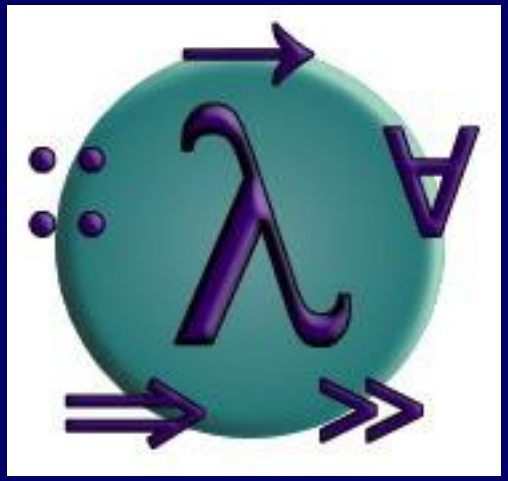
PROGRAMMING IN HASKELL
プログラミングHaskell



Chapter 4 - Defining Functions
関数定義

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、2013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

条件式 (Conditional Expressions)

注意: 条件文ではない

他のプログラミング言語と同様に、条件式 (if 式) を用いて関数を定義

```
abs  :: Int → Int
abs n = if n ≥ 0 then n else -n
```

abs は整数 n を取り、n が非負のとき n そのものを返し、それ以外は −n を返す

条件式の入れ子:

```
signum n = if n < 0
           then -1
           else
             if n == 0 then 0 else 1
```

```
signum  :: Int → Int
signum n = if n < 0 then -1 else
                 if n == 0 then 0 else 1
```

正の数	1
0	0
負の数	-1 を返す

注意:

Haskell では、条件式は必ず else 部を持つため、入れ子になった条件式の曖昧さ (dangling else) は生じない

ガード付き等式 (Guarded Equations)

条件式 (if 式) の代わりに、ガード式を用いて関数を定義

```
abs n | n ≥ 0      = n
      | otherwise  = -n
```

前ページの定義と同じ、ただしガード式を使用

ガード式を用いると、複数の場合分けによる関数定義が読みやすくなる:

```
signum n | n < 0      = -1
         | n == 0     = 0
         | otherwise = 1
```

注意:

「その他の場合」を表す otherwise は Prelude において True と定義されている

パターンマッチング（Pattern Matching）

関数の多くは、引数に対するパターンマッチにより、簡潔かつ直観的に定義できる

```
not       :: Bool → Bool
not False = True
not True  = False
```

not は False を True へ、True を False へ写像

関数定義におけるパターンマッチの書き方は一通りとは限らない。例えば、

```
(&&)              :: Bool → Bool → Bool
True  && True  = True
True  && False = False
False && True  = False
False && False = False
```

は、よりコンパクトにも書ける。

```
True && True = True
_    && _    = False
```

次の定義はより効率的である。1つめの引数が False のとき、2 つめの引数を評価しない:

```
True  && b = b
False && _ = False
```

注意:

下線文字 (アンダースコア) “_” は任意の値とマッチするワイルドカード

パターンは記述順 (上から下) にマッチを試される。例えば、次の定義は常に False を返す:

```
_     && _     = False
True  && True  = True
```

パターン中に同じ変数を 2 回使うことはできない。例えば、次の定義 (最初の式) はエラーとなる:

```
b && b = b
_ && _ = False
```

リストパターン (List Patterns)

空でないリストは、内部的には“cons”と呼ばれる演算子：(コロン文字)を繰り返し用いて構成されている

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

1:(2:(3:(4:[]))) を意味する。
1:2:3:4:[] とも書ける(演算子“:”は右結合)。

リストに対する関数は x:xs という形のパターンで定義できる

```
head        :: [a] → a
head (x:_)   = x

tail        :: [a] → [a]
tail (_:xs) = xs
```

head と tail は空でないリストをそれぞれ、
先頭要素、残りのリストに写像する

注意:

関数の典型的なパターンマッチ:

```
f []     = …
f (x:xs) = …
```

- x:xs パターンは非空リストにのみマッチする:

```
> head []
Error
```

- x:xs パターンは括弧でくくる必要がある。関数適用はリスト構成子 “:” より優先度が高い。例えば、次の定義はエラーになる:

(head x):_ = x と扱われる

```
head x:_ = x
```

Integer Patterns (非推奨)

Haskell 2010 で使用禁止

数学と同様に、整数上の関数を定義するのに n+k パターンが使える。ここで n は整数変数で、k は正の整数定数。

```
pred       :: Int → Int
pred (n+1) = n
```

pred は正の整数を 1 つ小さな値に写像する

注意:

- n+k パターンは k 以上の整数にのみマッチする

```
> pred 0
Error
```

n+k パターンは括弧でくくる必要がある。関数適用は加算の“+”より優先度が高い。例えば、次の定義はエラーになる:

```
pred n+1 = n
```

λ式 (Lambda Expressions)

λ式を用いて、名前を付けずに関数を構成できる

f x = x+x と同じ働きをするが、名前を持たない

λx → x+x

「数 x を取り x+x を結果として返す」
無名関数を表す

注意:

- 記号 λ はギリシャ文字の「ラムダ」で、キーボードからはバックスラッシュ “\” で入力する
  - 日本語キーボードでは “¥”
- 数学では、無名関数を記号 ↦ を用いて x ↦ x+x のように表す
- Haskell で無名関数の表記に λ を用いるのはλ算法からきている。λ算法は Haskell が基礎を置いている関数理論である。

λ式が有用な理由

λ式はカリー化された関数の形式的な意味付けに用いられる

例:

```
add x y = x+y
```

カリー化された add の意味

```
add = λx → (λy → x+y)
```

λ式は、関数を結果として返す関数を定義するときにも用いられる

例:

```
const    :: a → b → a
const x _ = x
```

2引数の関数:
第2引数は無視し、第1引数を返す

より自然に

```
const  :: a → (b → a)
const x = λ_ → x
```

1引数の関数:
引数が何であってもxを返す関数を返す

λ式は、1 回しか参照されない関数に名前を付けるのを避けるためにも用いられる

例:

```
odds 4 = map f [0..3] where …
       = map f [0, 1, 2, 3] where …
       = [1, 3, 5, 7]
```

```
odds n = map f [0..n-1]
         where
             f x = x*2 + 1
```

よりシンプルに

```
odds n = map (λx → x*2 + 1) [0..n-1]
```

map はリストの要素の全てに関数を適用し、結果のリストを返

セクション

演算子が 2 つの引数の間に置かれているとき、処理系内部では、括弧を付けて演算子をカリー化関数にして引数の前に置くように変換される

例:

```
> 1+2
3

> (+) 1 2
3
```

この変換において、演算子の引数を括弧の中に含んでもよい

例:

```
> (1+) 2
3

> (+2) 1
3
```

一般に、演算子 ⊕ と引数 x, y に対して、3種類の関数 (⊕), (x⊕), (⊕y) をセクションと呼ぶ

セクションが有用な理由

セクションを用いると、単純だが有用な関数を簡潔に定義できる

例:

- `(1+)` - successor function
- `(1/)` - reciprocation function (逆数関数)
- `(*2)` - doubling function
- `(/2)` - halving function

まとめ（4章）

- ガード式

```
abs n | n ≥ 0      = n
      | otherwise = -n
```

関数の典型的なパターンマッチ:

```
f []     = …
f (x:xs) = …
```

- パターンマッチング
  - リストパターン　`head (x:_) = x`
- λ式: 関数の記法、名前を付けずに関数を構成
  - カリー化された関数の意味
    `add x y = x+y` の意味は `add = λx → (λy → x+y)`
  - 関数を結果として返す関数

```
const  :: a → (b → a)
const x = λ_ → x
```

- セクション: 演算子を関数化し、さらに部分適用も
  - x ⊕ y に対して関数 (⊕), (x⊕), (⊕y)