Monad 실습

1 Haskell의 Maybe 타입

1.1 Maybe 정의 및 실습

- Maybe : 계산의 성공과 실패를 표현하기 위한 타입으로서, 이미 Haskell의 기본 기능으로 내재되어 있다.
- Maybe 는 type constructor로서 타입 변수(혹은 generics) a를 인수로 가지며, Just, Nothing 은 data constructor라고 함.

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

1.2 Ghci를 이용한 Maybe 테스트

- 다음과 같이 Ghci를 이용하여 data constructor와 type constructor를 테스트해 본다.

```
Prelude> :t Just
Just :: a -> Maybe a
Prelude> :t Nothing
Nothing :: Maybe a
Prelude> :kind Maybe
Maybe :: * -> *

Prelude> :t Just True
Just True :: Maybe Bool
Prelude> :t Just 'c'
Just 'c' :: Maybe Char

Prelude> Nothing ::  Maybe Int
Nothing
Prelude> Nothing ::  Maybe Bool
Nothing
Prelude> Nothing ::  Maybe Char
Nothing
Prelude> Nothing ::  Maybe [[[Int]]]
Nothing
Prelude> Nothing ::  Maybe (Maybe (Mabe [[Char]]))
```

- Just 는 어떤 타입 a 의 값을 Maybe a 타입으로 변환하는 기능을 함. Maybe 는 어떤 값을 담는 컨테이너(container) 인 데, Just 가 어떤 값을 Maybe 컨테이너로 래핑(wrapping) 하는 기능을 함.
- Nothing 은 **정의되지 않은 값** (undefined value)을 표현하는 구성자로서, 계산의 실패를 표현하기 위해 사용된다. 이 구성자는 인수를 갖지 않으며, 그 자체로서 모든 컨테이너 타입 Maybe a 의 값을 표현한다.
- 타입은 값들의 모음인데, 이론적으로 타입들의 모음은 타입이 될 수 없으며, 그 보다 더 상위 레벨인 kind에 속하게 된다. 위의 예에서, Maybe 의 kind는 한 개의 타입을 받아서 타입으로 변환하는 함수로서 인식되고 있음을 보여 준다.

2 Java의 Optional

- Optional 은 개념적으로 Haskell의 Maybe 과 동일하다.
- Maybe 의 Nothing 은 static 메소드인 Optional.empty() 함수로 표현된다. 여기서 Nothing 의 개념을 표현하는 것이 null 이 아니란 점에 유의해야 한다.
- Nothing 과 마찬가지로 Optional.empty() 는 여러 타입을 갖는다.

```
jshell> Optional<Integer> i = Optional.empty()
jshell> Optional<Boolean> b = Optional.empty()
```

- Optional 의 static 메소드인 Optional.of() 은 Haskell의 Just 처럼 값을 Optional 컨테이너로 래핑한다.

```
static <T> Optional<T> of(T value)
```

```
jshell> Optional<Integer> n = Optional.of(123)
jshell> Optional<Boolean> tt = Optional.of(true)
jshell> n.get()
$  ==> 123
jshell> tt.get()
$  ==> true
```

- Haskell은 함수형 프로그래밍으로서, Maybe 컨테이너 안에 들어 있는 값을 추출할 때 f (Just x) = ... 형태의 패턴 매칭으로 표현되므로, 별도의 함수나 특별한 표현이 필요치 않다. 그러나, 패턴 매칭이 없는 Java에서는 컨테이너 안의 값을 꺼내는 인스턴스 메소드 get() 를 이용하여 컨테이너 안의 값을 추출한다.
- Optional의 다른 메소들은 편리함을 위해서 제공되는 메소드들이고 보여 진다.
- Optional 의 map 은 개념적으로 Functor에 해당되는 메소드로서 Functor에서 설명하기로 한다.

3 펑터(Functor)

3.1 fmap

- Haskell에서 펑터는 아래와 같이 type class로서 정의된다.
- 다음 클래스에서 f 는 parameterized type (즉, 인수를 갖는 type constructor)이다. 흔히 우리는 이것을 f 컨테이너라고 부르기도 하는데, 위에 논의한 Optional 처럼 Java에서는 generics를 갖는 클래스가 컨테이너이다.

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

- Functor 클래스 f 에 대한 구현에 따라 List 펑터, Maybe 펑터 등의 다양한 형태의 펑터가 존재한다.
- fmap 의 타입 (a -> b) 는 함수를 표현하는 타입으로서, Java에서는 이와 같은 펑터를 계산하기 위한 함수를 mapper 함수라고 부른다. 여기서는 Haskell에서도 이 용어를 사용하기로 한다.
- mapper와 컨테이너가 주어지면, fmap 는 컨테이너 안에 있는 값을 추출하고 (타입 a), 이 값에 mapper(타입 a -> b) 를 적용하여 얻어진 값 (타입 b) 을 다시 컨테이너 안으로 래핑 (타입 f b) 하는 기능을 한다.
- 전체적으로 보면, fmap 은 마치 mapper 함수를 컨테이너 안에 넣어서 컨테이너 안의 값에 함수를 적용시킨 것 처럼 보인다.

- 펑터는 추상 대수학의 카테고리 이론(category theory)에서 연구된 결과를 구현한 것인데, 이론적으로 펑터의 `fmap` 함수는 **구조를 유지(structure-preserving)** 하는 특징을 갖는다.
- List 펑터에서 `fmap` 은 기능적으로 이미 구현된 함수 `map` 과 동일하다. 리스트 펑터에서 `map f [x,y,z]` 의 결과는 `[f x, f y, f z]` 로서, 주어진 데이터 `[x,y,z]` 가 `[f x, f y, f z]` 형태로 계산된 결과이다. 이때 펑터 계산은 컨테이너 원래의 구조 (즉, 리스트 원소의 순서 및 갯 수) 를 변화시키지 않는다.
- **구조 유지** 의 원리는 List 뿐만 아니라 모든 형태의 Functor에 대해서 적용되며, Functor 클래스 구현도 이 원리가 성립될 수 있도록 해야한다.

3.2 `List` Functor

- Haskell에서 리스트 타입 `[a]` 는 `(List a)` 의 구문적 치장(syntactic sugaring) 으로서, List 컨테이너를 표현한 것이다.

```
fmap :: (a -> b) -> List a -> List b     -- List Functor
fmap :: (a -> b) -> [a] -> [b]           -- 구문적 치장으로 표현된 형태
```

- `List` 펑터에 대한 `fmap` 의 기능은 이미 많이 사용되고 있는 `map` 과 동일하므로 그에 대한 구현은 `map` 으로 대신할 수 있다.

3.3 `Maybe` Functor

```
fmap :: (a -> b) -> Maybe a -> Maybe b    -- Maybe Functor
```

- 앞 부분에 설명한 대로 `Nothing` 은 타입에 상관없이 모든 `Maybe` 컨테이너에 소속되어 있다. 즉, `Maybe Int, Maybe Bool` 등은 모두 `Nothing` 을 포함하고 있다.
- `fmap` 이 `Maybe` 컨테이너에 대해 구조를 유지하는 방식은 mapper가 컨테이너의 원소에 적용될 때 타입에 상관 없이 `Nothing` 에 적용된 결과는 `Nothing` 으로 매핑되도록 정의하는 것이다.

```
Prelude> fmap (\x -> x * 10) (Just 2)
Just 20
Prelude> fmap even (Just 2)
Just True
Prelude> fmap (\x -> x * 10) Nothing
Nothing
Prelude> fmap even Nothing
Nothing
```

- `fmap (\x -> x * 10) (Just 2)` 는 `Maybe` 컨테이너 안에 있는 값 2을 꺼낸 후, 이 값에 주어진 함수 `(\x -> x * 10)` 을 적용하여 얻어진 값 20을 다시 `Just` 를 적용하여 `Maybe` 컨테이너 안으로 집어 넣는 기능을 한다.
- `Maybe` Functor에서 값이 `Nothing` 일 경우, `fmap` 은 어떤 함수를 적용하더라도 그 결과는 `Nothing` 이 됨을 볼 수 있다.

4 Java의 `Optional` 펑터

- Java의 `Optional` 펑터는 Haskell의 `Maybe` 와 동일한 개념으로 구현되었다.
- Haskell과는 달리 펑터 함수의 이름을 `fmap` 대신 `map` 으로 사용하고 있다.
- `map` 은 `Optional` 컨테이너 에 값이 있는 경우 그 값을 꺼내서, 주어진 mapper 함수로 그 값을 계산한 다음, 다시 `Optional` 컨테이너 않으로 넣는다. 만약 컨테이너의 원소가 empty 라면 어떤 mapper가 적용더라도 결과 값은 empty이다.

```
jshell> Optional<Integer> two = Optional.of(2)
jshell> two.map(x -> x*10)
$   ==> Optional[20]
jshell> two.map(x -> x % 2 == 0)
$   ==> Optional[true]

jshell> Optional.empty().map(x -> (Integer) x * 10)
$  ==> Optional.empty
```

5 Haskell의 Flattening 함수 join

5.1 중첩된 컨테이너 타입 m (m a)

- 컨테이너는 타입을 인수로 갖는 타입으로서, 컨테이너는 래핑되어 또 다른 컨테이너 안에 포함될 수 있다. 이때 원래의 컨테이너와 새로운 컨테이너는 동일한 타입이 될 수도 있고 그렇지않을 수도 있다.

```
Prelude> :t [Just True, Nothing]
[Just True, Nothing] :: [Maybe Bool]      -- List (Maybe Bool)

Prelude> :t (Just (Just True))
(Just (Just True)) :: Maybe (Maybe Bool)

Prelude> :t (Just (Just (Just True)))
(Just (Just (Just True))) :: Maybe (Maybe (Maybe Bool))
```

- 위의 예에서 두 번째의 경우 Bool 값이 동일한 컨테이너 Maybe 에 두 번 중첩되어 있는 경우이며, 세 번째는 세 번 중첩된 상황을 보여주는 예이다.
- 이처럼 동일하게 두 번 이상 중첩된 타입은 m (m a) 형태의 타입으로 표현된다. 세 번째의 경우, (Maybe Bool) 이 a 에 해당된다.

5.2 Join에 의한 Flattening

- Flattening 이란 같은 컨테이너에 맨 밖의 두 번 중첩된 래핑(wrapping)을 한 거플을 벗겨 내는 형태의 기능을 하는 오퍼레이션이다.
- 각 컨테이너 타입에 대한 Flattening 함수의 정의는 그 컨테이터 타입의 특징을 고려하여 정의된다.
- 하스켈의 경우, Control.Manad 패키지 소속의 join 이 이 기능을 한다.

```
join :: m (m a) -> m a
```

5.3 Maybe에 대한 Flattening

- 여러 겹으로 래핑된 Maybe 컨테이너를 반복적으로 flattening 할 수 있다.

```
Prelude> import Control.Monad
Prelude Control.Monad> join (Just(Just 8))
Just 8
Prelude Control.Monad> join (Just(Just(Just 8)))
Just (Just 8)
Prelude Control.Monad> join (Just(Nothing))
Nothing
Prelude Control.Monad> join (Just(Just(Nothing)))
Just Nothing
Prelude Control.Monad> join (join (Just(Just(Nothing))))
Nothing
```

- Maybe Flattening에서 Nothing 이 포함된 경우 결과는 Nothing 되도록 정의된다. 이러한 정의의 의도는 여러 단계의 계산을 수행하는 동안 한 번이라도 계산이 실패하는 경우가 발생하면 전체 계산의 결과를 실패로 처리하기 위한 것이다.

5.4 List에 대한 Flattening : join = concat

```
Prelude> import Control.Monad
Prelude Control.Monad> join [ [1,2], [3], [], [4]]
[1,2,3,4]
Prelude Control.Monad> concat [ [1,2], [3], [], [4]]
[1,2,3,4]
Prelude Control.Monad> join [[[1,2], [3,4]], [[5]]]
[[1,2],[3,4],[5]]
Prelude Control.Monad> join [[1], [2,3], [], [4]]
[1,2,3,4]
Prelude Control.Monad> join [[1,2,3,4], [], []]
[1,2,3,4]
Prelude Control.Monad> join [[], [1], [2,3,4]]
[1,2,3,4]
```

- 리스트는 임의의 갯 수의 값들을 래핑한다. 원소의 수가 0 개, 1 개, 2 개, ..., 무한 개인 경우도 래핑할 수 있다.
- [[a]] 는 리스트 내에 리스트가 내포된 형태인데, 안쪽 리스트 [a]와 바깥쪽 리스트 [[a]] 는 모두 불규칙적인 원소의 수를 갖는다. join 이 [[a]] 를 [a] 로 변환시키는 과정은 [[a]] 의 원소들을 순서적으로 나열하는 과정에 해당된다.
- Haskell에서 리스트에 대한 flattening 함수 join 은 concat 과 동일하다.

5.5 Flattening의 원리

- 펑터 계산이 컨테이너의 구조를 유지하는 것과는 달리, Flattening은 구조를 변경할 수 있는 특징이 있다.
- 예를 들어, Maybe 를 flattening 하는 경우, Just 가 Nothing 을 내포한 경우 결과는 Nothing 이 된다.
- Flattening의 정의는 각 컨테이너의 특징에 달려 있다.
- Maybe 의 경우 계산의 성공/실패를 표현하는 경우로서, 연속된 Flattening 과정에서 한 번이라도 계산의 실패가 있다면 전체 계산은 실패로 귀결된다.
- List는 임의의 갯 수의 값을 담기 위한 컨테이너이다. 즉, 값의 갯 수가 0개, 1개, ... 무한 개 까지의 값들을 담는 특성이 있다. 즉, 원소의 수가 고정되어 있지 않다. 대신 원소들이 나열되는 순서는 존중되어야 한다. 위의 예에서 join 이 적용됨에 따라 리스트의 원소 수가 변화되는 모습을 볼 수 있다.

6 Haskell의 Monad

6.1 bind (>>=) 오퍼레이터

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

- return 은 단순히 한 값을 컨테이너로 래핑하는 기능을 한다.

- 바인드 (>>=) 는 이진 함수로서 컨테이너로 래핑된 값과 그 컨테이너에 래핑된 값을 꺼냈을 때 적용할 수 있는 함수를 인수로서 받는다.
- 바인드는 대략적으로 말해서, 펑터와 Flattening의 두 연산을 연속적으로 결합한 것이다. 즉, 다음과 같이 함수의 합성 (flat . map) 의 형태의 기능을 한다 (이런 이유에서 Scala 언어 이후 대부분 언어에서는 바인드 대신 flatMap 이라는 이름을 사용하고 있다).

```
Prelude Control.Monad> fmap (\x -> [x+2]) [1,2,3] -- fmap 결과에 join 을 적용함
[[3],[4],[5]]
Prelude Control.Monad> join it
[3,4,5]

Prelude Control.Monad> [1,2,3] >>= \x -> [x+2]    -- 바인드에 의해 한 번에 처리됨
[3,4,5]
```

- 위의 예에서 `join it` 의 `it` 은 Ghci 기능으로서, 앞서 실행한 결과 값 (이 경우는 [[3],[4],[5]]) 을 의미한다.
- Haskell의 바인드는 위의 두 연속된 연산을 하나로 묶어서 처리하는 기능을 한다.

6.2 모나드에서 적용되는 펑터 타입

- 다음 펑터 타입에서 첫 번째 형식의 타입 `b` 가 `(m b)` 형태로 구체화되면 두 번째 형태의 모습을 갖게 된다.

```
fmap :: (a -> b) -> m a -> m b          -- 일반적인 펑터의 타입
fmap :: (a -> m b) -> m a -> m (m b)    -- 모나드에서 적용되는 타입
```

- 위의 두 번째 형태의 functor 수행 결과 얻게 되는 `m (m b)` 형태는 값 `b` 를 `m` 컨테이너에 두 겹으로 래핑된 모습니다. 모나드에 적용되는 펑터는 위의 두 번째 형태를 갖는다.
- `m (m b)` 를 `(m b)` 로 만들기 위해서는 `join` 이 적용되어야 한다.
- 바인드와 functor 사이에는 다음과 같은 공식이 성립된다.

```
xs >>= y
  =  join (fmap y xs)
```

- `xs` 의 타입은 `(m a)` 이고, y의 타입은 `(a -> m b)` 가 된다. 그러면 `fmap y xs` 의 타입은 `m (m b)` 이다. 그런데 바인드의 출력 타입은 `(m b)` 가 되어야 하므로, 여기서 `join` 이 적용하여 `(m b)` 타입의 결과를 갖게 된다.

6.3 하스켈 모나드 프로그래밍 예

```
Prelude Control.Monad> [1,2,3] >>= \x -> [x, x*10]
[1,10,2,20,3,30]

Prelude Control.Monad> (Just 3) >>= \x -> Just (x * x)
Just 9
```

6.4 Java의 모나드 프로그래밍 예

6.4.1 Java의 `flatMap`

```
Stream<R> stream<T>.flatMap(Function<T, Stream<R> mapper)
```

- 위의 하스켈 모나드 프로그래밍과 동일한 기능을 Java로 코딩한 경우이다. Java에서는 `flatMap` 은 Stream 에 적용되므로, List를 Stream으로, Stream을 List로 전환하는 과정이 필요하다.

```
jshell> List<Integer> list = List.of(1,2,3);
jshell> Stream<Integer> strm = list.stream();
jshell> strm.flatMap(x -> Stream.of(x,x*10)).collect(Collectors.toList());
$ ==> [1, 10, 2, 20, 3, 30]

jshell> Optional.of(3).flatMap(x -> Optional.of(x*x))
$  ==> Optional[9]
```

6.5 바인드와 flatMap이 펑터 함수처럼 동작되는 경우

- 바인드는 펑터 `fmap` 과 flattening 함수 `join` 이 모두 적용되어 운영되며, 모든 컨테이너 형태의 데이터를 처리할 수 있는 강력한 기능을 갖는다. 컨테이너 처리에 사용되는 대표적인 함수인 `map`, `filter` 등의 함수들은 펑터와 Flattening 기능을 적절히 조절함으로써 모나드 함수(Haskell의 바인드, Java의 flatMap)로서 대신 처리할 수 있다.

- `fmap` 의 mapper 함수 `f` 에 대해서 바인드의 두 번째 인수를 `\x -> return (f x)` 로 표현하는 경우 바인드는 펑터와 동일한 기능을 한다.

```
xs >>= \x -> return (f x)
  = join (fmap (\x -> return (f x)) xs)
  = fmap f xs
```

6.5.1 Haskell의 바인드가 `fmap` 처럼 동작

```
Prelude Control.Monad> fmap (\a -> a + 1) [1,2,3]
[2,3,4]
Prelude Control.Monad> [1,2,3] >>= \x-> return ((\a -> a + 1) x)
[2,3,4]

Prelude Control.Monad> fmap (\x -> x * 10) (Just 20)
Just 200
Prelude Control.Monad> (Just 20) >>= \x -> return (x * 10)
Just 200
```

6.5.2 Java의 `flatMap` 이 `map` 처럼 동작

- 위의 Haskell 코드는 Java로 다음과 같이 표현된다.

```
jshell> List<Integer> list = List.of(1,2,3)
jshell> list.stream()
            .flatMap(x -> Stream.of(x+1))
            .collect(Collectors.toList())
$ ==> [2, 3, 4]

jshell> Optional.of(20).flatMap(x -> Optional.of(x*10))
$ ==> Optional[200]
```

6.6 `filter` 처럼 동작하는 바인드와 flatMap

- `filter` 는 리스트의 원소 중에서 조건에 맞는 원소만을 선택하는 기능을 한다. `filter` 는 조건식을 사용하여 참 일 때는 `return x` 를 하지만, 거짓인 경우 `[]` 나 `Nothing` 을 return 한다.

6.6.1 `filter` 처럼 동작하는 Haskell의 바인드

```
Prelude Control.Monad> filter (>2) [1,2,3,4]
[3,4]
```

```
Prelude Control.Monad> [1,2,3,4] >>= \x -> if (x > 2) then (return x) else []
[3,4]
```

6.6.2 filter 처럼 동작하는 Java의 flatMap

```
jshell> List<Integer> list = List.of(1,2,3,4)
list ==> [1, 2, 3, 4]

jshell> list.stream()                // flatMap 적용
            .flatMap(x -> (x>2) ? Stream.of(x) : Stream.empty())
            .collect(Collectors.toList())

jshell> List.of(1,2,3,4).stream()    // filter 적용
            .filter(x -> x>2)
             .collect(Collectors.toList())
$2 ==> [3, 4]
```

6.7 바인드와 flatMap이 Flattening으로만 기능하는 경우

- 이 경우 펑터의 mapper는 아무런 처리 기능을 하지 않는 id, 혹은 (\x -> x) 형태로 표현된다.

```
Prelude Control.Monad> (Just(Just 2)) >>= \x -> x
Just 2

Prelude Control.Monad> [[1,2],[],[3,4]] >>= \x -> x
[1,2,3,4]
Prelude Control.Monad> [[1,2],[],[3,4]] >>= id
[1,2,3,4]

jshell> Optional<Optional<Integer>> ooTwo = Optional.of(Optional.of(2))
ooTwo ==> Optional[Optional[2]]       // nested Optional

jshell> ooTwo.flatMap(x -> x)         // flattening
$ ==> Optional[2]

jshell> Optional.of(Optional.of(2)).flatMap((Optional<Integer> x) -> x)
$ ==> Optional[2]

jshell> List<List<Integer>> llst = List.of(List.of(1,2), List.of(), List.of(3,4))
llst ==> [[1, 2], [], [3, 4]]

jshell> llst.stream()                       // Stream<List<Integer>>
            .flatMap(Collection::stream)    // Stream<Stream<Integer>>
            .collect(Collectors.toList())   // List<List<Integer>>
$ ==> [1, 2, 3, 4]
```