숙제 #6: 위치 기수법

건국대학교 소프트웨어학과 | 2019 컴퓨테이션이론 | 담당: 차리서

이 문서는 6차 숙제를 위한 문제지 겸 설명서입니다. 함께 첨부하는 `HW6.hs` 파일에 각 문제의 답에 해당하는 내용을 작성한 후, 완성된 `HW6.hs`만 숙제 제출 화면에 업로드하면 됩니다. 정시 제출 기한은 12/10(화) 밤 10시까지입니다.

위치 기수법(positional notation)이란 수를 나타내는 수많은 표기법들 중 하나로서, 일렬로 나열된 각 자리마다 한 세트의 숫자들 중 하나씩을 배치하는 방법이며 우리가 평소에 가장 흔히 사용하고 있는 표기법입니다. 위치 기수법 중에도 또다시 다양한 변종들이 있습니다만, 이 숙제에서는 둘 이상의 일정한 갯수의 숫자를 사용하고 자연수만 나타내는 (소수점이나 음수 부호 등이 없는) 가장 기본적인 형태만 다루겠습니다.

Contents

1 숫자

먼저, "숫자"들을 만들겠습니다. 숫자들을 만드는 방법은 여러가지가 있고, 가장 단순한 방법 중에는 서로 동치비교가 가능하고 모두 서로 다르게 판정되는 여러 값들을 리스트에 담는 방법이 있습니다. 예를 들어,

```haskell
digits = ['0'..'7']
```

라는 문자들의 리스트를 정의하고 이 리스트의 각 원소인 문자 하나 하나가 숫자라고 간주하거나,

```haskell
digits = ["zero", "one", "two", "three", "four", "five", "six", "seven"]
```

라는 문자열들의 리스트를 정의하고 이 리스트의 각 원소인 문자열 하나 하나가 숫자라고 간주하는 방법도 있습니다. 하지만, 이 방법을 사용하면 "가장 작은 숫자", "가장 큰 숫자", "다음 숫자", "이전 숫자" 등의 개념을 나타내기 위한 함수들을 하나 하나 정의해줘야 하는 번거로움이 있기에, Haskell의 표준 타입 클래스들이 제공하는 기능을 활용하기 위해 다른 방법을 사용하겠습니다. 숙제 코드 파일에는 이미 다음과 같은 타입 정의문이 들어있습니다:

file: HW6.hs

```
data Digit = Zero | One | Two | Three | Four | Five | Six | Seven | Eight | Nine
    deriving (Show, Eq, Ord, Bounded, Enum)
```

이 정의문의 첫줄은 대놓고 단순합니다. 마치 `Bool` 타입이 `True`와 `False`라는 딱 두 개의 무항 자료생성자들로만 값을 만들 수 있는 타입이었던 것처럼:

file: HW6.hs

```
data Bool = True | False
```

`Digit` 타입은 `Zero`부터 `Nine`까지 딱 열 개의 무항 자료생성자들로만 값을 만들 수 있는 타입일 뿐입니다.

위 정의문의 두번째 줄은 숫자들에 대한 몇 가지 편리한 함수들을 자동적으로 얻기 위한 문장입니다.

- `Show` 클래스에 자동 소속됨으로써 자료생성자의 형태 그대로 화면에 출력할 수 있게 되고,
- `Eq` 클래스에 자동 소속됨으로써 같은 자료생성자끼리만 서로 같고 (`==`) 그 외에는 모두 서로 다르게 (`/=`) 되며,
- `Ord` 클래스에 자동 소속됨으로써 자료생성자들이 타입 정의문에 등장한 순서대로 대소 (`<`) 관계를 갖게 되고,
- `Bounded` 클래스에 자동 소속됨으로써 타입 정의문에 가장 먼저 등장한 자료생성자가 최소경계값 (`minBound`), 가장 마지막에 등장한 자료생성자가 최대경계값 (`maxBound`) 을 각각 갖게 되며,
- `Enum` 클래스에 자동 소속됨으로써 다음 값 (`succ`), 이전 값 (`pred`), 몇번째 값인지 계산하기 (`fromEnum`), n번째 값 찾기 (`toEnum`), 나열한 리스트 만들기 (`[x..y]` 형태 등) 가 가능해집니다.

[WinGHCi window, module HW6 loaded]

```
*HW6> Two    -- Show
Two
it :: Digit
*HW6> [Two == Two, Two /= Two, Two == Five, Two /= Five]    -- Eq
[True,False,False,True]
it :: [Bool]
*HW6> [Two < Two, Two < Five, Five < Two]    -- Ord
[False,True,False]
it :: [Bool]
*HW6> [minBound, maxBound] :: [Digit]    -- Bounded
[Zero,Nine]
it :: [Digit]
*HW6> [succ Two, pred Two, toEnum 5]    -- Enum #1
[Three,One,Five]
it :: [Digit]
*HW6> fromEnum Two    -- Enum #2
2
it :: Int
*HW6> [Two .. Five]    -- Enum #3
[Two,Three,Four,Five]
it :: [Digit]
```

```
*HW6> [Two ..]   -- 무한하지 않습니다.
[Two,Three,Four,Five,Six,Seven,Eight,Nine]
it :: [Digit]
*HW6> succ Nine   -- 마지막 자료생성자의 다음은 없습니다.
*** Exception: succDigit: tried to take `succ' of last tag in enumeration
CallStack (from HasCallStack):
  error, called at HW6.hs:9:39 in main:HW6
*HW6>
```

마지막 두 입력에 대한 반응에 주목하셔요. `[n ..]` 형태로 무한히 나열하게 해도 자동적으로 마지막 자료생성자에서 멈춥니다. 이는 `Enum` 클래스에 속한 타입들의 절대적 속성이 아니라, `Digit` 타입이 `Enum` 클래스에 소속된 방식이 (별도의 인스턴스 선언을 통해 무한히 나열되게끔 달리 정의되지 않고) `deriving` 구문으로 자동 소속되는 방식이었기 때문입니다. 맨 마지막 입력인 `succ Nine`에 대해 오류가 발생한 이유도 마찬가지입니다.

1.1 숫자 변경

여기서, 당장의 코딩은 필요 없는 (하지만 중요한) 문제입니다: 지금 숫자는 `Zero`부터 `Nine`까지 총 열 개의 무항 자료생성자들로 이루어져있고 이 숙제의 실제 문제들이 진행되는 동안 내내 이 상태(?)를 기준으로 예시들을 보이며 설명하겠지만, **이 숫자들은 (후술할 몇 가지 규칙 하에) 다르게 바꿀 수 있어야** 하며 그렇게 숫자들만 다르게 바꾸고 코드 속의 다른 부분은 일체 수정하지 않아도 모든 함수들이 여전히 원래 의도했던 대로 올바르게 작동해야 합니다.

구체적으로 예를 들어서, 만일 (코드 속의 다른 부분은 일체 수정하지 않고) 숫자만 다음과 같이 16진법으로 바꾸거나:

an example of different digits

```
data Digit = Zero | One | Two | Three | Four | Five | Six | Seven | Eight | Nine
           | Ten | Eleven | Twelve | Thirteen | Fourteen | Fifteen
    deriving (Show, Eq, Ord, Bounded, Enum)
```

진법 뿐만 아니라 자료생성자의 이름까지 다음과 같이 바꿔도:

another example of different digits

```
data Digit = D0 | D1 | D2 | D3 | D4 | D5 | D6 | D7
    deriving (Show, Eq, Ord, Bounded, Enum)
```

다른 함수들은 그에 맞춰서 자연스럽게 원래 하던 일을 수행해야 합니다. 즉, 다른 함수들을 작성하는 동안 절대로 **각 자료생성자들의 이름과 그들의 갯수를 하드코딩하거나 함부로 가정하지 말라**는 뜻입니다.

위에서 언급했던, 숫자들을 다르게 바꿀 때의 "몇 가지 규칙"은 다음과 같습니다:

- `Digit`라는 타입생성자는 언제나 무항 타입생성자여야 합니다. 즉, 타입 자체의 이름이나 카인드는 바꿀 수 없습니다.
- 각각의 숫자는 무항 자료생성자여야 합니다. 즉, 각 숫자를 나타내는 자료생성자는 인자를 받으면 안되고 그 자체가 혼자 독립적으로 숫자 한 개를 나타내야 합니다.

- 숫자들의 갯수는 둘 혹은 그 이상이어야 합니다. 즉, 이 숙제에서는 1진법[1]은 다루지 않습니다.
- 숫자들의 갯수는 36개 혹은 그 이하여야 합니다. 테스트 중의 가독성 및 타이핑의 편의를 위한 함수들 몇 가지를 숙제 코드에 미리 만들어서 제공할텐데 (아래에 곧바로 이어지는 1.2절 참조), 이 "편의 기능" 들만 아니었다면 이 규칙은 사실은 없어도 되는 규칙이었습니다. (숫자와 영문 대문자의 갯수의 합이 36이기 때문입니다.)

1.2 편의 기능: 숫자를 나타내는 문자

이 소절의 내용은 숙제 문제를 푸는 데에 직접 필요한 내용은 아니고, 문제를 푼 (함수를 정의한) 후 결과를 테스트해볼 때 가독성과 입력의 편의를 위해 고려해두는 내용입니다.

숙제 코드 파일 초반에 다음과 같은 세 가지가 정의되어있는데:

file: HW6.hs

```
digitCount  =  length [(minBound::Digit)..maxBound]

digitCharCandids  =  ['0'..'9'] ++ ['A'..'Z']

digitChars  =  take digitCount digitCharCandids
```

`digitCount`는 `Digit` 타입의 정의로부터 숫자들의 갯수를 계산해둔 것이고, `digitCharCandids`는 추후 자연수를 가독성 좋게 표시하거나 편하게 입력하기 위해 각 숫자에 대응하는 문자의 후보 총 36개를 정의해둔 것이며 `digitChars`는 `digitCharCandids` 중에서 실제 숫자 갯수 만큼을 뽑아둔 것입니다.

단, 이를 이용해서 숫자 자체의 표시 방법을 (`Show Digit` 인스턴스 정의를 직접 작성해서) 바꾸지는 않으며, 숫자 자체는 여전히 `deriving Show`로 자동 생성된 표시 방법을 따릅니다. 다시 한 번 강조하지만, 여기서 준비해둔 "각 숫자에 대응하는 문자"는 나중에 숫자에 대한 함수나 자연수에 대한 함수들을 테스트할 때에만 사용될 예정입니다.

1.3 원형 순회

실질적인 첫번째 문제입니다: `Enum` 클래스의 (자동 생성된) 메서드들인 `succ`과 `pred`와 비슷하게 각각 "다음" 숫자와 "이전" 숫자를 리턴하지만, 양 끝에서 멈추는 이들 메서드들과는 달리 원형으로 돌면서 다음/이전 숫자를 리턴하는 두 함수

```
nextD :: Digit -> Digit
prevD :: Digit -> Digit
```

들을 각각 정의하십시오. 구체적으로 말하자면, 가장 큰 숫자의 `nextD`는 가장 작은 숫자고, 가장 작은 숫자의 `prevD`는 가장 큰 숫자입니다.

[WinGHCi window, module HW6 loaded]

```
*HW6> [nextD Five, nextD Nine, prevD Five, prevD Zero]
[Six,Zero,Four,Nine]
```

[1] Church numerals나 3차 숙제에서 `Z`와 `S`로 만들었던 자연수가 1진법에 가깝습니다.

```
it :: [Digit]
*HW6>
```

주의: 하드코딩하지 말라고 했던 1.1절의 '문제 아닌 문제'를 꼭 기억하십시오!

힌트: 숫자들은 `Eq` 클래스 소속으로서 동치 비교가 가능하고, `Bounded` 클래스 소속으로서 최소값과 최대값이 정해져있고 그 값들을 쉽게 지칭할 수 있으며, `Enum` 클래스 소속으로서 다음 값(`succ`)과 이전 값(`pred`)을 말할 수 있습니다. (`succ`과 `pred`를 이용해서 정의해도 된다는 뜻입니다.)

1.4 캐리

캐리에는 올림과 내림이 있지만, 이 숙제에서는 뺄셈은 다루지 않으므로 내림은 고려하지 않고 올림만 고려합니다. 덧셈은 항상 숫자 두 개만 서로 더하므로 (셋 이상의 숫자들을 한꺼번에 더하지 않고 둘씩 더하는 연산을 연이을 뿐이므로) 캐리도 여러가지 값을 갖는 것이 아니라 "있는지 없는지"만 나타냅니다. 따라서, 캐리는 다음과 같이 `Bool` 타입의 별명으로 간단히 정할 수 있습니다 (값이 `True`면 캐리가 있는 것이고, `False`면 없는 것입니다):

file: HW6.hs

```
type Carry = Bool
```

이제, "주어진 캐리를 고려하며" 두 숫자를 서로 더하는 연산을 생각하겠습니다. 즉, 이 연산은 인자가 두 개가 아니고 세 개(캐리 한 개와 숫자 두 개)입니다. (주어진 캐리를 고려하든 고려하지 않든 간에) 두 숫자를 더하면 결과 숫자(위치 기수법의 관점에서 단자리 값)와 함께 새로운 캐리도 발생할 수 있습니다. 여기서 (작은 상수배 차이 정도의) 계산 효율을 고려하자면 이런 계산을 한 번만 수행하면서 새로운 캐리와 결과 숫자를 동시에 얻는 게 좋겠지만, 추후 자연수 덧셈에 활용할 때의 실제 활용성은 오히려 떨어지므로 그냥 결과 숫자를 얻는 함수와 새로운 캐리를 얻는 함수로 따로 나누어 만들겠습니다.

문제입니다: 이미 주어진 (아마도 이전 단계—아래 자리—에서 넘어온) 캐리 하나와 두 숫자를 인자로 받아서, 그 캐리를 고려하며 두 숫자를 더했을 때의 (단자리) 결과 숫자를 리턴하는 함수

```
addDd :: Carry -> Digit -> Digit -> Digit
```

와, 다음 캐리를 리턴하는 함수

```
addDc :: Carry -> Digit -> Digit -> Carry
```

를 각각 정의하십시오. 이 함수는 반드시 어떻게 만들어야 한다는 제약이 없지만, 기왕이면 다음 사항들을 고려하기 바랍니다:

- 진법(숫자들의 갯수)이 b진법이라고 할 때, 재귀 호출의 횟수가 최대 $\lceil b/2 \rceil$번 이내가 되도록 만들 방법이 있습니다. (**힌트:** 숫자들은 `Eq`, `Bounded` 및 `Enum` 클래스 소속일 뿐만 아니라 `Ord` 클래스 소속이기도 합니다.)

- 두 함수 모두 (`where` 절 등을 이용해 별도의 함수를 정의하지 않고도, 직접) 꼬리 재귀로 작성할 방법이 있습니다.[2]

만일 필요하다면, 앞서 정의했던 `nextD`와 `prevD`를 사용해도 됩니다.

올바르게 정의된 두 함수는 다음과 같이 작동해야 합니다:

[*WinGHCi window, module* HW6 *loaded*]

```
*HW6> addDd False Seven Eight
Five
it :: Digit
*HW6> addDc False Seven Eight
True
it :: Carry
*HW6>
```

혹은, 숙제 코드에 미리 작성되어 주석처리되어있는 `testAdd` 함수의 주석을 풀고, 역시 미리 작성되어있는 테스트 케이스들 중 현재 숫자 표기 및 진법에 부합하는 테스트 케이스 하나의 (코드 파일에 최초에 주어졌던 영어식 이름 10개로 구성된 숫자들을 그대로 사용하고 있을 경우 `tc1`이라는 리스트의) 주석도 풀어서, 다음과 같이 테스트해보기 바랍니다:

[*WinGHCi window, module* HW6 *loaded*]

```
*HW6> testAdd tc1
    0 + 0 =  0
    0 + 3 =  3
    0 + 6 =  6
    0 + 9 =  9
    3 + 0 =  3
    3 + 3 =  6
    3 + 6 =  9
... (중략) ...
1 + 9 + 3 = 13
1 + 9 + 6 = 16
1 + 9 + 9 = 19
it :: ()
*HW6>
```

2 자연수

자연수를 위치 기수법으로 표현한다는 것은 결국 자연수를 "숫자들의 나열"로 표현한다는 뜻입니다. 나열 순서는 우리가 일상적으로 사용하는 순서 그대로, 가장 왼쪽 숫자가 가장 큰 자리이고 가장 오른쪽 숫자가 가장 작은 자리입니다. 따라서, 앞서 캐리를 `Bool` 타입의 별명으로 정의했던 것처럼, 자연수도 숫자들의 리스트 (`[Digit]`) 타입의 별명으로 정의할 수 있습니다:

[2] 어디까지나 연습을 위해 고려해보라는 것일 뿐, 이렇게 재귀 횟수가 상당히 적은 횟수로 제한되는 (수백, 수천 번 이상이 아닌) 경우에는 꼬리재귀든 아니든 실질적으로 별 차이가 없긴 합니다.

file: HW6.hs

```
type Nat = [Digit]
```

예를 들어, (숫자가 열 개고 영어식 이름을 사용한다는 가정 하에) 리스트 [Four, Two]는 흔히 사용되는 인도-아라비아숫자 십진 위치 기수법으로 42라고 표시되는 자연수를 의미합니다.

2.1 편의 기능: 자연수를 나타내는 문자열

참고로 지협적인 부연이지만, Haskell에서 타입을 정의할 때 data 문으로 별도의 독립된 타입을 정의하지 않고 이렇게 type 문으로 다른 기존 타입의 별명으로 정의하면, 이 타입은 바로 그 "다른 기존 타입"의 인스턴스 선언들을 무조건 따라갑니다. 예를 들어, Show 클래스 소속인 어떤 a 타입에 대해서 이 타입의 값들의 리스트인 [a] 타입도 Show 클래스 소속이 되도록 리스트의 인스턴스 선언이 만들어져있기 때문에, 방금 정의한 Nat 타입, 즉, [Digit] 타입도 (Digit가 Show 소속이므로) 이미 Show 클래스 소속입니다. 기존 타입에 대한 인스턴스 선언들이 별명에도 그대로 반영되므로 별도의 인스턴스 선언은 커녕 deriving조차 사용할 필요가 없다는 점은 장점이지만, 메서드를 다르게 정의하고 싶어도 (예를 들어, 자연수를 "[Four, Two]"라고 표시하는 대신 "Four_Two"나 "42"라고 표시하게끔 만들고 싶어도) 그렇게 만들 방법이 없다는 점은 단점입니다.[3] 인스턴스 선언을 다르게 하고싶으면 (예를 들어, 화면에 표시되는 방식을 리스트와 다르게 만들기 위해 Show Nat 인스턴스 선언을 다시 하면서 show 메서드를 다르게 정의하고 싶으면), 별명이 아니라 data 문을 사용해서 별도의 타입으로 정의해야 합니다.

각설하고, 위 단락에서 설명한 이유 때문에 이 Nat 타입은 Show나 Read 클래스에 이미 (불편한 형태로) 소속되어있고, 따라서 나중에 자연수에 대한 함수들의 동작을 테스트할 때 자연수 값을 입력하기도 번거롭고 출력된 자연수 값을 읽기도 불편하게 되었습니다. 그래서, 추후 편리한 테스트를 위해 (Show 클래스의 메서드와는 무관한, 별도의) 출력 함수 showN과 (Read 클래스의 메서드와는 무관한, 별도의) 입력 함수 readN을 만들어두겠습니다. 이들 두 함수는 숙제 코드 파일에 이미 작성되어 있습니다 (이미 수업 시간에 설명한 요소들로만 구성되어있으니 자세한 설명은 생략하겠습니다):

file: HW6.hs

```
showN :: Nat -> String
showN   []    =  ""
showN (d:ds)  =  digitChars !! fromEnum d : showN ds

readN :: String -> Nat
readN   ""    =  []
readN (c:cs)  =  toEnum (fstIdx c digitChars) : readN cs where
    fstIdx x (y:ys)
        | x == y     =  0
        | otherwise  =  1 + fstIdx x ys
    fstIdx _   _     =  error "Cannot be parsed as Natural number"
```

이들 두 함수는 예를 들어 다음과 같이 작동합니다:

[3] 실은 비표준 확장 기능을 쓰면 방법이 있긴 있지만, 권장하지 않습니다.

[*WinGHCi window, module* HW6 *loaded*]

```
*HW6> showN [Two,Zero,One,Nine,One,Two,Zero,One]
"20191201"
it :: String
*HW6> readN "20191201"
[Two,Zero,One,Nine,One,Two,Zero,One]
it :: Nat
*HW6> showN [Zero,Zero,Zero,Four,Two]
"00042"
it :: String
*HW6> readN "00042"
[Zero,Zero,Zero,Four,Two]
it :: Nat
*HW6> (showN [], readN "")
("",[])
it :: (String, Nat)
*HW6>
```

2.2 편의 기능: 자연수의 표준형

앞서 2절 초반에 숫자들의 리스트로 정의했던 자연수는, 리스트로서는 구조적으로 서로 다르지만 자연수로서는 산술적으로 서로 같은 경우가 있습니다. 예를 들어 [Four, Two]와 [Zero, Zero, Four, Two]는 리스트로서는 (일단 원소 갯수부터 다르니) 서로 다르지만 자연수로서는 둘 다 42를 나타내는 같은 값입니다. 2.1절 초반에 설명했던 이유 때문에, 동치 비교 중위 연산자 "=="의 두 피연산자로 이들 두 자연수(숫자 리스트)를 넣어주면 같지 않다고 판정(False로 계산)되고, 이를 거스를 (오버로딩 할) 방법이 없습니다. (그러려면 애초에 Nat 타입을 별명이 아닌 별도의 독립된 타입으로 정의했어야 합니다.) 다만, 이 숙제에서는 자연수의 동치 비교를 아예 안 할 예정이기 때문에 그냥 이대로 진행하겠습니다.

대신, 산술적으로 서로 같다고 간주하려는 자연수들의 공통된 "표준형"을 고려하겠습니다. 즉, 산술적으로 서로 같은 자연수들은 모두 예외 없이 하나의 표준형을 공유하고, 반대로 하나의 표준형을 공유하는 자연수들끼리는 (그리고 이런 자연수들끼리만) 반드시 서로 같습니다. 구체적으로, 표준형은 다음과 같이 정합니다:

- 모든 원소가 가장 작은 숫자인 리스트[4]는 가장 작은 자연수고, 가장 작은 자연수의 표준형은 "가장 작은 숫자 딱 한 개로 이루어진 리스트"입니다. 예를 들어 가장 작은 숫자가 Zero라고 할때, [], [Zero], [Zero,Zero], [Zero,Zero,Zero] 등은 모두 가장 작은 자연수를 나타내며, 이들의 표준형은 [Zero]입니다.
- 첫머리에 등장하는 가장 작은 숫자들의 갯수만 서로 다르고 나머지는 똑같은 리스트들은 서로 산술적으로 같은 자연수고, 이들 중 (위에서 정한) '가장 작은 자연수'들을 제외한 다른 리스트들의 표준형은 첫머리의 가장 작은 숫자들을 모두 제거한 리스트입니다. 예를 들어 가장 작은 숫자가 Zero고 그 다음 숫자가 One이라고 할 때, [One,One]과 [Zero,One,One]과 [Zero,Zero,One,One] 등은 모두 같은 자연수입니다.

숙제 코드에 정의되어있는 함수 normalize는 임의의 자연수 하나를 인자로 받아서 그 자연수의 표준형을 리턴

[4] 이렇게 "모든 원소가..."라고 말하면 자동적으로 빈 리스트도 이 경우에 포함됩니다.

하는 함수입니다:

file: HW6.hs

```
normalize :: Nat -> Nat
normalize (d:ds)
    ¦ d == minBound  =  normalize ds
    ¦ otherwise      =  d:ds
normalize   []       =  [minBound]
```

여기서 또 한 번, 당장 코딩할 필요는 없는 (하지만 역시 중요한) 문제입니다: 이후 숙제 문제를 풀 때,

- 자연수 인자를 받는 함수를 작성할 경우, 그 자연수 **인자가 표준형이든 아니든 관계 없이 항상 올바르게 작동하도록** 작성해야합니다. 다시 말하자면, 인자가 표준형으로만 주어진다는 보장이 없으니 표준형이 아닌 인자에도 대비해야한다는 뜻입니다.
- 자연수를 리턴하는 함수를 작성할 경우, 그 자연수 리턴값이 산술적으로 올바른 자연수이기만 하면 그 자연수가 표준형이든 아니든 관계 없습니다. 간단히 말해서, 표준형이 아닌 자연수를 리턴해도 됩니다.

2.3 자연수 증감

자연수를 하나 증가시키는 함수 `incN`과, 가장 작지는 않은 자연수를 하나 감소시키는 함수 `decN`을 다음과 같이 정의해두겠습니다:

file: HW6.hs

```
incN :: Nat -> Nat
incN  =  reverse . fn . reverse  where
    fn (d:ds)
        ¦ d == maxBound  =  minBound : fn ds
        ¦ otherwise      =  succ d : ds
    fn   []              =  [succ minBound]

decN :: Nat -> Nat
decN  =  reverse . fn . reverse  where
    fn (d:ds)
        ¦ d == minBound  =  maxBound : fn ds
        ¦ otherwise      =  pred d : ds
    fn   []              =  error "Cannot decrease more"
```

즉, 이 함수들은 (숫자들이 초기에 주어진 영어 이름 열 개로 구성되어있을 경우) 다음과 같이 동작합니다:

[WinGHCi window, module HW6 loaded]

```
*HW6> map incN [[Three,Five,Seven],[Three,Five,Nine],[Three,Nine,Nine],[Nine,Nine,Nine]]
[[Three,Five,Eight],[Three,Six,Zero],[Four,Zero,Zero],[One,Zero,Zero,Zero]]
it :: [Nat]
*HW6> map incN [[],[Zero,Zero,Zero]]
[[One],[Zero,Zero,One]]
it :: [Nat]
*HW6> map decN [[Three,Five,Seven],[Three,Five,Zero],[Three,Zero,Zero],[One,Zero,Zero]]
```

```
[[Three,Five,Six],[Three,Four,Nine],[Two,Nine,Nine],[Zero,Nine,Nine]]
it :: [Nat]
*HW6> decN [Zero,Zero,Zero]
*** Exception: Cannot decrease more
CallStack (from HasCallStack):
  error, called at HW6.hs:125:29 in main:HW6
*HW6>
```

2.4 자연수에 대한 술어

자연수 하나를 인자로 받아서 그 자연수가 (표준형이든 아니든, 산술적으로) 가장 작은 자연수인지 아닌지를 판별하는 단항 술어 `isNull`과, 역시 자연수 하나를 인자로 받아서 그 자연수가 (역시 표준형 여부에 관계 없이 산술적으로) 두번째로 작은 자연수인지 (흔히 사용하는 십진 기수법으로 "1"인지) 아닌지를 판별하는 단항 술어 `isMono`, 그리고 두 자연수를 인자로 받아서 그 자연수들이 그 순서대로 '작거나 같은' 관계인지를 판별하는 이항 술어 `leN`을 정의해두겠습니다 (행번호는 실제 코드 파일의 행번호와 다를 수 있습니다):

file: HW6.hs

```
128 isNull, isMono :: Nat -> Bool
129 isNull n  =  all (minBound ==) n
130
131 -- isMono n  =  f (reverse n)  where
132 --     f   []    =  False
133 --     f (d:ds)  =  d == succ minBound && isNull ds
134
135 -- isMono n  =  not (isNull n) && isNull (init n) && last n == succ minBound
136
137 isMono n  =  normalize n == [succ minBound]
138
139 leN :: Nat -> Nat -> Bool
140 leN n m  =  fn (normalize n) (normalize m)  where
141     fn x y  =  length x < length y || (length x == length y && x <= y)
```

위에서 `isMono` 함수는 결국 137행에서 `normalize` 함수와 리스트 간의 동치 비교 연산을 이용해서 정의하고 있지만, 그 외에 주석 처리된 다른 두 가지 정의 방식(131–133행의 정의, 135행의 정의)도 참고로 보아두십시오. 이렇게 여러가지 방식으로 정의하는 예제들이 기말고사 문제 풀이에 힌트가 될 수도 있습니다. 또한 지협적인 참고로, 128행처럼 같은 타입을 갖는 둘 이상의 함수들의 타입을 한 행에서 (쉼표로 나열하여) 한꺼번에 선언하는 것도 가능합니다.

앞서 설명했듯이 자연수를 직접 `Ord` 클래스에 소속시키면서 이에 대한 `<=` 중위 연산자를 정의하지 못하는 이유는 자연수가 별도의 타입이 아닌 '숫자들의 리스트'의 별칭일 뿐이기 때문입니다. 대신, 리스트의 길이는 정수(정확히는 `Int` 타입)로서 이미 대소 비교가 가능하고, (대소 비교가 가능한 타입의 값들이 들어있는) 리스트끼리도 대소 비교가 가능하므로, 이를 이용해서 `leN`을 정의합니다. 주의할 점은, 리스트 간의 대소 비교는 "사전식 비교"라는 점입니다. 따라서, 자연수들의 크기를 비교하려면, 먼저 그 자연수들 각각의 표준형들의 길이부터 비교하고 (짧은 쪽이 무조건 작음), 표준형의 길이가 같을 경우 내장된 `<=` 연산자를 이용해서 리스트

자체를 (사전식으로) 비교하면 됩니다. 참고로, '숫자' 자체는 이미 (`deriving Ord`에 의해서) `Ord` 클래스에 속해있습니다.

2.5 자연수 덧셈

이제 두 자연수 간의 덧셈 연산을 생각해보겠습니다. 먼저, 3차 숙제에서 Church numerals와 유사한 방식(`Z`와 `S`)으로 정의했던 자연수에 대한 덧셈 연산을 비슷하게 흉내내보겠습니다. (이번에는 자연수끼리의 대소 비교가 가능해서) 그때와는 정의 방식이 살짝 다르지만, 여전히 그때처럼 재귀 횟수는 두 인자 중 작은 쪽의 (자연수로서의) 크기에 비례하며, 심지어 이번에는 꼬리재귀 형태입니다. (대소 비교를 번번이 반복하지 않고 한 번만 하도록 만들기 위해 대소 비교부터 먼저 수행한 후 내부 함수에 넘겨주는 방식입니다.) 단, 이번 숙제에서는 이것보다 압도적으로 빠른 덧셈을 (여러분이) 곧 만들 예정이므로, 이 덧셈은 상대적으로 "느린" 덧셈임을 함수 이름에 명시해두겠습니다:

file: HW6.hs

```haskell
addNslow :: Nat -> Nat -> Nat
addNslow x y  =  if leN x y then add x y else add y x  where
    add a b
        | isNull a   =  b
        | otherwise  =  add (decN a) (incN b)
```

이번에는, 이 숙제 코드 파일에 위치 기수법으로 정의되어있는 자연수까리 더하되, 이전 (하나 낮은) 자리에서 넘어온 캐리를 고려해가며 단자리는 단자리끼리[5], (십진법 기준) 십의 자리는 십의 자리끼리, 백의 자리는 백의 자리끼리 서로 더하고 그 결과로 발생한 새로운 캐리를 다음 (하나 높은) 자리에 넘겨주는 방식으로 더하는 `addNfast` 함수를 생각해보겠습니다. 즉, 일상 생활에서 수동으로 (종이와 연필을 이용해서) 여러 자리 자연수 두 개를 서로 더할 때 실제로 흔히 사용하는 방식을 그대로 구현해보는 것입니다.

이 덧셈의 재귀 횟수는 두 인자 중 작은 쪽의 (혹은 구현 방식에 따라서는 큰 쪽의) "자리수"에 비례하며, 숫자 체계가 b진법이라고 하고 인자의 (자연수로서의) 크기를 n이라고 할 때 이 덧셈의 재귀 횟수는 $\log_b n$에 비례합니다. 예를 들어, (십진법 기준으로) 인자들이 자연수로서 백 배 커지면 재귀 횟수는 두 번 늘어나고, 만 배 커지면 네 번 늘어나며, 억 배 커지면 여덟 번 늘어납니다.

오랜만에 (당장 코딩할) 문제입니다: 위에서 설명한 "각 자리 숫자끼리 더해서 두 자연수를 더하는" 함수

```haskell
addNfast :: Nat -> Nat -> Nat
```

를 정의하십시오. 가장 중요한 필수 요구 사항은:

- 재귀 호출 횟수가 두 자연수 중 (큰 쪽이든 작은 쪽이든) 한 쪽의 자릿수에 비례해야 하고,
- 1.1절의 문제(숫자 이름/갯수 변경)와 2.2절의 문제(표준형이 아닌 인자)를 잊지 말아야하며,
- 자연수와 정수 (`Int`나 `Integer`) 간의 상호 변환 함수를 만들어서 정수 덧셈 (`+`)를 통해 정의하면 안되고,
- 마찬가지로, 자연수와 문자열 간의 상호 변환 함수들인 `showN`과 `readN`은 어디까지나 테스트의 편의를 위한 함수들이므로, 이들을 이용해서 정의하면 안됩니다.

[5] 물론, 단자리끼리 더할 때 '이전 자리에서 넘어온 캐리'는 언제나 '캐리 없음'입니다.

그 외의 (힌트 겸) 안내 사항은:

- (숫자와 정수 간의 변환 함수인 `fromEnum`/`toEnum`와 자연수와 문자열 간의 변환 함수인 `showN`/`readN`만 제외하고) 이 숙제 문제지나 코드에 언급/사용/정의된 어떤 함수든 자유롭게 사용할 수 있습니다. 특히, `addDd`와 `addDc`를 잘 활용해보셔요.
- 이 숙제 문제지나 코드에 언급/사용/정의되지 않은 다른 Haskell 내장 함수를 사용하지 않고도 이 덧셈 함수를 만들 수 있음이 분명하지만, 혹시 다른 문서나 다른 숙제에서 봤던 내장 함수가 필요하다고 느낀다면 사용해도 됩니다. (`id`, `head`, `tail`, `init`, `last` 등등)
- 2.2절의 문제(표준형이 아닌 인자)에 대응하는 방법은 두 가지인데:
  - 인자를 받자마자 일단 표준형으로 바꿔놓고, 표준형에 대해서만 올바르게 동작하는 함수를 작성
  - 표준형이든 아니든 상관 없이 모든 '숫자 리스트'에 대해 올바르게 동작하는 함수를 작성

  이 중 어떤 방법을 사용해도 점수 차이는 없지만, 훈련 목표 상 후자의 방식으로 시도하기를 추천합니다. 게다가, 얼핏 생각하면 전자가 쉬워보이겠지만, 재귀의 종료 조건(대체로 인자가 빈 리스트인 경우)을 작성하려다보면 꼭 그렇지만도 않음을 알게 될 겁니다.
- 꼬리재귀 형태로 정의하면 (시험 대비 훈련으로서) 더 좋지만, 숙제 점수 자체를 기준으로 보자면 꼬리재귀는 필수 사항이 아닙니다 (점수 차이 없음).
- 두 인자 자연수들 중 (표준형으로 바꾼 형태를 기준으로 하든 인자로 주어진 형태 그대로를 기준으로 하든 간에) 길이가 짧은 쪽에 재귀횟수가 비례하도록 만들면 더 좋지만, 역시 필수 사항은 아니며 긴 쪽에 비례해도 점수 차이는 없습니다.
- 리스트를 대상으로 재귀 함수를 만들 때에는 리스트의 맨 앞 (왼쪽) 원소부터 짚어가도록 만드는 게 편하므로, 주어진 인자를 (필수 요구사항은 아니고, 이렇게 하지 않는 방법도 있지만) 일단 뒤집어놓고 생각하는 게 편할 겁니다. 다만, 뒤집은 후의 처리 방식에 따라서, 어떤 방식으로 처리할 경우에는 "뒤집은 수들을 처리한 결과를 다시 뒤집어서 리턴해야" 할 수도 있는 반면, 또 어떤 방식으로 처리할 경우에는 "뒤집은 수들을 처리한 결과를 그대로 리턴해야" 할 수도 있습니다.
- 각각 다섯 자리, 네 자리인 두 자연수 $d_1d_2d_3d_4d_5$와 $d_6d_7d_8d_9$가 있다고 합시다. 그리고, 위 두 자연수 각각의 단자리 숫자들인 d_5와 d_9를 '숫자로서 서로 더한' 결과 캐리와 결과 숫자를 각각 c와 d라고 합시다. 그러면, 이 두 자연수 $d_1d_2d_3d_4d_5$와 $d_6d_7d_8d_9$를 '캐리 없음'이라는 초기 캐리를 고려하며 더한 결과는, "각각 네 자리, 세 자리인 두 자연수 $d_1d_2d_3d_4$와 $d_6d_7d_8$을 c라는 캐리를 고려하며 서로 더한 결과"의 맨 뒤에 d를 붙인 것입니다. (큰따옴표 속이 재귀 호출입니다.)

테스트할 때에는 `showN`과 `readN`을 활용하기 바랍니다:

[*WinGHCi window, module* HW6 *loaded*]

```
*HW6> :set +s
*HW6> 7456087 + 2849796
10305883
it :: Num a => a
(0.00 secs, 64,024 bytes)
*HW6> showN $ addNslow (readN "7456087") (readN "2849796")
"10305883"
it :: String
```

```
(19.45 secs, 5,434,288,520 bytes)
*HW6> showN $ addNfast (readN "7456087") (readN "2849796")
"10305883"
it :: String
(0.00 secs, 115,632 bytes)
*HW6>
```

더 큰 자연수들도 더해보면서 시간 복잡도 한 단계($O(n)$ vs. $O(\log n)$)가 가져오는 극명한 차이를 느껴보기 바랍니다:

[*WinGHCi window, module* HW6 *loaded*]

```
*HW6> showN $ addNfast (readN "74568746646545604850484") (readN "4849796345353534098405")
"7941854299189913894888 9"
it :: String
(0.00 secs, 250,512 bytes)
*HW6>
```

아주 간단한 추가 문제(라기보다는 작업)입니다: 숙제 코드 파일 속에는 다음과 같이 `addN` 함수가 `addNslow`를 이용해서 정의되어 있는데:

file: HW6.hs

```
addN :: Nat -> Nat -> Nat
addN  =  addNslow      -- addNslow 혹은 addNfast
```

만일 본인이 성공적으로 `addNfast` 함수를 작성했다면, 위 `addN` 함수의 정의를 다음과 같이 수정하십시오:

file: HW6.hs

```
addN :: Nat -> Nat -> Nat
addN  =  addNfast      -- addNslow 혹은 addNfast
```

향후 곱셈에 관한 함수를 정의하는 도중에 덧셈이 필요할 경우, 기본적으로 `addN`을 사용하게 됩니다.

2.6 자연수 곱셈

곱셈에 관해서도, 먼저 3차 숙제에서 만들었던 곱셈과 비슷한 방식으로 "느린" 곱셈부터 만들어보겠습니다. 이 곱셈은 일단 꼬리재귀 형태고, 재귀 횟수가 두 인자 중 작은 쪽의 크기에 의존하긴 하지만, 작은 쪽의 '리스트로서의 크기 (자릿수)'가 아닌 '자연수로서의 크기'에 의존합니다. 즉, 0에 큰 인자를 작은 인자번 (작은 인자 횟수 만큼) 계속 더하는 방식입니다:

file: HW6.hs

```
mulNslow :: Nat -> Nat -> Nat
mulNslow x y  =  if leN x y then mul x y [minBound] else mul y x [minBound]  where
    mul a b acc
        | isNull a   =  acc
        | otherwise  =  mul (decN a) b (addN b acc)
```

예를 들어, 아래 테스트에서 입력한 두 값을 곱하려면 959054(에 해당하는 숫자 리스트, 즉 자연수)를 무려 597062번 서로 (혹은 0에 누적해서 계속) 더해야합니다:

[*WinGHCi window, module* HW6 *loaded*]

```
*HW6> 959054 * 597062
572614699348
it :: Num a => a
(0.00 secs, 70,456 bytes)
*HW6> showN $ mulNslow (readN "959054") (readN "597062")
"572614699348"
it :: String
(48.52 secs, 8,798,286,080 bytes)
*HW6>
```

덧셈으로서 비약적인 효율 향상을 보여주었던 `addNfast`를 `addN`으로서 사용하고 있고 이를 이용해서 정의된 곱셈임에도 불구하고, 재귀 호출 597062번이 가져오는 부담은 여전히 큽니다.

그래서, 곱셈도 종이에 연필로 곱하듯이 자리별로 곱하는 방식의 함수 `mulNfast`를 정의해보고자 합니다. 즉, 다음과 같이 곱하는 과정을 그대로 따라하는 함수를 뜻합니다:

```
              9 5 9 0 5 4
            x 5 9 7 0 6 2
           ---------------
            1 9 1 8 1 0 8
          5 7 5 4 3 2 4
                  0
        6 7 1 3 3 7 8
      8 6 3 1 4 8 6
  + 4 7 9 5 2 7 0
 ---------------------------
    5 7 2 6 1 4 6 9 9 3 4 8
```

즉, 이 곱셈의 재귀 횟수는 인자 자연수들 중 하나의 '자연수로서의' 크기에 비례하는 것이 아니라 '리스트로서의' 크기(즉, 자릿수)에 비례합니다.

마지막 문제입니다: 바로 위에서 설명한 대로 동작하는 곱셈 함수

```
mulNfast :: Nat -> Nat -> Nat
```

를 정의하십시오. 가장 중요한 필수 요구 사항은:

- "`mulNfast` 자체"를 재귀 호출하는 횟수가 두 자연수 중 (큰 쪽이든 작은 쪽이든) 한 쪽의 자릿수에 비례해야 하고,
- 1.1절의 문제(숫자 이름/갯수 변경)와 2.2절의 문제(표준형이 아닌 인자)를 잊지 말아야하며,
- 자연수와 정수 (`Int`나 `Integer`) 간의 상호 변환 함수를 만들어서 정수 곱셈 (`*`)를 통해 정의하면 안되고,
- 마찬가지로, 자연수와 문자열 간의 상호 변환 함수들인 `showN`과 `readN`은 어디까지나 테스트의 편의를 위한 함수들이므로, 이들을 이용해서 정의하면 안됩니다.

그 외의 (힌트 겸) 안내 사항은:

- 답(함수 정의)은 여러 방식이 있을 수 있겠지만, 출제 의도 대로 정의된 곱셈 함수의 코드는 앞서 정의했던 덧셈 함수 `addNfast`보다도 오히려 더 짧고 간단합니다.
- (숫자와 정수 간의 변환 함수인 `fromEnum`/`toEnum`와 자연수와 문자열 간의 변환 함수인 `showN`/`readN`만 제외하고) 이 숙제 문제지나 코드에 언급/사용/정의된 어떤 함수든 자유롭게 사용할 수 있습니다. **특히, `mulNslow`는 (100%까진 아니어도) 아마 필요할 겁니다.** 위 요구사항에서 자릿수에 비례하라고 한 것은 전체 곱셈의 재귀 횟수를 말했던 것이고, 작은 부분 문제(예를 들어, 십진법 기준으로 열 번 이내의 곱셈 문제)를 풀 때에는 `mulNslow`를 사용하더라도 전체 효율에 큰 영향이 없습니다.
- 이 숙제 문제지나 코드에 언급/사용/정의되지 않은 다른 Haskell 내장 함수를 사용하지 않고도 이 곱셈 함수를 만들 방법은 분명히 있지만, 아마 `init`와 `last`는 사용하는 게 더 좋을 겁니다. (이들 두 내장 함수를 사용하는 정의 방식이 문제에서 의도한 정의 방식입니다.) 물론, 그 외의 다른 내장 함수들도 정 필요하면 사용해도 됩니다.
- 바로 위 힌트와 이어지는 내용인데, 곱셈은 굳이 "뒤집지" 않고도 정의할 수 있고, 그게 더 간단합니다. (그래서 `init`와 `last`는 사용하는 게 좋다고 한 겁니다.)
- 굳이 꼬리재귀 형태로 정의하려고 끙끙대지 않기를 권장합니다. (문제에서 의도한 정의 방식이 아닙니다.)
- 두 인자 자연수들 중 (표준형으로 바꾼 형태를 기준으로 하든 인자로 주어진 형태 그대로를 기준으로 하든 간에) 길이가 짧은 쪽에 재귀횟수가 비례하도록 만들면 더 좋지만, 역시 필수 사항은 아니며 긴 쪽에 비례해도 점수 차이는 없습니다.
- 어느 부분에 재귀 호출을 걸어야할지 잘 생각해보기 바랍니다. 이걸 스스로 생각해내는 훈련이 가장 중요한 훈련이므로, 덧셈과는 달리 더이상의 힌트는 주지 않겠습니다. 위 곱셈 전개 도식에서 비슷한 패턴으로 반복되는 부분을 한 덩어리로 묶는다고 생각하셔요.

역시 `showN`과 `readN`을 이용해서 테스트해보기 바랍니다:

[*WinGHCi window, module* HW6 *loaded*]

```
*HW6> 959054 * 597062
572614699348
it :: Num a => a
(0.00 secs, 70,456 bytes)
*HW6> showN $ mulNslow (readN "959054") (readN "597062")
"572614699348"
it :: String
(48.52 secs, 8,798,286,080 bytes)
*HW6> showN $ mulNfast (readN "959054") (readN "597062")
"572614699348"
it :: String
(0.01 secs, 530,000 bytes)
*HW6>
```

더 큰 자연수들도 곱해보면서 시간 복잡도 한 단계($O(n)$ vs. $O(\log n)$)가 가져오는 현격한 차이를 즐겨보셔요:

[*WinGHCi window, module* HW6 *loaded*]

```
*HW6> showN $ mulNfast (readN "74568746646545604850484") (readN "4849796345353534098405")
"361643234964010475894568942299511255767878020"
```

```
it :: String
(0.09 secs, 6,861,192 bytes)
*HW6>
```

마지막 첨언으로, 숙제 코드 파일의 마지막 부분에는 `mulNslow`를 이용해서 `mulN` 함수가 정의되어 있고, 다시 이 `mulN` 함수를 이용해서 `facN` 함수가 (꼬리재귀 형태로) 정의되어 있습니다:

file: HW6.hs

```
mulN :: Nat -> Nat -> Nat
mulN  =  mulNslow      -- mulNslow 혹은 mulNfast

facN :: Nat -> Nat
facN n  =  f n [succ minBound]  where
    f x acc
        ¦ isNull x   =  acc
        ¦ otherwise  =  f (decN x) (mulN x acc)
```

일단 이 상태 (`mulN`이 `mulNslow`인 상태) 그대로 계승 계산 속도와 메모리 사용량을 확인해보면 다음과 같습니다:

[WinGHCi window, module HW6 loaded]

```
*HW6> showN $ facN $ readN "10"
"3628800"
it :: String
(0.01 secs, 375,544 bytes)
*HW6> showN $ facN $ readN "100"
"933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000"
it :: String
(4.23 secs, 603,906,824 bytes)
*HW6> showN $ facN $ readN "200"
"788657867364790503552363213932185062295135977687173263294742533244359449963403342920304284011984623904177212138919638830257642790242637105061926624952829931113462857270763317237396988943922445621451664240254033291864131227428294853277524242407573903240321257405579568660226031904170324062351700858796178922222789623703897374720000000000000000000000000000000000000000000000000000"
it :: String
(39.39 secs, 5,692,463,608 bytes)
*HW6>
```

이번에는 `mulN`의 정의를 `mulNfast`로 바꾸고 계승 계산 속도와 메모리 사용량을 다시 확인해보면 다음과 같습니다:

[WinGHCi window, module HW6 loaded]

```
*HW6> showN $ facN $ readN "10"
"3628800"
it :: String
(0.01 secs, 522,632 bytes)
*HW6> showN $ facN $ readN "100"
```

```
"933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000000"
it :: String
(1.55 secs, 347,669,712 bytes)
*HW6> showN $ facN $ readN "200"
"788657867364790503552363213932185062295135977687173263294742533244359449963403342920304284011984623904177212138919638830257642790242637105061926624952829931113462857270763317237396988943922445621451664240254033291864131227428294853277524242407573903240321257405579568660226031904170324062351700858796178922222789623703897374720000000000000000000000000000000000000000000000000000"
it :: String
(7.74 secs, 2,459,981,712 bytes)
*HW6>
```

차이가 분명히 나긴 나지만, `mulNslow`와 `mulNfast`가 그 자체로서 보여주었던 만큼의 드라마틱한 차이가 나지는 않습니다. 이 숙제에서 구현한 방식은, 적은 갯수의 아주 큰 자연수들을 더하거나 곱할 때에는 엄청난 효율을 보여주지만, 아주 많은 갯수의 작은 자연수들을 더하거나 곱할 때에는 별로 효과를 보기 힘든 방식이기 때문입니다.

이 숙제가 마지막 숙제일 확률은 1이 아닙니다. 약 0.999입니다.