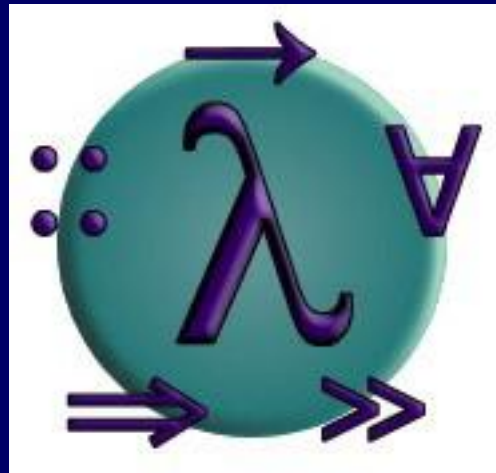


# PROGRAMMING IN HASKELL

## プログラミングHaskell



## Chapter 6 - Recursive Functions

### 再帰関数

# イントロダクション

これまで見てきたように、多くの関数は他の関数を利用して定義できる

```
factorial  :: Int → Int  
factorial n = product [1..n]
```

factorial は、任意の整数  $n$  を、  
1 から  $n$  の整数の積に写像する

式は、関数とその引数に適用する手続きにより、段階的に評価される

例:

```
factorial 4  
=  
product [1..4]  
=  
product [1,2,3,4]  
=  
1*2*3*4  
=  
24
```

# 再帰関数(Recursive Functions)

Haskell では、自分自身を用いて関数を定義できる。  
そのような関数を再帰的(recursive)と呼ぶ。

```
factorial 0      = 1
```

```
factorial n      = n * factorial (n-1)
```

```
n+kパターン: factorial (n+1) = (n+1) * factorial n
```

factorial は 0 を 1 に写像し、その他の正の整数を、  
その値 × 1つ小さい値の factorial に写像する

For example:

$$\begin{aligned} & \text{factorial } 3 \\ = & 3 * \text{factorial } 2 \\ = & 3 * (2 * \text{factorial } 1) \\ = & 3 * (2 * (1 * \text{factorial } 0)) \\ = & 3 * (2 * (1 * 1)) \\ = & 3 * (2 * 1) \\ = & 3 * 2 \\ = & 6 \end{aligned}$$

## 注意:

- $\text{factorial } 0 = 1$  は 1 が乗法の単位元なので適切である:  $1 * x = x = x * 1$
- 0 より小さい整数に対して、基底ケースに到達できないため発散する:

```
> factorial (-1)
```

```
Error: Control stack overflow
```

# 再帰が役に立つ理由

- factorial のように、他の関数(例えば product)を利用して定義するとより簡潔に定義できる関数もある
- しかし、以降の章で分かるように、自分自身を利用して自然に定義できる関数も多い
- 再帰を用いて定義した関数の性質は、単純かつ強力な数学的帰納法を用いて証明できる(13章)

# リストに対する再帰

再帰は、整数だけに限られるのではなく、リスト関数にも使える

```
product      :: [Int] → Int
product []   = 1
product (n:ns) = n * product ns
```

product は空リストを 1 に写像し、非空リストを、  
先頭要素 × 残りのリストの product に写像する



For example:

```
product [2,3,4]
=
2 * product [3,4]
=
2 * (3 * product [4])
=
2 * (3 * (4 * product []))
=
2 * (3 * (4 * 1))
=
24
```

product と同じ再帰パターンで、リストの長さを返す関数を定義する

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

length は空リストを 0 に写像し、非空リストを、**残りのリストのlengthより1つ大きな値** に写像する

For example:

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

同じ再帰パターンで、リストを反転する関数を定義する

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

reverse は空リストを空リストに写像し、非空リストを、  
(残りのリストのreverse)と(先頭要素だけのリスト)  
を連結したリスト に写像する

For example:

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

# 複数の引数

複数の引数を取る関数も再帰的に定義できる  
例えば:

- 2つのリストの要素同士を組にする:  
Zipping the elements of two lists:

```
zip      :: [a] → [b] → [(a,b)]
zip []   _      = []
zip _    []     = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

## ■ リストの先頭から n 要素を取り除く:

```
drop      :: Int → [a] → [a]
drop 0 xs = xs
drop n [] = []
drop n (_:xs) = drop (n-1) xs
n+kパターン: drop (n+1) [] = []
              drop (n+1) (_:xs) = drop n xs
```

## ■ 2つのリストを結合する:

```
(++)      :: [a] → [a] → [a]
[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

# クイックソート

整数リストを整列するクイックソートのアルゴリズムは、2つの規則で定式化される:

- 空リストはソート済み
- 非空リストのソートは以下の3つのリストを連結したリスト
  - (残りのリスト中の先頭要素以下の要素)をソートしたリスト
  - 先頭要素のみのリスト
  - (残りのリスト中の先頭要素より大きな要素)をソートしたリスト



再帰を用いて仕様を直接的に実装に変換できる:

```
qsort      :: [Int] → [Int]
qsort []   = []
qsort (x:xs) =
    qsort smaller ++ [x] ++ qsort larger
  where
    smaller = [a | a ← xs, a ≤ x]
    larger  = [b | b ← xs, b > x]
```

注意:

- 他のプログラミング言語と比べて、おそらく最も簡潔な実装!

例 (qsort を q と省略):

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



q [2, 1] ++ [3] ++ q [4, 5]



q [1] ++ [2] ++ q []

q [] ++ [4] ++ q [5]



[1]

[]

[]

[5]

# 相互再帰

- 2つ以上の関数が、お互いを参照し合う

- `even, odd :: Int -> Bool`

- `even 0 = True`

- `even n = odd (n - 1)`

- `odd 0 = False`

- `odd n = even (n - 1)`

- 実際の定義(上の定義は非効率)

- `even, odd :: (Integral a) => a -> Bool`

- `even n = n `rem` 2 == 0`

- `odd n = not (even n)`

rem は習っていない  
ので mod と考える

## 再帰の秘訣(product)

- 型を定義する(数値リストの要素の積を求める)

```
product :: [Int] → Int
```

- 場合分けをする

```
product [] =
```

```
product (n:ns) =
```

- 簡単な方を定義する

```
product [] = 1
```

```
product (n:ns) =
```

- 複雑な方を定義する

```
product [] = 1
```

```
product (n:ns) = n * product ns
```

- 一般化し単純にする

```
product :: Num a => [a] → a
```

# 再帰の秘訣(drop)

- 型を定義する(リストの先頭から n 要素を取り除く)

$\text{drop} :: \text{Int} \rightarrow [a] \rightarrow [a]$

- 場合分けをする

$\text{drop } 0 [] = \text{drop } n [] = []$   
 $\text{drop } 0 (x:xs) = \text{drop } n (x:xs) = \dots$

- 簡単な方を定義する

$\text{drop } 0 [] = []$   
 $\text{drop } 0 (x:xs) = x:xs$   
 $\text{drop } n [] = []$

$\text{drop } 0 \text{ xs} = \text{xs}$

省略すれば実行時エラー

- 複雑な方を定義する

$\text{drop } n (x:xs) = \text{drop } (n-1) \text{ xs}$

- 一般化し単純にする

$\text{drop } n (\_ : xs) = \dots$

$\text{drop} :: \text{Integral } b \Rightarrow b \rightarrow [a] \rightarrow [a]$

# まとめ(6章)

## ■ 再帰関数

- 自然な定義
- 数学的帰納法との好相性

```
factorial 0 = 1
```

```
factorial n = n * factorial (n-1)
```

- 相互再帰: 2 つ以上の関数が、お互いを参照し合う
- リストに対する再帰

```
product [] = 1
```

```
product (n:ns) = n * product ns
```

## ■ 再帰の秘訣: 5 段階の工程

型定義、場合分け、基底部、再帰部、一般化

# 問題

- product に関する再帰の秘訣(6.6節、p.66)において、 $\text{product} [] = 1$  と定義する理由を、“1 は乗法の単位元だから” としている。これを解説せよ。