숙제 #3: 패턴 매칭과 타입 Rev 1

건국대학교 소프트웨어학과 2019 컴퓨테이션이론 담당: 차리서

이번 숙제는 기초적인 재귀적 함수 정의 및 인자의 패턴 매칭, 그리고 타입 정의에 관한 숙제입니다. 강의 시간에 제가 설명했던 요소들만 숙제에 포함했으며, 이미 설명했던 사항 중 일부도 이 문서에서 다시 설명하고 있습니다. 즉, 이 숙제를 하기 위해 강의 시간에 설명하지 않았던 요소를 혼자 찾아내서 공부해야할 필요가 없도록 구성했습니다.

이 문서를 보면서, `HW3.hs` 파일의 적절한 곳을 수정하거나 코드를 추가한 후, 최종적으로 `HW3.hs` 파일만 업로드하면 됩니다.

Contents

1 패턴 매칭

가장 익숙하고 자주 다루는 자료형인 정수를 소재로, 패턴 매칭의 기본 사항을 다시 짚어보겠습니다.

정의하려는 함수의 인자가 정수인 경우, 매개변수 n이나 x 등 뿐만 아니라 0이나 1, 2 등의 정수 literal도 인자 자리에 사용될 수 있습니다. 이렇게 함수의 인자 자리에 매개변수 대신 정수 literal이 사용될 경우, 실제 주어진 인자가 바로 그 literal에 해당하는 값이어야만 그 정의문에 대응됩니다.

예를 들어, 함수 `f`가 다음과 같이 정의되어있을 경우:

file: HW3.hs

```
f :: Integer -> Bool
f 3  =  True
f 5  =  False
```

이 `f`는 정의역인 `Integer`의 모든 원소에 대해 정의된 total function이 아니라, 정의역의 원소들 중 일부 원소에 대해서만 정의된 partial function입니다.

Haskell에서 위와 같은 정의는 일단 문제 없이 받아들여지지만, 실제로 `f` 함수가 사용될 때 정의되지 않은 인자가 주어지면 런타임 오류를 일으킵니다.

[WinGHCi window, module HW3 loaded]

```
*HW3> f 3
True
it :: Bool
*HW3> f 5
False
```

```
it :: Bool
*HW3> f 10
*** Exception: HW3.hs:(6,1)-(7,13): Non-exhaustive patterns in function f

*HW3>
```

여기서 non-exhaustive patterns란 "패턴이 (정의역의 모든 원소에 대해) 완전히 포괄적이지 않다"는 뜻입니다.

반면에, 함수 `f'`이 다음과 같이 정의되어 있을 경우:

file: HW3.hs

```
f' :: Integer -> Bool
f' 3  =  True
f' n  =  False
```

이 `f'`은 정의역인 `Integer`의 모든 원소에 대해 정의된 total function입니다. Haskell은 함수에 실제 인자가 주어지면 그때 함수의 정의들을 찾아보는데, 위 `f'`처럼 두 줄로 정의된 경우 위에서부터 순서대로 확인하면서 실제 주어진 인자가 패턴(`3` 혹은 `n`)에 맞아떨어질 방법이 있는지 확인합니다. 따라서, 실제 인자가 `10`일 경우 첫번째 정의의 인자 패턴인 `3`에는 맞아떨어질 방법이 없어서 그냥 지나가지만, 두번째 정의의 인자 패턴인 `n`에는 맞아떨어질 방법이 있으므로 (즉, `n`을 `3`으로 간주할 수 있으므로) 해당 정의에 따라 `False`를 리턴합니다.

[WinGHCi window, module HW3 loaded]

```
*HW3> f' 3
True
it :: Bool
*HW3> f' 10
False
it :: Bool
*HW3>
```

그래서, 이 패턴 매칭은 순서가 무척 중요합니다. 만일 다음과 같이 두 정의의 순서가 바뀌어있을 경우:

file: HW3.hs

```
f'' :: Integer -> Bool
f'' n  =  False
f'' 3  =  True
```

첫번째로 확인해보는 패턴인 `n`이 모든 정수에 대해 맞아떨어질 수 있는 형태기 때문에, 실제 인자가 `3`이더라도 두번째 정의를 찾아볼 겨를도 없이 무조건 첫번째 정의에만 대응됩니다:

[WinGHCi window, module HW3 loaded]

```
*HW3> f'' 3
False
it :: Bool
*HW3> f'' 10
False
it :: Bool
*HW3>
```

1.1 자연수: 계승 (factorial)

자연수(0과 양의 정수)에 대한 계승은 수학적으로 다음과 같이 정의됩니다. 단, 이항 중위 연산자 $\times$보다 단항 후위 연산자 !의 결합 강도가 더 강합니다:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{otherwise} \end{cases}$$

참고로, $0! = 1$이라는 점에 주의하십시오. 아무 것도 더하지 않은 (0개의 수들을 더한) 결과를 0으로 간주하는 이유는 세상의 기본이 0이어서가 아니라 어디까지나 덧셈의 항등원이 0이기 때문입니다. 즉, 여러개의 수들의 합(summation)이라는 개념은 그 여러개의 수들을 덧셈의 항등원인 0에 반복적으로 더한 결과를 뜻합니다. 마찬가지로, 아무 것도 곱하지 않은 (0개의 수들을 곱한) 결과는 곱셈의 항등원인 1입니다. 여러개의 수들의 곱(product)이라는 개념은 그 여러개의 수들을 곱셈의 항등원인 1에 반복적으로 곱한 결과를 뜻합니다. (이 외에도 다른 다양한 이유들 때문에, $0! = 1$로 정하는 편이 훨씬 유리합니다. 이는 $0^0 = 1$인 이유와도 일맥상통합니다.)

첫번째 문제입니다: 정수 (`Integer` 타입의 값) 하나를 인자로 받아서, 그 수가 0 혹은 그 이상일 경우, 그 수의 계승(factorial)에 해당하는 자연수를 리턴하는 함수

```
fac :: Integer -> Integer
```

을 정의하십시오. 단, 인자로 주어진 정수가 0 미만일 경우 어떤 오동작(예를 들어, 무한 재귀)을 해도 상관없습니다. (이 문제를 어느 정도라도 해결하는 방법들 중 하나는 이 문서 후반에 잠깐 언급되지만, 아직 제대로 설명하지 않은 요소이므로 이번 숙제에서는 다루지 않습니다.)

정의된 함수 `fac`은 다음과 같이 동작해야 합니다:

[WinGHCi window, module HW3 loaded]

```
*HW3> map fac [0..10]
[1,1,2,6,24,120,720,5040,40320,362880,3628800]
it :: [Integer]
*HW3>
```

1.2 리스트: 반전

먼저, Haskell에는 두 리스트를 이어붙이는 이항 중위 연산자 `++`가 이미 정의되어 있습니다 (오른쪽으로 먼저 결합하고, 결합 강도는 5입니다):

built-in module: GHC.Base

```
(++) :: [a] -> [a] -> [a]
[]     ++ l  =  l
(x:xs) ++ l  =  x : (xs ++ l)

infixr 5  ++
```

예를 들어, 다음과 같이 동작합니다:

[*WinGHCi window*]

```
Prelude> [9,1,8,2,7] ++ [3,6,4,5]
[9,1,8,2,7,3,6,4,5]
it :: Num a => [a]
Prelude> "Kon" ++ "kuk"
"Konkuk"
it :: [Char]
Prelude>
```

참고로, 만일 이 연산자의 두 피연산자들 중 최소 하나 이상이 무한 리스트일 경우 이어붙인 결과도 무한 리스트며, 특히 왼쪽 피연산자가 무한 리스트일 경우 이어붙인 결과는 (오른쪽 피연산자는 무시되고) 왼쪽 피연산자와 동일한 무한 리스트입니다.

두번째 문제입니다: (만일 필요하다면 위 ++연산자를 이용해서) 임의의 리스트 하나를 인자로 받아서 그 리스트를 거꾸로 뒤집는 (원소들이 역순으로 들어있는 리스트를 리턴하는) 단항 함수

```
rev :: [a] -> [a]
```

를 정의하십시오. 정의된 rev는 다음과 같이 동작해야 합니다:

[*WinGHCi window, module* HW3 *loaded*]

```
*HW3> rev [4,6,8,7,5]
[5,7,8,6,4]
it :: Num a => [a]
*HW3> rev "Haskell Brooks Curry"
"yrruC skoorB lleksaH"
it :: [Char]
*HW3>
```

단, 인자로 무한 리스트가 주어지면 무한 재귀에 빠져도 상관 없습니다.

2 타입 정의

강의 시간에 이미 설명했지만, 새로운 타입을 만들려면 data로 시작하는 타입 정의문을 작성해야 합니다. 일단 설명을 위해서, 실제 Haskell에 들어있는 리스트 말고 우리 나름대로의 리스트를 정의해보겠습니다. 이 타입은 어디까지나 설명을 위해 만들어보는 타입일 뿐이며, 숙제 파일에 들어있기는 하지만 실제로 사용되지는 않습니다:

file: HW3.hs

```
data List t  =  EmptyList | ListCons t (List t)
```

위 정의문은 (임의의 타입 t에 대해서) List t 타입이 무엇인지를 정의하는 문장이지만, 실제로는 이 문장 하나로 인해 아래 세 가지 요소(타입 생성자 한 개와 자료 생성자 두 개)들이 한꺼번에 정의된 것입니다:

- 아무 타입 1개를 인자로 받아서 새로운 타입을 생성하는 (리턴하는) 단항 타입 생성자

```
List :: * -> *
```

- 0개의 값을 인자로 받아서, 즉, 인자 없이 (어떤 `t` 타입에 대해서든) `List t` 타입의 값을 리턴하는 무항 자료 생성자

  ```
  EmptyList :: List t
  ```

- 임의의 `t` 타입의 값 1개와 바로 그 `t` 타입 원소들의 리스트인 `List t` 타입의 값 1개, 이렇게 총 2개의 인자를 받아서 `List t` 타입의 값을 리턴하는 이항 자료 생성자

  ```
  ListCons :: t -> List t -> List t
  ```

참고로, 강의 시간에는 설명의 편의를 위해 타입 생성자의 정의역이나 공변역이 마치 "`Type`"인 것처럼 설명했었고 개념적으로는 딱히 틀리지 않은 설명이었지만, 엄밀히 말하자면 Haskell에는 "`Type`"이라는 타입은 없습니다.[1] 타입 생성자 `List`의 타입은 "`Type -> Type`"이 아니라 "`* -> *`"이며, 심지어 이것을 "타입"이라고 부르지도 않고 "카인드(kind)"라고 따로 부릅니다. 즉, 타입과 타입 생성자[2]들은 "타입"을 갖는 게 아니라 "카인드"를 갖습니다.

WinGHCi에서도 이 점을 확인할 수 있습니다. 아래 프롬프트 입력 명령 중, `:t` 혹은 `:type` 명령이 대상 값(혹은 자료 생성자)의 타입을 보여주듯이, `:k` 혹은 `:kind` 명령은 대상 타입(혹은 타입 생성자)의 카인드를 보여줍니다:

[*WinGHCi window, module* HW3 *loaded*]

```
*HW3> :t List

<interactive>:1:1: error: Data constructor not in scope: List
*HW3> :k List
List :: * -> *
*HW3> :t EmptyList
EmptyList :: List t
*HW3> :t ListCons
ListCons :: t -> List t -> List t
*HW3>
```

참고로, 강의 시간에 이미 여러번 다뤄봤던 (Haskell에 실제로 들어있는) 리스트의 타입 정의를 **잠깐만** 살펴보고 지나가겠습니다. WinGHCi 프롬프트에서 `:i []` 명령을 실행해보면 다음과 같은 결과가 나옵니다:

[*WinGHCi window*]

```
Prelude> :i []
data [] a = [] | a : [a]        -- Defined in ‘GHC.Types’
-- ... 중간 생략
Prelude>
```

명심할 점은, 실제로 `GHC.Types` 모듈의 100행 즈음에 저런 구문이 존재하긴 하지만:

built-in module: GHC.Types

```
92 {- ********************************************************************
93 *                                                                      *
```

[1] 실제로 `Type` 타입이 존재하는 프로그래밍 언어나 증명 시스템도 있으며, 그 중에는 심지어 $3 : \texttt{Int}$고, $\texttt{Int} : \texttt{Type}_1$이며, $\texttt{Type}_1 : \texttt{Type}_2$고, $\texttt{Type}_2 : \texttt{Type}_3$, ...하는 식으로 무한히 전개되는 시스템도 있습니다.

[2] 마치 '타입'과 '타입 생성자'가 별개인 것처럼 말했지만, 사실은 타입은 그저 무항 타입 생성자일 뿐입니다.

```
94                  Lists
95
96     NB: lists are built-in syntax, and hence not explicitly exported
97  *                                                              *
98  ****************************************************************-}
99
100 data [] a = [] | a : [a]
```

주석을 잘 보면, "리스트는 내장 구문이며, 따라서 (이 정의는) 다른 곳으로 유출되지 않는다"는 말이 쓰여있습니다. 즉, 사실 저 100행의 타입 정의는 반 쯤은 가짜입니다. (일반적인 방식으로 타입을 정의할 때 저렇게 로마자가 아닌 특수 기호를 타입 생성자나 자료 생성자로 사용할 수 없습니다.)

하지만, 일반적인 방식이 아닌 다른 방식으로 어딘가에 별개로 내장시켜놓은 구문 때문에, "`[]`"와 "`:`"를 실제로 우리가 사용할 수 있기는 합니다. 그래서, WinGHCi에서 다음과 같은 결과를 확인할 수 있습니다:

[*WinGHCi window*]

```
Prelude> :k [] -- :t가 아니라 :k입니다.
[] :: * -> *
Prelude> :t []
[] :: [a]
Prelude> :t (:)
(:) :: a -> [a] -> [a]
Prelude>
```

그리고, 이 정의를 그대로 쓸 수 있다고 해도 (게다가 우측 우선 결합이라는 사실까지 활용한다고 해도), 사실 리스트는 언제나

```
1 : 2 : 3 : 4 : 5 : []
```

형태로만 쓸 수 있어야 정상입니다. 이걸

```
[1,2,3,4,5]
```

라고도 쓸 수 있는 이유는 이것이 별도의 문법적 규칙으로 따로 허용되어있기 때문이며, 이렇게 원래 그것을 표기하는 원론적이고 보편적인 방법이 있음에도 불구하고 더 편한 형태로 표기할 수 있도록 따로 정해놓은 추가 문법을 syntactic sugar라고 합니다.[3]

2.1 자연수 타입

이제, Church numerals와 비슷한 방식으로 자연수를 정의해보겠습니다. 즉, 자연수란 `Z`거나 어떤 자연수의 `S`입니다.[4] (물론 여기서 "어떤 자연수"란 다시 재귀적으로 `Z`거나 어떤 자연수의 `S`입니다.)

[3] 이걸 "구문 설탕"이라고 번역하는 문서들이 종종 있는데, 저는 아무래도 오글거려서 사용하지 않고 그냥 원어로만 말하겠습니다.

[4] 수식 "$f(x)$"나 "$f\ x$"를 영어로 "f of x"라고 읽습니다. 하필 f와 x여서 와닿지 않을 수 있는데, 좀 더 와닿을 만한 예는 "`triple`(5)"를 "triple of five"라고 읽는 경우입니다. "어떤 자연수의 `S`"라는 말이 너무 노골적인 번역체여서 어색하게 느껴지거나 거부감이 들 수도 있겠지만, "어떤 자연수에 함수 (정확히는 자료 생성자) `S`를 적용한 것"이라고 길게 말하는 것보다 짧기도 하고 개념적으로 익숙해지라는 뜻에서 일부러 번역체를 사용했습니다.

file: HW3.hs

```
data Nat  =  Z
          |  S Nat
    deriving Show
```

위 정의는 줄바꿈 없이 한 줄에 이어서 써도 되지만, 이렇게 각 자료 생성자 별로 각각 다른 줄에 나눠 쓰는 방식도 가능하다는 것을 보이기 위해 (때로는 이렇게 쓰는 게 가독성이 높은 경우가 있으므로) 의도적으로 줄을 바꿔서 썼습니다.

앞서 `List`의 경우와 마찬가지로, 여기서도 위 한 문장을 통해 동시에 세 가지 요소가 정의된 것입니다:

- 0개의 타입을 인자로 받아서 타입을 리턴하는 무항 타입 생성자 `Nat :: *`
- 0개의 값을 인자로 받아서 `Nat` 타입의 값을 리턴하는 무항 자료 생성자 `Z :: Nat`
- `Nat` 타입의 값 1개를 인자로 받아서 `Nat` 타입의 값을 리턴하는 단항 자료 생성자 `S :: Nat -> Nat`

당연히, `Z`는 0을 뜻하고, `S`는 어떤 자연수의 "다음 (하나 큰)" 자연수를 뜻합니다.

뒷부분의 "`deriving Show`"에 대해서는 아직 설명하지 않았고 다음 수업 때 설명하겠지만, 간단히 말하자면 `Show` 타입 클래스에 (구체적인 인스턴스 선언 없이) 자동적으로 소속되기 위한 구문입니다. (타입 자체의 구조로부터 자동 추론해서, `Show` 클래스의 인스턴스가 되기 위한 적절한 메서드들—여기서는 `show` 함수 한 개—을 알아서 만들어줍니다.) 그래서, 다음과 같이 값을 표시해볼 수 있습니다:

[WinGHCi window, module HW3 loaded]

```
*HW3> S (S (S Z))
S (S (S Z))
it :: Nat
*HW3>
```

반면에, 저 "`deriving Show`"를 잠시 주석 처리하면:

file: HW3.hs, temporarily modified

```
data Nat =  Z
         |  S Nat
    -- deriving Show
```

여전히 타입도 확인해볼 수 있고 자연수에 대한 함수 정의도 가능하지만, 값을 표시해볼 수는 없게 됩니다:

[WinGHCi window, module HW3 (modified) loaded]

```
*HW3> :t S (S (S Z))
S (S (S Z)) :: Nat
*HW3> S (S (S Z))

<interactive>:66:1: error:
    ? No instance for (Show Nat) arising from a use of ‘print’
    ? In a stmt of an interactive GHCi command: print it
*HW3>
```

따라서, 만일 직접 효과를 확인해보기 위해 "`deriving Show`"를 주석 처리했다면, 숙제를 진행할 때에는 주석을 다시 풀기 바랍니다.

이제, 음이 아닌 정수 (`Integer` 혹은 `Int` 타입의 값) 하나가 인자로 주어지면 그에 대응하는 (산술적으로 같은) 자연수 (`Nat` 타입의 값) 하나를 리턴하는 함수 `toNat`을 만들어보겠습니다:

file: HW3.hs

```
toNat :: Integral i => i -> Nat
toNat 0          =  Z
toNat n | n > 0  =  S (toNat (n-1))
```

마지막 줄의 "`| n > 0`" 부분은 패턴 가드 (guard) 구문인데, 수업 시간에 지나가듯이 본 적은 몇 번 있지만 아직 제대로 설명하지 않았습니다. 추후 자세히 설명하겠지만 일단 간단히만 말하자면, 마지막 줄은 (0이 아닌) 모든 `n`에 대해 작동하는 정의가 아니라 0보다 큰 `n`에 대해서만 작동하는 정의입니다. 즉, 위의 `toNat` 함수는 정의역인 정수들 중 음수에 대해서는 정의가 안 되어있는 partial function입니다. 따라서, 이 함수에 음의 정수를 인자로 주면 이 문서 맨 첫머리에서 언급했던 바와 같이 "패턴이 완전히 포괄적이지 않다"는 에러메세지를 뿌립니다:

[WinGHCi window, module HW3 loaded]

```
*HW3> toNat (-5)
*** Exception: HW3.hs:(53,1)-(54,35): Non-exhaustive patterns in function toNat

*HW3>
```

물론, 0이나 양의 정수에 대해서는 제대로 작동합니다:

[WinGHCi window, module HW3 loaded]

```
*HW3> toNat 0
Z
it :: Nat
*HW3> toNat 5
S (S (S (S (S Z))))
it :: Nat
*HW3>
```

굳이 이런 패턴 가드를 붙여둔 이유는, 런타임 오류 중에서도 차라리 이렇게 명시적인 오류 메세지를 뿌리는 편이 "조용히 무한 재귀에 빠져버리는" 것보다는 훨씬 낫기 때문입니다.

이제 문제입니다: 자연수 (`Nat` 타입의 값) 하나를 인자로 받아서 그와 산술적으로 같은 정수 (`Integer` 혹은 `Int` 타입의 값) 하나를 리턴하는 함수

```
fromNat :: Integral i => Nat -> i
```

을 정의하십시오. 이 함수는 다음과 같이 동작해야 합니다:

[WinGHCi window, module HW3 loaded]

```
*HW3> fromNat (S (S (S (S (S Z)))))
5
it :: Integral i => i
*HW3> fromNat Z
```

```
0
it :: Integral i => i
*HW3>
```

힌트: 자연수가 가질 수 있는 "두 가지 구조"에 따라 각각 정의하면 됩니다. 또한, 이 함수의 계산 (시간) 복잡도는 인자 자연수의 크기에 선형으로 비례($O(n)$)합니다.

연계 문제입니다: 자연수 두 개를 인자로 받아서 그 둘의 합에 해당하는 자연수를 리턴하는 함수

```
addNat :: Nat -> Nat -> Nat
```

을 정의하십시오. 단, 인자 자연수들을 `fromNat`을 이용해서 Haskell 내장 정수로 바꾼 후 그들을 `+` 연산자로 더한 결과를 다시 `toNat`을 이용해서 자연수로 바꾸는 방식은 안됩니다. 반드시 (Haskell 내장 정수를 거치지 말고) 자연수 자체에 대해 직접적으로 덧셈을 정의하셔요.

또한, 계산 시간 복잡도가 첫번째 인자나 두번째 인자 중 무조건 어느 (고정된) 한 쪽의 크기에만 비례하게 만들지 말고, 두 인자들 중 작은 쪽의 크기에 비례하게 만드십시오.

이 함수를 만들면서 제대로 작동하는지 테스트할 때, 앞서 정의했던 `toNat`과 `fromNat`을 활용하십시오. 예를 들어, 다음과 같이 테스트해볼 수 있습니다:

[*WinGHCi window, module* HW3 *loaded*]

```
*HW3> fromNat (addNat (toNat 17) (toNat 26))
43
it :: Integral i => i
*HW3>
```

또 연계 문제입니다: 자연수 두 개를 인자로 받아서 그 둘의 곱에 해당하는 자연수를 리턴하는 함수

```
mulNat :: Nat -> Nat -> Nat
```

을 정의하십시오. 단, 위 `addNat`의 경우와 마찬가지로, 인자 자연수들을 `fromNat`을 이용해서 Haskell 내장 정수로 바꾼 후 그들을 `*` 연산자로 곱한 결과를 다시 `toNat`을 이용해서 자연수로 바꾸는 방식은 안됩니다. 반드시 (Haskell 내장 정수를 거치지 말고) 자연수 자체에 대해 직접적으로 곱셈을 정의하셔요.

물론, 이번에도 계산 시간 복잡도는 두 인자들 중 작은 쪽의 크기에 비례하게 만드십시오. 힌트:

$$(1+x)(1+y) = 1 + x + y + xy$$

한 번 더 연계 문제입니다: 자연수 하나를 인자로 받아서 그 자연수의 계승(factorial)에 해당하는 자연수를 리턴하는 함수

```
facNat :: Nat -> Nat
```

을 정의하십시오. Haskell 내장 정수를 거치지 말고 자연수 그대로... 이하 동문입니다.

마지막 연계 문제였던 `facNat`까지 정의하고 나면, WinGHCi 프롬프트에 다음과 같은 표현식을 입력하여 (테스트 겸) 평가해보기 바랍니다: (경고: 일단은 "5"까지만 테스트하셔요. 이유는 곧 알게 됩니다.)

[*WinGHCi window, module* HW3 *loaded*]
```
*HW3> map (fromNat . facNat . toNat) [0..5]
[1,1,2,6,24,120]
it :: Integral b => [b]
*HW3>
```

이제 WinGHCi 설정 창에서 "Print timing/memory stats after each evaluation" 부분을 체크하거나, 혹은 (GUI가 아닌 터미널 + ghci일 경우) 프롬프트에

[ghci *prompt in a terminal, module* HW3 *loaded*]
```
*HW3> :set +s
*HW3>
```

라고 명령합니다. 그 후, (본인의 컴퓨터 성능을 감안해가면서) 다음과 같이 진행해보기 바랍니다:

[*WinGHCi window, module* HW3 *loaded*]
```
*HW3> (fromNat . facNat . toNat) 5
120
it :: Integral i => i
(0.01 secs, 118,320 bytes)
*HW3> (fromNat . facNat . toNat) 6
720
it :: Integral i => i
(0.01 secs, 427,408 bytes)
*HW3> (fromNat . facNat . toNat) 7
5040
it :: Integral i => i
(0.02 secs, 2,664,136 bytes)
*HW3> (fromNat . facNat . toNat) 8
40320
it :: Integral i => i
(0.07 secs, 20,821,200 bytes)
*HW3> (fromNat . facNat . toNat) 9
362880
it :: Integral i => i
(0.65 secs, 186,886,568 bytes)
*HW3> (fromNat . facNat . toNat) 10
3628800
it :: Integral i => i
(6.59 secs, 1,868,336,080 bytes)
*HW3>
```

혹시 10이 놀랍지 않다면, 컴퓨터 성능이 좋은가보군요. 11을 넣어보셔요.

물론, 이 시간/공간 문제는 맨 처음에 작성했던 `fac :: Integer -> Integer` 함수에서는 일어나지 않는 문제입니다:

[*WinGHCi window, module* HW3 *loaded*]
```
*HW3> fac 10
3628800
```

```
it :: Integer
(0.00 secs, 64,144 bytes)
*HW3> fac 100
933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639
761565182862536979208272237582511852109168640000000000000000000000
it :: Integer
(0.00 secs, 332,792 bytes)
*HW3>
```

3 마치며

타입 클래스에 관해서는 기본 개념만 설명했을 뿐 코딩을 해볼 만큼 설명하지 않았기에, 관련 숙제는 일단 다음 주로 미뤘습니다. 게다가, 타입 클래스는 이 강좌의 요점이 아니어서, 어쩌면 (시험에 안 낼 생각이면) 숙제 없이 지나갈 수도 있습니다.

위에서 정의했던 자연수는 시간 효율도 공간 효율도 그야말로 극악무도합니다. 당연히, 실제로 이런 식으로 자연수 자료형을 만들어서 쓰라는 의미로 낸 숙제가 아닙니다. 마지막에 만들어본 `facNat`의 의미는 "복잡도를 고려하지 않은 자료형과 알고리즘"이 얼마나 문제가 큰지를 느껴보라는 뜻입니다.

그리고, 이것은 절자형 언어냐 함수형 언어냐의 여부와도 상관 없는 문제입니다. 왜 그런지는 다음번 숙제와 다음 다음번 숙제를 통해 실제로 느껴볼 수 있을 겁니다.

(느리게 짜니까 느린 거에요. 함수형 언어라서 느린 게 아니라구요!)