      | x > y  = x + y
      | otherwise = 2 * y
```
- `(Num a, Ord a) => a -> a -> a`