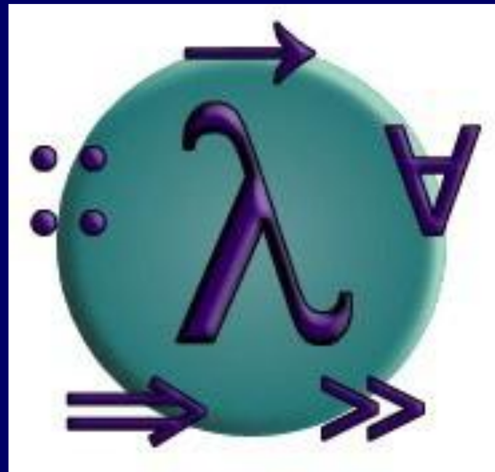
PROGRAMMING IN HASKELL
プログラミングHaskell



Chapter 10 - Declaring Types and Classes
型とクラスの定義

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

型宣言 (Type Declarations)

型宣言を用いて既存の型に別名を付けることができる

```
type String = [Char]
```

String は [Char] 型の別名

型宣言を用いて別名を付けると型が読みやすくなる。
例えば、以下のように型を定義すると、

```
type Pos = (Int,Int)
```

次のように使用できる:

```
origin      :: Pos
origin       = (0,0)

left        :: Pos → Pos
left (x,y) = (x-1,y)
```

関数定義と同様に、型宣言でも引数が使える。
例えば、以下のように型を定義すると、

a 型の値の 2 項組

```
type Pair a = (a,a)
```

次のように使用できる:

```
mult       :: Pair Int → Int
mult (m,n) = m*n

copy       :: a → Pair a
copy x      = (x,x)
```

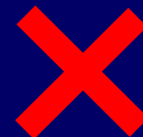
型宣言は入れ子になってもよい:

```
type Pos   = (Int,Int)

type Trans = Pos → Pos
```

しかし、型宣言 (type) では型の再帰は許されない:

```
type Tree = (Int,[Tree])
```



データ宣言 (Data Declarations)

データ宣言を用いて、まったく新しい型をその要素の値と共に定義できる

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

Bool は 2 つの新しい値
False と True を持つ新しい型

Note:

- 二つの値 False と True は Bool 型の (データ) 構成子 (data constructor) という
- 型名と構成子名は大文字で始まる

注意: 変数、関数、型変数名は小文字で始まる

- データ宣言は文脈自由文法に似ている
  - データ宣言は型に含まれる値を定義する
  - 文脈自由文法は言語に含まれる文を定義する

新しい型の値は、組み込みの型の値と同様に使える。例えば、以下のように型を定義すると、

```
data Answer = Yes | No | Unknown
```

次のように使用できる:

```
answers      :: [Answer]
answers       = [Yes,No,Unknown]

flip         :: Answer → Answer
flip Yes      = No
flip No       = Yes
flip Unknown  = Unknown
```

構成子による場合分け

データ宣言の構成子は引数を取ることもできる。
例えば、次のように型を定義すると、

```
data Shape = Circle Float
           | Rect Float Float
```

次のように使用できる:

```
square          :: Float → Shape
square n         = Rect n n

area            :: Shape → Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y
```

構成子による場合分けと値の取り出し

注意:

- Shape 型の値は次のいずれかの形をしている
  - Circle r　　r は Float 型
  - Rect x y　　x と y は Float 型
- 構成子 Circle と Rect は、Shape 型の値を作成する関数と見なせる:

```
Circle :: Float → Shape

Rect   :: Float → Float → Shape
```

Rect は 2 つの Float を受け取り Shape を返す関数。
ただし、Rect に関する計算規則は無い。

データ宣言それ自身も引数を取ることができる。例えば、以下のように型を定義すると、

注意(重要)：異常を正常値と分離

```
data Maybe a = Nothing | Just a
```

次のように使用できる:

```
safediv    :: Int → Int → Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m `div` n)

safehead   :: [a] → Maybe a
safehead [] = Nothing
safehead xs = Just (head xs)
```

エラー付きの Int

エラー付きの a

構成子による場合分けと値の抽出 (case式)

- 式の値 (形) によって場合分けしたい
  - リスト型 (構成子は [] と cons (:))

```
case exp of
  []       → …
  (x:xs)   → …
```

```
f :: [a] → …
f []       = …
f (x:xs)   = …
```

  - Maybe 型 (構成子は Nothing と Just)

```
case exp of
  Nothing → …
  Just x  → …
```

```
f :: Maybe a → …
f Nothing  = …
f (Just x) = …
```

再帰型

データ型は自分自身を使って定義することもできる。
すなわち、型は再帰的に定義されることもある。

```
data Nat = Zero | Succ Nat
```

Nat は 2 つの構成子 Zero :: Nat と
Succ :: Nat → Nat を持つ新しい型

Note:

- Nat 型の値は Zero か、あるいは Succ n の形をしている (ただし、n :: Nat)。すなわち、Nat は以下のような値の無限列を含む:

```
Zero

Succ Zero

Succ (Succ Zero)
```

⋮

- Nat 型の値は自然数とみなせる。すなわち、Zero は 0 を、Succ は 1 つ大きな整数を返す関数 (1+) を表している。

- 例えば、以下の値は、

```
Succ (Succ (Succ Zero))
```

自然数の 3 を表している

```
1 + (1 + (1 + 0))
```
=
```
3
```

再帰を用いて、Nat 型の値と Int 型の値の変換関数を容易に定義できる:

```
nat2int            :: Nat → Int
nat2int Zero        = 0
nat2int (Succ n)    = 1 + nat2int n

int2nat            :: Int → Nat
int2nat 0           = Zero
int2nat n           = Succ (int2nat (n - 1))
```

n+kパターン　`int2nat (n+1) = Succ (int2nat n)`

2 つの自然数 Nat を足すには、最初に自然数を整数 Int に変換し、整数を足し合わせ、結果を自然数 Nat に戻せばよい:

```
add     :: Nat → Nat → Nat
add m n = int2nat (nat2int m + nat2int n)
```

しかし、変換しないで済む関数 add を再帰を用いて定義できる:

```
add Zero     n = n
add (Succ m) n = Succ (add m n)
```

例えば:

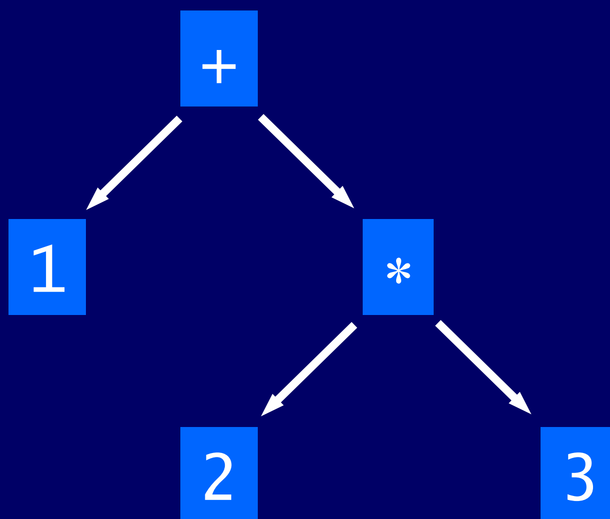
```
  add (Succ (Succ Zero)) (Succ Zero)
=
  Succ (add (Succ Zero) (Succ Zero))
=
  Succ (Succ (add Zero (Succ Zero))
=
  Succ (Succ (Succ Zero))
```

注意:

- add の再帰的な定義は次の法則と関係している
  $0+n = n$ $(1+m)+n = 1+(m+n)$

数式

整数上の加算と乗算によって構成される単純な数式を考えてみる



再帰を用いて、この数式を表すための新しい型を定義する:

```
data Expr = Val Int
          | Add Expr Expr
          | Mul Expr Expr
```

例えば、前ページの数式は次のように表される:

```
Add (Val 1) (Mul (Val 2) (Val 3))
```

再帰を使うと、数式を処理する関数を容易に定義できる
例えば:

```
size            :: Expr → Int
size (Val n)     = 1
size (Add x y) = size x + size y
size (Mul x y) = size x + size y

eval            :: Expr → Int
eval (Val n)     = n
eval (Add x y) = eval x + eval y
eval (Mul x y) = eval x * eval y
```

Note:

- 3 つの構成子の型:

```
Val :: Int → Expr
Add :: Expr → Expr → Expr
Mul :: Expr → Expr → Expr
```

- 数式を扱う関数の多くは、構成子を別の関数で置き換える畳み込み関数 fold′ で定義できる
例えば:

```
eval = fold' id (+) (*)
```

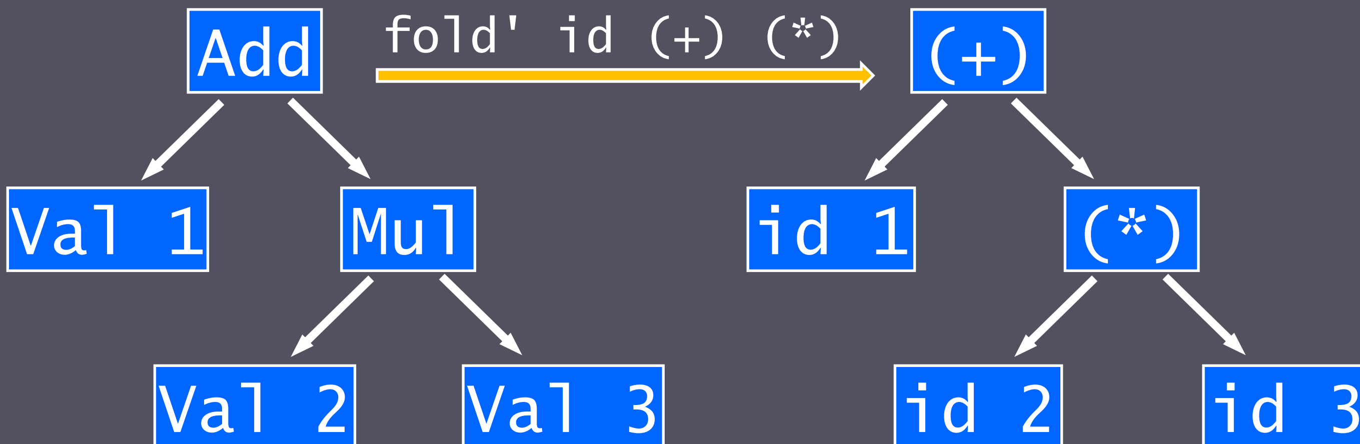
fold' の補足

```
id :: x -> x
id x = x
```

- 構成子 Val を id に、Add を (+) に、Mul を (*) に置き換えれば、自然な形で eval を実現できる

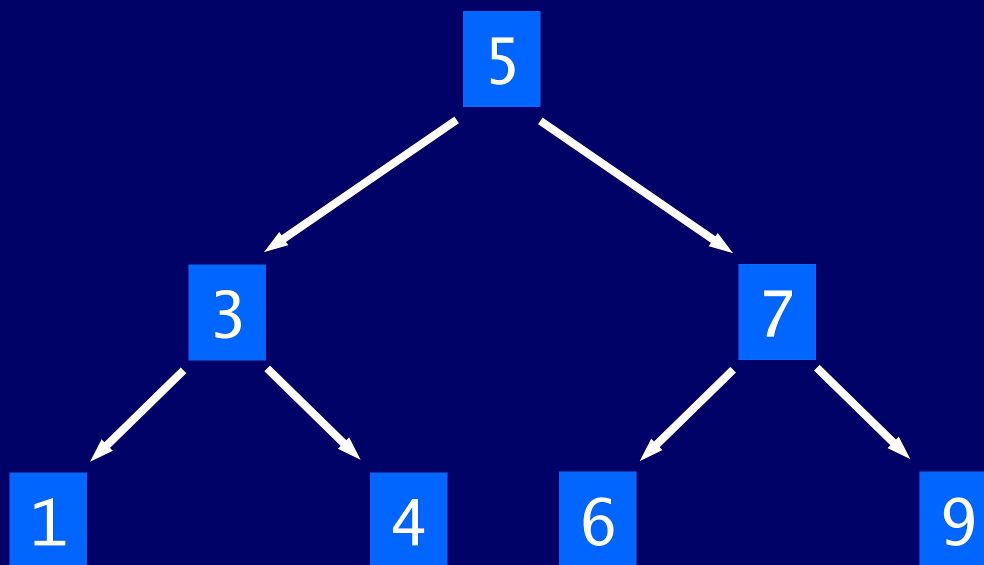
```
fold' v a m (Val x)   = v x
fold' v a m (Add x y) = a (fold' v a m x) (fold' v a m y)
fold' v a m (Mul x y) = m (fold' v a m x) (fold' v a m y)

  fold’ id (+) (*) (Add (Val 1) (Mul (Val 2) (Val 3)))
=                   (+) (id  1) ((*) (id  2) (id  3))
= 7
```



二分木

計算機上で、二分木と呼ばれる 2 つに分岐するデータ構造にデータを保存することが多い



再帰を用いて、二分木を表す新しい型を定義できる:

```
data Tree = Leaf Int
          | Node Tree Int Tree
```

例えば、前ページの木は次のように表される:

```
Node (Node (Leaf 1) 3 (Leaf 4))
     5
     (Node (Leaf 6) 7 (Leaf 9))
```

与えられた整数が二分木の中に存在するか否かを判定する関数:

```
occurs                    :: Int → Tree → Bool
occurs m (Leaf n)          = m==n
occurs m (Node l n r) = m==n
                           || occurs m l
                           || occurs m r
```

ただし、整数が木の中に存在しないとき最悪の場合となり、この関数は木全体を走査する

木に含まれる全ての整数のリストを返す関数 flatten を考える(左(右)部分木中の値は自分より前(後ろ)に):

```
flatten                   :: Tree → [Int]
flatten (Leaf n)          = [n]
flatten (Node l n r) =     flatten l
                        ++ [n]
                        ++ flatten r
```

木を flatten で平らにしたときに、結果が整列している木を検索木(search tree)という。例の木は、flatten すると [1,3,4,5,6,7,9] となるので検索木である。

検索木の重要性は、木の中の値を探すとき、ノードの 2 つの部分木のどちらに値が出現し得るか常に決定できることにある:

```
occurs m (Leaf n)               = m==n
occurs m (Node l n r) | m==n = True
                      | m<n  = occurs m l
                      | m>n  = occurs m r
```

新しい occurs は、木の中の 1 つの(根から葉へ至る)経路だけを辿るので前の定義より効率的である

クラス宣言

- 予約語 class を用いて、新しいクラスを定義できる
  - ある型 a がクラス Eq のインスタンスになるためには、(==) と (/=) が必要
  - (/=) のデフォルト定義は、(==) を用いて (/=) を定義する

```
class Eq a where
  (==), (/=)    :: a -> a -> Bool
  x /= y        = not (x == y)
```

上書き可能

  - (==) を与えて Bool を Eq クラスのインスタンスにする

```
instance Eq Bool where
  False == False     = True
  True  == True      = True
  _     == _         = False
```

インスタンスの自動導出

- データ宣言とは別に，インスタンス宣言を書くのは面倒
- deriving により、データ宣言と同時に，自動的に主要なクラスのインスタンスにする
- 例:

```
data Bool = False | True
  deriving (Eq, Ord, Show, Read)
```

  - Bool は Eq などのインスタンスになる

まとめ (10章)

- 型宣言: 既存の型に別名を与える
  - `type String = [Char]`
- データ宣言: 新しい型とその値を定義 (Answerは3つの値を持つ)
  - `data Answer = Yes | No | Unknown`
    (データ) 構成子
  - 構成子は引数を取れる
    - `data Shape = Circle Float | Rect Float Float`
  - データ宣言も引数を取れる (Maybeは異常を正常値と分離)
    - `data Maybe a = Nothing | Just a`
  - case と関数定義による場合分けと値の抽出

```
case exp of
  Nothing → …
  Just x  → …
```

```
f :: Maybe a → …
f Nothing  = …
f (Just x) = …
```

- 再帰型
  - `data Nat = Zero | Succ Nat`