関数型プログラミング
第2回　数値と関数

萩野　達也
hagino@sfc.keio.ac.jp

Slide URL

https://vu5.sfc.keio.ac.jp/slide/

プログラミング言語の構成要素

- 手続き型プログラミング言語
  - データ構造
    - 基本データ
      - 整数
      - 浮動小数点
      - 文字
    - 複雑なデータ
      - 配列
      - 構造体
      - オブジェクト
  - 制御構造
    - 基本文
      - 代入文
    - 複雑な構造
      - if 文
      - while 文
      - for 文
      - 関数
      - クラス・オブジェクト
- 関数型プログラミング言語
  - 型
    - 基本的な型
      - 整数
      - 浮動小数点
      - 文字
    - 複雑な型
      - リスト
      - 関数
      - データ型
  - 関数
    - 再帰呼び出し
    - 高階関数

Haskellの基本的な値と型

- Haskellで取り扱うことができる基本的な値と型には以下のようなものがあります.
  - 真偽値
    - `Bool`型
  - 数値
    - `Int`型, `Integer`型, `Float`型, `Double`型
  - 文字
    - `Char`型
  - 文字列
    - `String`型 = `[Char]`型
  - タプル
    - `(a,b)`型
  - ユニット
    - `()`型
  - リスト
    - `[a]`型
  - 関数
    - `a -> b`型

真偽値

- **Bool型**

- 真偽値の値（定数）
  - **True**
  - **False**

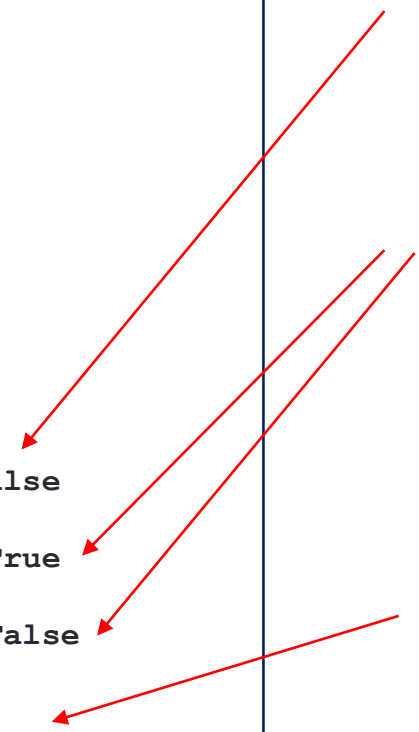
- **Bool型に関する関数と演算**

使用例	意味
`not x`	**x** が **True** なら **False** を返す. **x** が **False** なら **True** を返す.
`x && y`	**x** と **y** の両方が **True** ならば **True** を返す. それ以外のときは **False** を返す.
`x || y`	**x** と **y** のいずれかが **True** ならば **True** を返す. それ以外のときは **False** を返す.

確かめてみよう

- ghciを使って真偽値について演算の優先順位などを確認

```
% stack ghci
...
Prelude> True
True
Prelude> :type False
False :: Bool
Prelude> not True
False
Prelude> True && False
False
Prelude> False || True
True
Prelude> True && True && False
False
Prelude> False && True || True
True
Prelude> False || True && False
False
Prelude> not True || True
True
Prelude> :quit
```



- `&&` も `||` も左結合
  `x && y && z` は
  `(x && y) && z` のこと
- `&&` は `||` より優先される
  `x && y || z` は
  `(x && y) || z` のこと
  `x || y && z` は
  `x || (y && z)` のこと
- `not` は優先される
  `not x || y` は
  `(not x) || y` のこと

左結合と右結合

- 演算子には左結合のものと右結合のものがある
  - どちらでもない（結合を許さない）ものもある
- 左結合
  - 3つ以上を同じ演算子で結合するときに左から右に結合する
  - **x+y+z → (x+y)+z**
  - **x+y+z+u → ((x+y)+z)+u**
- 右結合
  - 3つ以上を同じ演算子で結合するときに右から左に結合する
  - **x+y+z → x+(y+z)**
  - **x+y+z+u → x+(y+(z+u))**
  - 実際には **+** は左結合
- 結合法則が成り立つとは
  - 左結合でも右結合でも同じ値になる
  - **+** は結合法則が成り立つが，**–** は成り立たない

演算子の優先順位

- 複数の演算子を組み合わせた式の場合，優先順位の高い演算子から先に結合する

- `*` の優先順位は `+` より高い
  - `1+2*3` → `1+(2*3)`

- `&&` の優先順位は `||` より高い
  - `False || True && False` → `False ||(True && False)`

関数の定義

```
func  param1  param2  ・・・・ = body
```

- 関数 ***func*** を定義する.
  - ***param1 param2 ・・・・*** は引数
  - ***body*** は関数の本体

```
notnot x = not(not x)
```

- **notnot** を定義
  - 引数 **x** を受け取る
  - 本体は **not(not x)**
  - **x** に **not** に適用した結果を **not** に与える
    - **x** の **not** の **not** を計算

関数への引数の適用

func arg1 arg2 ····

- 関数に引数を適用する:
  - *func arg*
- 引数が2つある場合:
  - *func arg1 arg2*
- 引数が3つの場合:
  - *func arg1 arg2 arg3*
- 括弧は必要ない:
  - *func arg1 arg2* → ((*func arg1*) *arg2*)
  - *func arg1 arg2 args* → (((*func arg1*) *arg2*) *arg3*)

func arg1 arg2 ≠ *func*(*arg1, arg2*)

関数の実行（評価）

- 関数に引数が与えられると，関数の本体を，与えられた引数で評価する

```
notnot x = not(not x)
```

- 実際の実行では **x** を **True** に置き換えて評価するのではなく，
- **x** が **True** であるという束縛環境の下で **not(not x)** を評価する

```
% stack ghci
Prelude> notnot x = not(not x)
Prelude> notnot True
True
```

バージョンによっては
`let notnot x = not(not x)`
としなくてはいけない

if 式を使って not を定義

```
if 条件式 then 式1 else 式2
```

- if式の値
  - 条件式 の値が True ならば 式1 の値になり，False ならば 式2 の値になる

- C言語やjavaのif文とは違う
  - 文ではなく式の一部です
  - 3項演算子の「式1?式2:式3」と同じ

noot.hs

```
noot x = if x then False else True
```

- x が True なら False で，False なら True
  - noot True → False
  - noot False → True

練習問題2－1

- && と || を if を使って定義しなさい.
  - && も || も2つの引数を持つ関数とみなすことができます.
    - aand True True → True
    - aand True False → False
    - aand False False → False
    - aand False False → False
    - oor True True → True
    - oor True False → False
    - oor False False → False
    - oor False False → False

aand.hs

```
aand x y = if x then if y then True else False else False
```

```
aand x y = if x then(if y then True else False)else False
```

oor.hs

```
oor x y = if x then ...
```

演算子と関数

- 2項演算子は2引数の関数
  - `x && y`
  - `x || y`
  - 演算子を ( ) で囲むと通常の関数となる
  - `(&&) :: Bool -> Bool -> Bool`
  - `(||) :: Bool -> Bool -> Bool`

```
% stack ghci
Prelude> True && False
False
Prelude> (&&) True False
False
```

- 2引数の関数は2項演算子
  - `aand::Bool -> Bool -> Bool`
  - `oor::Bool -> Bool -> Bool`
  - 関数を ` ` で囲むと演算子となる
  - `True `aand` False` → `False`
  - `False `oor` True` → `True`

数値

- 整数型
  - `Int`型　　比較的小さな符号付整数
  - `Integer`型　　制限のない整数
- 整数リテラル
  - 10進表記　　`5`, `999`, `12345678901234567890`
  - 8進表記　　`0o644`
  - 16進表記　　`0x1f`
  - リテラルは文脈で `Int` 型か `Integer` 型が決まる
  - `(16::Int)` などで明示的に指定も可能
- 浮動小数点型
  - Float型　　単精度浮動小数点
  - Double型　　倍精度浮動小数点
- 浮動小数点リテラル
  - `1.5`
  - `3.141592`
  - `0.1543e+2`
  - `1343e-3`
  - `Float` か `Double` かは文脈で決まる
  - `(1.5::Double)`

数値の演算

演算子	意味
`x + y`	x と y の和
`x - y`	x と y の差
`x * y`	x と y の積
`x / y`	x と y の商（浮動小数点同士のみ）
`x `div` y`	x と y の整除（整数同士のみ，負の無限大に向かってまるめる）
`x `quot` y`	x と y の整除（整数同士のみ，ゼロに向かってまるめる）
`x `mod` y`	`(x `div` y)` のあまり（整数同士のみ） y の符号と一致
`x `rem` y`	`(x `quot` y)` のあまり（整数同士のみ） x の符号と一致
`-x`	x の符号反転
`negate x`	x の符号反転

演習問題2—2

- **div**, **quot**, **mod**, **rem** の違いを確かめなさい.
  - **x** と **y** が正の整数であれば, **div** と **quote**, **mod** と **rem** は等しい
  - 次の値を ghci で確かめなさい.

式	値
`7 `div` 3`	2
`7 `quot` 3`	2
`(-7) `div` 3`	
`(-7) `quot` 3`	
`7 `div` (-3)`	
`7 `quot` (-3)`	
`(-7) `div` (-3)`	
`(-7) `quot` (-3)`	

式	値
`7 `mod` 3`	1
`7 `rem` 3`	1
`(-7) `mod` 3`	
`(-7) `rem` 3`	
`7 `mod` (-3)`	
`7 `rem` (-3)`	
`(-7) `mod` (-3)`	
`(-7) `rem` (-3)`	

```
y == (x `div` y)* y + (x `mod` y)
y == (x `quot` y)* y + (x `rem` y)
```

数値の比較

演算子	意味
`x == y`	`x` と `y` が等しいときに `True`, そうでないときに `False`
`x /= y`	`x` と `y` が等しくないときに `True`, そうでないときに `False`
`x > y`	`x` が `y` より大きいとき `True`, そうでないときに `False`
`x < y`	`x` が `y` より小さいとき `True`, そうでないときに `False`
`x >= y`	`x` が `y` 以上のとき `True`, そうでないときに `False`
`x <= y`	`x` が `y` 以下のとき `True`, そうでないときに `False`

- 比較の演算子は非結合
  - `x == y == z` はダメ
  - 3つの数が等しいかどうかを調べるには, `x == y && y == z`

練習問題2－3

- 与えられた2つの数値の大きな方を求める関数 **max2** を定義しなさい.

max2.hs

```
max2 x y = if ... then ... else ...
```

```
% stack ghci
Prelude> :load max2.hs
*Main> max2 7 3
7
```

- 与えられた2つの数値の小さな方を求める関数 **min2** を定義しなさい.

min2.hs

```
min2 x y = if ... then ... else ...
```

練習問題2—4

- 与えられた3つの数値の大きな方を求める関数 **max3** を定義しなさい.

max3.hs

```
max3 x y z = if ... then ... else ...
```

```
% stack ghci
Prelude> :load max3.hs
*Main> max3 7 3 9
9
```

- 与えられた3つの数値の小さな方を求める関数 **min3** を定義しなさい.

min3.hs

```
min3 x y z = if ... then ... else ...
```

再帰呼び出しと分割統治

- 再帰呼び出し(recursive call)
  - 関数の本体で自分自身を呼び出す

fact.hs

```
fact n = if n == 0 then 1 else n * fact(n - 1)
```

- 関数型には **while** や **for** のような繰り返しはない
  - 再帰呼び出しで同等のことを行う

- 分割統治
  - 与えられた問題をより小さな問題に分割して解く
  - 十分小さくなるまで分割を繰り返す
  - 分割した問題が元の問題とサイズが違うだけで同じとき，再帰呼び出しで実装できる
    - `fact n` を `fact(n-1)` の分割している
    - `fact 0` になればそのまま解ける

再帰呼び出しの実行（評価）

fact.hs

```
fact n = if n == 0 then 1 else n * fact(n - 1)
```

- 通常の関数と同じように，関数呼び出しを本体で書き換えればよい

```
fact 3 ⇒ if 3==0 then 1 else 3*fact(3-1) ⇒ if False then 1 else 3*fact(3-1)
⇒ 3*fact(3-1) ⇒ 3*fact 2 ⇒ 3*(if 2==0 then 1 else 2*fact(2-1))
⇒ 3*(if False then 1 else 2*fact(2-1)) ⇒ 3*(2*fact(2-1)) ⇒ 3*(2*fact 1)
⇒ 3*(2*(if 1==0 then 1 else 1*fact(1-1)))
⇒ 3*(2*(if False then 1 else 1*fact(1-1))) ⇒ 3*(2*(1*fact(1-1)))
⇒ 3*(2*(1*fact 0)) ⇒ 3*(2*(1*(if 0==0 then 1 else 0*fact(0-1))))
⇒ 3*(2*(1*(if True then 1 else 0*fact(0-1)))) ⇒ 3*(2*(1*1)) ⇒ 6
```

練習問題2−5

- n 個のものから m 個を選ぶ組み合わせの数 ${}_nC_m$ を計算しなさい.

comb.hs

```
comb n m = ...
```

```
% stack ghci
Prelude> :load comb.hs
*Main> comb 5 2
10
```

- 分割統治
  - n 個のものから m 個を選ぶ方法は，n 個のある1つを考えて，それを選ぶかどうかの2つ考えられる
    - 選んだ場合には，残りの $n-1$ 個から $m-1$ 個を選ぶ
    - 選ばなかった場合には，残りの $n-1$ 個から m 個を選ぶことになる
  - ${}_nC_m = {}_{n-1}C_m + {}_{n-1}C_{m-1}$
  - 分割不要
    - n 個のものから 0 個を選ぶ方法は1通り
    - n 個のものから n 個を選ぶ方法も1通り

練習問題2—6

- 正の整数 **x** と **y** に対して **x** の **y** 乗 **beki x y** を計算しなさい.
  - 組み込みのべき乗 **x^y** は利用してはいけません.

beki.hs

```
beki x y = ...
```

```
% stack ghci
Prelude> :load beki.hs
*Main> beki 2 10
1024
```

- 分割統治(1)
  - **x** の **y** 乗は **x** の **y-1** 乗に **x** を掛けたものである
  - **x** の **0** 乗は **1**

- 分割統治(2)「もっと効率よく」
  - **y** が偶数のとき **y** の半分を **y2** とすると, **x** の **y** 乗は **x*x** の **y2** 乗
  - **y** が奇数のときは分割統治(1)と同じようにする

まとめ

- 真偽値
  - Bool型
  - `not`, `&&`, `||`
  - `if` 式
- 数値
  - 整数，浮動小数点
  - 四則演算
- 関数
  - 関数の定義と適用
  - 再帰呼び出し