숙제 #4: 함수 재귀 Part 1

건국대학교 소프트웨어학과 2019 컴퓨테이션이론 담당: 차리서

이 문서는 4차 숙제에서 제출할 `HW4-b2.hs` 파일에 대한 설명이자 문제지이며, 비록 제목에 "Part 1"이라는 말이 있지만 이 문서는 분명히 이 버전이 **완성판**입니다. (이후 소소한 오류 수정 정도는 있을 수 있습니다.)

이 문서의 내용은 11/15(금) 낮에 업로드했던 "미완성판 b2"의 내용과 거의 똑같습니다. 정확히 말하자면, 숙제 문제 및 관련 내용 설명은 완전히 똑같고, 이 문서 자체에 대한 (메타-) 설명과 제목, 조판 형식 등만 약간 바뀌었을 뿐입니다.

원래는 그 "미완성판 b2"의 뒷부분에 숙제 문제 및 설명이 추가된 문서를 11/15(금) 저녁까지 다시 업로드할 예정이었지만, 계획을 바꿨습니다: 그 "미완성판 b2"의 내용 그대로인 (뒷부분에 문제나 설명이 추가되지 않은) 지금 이 문서를 4차 숙제용 문서의 완성판으로 삼아서 4차 숙제는 이걸로 마무리하고, 그 "미완성판 b2"의 뒷부분에 원래 추가하려던 문제와 설명들은 아예 별도의 문서로 만들어서 별도의 기한까지 5차 숙제로 따로 제출하게 하려고 합니다. 이유는, 추가 내용 작성 시간이 생각보다 너무 길어지는 바람에 예정대로 진행하기에는 제출 마감까지의 남은 시한이 너무 촉박해졌고, 그렇다고 제출 마감을 연장하기에는 기존 마감 시한을 믿고 열심히 주말에 고생한 학생들에게 허탈감을 줄 수 있어서, 이미 (미완성판 b2를 통해) 게시됐던 문제들은 예정했던 마감 시한까지 받고 이후 추가되는 내용은 풀이 시간을 더 주기 위해서입니다.

즉, 이 4차 숙제의 정시 제출 마감은 예정대로 11/17(일) 오전 11시까지입니다. 기말까지 (약간의 지각 감점과 함께) 지각 제출을 받긴 하지만, 11/18(월) 오전 수업 시간에 이 4차 숙제에 대한 풀이와 설명을 진행할 예정이어서 그 이후에는 (답을 보고나면) 숙제의 취지인 "연습" 효과가 떨어질 수도 있습니다.

참고로, 하나 더 이전 판인 "미완성판 b1"과 비교해서 1절의 내용은 전혀 바뀌지 않았고 2절만 추가됐습니다.

Contents

1 함수의 반복 적용

1.1 직접적 정의

어떤 단항 함수의 정의역과 공변역이 서로 (어떤 임의의 t 타입으로서) 같다면, 이러한 단항 함수 $f : t \to t$를 t 타입의 인자 $x : t$에 적용한 그 결과인 $f\ x$도 역시 t 타입이기 때문에 (즉, $f\ x : t$), 이 결과에 이 함수를 몇

번이라도 다시 적용할 수 있습니다.

$$f\ (f\ (f\ (f \cdots (f\ x) \cdots)))$$

이렇게 함수 f를 인자 x에 여러번 반복 적용할 경우, 그 반복 적용 횟수를 함수의 윗첨자로 표시합니다. 예를 들어,

- f를 x에 한 번도 적용하지 않은 그냥 x 자체는 $f^0\ x$와 같고,
- f를 x에 한 번만 적용한 $f\ x$는 $f^1\ x$며,
- f를 x에 다섯 번 적용한 $f\ (f\ (f\ (f\ (f\ x))))$는 $f^5\ x$입니다.

참고로, 이런 윗첨자 표기는 역함수 표기인 f^{-1}과도 일맥 상통합니다만, 이 부분은 이번 숙제와는 관련이 멀어지므로 생략하고 이번 숙제에서는 윗첨자 자리에는 자연수만 들어갈 수 있고 음수는 들어갈 수 없다고 정하겠습니다. 따라서, 이 "함수의 윗첨자 표기"를 정형적으로 정의하면 다음과 같습니다. n이 정수일 때:

$$f^n\ x = \begin{cases} \texttt{ERROR} & \text{if } n < 0 \\ x & \text{if } n = 0 \\ f^{n-1}\ (f\ x) & \text{otherwise} \end{cases} \quad \text{혹은} \quad f^n\ x = \begin{cases} \texttt{ERROR} & \text{if } n < 0 \\ x & \text{if } n = 0 \\ f\ (f^{n-1}\ x) & \text{otherwise} \end{cases}$$

이 두 정형적 정의는 수학적 관점에서는 의미가 완전히 똑같지만, 전산학적 관점에서는 차이가 있습니다. 왼쪽 정의는 꼬리 재귀(tail recursion)로서 재귀 호출 이후로 연기되어 남겨진 연산(deferred operation)이 전혀 없지만, 오른쪽 정의는 꼬리재귀가 아니며 재귀 호출 이후 그 결과에 f를 한 번 적용하는 연산이 계속 남겨져 쌓입니다.

문제입니다: 위 두 정형적 정의들 중 왼쪽 (꼬리 재귀) 정의에 그대로 따라서, 순서대로 세 개의 인자:

- 정의역과 공변역이 같은 단항 함수 `f`,
- 정수 (`Integer` 혹은 `Int` 타입의 값) `n`,
- 위 함수의 정의역의 값 `x`

를 받아서 함수 `f`를 `x`에 `n`번 반복 적용한 결과를 리턴하는 함수

```
iterApp :: Integral i => (t -> t) -> i -> t -> t
```

을 정의하십시오. 힌트:

- 정수 인자의 값에 따라 경우를 나누되, 0인 경우와 0이 아닌 경우로 나누는 것이 아니라, 0보다 작은 경우와 0인 경우와 그 외의 경우로 나눠야 하며, 이를 위해서는 패턴 매칭이 아니라 패턴 가드 (guard) 문법이 필요할 겁니다.
- 가드 속에서 값을 비교할 때, 동치 비교 연산자는 `==` 이고 대소 비교 연산자는 `<`입니다.
- 정수 인자가 0보다 작을 때 명시적으로 오류를 일으켜야 하는데, 이를 위해서는 온라인 입문서의 3.5 절에 나왔던 `error :: String -> a` 함수를 사용하면 됩니다.

정의된 `iterApp` 함수는 다음과 같이 작동해야 합니다:

[*WinGHCi window, module* HW4 *loaded*]

```
*HW4> iterApp (+1) 10 0
10
it :: Num t => t
*HW4> iterApp (*2) 10 1
1024
it :: Num t => t
*HW4> iterApp (*2) 0 1
1
it :: Num t => t
*HW4> iterApp (*2) (-1) 1
*** Exception: Cannot apply negative times!
CallStack (from HasCallStack):
  error, called at HW4-r2.hs:5:21 in main:HW4
*HW4>
```

관련 문제입니다: 앞서의 두 정형적 정의들 중 오른쪽 정의에 그대로 따라서, `iterApp`과 수학적으로는 똑같지만 전산학적으로는 지연된 연산을 남기며 재귀하는 함수 `iterApp'`을 정의하십시오.

힌트: `iterApp`과는 재귀 호출 부분만 다르고 나머지는 똑같습니다.

1.2 간접적 정의

앞서 정의했던 `iterApp`과 `iterApp'`은 주어진 함수가 적용될 초기 인자(매개변수명 "x")를 직접 이용해서 정의되어 있습니다. 하지만, 살짝 다른 관점에서 생각해보면 이 "함수의 윗첨자 표기"를 초기 인자 "x" 없이 함수 이름과 적용 횟수만 가지고도 정의할 수 있습니다.

$$f\ (f\ (f\ (f\ (f\ x)))) = (f \circ (f \circ (f \circ (f \circ f)))) \ x$$

함수 합성 연산자 $\circ$는 원래 오른쪽부터 먼저 묶이므로, 위 식에서 우변의 괄호들 중 일부 (최외곽 괄호를 제외한, 내부) 괄호들은 필요 없었습니다. 따라서:

$$f\ (f\ (f\ (f\ (f\ x)))) = (f \circ f \circ f \circ f \circ f) \ x$$

게다가, 사실은 $\circ$는 결합법칙이 성립하는 연산자여서, 왼쪽을 먼저 명시적으로 묶는다고 해도 결과는 똑같습니다:

$$f\ (f\ (f\ (f\ (f\ x)))) = ((((f \circ f) \circ f) \circ f) \circ f) \ x$$

즉, "f^n이란 n개의 함수 f들을 줄줄이 합성해놓은 것"이라고도 볼 수 있습니다. 단, 함수의 합성이란 기본적으로 두 개의 함수를 서로 합성하는 개념이기 때문에, 주의할 사항이 두 가지 있습니다:

- f^1은 1 개의 함수를 합성한 결과인데, 이 f^1을 x에 적용하면 그 결과는 $f\ x$여야 합니다. 따라서, 1 개의 함수를 합성한 결과는 그 함수 그대로입니다.

- f^0은 0 개의 함수를 합성한 결과인데, 이 f^0을 x에 적용하면 그 결과는 x 그대로여야 합니다. 따라서, 0 개의 함수를 합성한 결과는 인자 하나를 받아서 그 인자를 그대로 리턴하는 단항 함수, 즉 항등 함수 (identity function)여야 합니다.

항등 함수를 표현하기 위해서 f^0을 $\lambda x.x$라고 (즉, `\x -> x` 라고) 정의해도 됩니다만, 사실은 Haskell에는 항등 함수 `id`가 이미 정의되어 있습니다:

built-in module: `GHC.Base`

```haskell
-----------------------------------------
-- The function type
-----------------------------------------

-- | Identity function.
--
-- > id x = x
id                    :: a -> a
id x                  =  x
```

따라서, f^n의 정형적 정의는 다음과 같습니다:

$$f^n = \begin{cases} \texttt{ERROR} & \text{if } n < 0 \\ \texttt{id} & \text{if } n = 0 \\ f & \text{if } n = 1 \\ f \circ f^{n-1} & \text{otherwise} \end{cases}$$

그런데, 0이 덧셈에 대한 항등원이고 1이 곱셈에 대한 항등원이듯이, 사실 `id`는 함수 합성에 대한 항등원입니다:

$$\forall T^{\texttt{Type}}, \forall f^{T \to T}, (f \circ \texttt{id}) = f \wedge (\texttt{id} \circ f) = f$$

따라서, 위의 정형적 정의 중 세번째 줄 (n이 1인 경우)은 별도로 필요하지 않았으며, f^n의 정형적 정의는 최종적으로 다음과 같이 정리됩니다:

$$f^n = \begin{cases} \texttt{ERROR} & \text{if } n < 0 \\ \texttt{id} & \text{if } n = 0 \\ f \circ f^{n-1} & \text{otherwise} \end{cases}$$

(수들의 합이 0에 그 수들을 줄줄이 더한 것이고 수들의 곱이 1에 그 수들을 줄줄이 곱한 것이듯이, 함수들의 합성이란 `id`에 그 함수들을 줄줄이 합성한 것입니다.)

여기서 문제입니다: 순서대로 두 개의 인자:

- 정의역과 공변역이 같은 단항 함수 `f`,
- 정수 (`Integer` 혹은 `Int` 타입의 값) `n`

만을 받아서 함수 `f^n`을 리턴하는 함수

```
iterApp'' :: Integral i => (t -> t) -> i -> t -> t
```

을 정의하십시오. (인자의 갯수가 다른데 타입이 왜 `iterApp`이나 `iterApp'`이랑 똑같은지는 각자 생각해보기 바랍니다.) 단,

- 람다 구문(`\x ->`)을 일체 사용하지 말고
- `where` 절도 사용하지 말고
- 앞서 정의했던 `iterApp`과 `iterApp'`도 사용하지 말고,
- 대신 내장 함수 `id`와 중위 연산자 `.`를 이용해서 정의하십시오.

이 함수 `iterApp''`은 꼬리 재귀가 아니어도 상관 없습니다.

2 반복 적용열

때로는, 최종 $f^n\ x$ 값 뿐만 아니라 반복 적용 각 단계 마다의 값인

$$f^0\ x,\ f^1\ x,\ f^2\ x,\ \ldots,\ f^n\ x$$

들이 순서대로 모두 필요할 때도 있습니다. 따라서, 초기 인자에 함수를 0번 적용한 결과(초기 인자 자체)부터 n번 반복 적용한 결과까지를 순차적으로 원소로 갖는 리스트를 만드는 삼항 함수를 생각해 볼 수도 있습니다. 즉, 이 삼항 함수에 음이 아닌 정수 n을 두번째 인자로 주면 결과 리스트의 길이는 언제나 $n+1$이 됩니다. (두번째 인자가 0일 경우 세번째 인자 (즉, 초기 인자) 하나만을 원소로 갖는 길이 1짜리 리스트가 만들어지고, 두번째 인자가 5일 경우 $f^0\ x$부터 $f^5\ x$까지 총 6개의 원소를 갖는 리스트가 만들어지기 때문입니다.)

그런데, 실은 Haskell에는 이와 (똑같지는 않지만) 비슷한 이항 함수가 이미 정의되어 있습니다:

built-in module: GHC.List

```
440 -- | 'iterate' @f x@ returns an infinite list of repeated applications
441 -- of @f@ to @x@:
442 --
443 -- > iterate f x == [x, f x, f (f x), ...]
444 --
445 -- Note that 'iterate' is lazy, potentially leading to thunk build-up if
446 -- the consumer doesn't force each iterate. See 'iterate\'' for a strict
447 -- variant of this function.
448 {-# NOINLINE [1] iterate #-}
449 iterate :: (a -> a) -> a -> [a]
450 iterate f x =  x : iterate f (f x)
```

위에서 언급했던 삼항 함수와의 차이는, 이 이항 함수 `iterate`는 반복 횟수를 인자로 받지 않고 그냥 무한 리스트를 만들어버린다는 차이 뿐입니다.

[WinGHCi window]

```
Prelude> iterate (*2) 1
[1,2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384,32768,65536,131072,262144,524288,
1048576,2097152,4194304,8388608,16777216,33554432,67108864,134217728,268435456,536870912,
```

```
1073741824,2147483648,4294967296,8589934592,17179869184,34359738368,68719476736,137438953
472,274877906944,549755813888,1099511627776,2199023255552,4398046511104,8796093022208,175
92186044416,35184372088832,70368744177664,140737488355328,281474976710656,562949953421312
,1125899906842624Prelude> Interrupted.

Prelude>
```

Haskell에는 이런 식으로 무한 리스트를 만드는 함수들이 무척 많은데, 일단 "원본 데이터"는 이런 식으로 무한 리스트로 만들어 놓고 중간 과정에서도 계속 무한 리스트로 다루다가, 최종적으로 필요할 때 `take` 함수로 필요한 만큼의 앞부분만 계산해서 떼어내거나, `!!` 중위연산자로 필요한 인덱스의 원소만 꺼내올 수 있기 때문입니다.

[*WinGHCi window*]

```
Prelude> take 11 (iterate (*2) 1)
[1,2,4,8,16,32,64,128,256,512,1024]
it :: Num a => [a]
Prelude> iterate (*2) 1 !! 10
1024
it :: Num a => a
Prelude>
```

따라서, 위에서 언급했던 $f^0\ x$부터 $f^n\ x$까지의 리스트를 만드는 삼항 함수는 다음과 같이 정의할 수 있습니다:

file: HW4-r3.hs

```
progress0 f n x  =  take (n+1) (iterate f x)
```

하지만, 이 숙제에서는 `iterate`를 일체 사용하지 않고 정의하는 훈련을 하려고 합니다. `iterate`를 사용한 위 정의에 관해서는 추후 2.6절에서 (숙제와는 관련 없이) 좀 더 부연하겠습니다.

지금부터 이어지는 각 소절마다 이 삼항 함수를 각각 다른 방식으로 정의해보게 됩니다. 각 방식마다 제시하는 (지켜야하는) 각각의 규칙이 있습니다만, 그에 앞서 **모든 방식들에서 공통적으로 지켜야하는 규칙**은 다음과 같습니다:

- `iterate`나 `take`는 사용하지 말아야합니다.
- 반복 횟수로 음수가 주어지면 명시적으로 오류 메세지를 뱉어야 합니다. (아래 템플릿 팁 참고.)
- 위 `progress0`를 포함해서, 다른 방식으로 정의된 다른 `progress#` 시리즈를 사용해선 안됩니다. (예를 들어, `progress2`을 정의할 때 `progress1`이나 `progress3`을 사용해선 안됩니다.) 물론, 각 문제에서 허용/요구하는 경우 자기 자신을 재귀 호출할 수는 있습니다.
- 계산 (시간/공간) 복잡도는 신경쓸 필요 없고, 꼬리 재귀 형태가 아니어도 됩니다.

또한, (다는 아닐지라도) 여러 소절에 공통되는 팁들이 좀 더 있습니다. 첫째로, 추후 좀 더 자세히 설명하겠지만, 재귀적 함수 정의란 귀납적 논리 추론의 쌍둥이입니다. 귀납적 논리 추론의 핵심은, 일단 귀납적 추론 없이 즉시 확실히 증명할 수 있는 일부 사항들을 증명해놓고 (이걸 "귀납 기초"라고 부릅니다), 어떤 일부가 이미 (어떤 방식으로든) 증명되어 있다고 가정할 때 (이걸 "귀납 가설"이라고 부릅니다) 그 귀납 가설과 위에서

이미 증명한 귀납 기초들을 어떻게 이용하고 조합해서 최종 결론을 증명할지를 생각하는 것입니다. 마찬가지로 재귀적 함수 정의의 핵심은, 일단 재귀 없이 즉시 만들어낼 수 있는 일부 값들을 만들어놓고 (이걸 "종료 조건"이라고 부릅니다), 어떤 일부 값이 이미 (어떤 방식으로든) 계산되어 만들어져있다고 가정할 때 (특별히 통일된 명칭은 없지만, 여기서는 "재귀 호출의 리턴값"이라고 부르겠습니다) 이 재귀 호출의 리턴값과 위에서 이미 만들어둔 종료 조건들을 어떻게 이용하고 조합해서 최종 리턴값을 만들어낼지를 생각하는 것입니다.

다른 한 가지 팁은, 우리가 앞으로 각 소절들마다 하나씩 만들고자 하는 삼항 함수들은 재귀 호출 과정에서 중간값으로 빈 리스트를 만들어내는 일은 있어도 최종적으로는 (차라리 오류를 뱉으면 뱉었지) 빈 리스트는 결코 만들어내지 않는다는 점입니다. 반복 횟수가 음수면 오류를 일으킬 예정이며, 반복 횟수가 0번이면 $f^0\ x$, 즉 초기 인자인 x 자체를 유일한 원소로 갖는 길이 1짜리 리스트가 만들어져야하기 때문입니다. 따라서, 최종 리스트를 둘로 쪼개서 생각할 때, 최종 리스트가 (둘로 쪼갤 수 없는) 빈 리스트인 경우는 걱정할 필요가 없습니다.

마지막 공통 팁은, 이후 소절에서 정의할 함수들은 모두 다음과 같은 템플릿을 공유한다는 사실입니다.

common template

```
progress# :: Integral i => (t -> t) -> i -> t -> [t]
progress# f n x
    | n <  0     =  error "Cannot apply negative times!"
    |
```

각 소절의 답을 작성할 때, 위 템플릿 중 두 곳의 "#" 문자를 적절한 숫자로 바꾸고 아래쪽 빈 부분을 채우면 됩니다.

2.1 앞부분 리스트와 마지막 원소로 나누기

최종적으로 만들고자 하는 리스트인

$$[\ f^0\ x,\ f^1\ x,\ f^2\ x,\ \ldots,\ f^n\ x\ ]$$

를 다음과 같이 "마지막 원소를 제외한 (어쩌면 비어있을지도 모르는) 앞부분 리스트"와 "마지막 원소 하나만을 원소로 갖는 길이 1짜리 리스트"라는 두 리스트로 나누어 생각해볼 수 있습니다:

$$[\ f^0\ x,\ f^1\ x,\ f^2\ x,\ \ldots,\ f^{n-1}\ x\ ] \qquad [\ f^n\ x\ ]$$

예를 들어, 반복 횟수가 3일 경우에는 다음과 같은 두 리스트로 나누어 생각할 수 있고:

$$[\ f^0\ x,\ f^1\ x,\ f^2\ x\ ] \qquad [\ f^3\ x\ ]$$

반복 횟수가 0일 경우에는 다음과 같은 두 리스트로 나누어 생각할 수 있습니다:

$$[\,] \qquad [\ f^0\ x\ ]$$

즉, f와 n, x가 주어졌을 때 위의 두 리스트

$$[\ f^0\ x,\ f^1\ x,\ f^2\ x,\ \ldots,\ f^{n-1}\ x\ ] \qquad [\ f^n\ x\ ]$$

를 각각 어떻게 표현해야할지, 그리고 그 둘을 어떻게 "연결"해야할지를 생각하면 됩니다.

팁으로서, 이렇게 리스트를 둘로 나누면 왼쪽 리스트의 맨 마지막 원소의 함수 윗첨자가 $n-1$이 되는데, 이 $n-1$자리에 (심지어 n이 0일 때에도) 실제로 음수가 들어가는 일은 발생하지 않는다는 점에 주목하십시오. n이 3일 때는 왼쪽 리스트의 마지막 원소가 $f^2\ x$지만, n이 0일 때는 왼쪽 리스트의 마지막 원소가 $f^{-1}\ x$가 되는 게 아니라 그냥 왼쪽 리스트가 빈 리스트가 됩니다. (앞서 공통 팁 부분에서, 재귀 호출의 중간 과정에서는 빈 리스트가 만들어지기도 한다고 이야기했었습니다.)

문제입니다: 바로 위에서 설명한 발상을 그대로 따라서, 단항 함수 f 하나와 반복 횟수 n, 초기 인자 x를 받아서

$$[\ f^0\ x,\ f^1\ x,\ f^2\ x,\ \ldots,\ f^n\ x\ ]$$

라는 리스트를 만들어내는 삼항 함수

```
progress1 :: Integral i => (t -> t) -> i -> t -> [t]
```

을 정의하십시오. 단, 뒤에서 설명할 `map` 함수는 아직 사용해선 안되며, 산술 수열 (`[0..n]` 문법) 또한 아직 사용해선 안됩니다.

힌트: 이 발상에 따라 구현할 때에는 필연적으로 `progress1` 자신을 재귀호출하게 되고, 앞서 정의했던 `iterApp` 함수가 필요해집니다. 또한, 두 리스트를 어떻게 "연결"하는지는 다른 숙제나 수업시간에 몇 번 소개했었습니다. 결정적으로, 비록 최종 결과 리스트를 "앞부분"과 "마지막"으로 나누어 생각하라고 하긴 했지만, 이 말은 재귀 호출 부분을 작성할 때 최종 결과를 만들어내는 방식을 그렇게 생각하라는 뜻이지, 함수 정의문의 매개변수 가드 문법을 그렇게 나누라는 뜻이 아닙니다. 가드 문법은 (템플릿 팁에 이미 반복 횟수가 음수인 경우가 따로 나누어져 있듯이) 그저 반복 횟수만을 기준으로 나누면 됩니다. (반복 횟수가 0인 경우를 종료 조건으로서 따로 나눠야 할 겁니다.)

2.2 첫 원소와 뒷부분 리스트로 나누기: without map

이번에는 반대로, 최종 결과 리스트

$$[\ f^0\ x,\ f^1\ x,\ f^2\ x,\ \ldots,\ f^n\ x\ ]$$

를 다음과 같이 "첫번째 원소"와 "첫번째 원소를 제외한 (어쩌면 비어있을지도 모르는) 나머지 뒷부분 리스트"로 나누어 생각해볼 수 있습니다:

$$f^0\ x \qquad [\ f^1\ x,\ f^2\ x,\ \ldots,\ f^n\ x\ ]$$

(나누어진 두 부분 중 하나가 원소 한 개 짜리 리스트였던 앞서와는 달리, 이번에는 왼쪽 부분이 첫번째 원소 하나를 유일한 원소로 갖는 길이 1 짜리 리스트가 아니라 그냥 "첫번째 원소 그 자체"인 점에 주의하기 바랍니다.) 여기서 오른쪽 리스트인

$$[\ f^1\ x,\ f^2\ x,\ \ldots,\ f^n\ x\ ]$$

는 첫번째 원소가 $f^0\ x$가 아닌 $f^1\ x$로 시작하므로 마치 우리가 정의하려는 삼항 함수로는 만들어낼 수 없는 (결과물로서 적합한 형태가 아닌) 리스트처럼 보일 수도 있지만, 발상을 조금만 달리 해보면 이 리스트도 우리가 정의하려는 삼항 함수로 만들어낼 수 있습니다. 즉, 이 리스트가 어떤 초기 인자에 어떤 함수를 몇 번 반복

적용한 결과일지를 떠올리면 됩니다. 힌트는, 1보다 크거나 같은 모든 k에 대해서

$$f^k\ x\ =\ f^{k-1}\ (f\ x)$$

라는 사실입니다.

문제입니다: 바로 위에서 설명한 발상을 그대로 따라서, 단항 함수 f 하나와 반복 횟수 n, 초기 인자 x를 받아서

$$[\ f^0\ x,\ f^1\ x,\ f^2\ x,\ \ldots,\ f^n\ x\ ]$$

라는 리스트를 만들어내는 삼항 함수

```
progress2 :: Integral i => (t -> t) -> i -> t -> [t]
```

를 정의하십시오. 단, 뒤에서 설명할 `map` 함수는 아직 사용해선 안되며, 산술 수열 (`[0..n]` 문법) 또한 아직 사용해선 안됩니다. 또한, 앞서 정의했던 `iterApp` 함수도 이번에는 사용해선 안됩니다.

힌트: 이 발상에 따라 구현할 때에는 필연적으로 `progress2` 자신을 재귀호출하게 되지만, 앞서 정의했던 `iterApp` 함수는 필요 없습니다. 또한, 새 원소를 기존 리스트 맨 앞에 추가한 새 리스트를 어떻게 만드는지는 너무 여러번 소개되었고 사용도 해봤습니다. 이번에도, 반복 횟수가 0인 경우를 종료 조건으로서 따로 나눠야 할 겁니다.

2.3 첫 원소와 뒷부분 리스트로 나누기: with map

앞 절에서와 똑같이 최종 결과 리스트

$$[\ f^0\ x,\ f^1\ x,\ f^2\ x,\ \ldots,\ f^n\ x\ ]$$

를 다음과 같이 둘로 나누어 놓고 생각하겠습니다:

$$f^0\ x \qquad [\ f^1\ x,\ f^2\ x,\ \ldots,\ f^n\ x\ ]$$

단, 이번에는 오른쪽 리스트를 만들어내는 다른 방법을 생각해보고자 합니다.

수업 시간에 여러 번 소개했던 `map` 함수는 리스트의 각 원소마다 일괄적으로 똑같은 함수를 전부 적용해줍니다. 즉, `f`가 단항 함수고 `lst`가 `f`의 정의역의 값들로 이루어진 리스트일 때, 다음 두 결과는 서로 같습니다:

```
map f lst  =  [ f e | e <- lst ]
```

이 `map` 함수를 이용하면, 위에서 최종 결과 리스트를 둘로 나눈 부분들 중 오른쪽 리스트인

$$[\ f^1\ x,\ f^2\ x,\ \ldots,\ f^n\ x\ ]$$

를 약간 다른 방식으로 만들 수 있습니다. 즉, 이 리스트가 어떤 리스트에 어떤 함수를 일괄적으로 "mapping" 한 결과일지를 떠올리면 됩니다. 이 발상의 핵심은, 1보다 크거나 같은 모든 k에 대해서

$$f^k\ x\ =\ f\ (f^{k-1}\ x)$$

라는 사실입니다. (앞 절에서 제시했던 등식과 비슷해보이지만 명백히 다르게 생겼습니다. 유심히 보기 바랍니다.)

문제입니다: 바로 위에서 설명한 발상을 그대로 따라서, 단항 함수 f 하나와 반복 횟수 n, 초기 인자 x를 받아서

$$[\, f^0\, x,\ f^1\, x,\ f^2\, x,\ \ldots,\ f^n\, x\,]$$

라는 리스트를 만들어내는 삼항 함수

```
progress3 :: Integral i => (t -> t) -> i -> t -> [t]
```

를 정의하십시오. 단, 반드시 `map` 함수를 사용해야 하며, 산술 수열(`[0..n]` 문법)과 `iterApp` 함수는 사용해선 안됩니다.

힌트: 이 발상에 따라 구현할 때에는 필연적으로 `progress3` 자신을 재귀호출하게 됩니다. 그 외의 힌트는 앞 절의 `progress2`의 경우와 같습니다.

2.4 재귀 호출 없이 map과 산술 수열로 정의하기

산술 수열은 수(數)들이 순차적으로 나열된 리스트를 만들어내는 특수 문법이며, 사실은 수 뿐만 아니라 "다음 것"을 말할 수 있는 모든 타입(즉, `Enum` 클래스의 소속 타입)의 값들의 나열을 만들 수 있습니다. 예를 들어, 0부터 5까지 순서대로 나열된 리스트는

```
[0..5]
```

라고 입력하여 만들 수 있고, 심지어 `m`과 `n`이 정수일 경우

```
[m..n]
```

처럼 변수를 사용하여 만들 수도 있습니다.

이번에는 최종적으로 만들려는 리스트 자체를 부분 별로 쪼개는 대신, 최종적으로 만들려는 리스트의 각 원소 값들을 기준으로 생각해보겠습니다. ("미완성인 일부를 이용해서 완성된 전체를 만든다"는 차원에서는, 앞서 리스트를 부분 별로 쪼갰던 방식이나 지금 이 절에서 리스트 원소 값들을 그 값의 "미완성된 일부 구성 부분"에 따라 생각하는 방식이나 근본적으로는 같은 맥락입니다.) 즉, 다음과 같은 정수들의 리스트:

$$[\, 0,\ 1,\ 2,\ \ldots,\ n\,]$$

의 각 원소마다 어떤 함수를 일괄적으로 mapping하면 다음과 같은 리스트:

$$[\, f^0\, x,\ f^1\, x,\ f^2\, x,\ \ldots,\ f^n\, x\,]$$

가 만들어질지를 떠올리면 됩니다. 다시 말해서, 정수 k를 인자로 받아서 $f^k\, x$를 리턴하는 함수를 생각하면 됩니다.

문제입니다: 바로 위에서 설명한 발상을 그대로 따라서, 단항 함수 f 하나와 반복 횟수 n, 초기 인자 x를 받아서

$$[\, f^0\, x,\ f^1\, x,\ f^2\, x,\ \ldots,\ f^n\, x\,]$$

라는 리스트를 만들어내는 삼항 함수

```
progress4 :: Integral i => (t -> t) -> i -> t -> [t]
```

를 정의하십시오. 단, 반드시 `map` 함수와 산술 수열 문법을 사용해야 하며, 이번에는 `progress4` 자신을 재귀 호출하면 안됩니다. `iterApp` 함수는 필요하다면 사용할 수 있습니다.

힌트: 이 발상에 따라 구현할 때에는 (이미 템플릿에 나누어져있는) 반복 횟수가 음수인 경우 외에는 더이상 추가로 경우를 나눌 필요가 없습니다. (재귀 호출이 없으니 종료 조건도 필요 없다는 뜻입니다. 게다가, `[0..0]`의 결과는 `[0]`입니다.) 또한, 필요하다면 람다 구문("`\e ->`" 형식)을 사용할 수 있습니다.

2.5 progress 정의 선택

이 소절에서 할 일은 간단합니다. 이후 고정점에 대한 접근열을 만들 때 사용하기 위해 `progress` 함수를 정의하는데, 새로 정의하는 게 아니라 앞서 정의했던 `progress1`부터 `progress4`까지 중 하나를 선택해서 그것을 `progress`로 사용합니다. 이 소절에 한해서, 앞서 "다른 `progress#`를 이용해서 정의하면 안된다"는 규칙은 적용하지 않습니다.

문제입니다: 숙제 코드 파일의 주석과 예제를 참고하여, 주석을 풀고 `progress`를 정의하십시오. 넷 중 본인이 성공적으로 정의한 것 하나를 마음대로 고르기만 하면 됩니다.

2.6 참고: iterate와 take

이 소절은 숙제 문제와는 직접 상관이 없으니 시간이 부족하면 패스해도 됩니다만, 기말 시험과는 관련이 있을 수도 있으니 아예 버리진 말고 언젠가는 읽어보기 바랍니다.

앞서 `iterate`를 이용해서 `progress0`를 다음과 같이 정의했었습니다:

file: HW4-r3.hs

```
progress0 :: Integral i => (t -> t) -> i -> t -> [t]   -- incorrect
progress0 f n x  =  take (n+1) (iterate f x)
```

다만, 이렇게 정의한 `progress0` 함수는 1절에서 정의하던 스타일과는 두 가지 측면에서 다릅니다.

첫째로, 1절에서 반복 횟수를 나타내는 인자는 `Integral i => i` 타입이었습니다. 즉, `Integer` 타입 혹은 `Int` 타입 중 어느 타입이든 가능했었습니다. 반면에, 지금 이 `progress0` 함수의 타입을 위 코드에서와 같이

```
progress0 :: Integral i => (t -> t) -> i -> t -> [t]
```

라고 명시적으로 지정해주면 오류를 일으킵니다:

[WinGHCi window, failing to load module HW4]

```
*HW4> :r
[1 of 1] Compiling HW4              ( HW4-r3.hs, interpreted )
Failed, no modules loaded.
Prelude>
HW4-r3.hs:29:30: error:
    ? Couldn't match expected type ‘Int’ with actual type ‘i’
      ‘i’ is a rigid type variable bound by
```

```
        the type signature for:
          progress0 :: forall i t. Integral i => (t -> t) -> i -> t -> [t]

      ... (중략) ...

Prelude>
```

이런 오류 메세지를 만날 때에는, 문제를 해결하는 방법은 대체로 오류 메세지 속에 있습니다. 기대했던 타입인 'Int'를 실제 (명시적으로 지정한) 타입인 'i'에 대응시킬 방법이 없다고 말하고 있습니다.

이 오류 메세지만으로는 무엇이 문제인지 파악하기 어려울 경우 그 다음으로 시도해볼 일은, 자신이 직접 명시적으로 지정해주었던 타입을 주석 처리하고 Haskell이 타입을 어떻게 추론하는지 확인해보는 것입니다. (이 과정에서 자신이 뭔가 잘못 생각했었다는 사실과, 구체적으로 무엇을 어떻게 잘못 기획/설계했는지 알게 됩니다.)

file: HW4-r3.hs

```haskell
-- progress0 :: Integral i => (t -> t) -> i -> t -> [t]   -- incorrect
progress0 f n x  =  take (n+1) (iterate f x)
```

[WinGHCi window, module HW4 loaded]

```
Prelude> :r
[1 of 1] Compiling HW4              ( HW4-r3-ans.hs, interpreted )
Ok, one module loaded.
*HW4> :t progress0
progress0 :: (a -> a) -> Int -> a -> [a]
*HW4>
```

결국 progress0 함수는 반복 횟수 인자로 Integer 타입의 값은 받지 못하고 반드시 Int 타입의 값을 받아야 한다는 뜻입니다.

이 원인은 progress0를 정의하는 과정에서 내장 함수 take를 사용했기 때문입니다. take는 자신의 첫번째 인자(가져올 리스트의 길이)로서 반드시 Int 타입의 인자를 요구하며, 사실은 리스트의 길이나 인덱스를 다루는 거의 대부분의 다른 내장 함수들도 그 길이나 인덱스로 Int 타입의 값만 허용합니다. 최대/최소 한계치가 없는 Integer라는 (범용성 측면에서) 더 좋은 정수형이 있는데도 굳이 Int를 강제하는 건 아무래도 효율성 때문이며, 실은 Int 타입으로도 거의 대부분의 경우에는 충분합니다. Int는 최대/최소 한계치가 있고 각각 다음과 같은데:

[WinGHCi window]

```
Prelude> (minBound, maxBound) :: (Int, Int)
(-9223372036854775808,9223372036854775807)
it :: (Int, Int)
Prelude>
```

이는 C 언어의 signed long long int 타입의 크기 및 범위와 같습니다. (minBound와 maxBound는 Bounded 클래스의 유이한 조건들이며, 다음 시간에 설명하겠습니다.)

아무튼, 이렇게 progress0를 take를 이용해서 정의하려면, 부득이 progress0의 타입도 take에 맞게 조정해주는 수 밖에 없습니다:

file: HW4-r3.hs

```haskell
-- progress0 :: Integral i => (t -> t) -> i -> t -> [t]   -- incorrect
progress0 :: (t -> t) -> Int -> t -> [t]                  --   correct
progress0 f n x  =  take (n+1) (iterate f x)
```

1절의 스타일과의 또 한 가지 차이는 음수에 대한 반응입니다. 위와 같이 정의한 `progress0`는 두번째 인자로 음수가 들어오면 오류를 뱉는 대신 그냥 조용히 빈 리스트를 리턴합니다:

[WinGHCi window, module HW4 loaded]

```
*HW4> progress0 (*2) (-5) 1
[]
it :: Num t => [t]
*HW4>
```

이 또한 내장 함수 `take` 때문입니다. `take`는 첫번째 인자로 음수가 들어오면 오류를 뱉는 대신 조용히 빈 리스트를 리턴합니다. 이 차이를 없애려면 인자 `n`이 음수인 경우와 그렇지 않은 경우 각각을 가드 문법으로 나누어 정의해야 합니다:

file: HW4-r3.hs

```haskell
progress0' :: (t -> t) -> Int -> t -> [t]
progress0' f n x
    | n < 0      =  error "Cannot apply negative times!"
    | otherwise  =  take (n+1) (iterate f x)
```

다음 숙제인 HW5에서 함수 재귀 Part 2가 이어집니다. I'll be back!