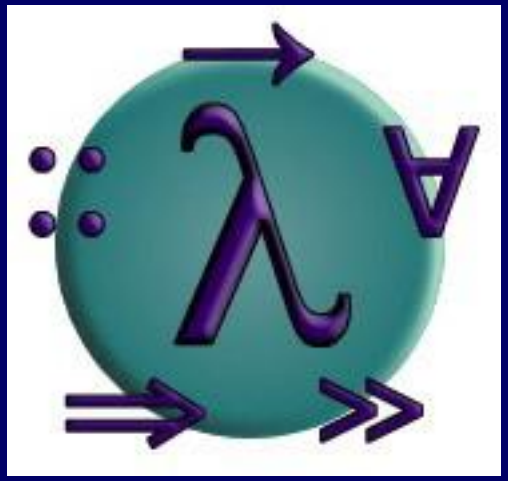
PROGRAMMING IN HASKELL



Chapter 7 - Higher-Order Functions

愛知県立大学 情報科学部 計算機言語論(山本晋一郎・大久保弘崇、2011年)講義資料。
オリジナルはhttp://www.cs.nott.ac.uk/~gmh/book.htmlを参照のこと。

Introduction

カリー化により 2 引数以上の関数の戻り値は関数となるため、多くの関数が厳密には高階関数である。より限定して引数として関数を取る関数という立場も。

関数を引数とする、または返り値とする関数を高階関数という

```
twice    :: (a → a) → a → a
twice f x = f (f x)
```

twice は第 1 引数に関数を取るので高階関数

高階関数は有用か?

- 高階関数によって広く用いられるプログラミングのイディオムを自然に表現できる
- 高階関数を用いてドメイン固有言語(domain specific language, DSL)を定義できる
- 高階関数の代数的性質はプログラムに関する論証に用いられる

The Map Function

高階ライブラリ関数 map は、引数として与えられた関数をリストの要素全てに適用する

```
map :: (a → b) → [a] → [b]
```

For example:

```
> map (+1) [1,3,5,7]

[2,4,6,8]
```

map 関数はリスト内包表記を用いてとても簡潔に定義できる:

```
map f xs = [f x | x ← xs]
```

別な方法として、主に証明に使うために、再帰を用いても定義できる:

```
map f []     = []
map f (x:xs) = f x : map f xs
```

The Filter Function

高階ライブラリ関数 filter は、リストから述語を満たす要素だけを選び出す

```
filter :: (a → Bool) → [a] → [a]
```

For example:

```
> filter even [1..10]

[2,4,6,8,10]
```

filter 関数はリスト内包表記を用いて定義できる:

```
filter p xs = [x | x ← xs, p x]
```

別な方法として、再帰を用いても定義できる:

```
filter p []     = []
filter p (x:xs)
   | p x        = x : filter p xs
   | otherwise  = filter p xs
```

The Foldr Function (右畳み込み)

リスト関数の多くは、単純な再帰パターンによって定義される(演算子 ⊕ と初期値 v がパラメータ):

```
f []     = v
f (x:xs) = x ⊕ f xs
```

f は空リストを初期値 v に写像し、非空リストを、
(先頭要素)と(残りのリストを f に適用した結果)
を演算子 ⊕ に適用した結果に写像する

For example:

```
sum []     = 0
sum (x:xs) = x + sum xs
```

$v = 0$
$\oplus = +$

```
product []     = 1
product (x:xs) = x * product xs
```

$v = 1$
$\oplus = *$

```
and []     = True
and (x:xs) = x && and xs
```

$v = \text{True}$
$\oplus = \&\&$

高階ライブラリ関数 foldr (fold right, 右畳み込み)は、関数 (⊕) と初期値 v を引数として、この単純な再帰パターンを表現する

For example:

教科書はポイントフリー(引数を明示しない)スタイルで書かれているので、両辺に引数を補って考える

```
sum xs      = foldr (+) 0 xs

product xs = foldr (*) 1 xs

or xs       = foldr (||) False xs

and xs      = foldr (&&) True xs
```

foldr 自体は再帰を用いて定義できる:

```
foldr :: (a → b → b) → b → [a] → b
foldr f v []     = v
foldr f v (x:xs) = f x (foldr f v xs)
```

または

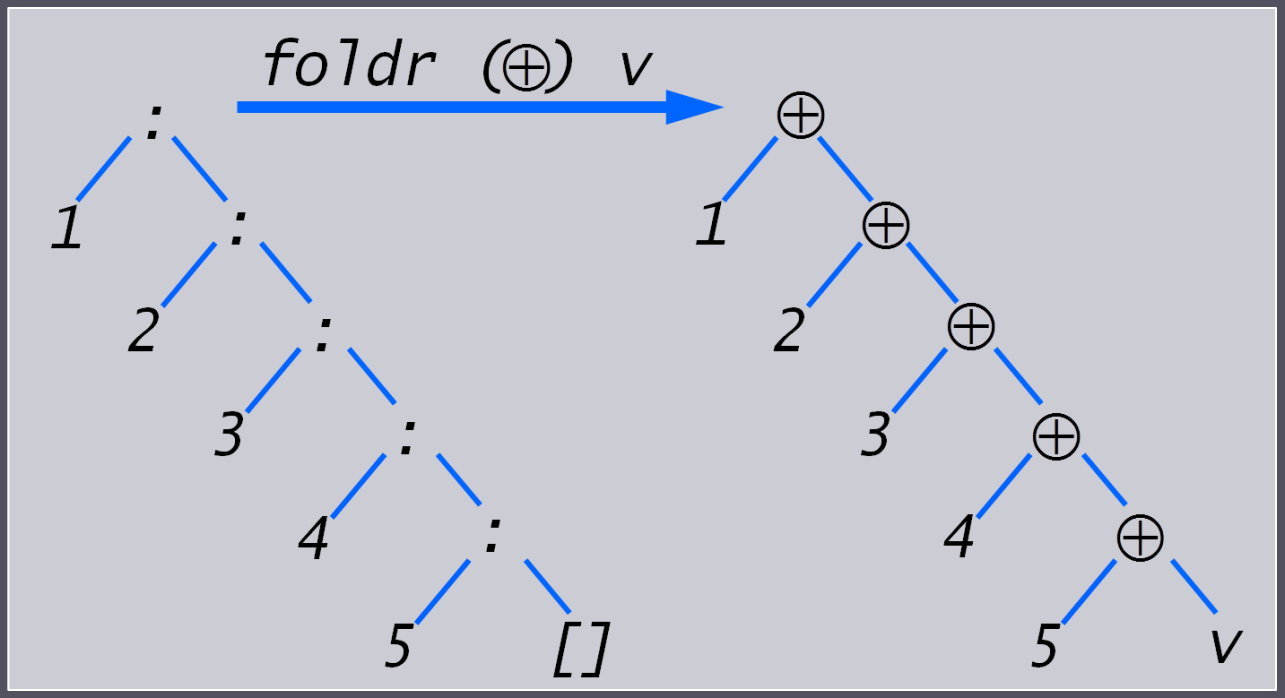
```
foldr f v (x:xs) = x `f` (foldr f v xs)
```

実際には、foldr を再帰的に理解するのではなく、
リストの cons (:) を演算子 ⊕ に、
終端 [] を初期値 v に、同時に置き換えると理解すべき

第 1, 2 引数は変化しない(持ち回される)

cons (:) を ⊕ に、終端 [] を v に置換

- foldr (⊕) v [x0, x1, ..., xn]
  = foldr (⊕) v (x0 : x1 : ... : xn : [])
  = x0 ⊕ (x1 ⊕ (... (xn ⊕ v)))



For example:

```
  sum [1,2,3]
=
  foldr (+) 0 [1,2,3]
=
  foldr (+) 0 (1:(2:(3:[])))
=
                1+(2+(3+0))
=
  6
```

Replace each (:) by (+) and [] by 0.

For example:

```
  product [1,2,3]
=
  foldr (*) 1 [1,2,3]
=
  foldr (*) 1 (1:(2:(3:[])))
=
            1*(2*(3*1))
=
  6
```

Replace each (:) by (*) and [] by 1.

foldr を用いた他の例

foldr は単純な再帰をパターン化しただけだが、
想像以上に多様な関数を定義できる

length 関数について考えてみる:

```
length        :: [a] → Int
length []      = 0
length (_:xs) = 1 + length xs
```

例えば:

Replace each (:)
by λ_ n → 1+n and [] by 0.

```
  length [1,2,3]
=
  length (1:(2:(3:[])))
=
        1+(1+(1+0))
=
  3
```

従って:

```
length xs = foldr (λ_ n → 1+n) 0 xs
```

λ_ n → 1+n はどうやって求める?

```
  length [1, 2, 3]
= foldr f v (1 : (2 : (3 : [])))
= 1 `f` (2 `f` (3 `f` v))
    where
      v = 0
      f x y = 1 + y
```

注意: 3 `f` v == f 3 v

- f 3 v より、f x y は要素 3 を x に、初期値 v を y に取って、[3] の長さ 1 を返す
- f 2 (f 3 v) より、f x y は要素 2 を x に、残りのリストの長さ 1 を y に取って、[2, 3] の長さ 2 を返す
- f x y は、要素を x に、残りのリストの長さを y に取って、y に 1 を加えて返せば良い(v は 0)
  f x y = 1 + y　　(x は利用しないので _ で十分)

reverse について考えてみる:

```
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

例えば:

```
  reverse [1,2,3]
=
  reverse (1:(2:(3:[])))
=
  (([] ++ [3]) ++ [2]) ++ [1]
=
  [3,2,1]
```

Replace each (:)
by λx xs → xs ++ [x]
and [] by [].

従って:

```
reverse xs =
    foldr (λx xs → xs ++ [x]) [] xs
```

最後に、リストの連結 (++) を foldr によって極めて簡潔に定義する:

```
(++) xs ys = foldr (:) ys xs
```

Replace each (:) by (:) and [] by ys.

λx xs → xs ++ [x] はどうやって求める?

```
   reverse [1,2,3]
= foldr f v (1 : (2 : (3 : [])))
= 1 `f` (2 `f` (3 `f` v))
    where
      v = []
      f x y = y ++ [x]
```

注意: 3 `f` v == f 3 v

- f 3 v より、f x y は要素 3 を x に、初期値 v を y に取って、[3] の反転 [3] を返す
- f 2 (f 3 v) より、f x y は要素 2 を x に、残りのリストの反転 [3] を y に取って、[2,3] の反転 [3,2] を返す
- f x y は、要素を x に、残りのリストの反転を y に取って、y の後ろに [x] を結合したリストを返せば良い(v は[])
  f x y = y ++ [x]

reverse の補足

- 要素をリストの末尾に追加する関数 snoc を導入
  snoc x xs = xs ++ [x]

```
  reverse [1,2,3]
= r (1  :     (2  :     (3  :     []))
=    1 `snoc` (2 `snoc` (3 `snoc` []))
= (([] ++ [3]) ++ [2]) ++ [1]
= [] ++ [3] ++ [2] ++ [1]
= [3,2,1]
```

- snoc == λx xs → xs ++ [x]

従って:

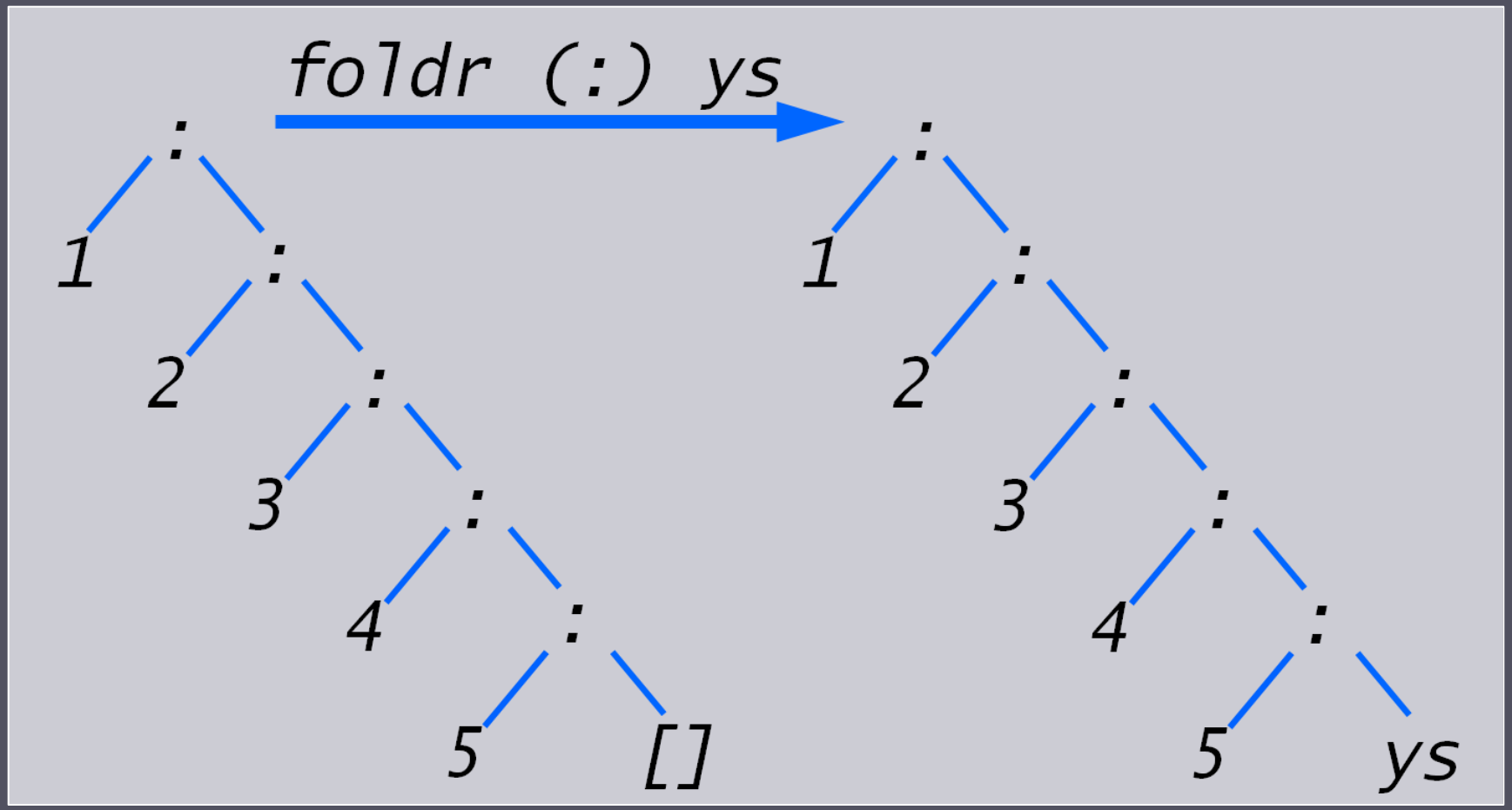
```
reverse xs =
   foldr (λx xs → xs ++ [x]) [] xs
```

最後に、リストの連結 (++) を foldr によって極めて簡潔に定義する:

```
(++) xs ys = foldr (:) ys xs
```

Replace each (:) by (:) and [] by ys.

append を foldr によって表現

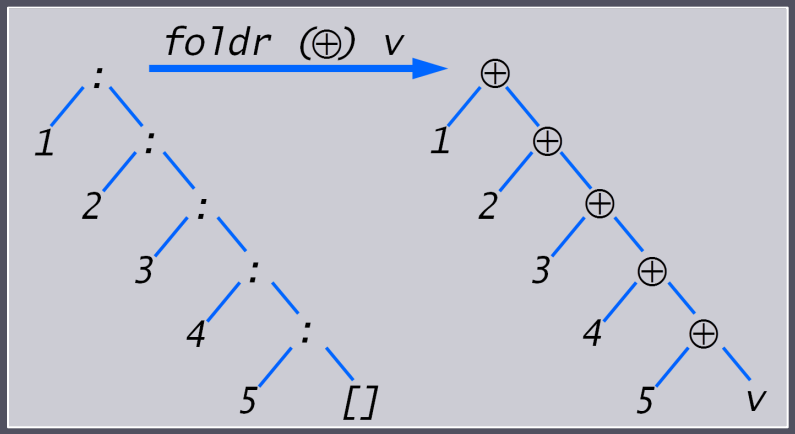


foldr は有用か?

- sum のようなリスト関数をより簡潔に定義できる
- foldr を用いて定義された関数の性質は、foldr の代数的性質(fusion や banana split 規則)を用いて証明できる
- 明示的な再帰の代わりに foldr を用いると、高度なプログラムの最適化が容易になる

foldr (右畳み込み)のまとめ

- `f []     = v`
  `f (x:xs) = x ⊕ f xs`
  と定義される関数 f xs は、foldr (⊕) v xs と書ける
- foldr (⊕) v [x0, x1, ..., xn] = x0 ⊕ (x1 ⊕ (... (xn ⊕ v)))



- `sum xs     = foldr (+) 0 xs`
- `product xs = foldr (*) 1 xs`
- `and xs     = foldr (&&) True xs`

ここから foldl の説明が始まるが、foldr を
初めて学んだ人は後回しにしても良い

The Foldl Function (左畳み込み)

リスト関数の多くは、単純な再帰パターンによって定義される(演算子 ⊕ と蓄積変数の初期値 v がパラメータ):

```
f xs = f' v xs
f' ac []     = ac
f' ac (x:xs) = f' (ac ⊕ x) xs
```

f': f の補助関数
ac: それまで処理した中間結果を保持する蓄積変数

f' は、空リストを蓄積変数の値に、非空リストを、
(蓄積変数と先頭要素に ⊕ を適用した結果)と
(残りのリスト)を f' に適用した結果に写像する

For example:

```
sum xs = sum' 0 xs
sum' ac []     = ac
sum' ac (x:xs) = sum' (ac+x) xs
```

v = 0
⊕ = +

```
product xs = prod' 1 xs
prod' ac []     = ac
prod' ac (x:xs) = prod' (ac*x) xs
```

v = 1
⊕ = *

```
rev xs = rev' [] xs
rev' ac []     = ac
rev' ac (x:xs) = rev' (x:ac) xs
```

v = [] ⊕ = λys y -> y:ys

高階ライブラリ関数 foldl (fold left, 左畳み込み)は、関数 (⊕) と蓄積変数の初期値 v を引数として、この単純な再帰パターンを表現する
For example:

```
sum xs      = foldl (+) 0 xs

product xs = foldl (*) 1 xs

reverse xs = foldl (λys y -> y:ys) [] xs
```

foldl 自体は再帰を用いて定義できる:

```
foldl :: (a → b → a) → a → [b] → a
foldl f ac []     = ac
foldl f ac (x:xs) = foldl f (f ac x) xs
```

または

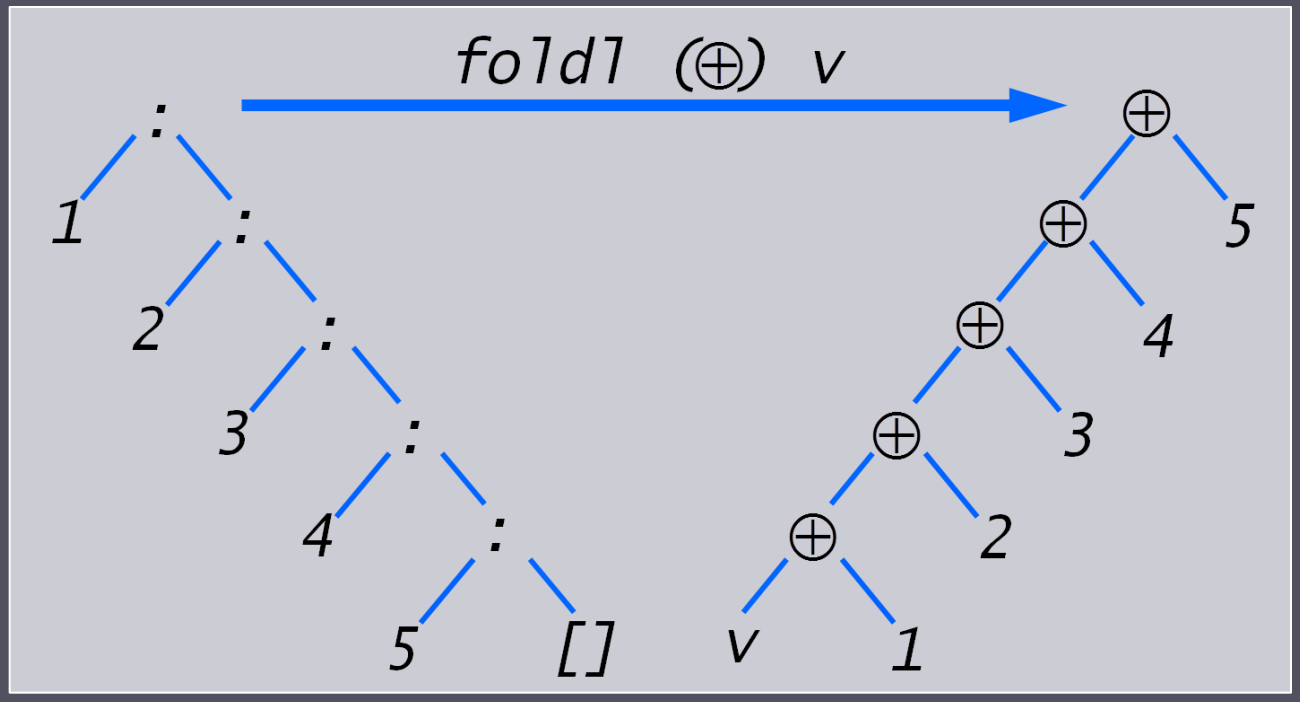
```
foldl f ac (x:xs) = foldl f (ac `f` x) xs
```

実際には、foldl を再帰的に理解するのではなく、蓄積変数の初期値を v とし、左結合の演算子 ⊕ を用いてリストから式を構成すると理解すべき

蓄積変数 ac に結果を蓄えていき、最後にその値を返す。
第 1 引数は変化しない(持ち回される)。

蓄積変数の初期値を v とし、左結合の ⊕ を用いてリストから式を構成

- foldl (⊕) v [x0, x1, ..., xn]
  = foldl (⊕) v (x0 : x1 ... : xn : [])
  = (((v ⊕ x0) ⊕ x1) ...) ⊕ xn



For example:

```
  sum [1,2,3]
=
  foldl (+) 0 [1,2,3]
=
  foldl (+) 0 (1:(2:(3:[])))
=
        ((0+1)+2)+3
=
  6
```

リストの前に蓄積変数の初期値 0 を置いて、(+) で左から演算する

For example:

```
  product [1,2,3]
=
  foldl (*) 1 [1,2,3]
=
  foldl (*) 1 (1:(2:(3:[])))
=
        ((1*1)*2)*3
=
  6
```

リストの前に蓄積変数の初期値 1 を置いて、(*) で左から演算する

reverse を foldl を用いて実現

```
    rev [1,2,3]
  = foldl (λys y -> y:ys) [] [1,2,3]
  = foldl (…) ((…) [] 1) [2,3]
  = foldl (…) (1:[]) [2,3]
  = foldl (…) ((…) [1] 2) [3]
  = foldl (…) (2:[1]) [3]
  = foldl (…) ((…) [2,1] 3) []
  = foldl (…) (3:[2,1]) []
  = [3,2,1]
```

蓄積変数 ys には、既に処理した前方のリストを反転した結果が渡される

λys y -> y:ys はどうやって求める?

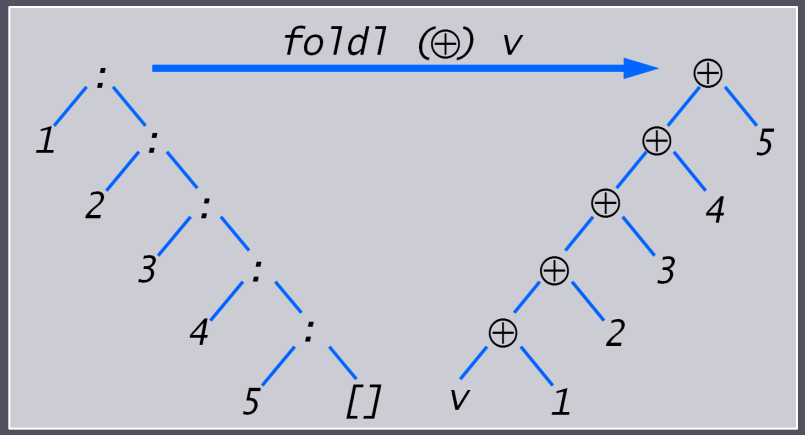
```
   reverse [1,2,3]
= foldl f v (1 : (2 : (3 : [])))
= ((v `f` 1) `f` 2) `f` 3
    where
      v = []
      f x y = y:x
```

注意: v `f` 1 == f v 1

- f v 1 より、f x y は初期値 v を x に、要素 1 を y に取って、[1] の反転 [1] を返す
- f (f v 1) 2 より、f x y は前方のリストの反転 [1] を x に、要素 2 を y に取って、[1,2] の反転 [2,1] を返す
- f x y は、前方のリストの反転を x に、要素を y に取って、x の前に y を cons したリストを返せば良い(v は[])
  f x y = y:x

foldl (左畳み込み)のまとめ

- `f xs = f' v xs`
  `f' ac []     = ac`
  `f' ac (x:xs) = f' (ac ⊕ x) xs`
  と定義される関数 f xs は、foldl (⊕) v xs と書ける
  - 空リストを蓄積変数の値に、非空リストを、(蓄積変数と先頭要素に ⊕ を適用した結果)と(残りのリスト)を f' に適用した結果に写像
- foldl (⊕) v [x0, x1, ..., xn] = (((v ⊕ x0) ⊕ x1) ...) ⊕ xn



- `reverse xs = foldl (¥ys y -> y:ys) [] xs`

foldl の説明終わり

Other Library Functions

高階ライブラリ関数 (.) は、2 つの関数を合成した関数を返す

```
(.)    :: (b → c) → (a → b) → (a → c)
f . g  = λx → f (g x)
```

For example:

```
odd :: Int → Bool
odd  = not . even
```

```
odd    = not . even
odd    = λx → not (even x)
odd n = (not . even) n
odd n = not (even n)
odd n = not $ even n
```

高階ライブラリ関数 all は、リストの全ての要素が与えられた述語を満たすか判定する

```
all      :: (a → Bool) → [a] → Bool
all p xs = and [p x | x ← xs]
```

For example:

```
> all even [2,4,6,8,10]

True
```

all と双対な高階ライブラリ関数 any は、リストの要素の少なくとも 1 つが与えられた述語を満たすか判定する

```
any      :: (a → Bool) → [a] → Bool
any p xs = or [p x | x ← xs]
```

For example:

```
> any isSpace "abc def"

True
```

高階ライブラリ関数 takeWhile は、リストの先頭から述語を満たす区間を取り出したリストを返す

```
takeWhile :: (a → Bool) → [a] → [a]
takeWhile p []      = []
takeWhile p (x:xs)
    | p x           = x : takeWhile p xs
    | otherwise     = []
```

For example:

```
> takeWhile isAlpha "abc def"

"abc"
```

takeWhile と双対な高階ライブラリ関数 dropWhile は、リストの先頭から述語を満たす区間を除いたリストを返す

```
dropWhile :: (a → Bool) → [a] → [a]
dropWhile p []      = []
dropWhile p (x:xs)
    | p x           = dropWhile p xs
    | otherwise     = x:xs
```

For example:

```
> dropWhile isSpace "   abc"

"abc"
```

まとめ(7章)

- 高階関数: 関数を引数、または返り値とする関数
- map: 与えられた関数をリストの要素全てに適用
- filter: リストから述語を満たす要素を選び出す
- foldr: 右畳み込み
  foldr (⊕) v [x0, x1, ..., xn] = x0 ⊕ (x1 ⊕ (... (xn ⊕ v)))
  - `sum xs = foldr (+) 0 xs`
  - `product xs = foldr (*) 1 xs`
  - `and xs = foldr (&&) True xs`
- (.): 関数合成 `f . g = λx → f (g x)`
- all: リストの全ての要素が述語を満たすか判定
- takeWhile: リストの先頭から述語を満たす区間を取り出す

まとめ(foldr と foldl)

- 先頭要素は最外、v は最内右、foldr は結果の中へ

```
                    foldr (+) 0 (1:(2:(3:[])))
= 1 +              (foldr (+) 0    (2:(3:[])))
= 1 + (2 +         (foldr (+) 0       (3:[])))
= 1 + (2 + (3 + (foldr (+) 0          [])))
= 1 + (2 + (3 + 0))
= 6
```

- 先頭要素は最内、v は最内左、結果は foldl の蓄積変数へ

```
  foldl (+)     0                    (1:(2:(3:[])))
= foldl (+)   (0 + 1)                 (2:(3:[]))
= foldl (+)  ((0 + 1) + 2)               (3:[])
= foldl (+) (((0 + 1) + 2) + 3)             []
=            ((0 + 1) + 2) + 3
= 6
```