基本データ型

胡　振江

整数型

-2^31, …, -2, -1, 0, 1, 2, …, 2^31-1 :: **Int**

Some operations:

2+3 ⟶ 5
2*3 ⟶ 6
2^3 ⟶ 8

二項中置演算子

div 7 2 ⟶ 3
mod 7 2 ⟶ 1

二項前置演算子

浮動小数点数（実数）型

1.5, 0.425, 3.14159… :: **Float**

Some operations:

2.5 + 1.5 ⟶ 4.0
3 - 1.2 ⟶ 1.8
1.4142^2 ⟶ 1.99996

1 / 3 ⟶ 0.333333
sin (pi/4) ⟶ 0.707107

結合順位

- 結合順位：
  関数適用　　強い
  ^
  * / div mod
  + -　　弱い
  - 例
    - 3 ^ 4 * 5
    - 3 * 7 + 4.1
    - square 3 * 4

- 結合順序（同じの演算）
  - 左結合： 5-4-3
  - 右結合： 5^4^3
  - 結合性： 5+4+3

二項演算子とセクション

- セクション：括弧でくくられた演算子
  (+) :: Int → Int → Int
  (+) x y = x + y
- 更に拡張： 引数を演算子とともに括弧でくくる
  (x⊕) y = x ⊕ y
  (⊕x) y = y ⊕ x
  (*2):　2倍する　関数
  (1/):　逆数を求める　関数
  (/2):　2分する　関数
  (+1):　つぎの値を得る　関数

練習問題

- つぎの関数はどのような引数に対して Trueを返すか。
  - (==9) . (2+) . (7*)
- 正しいのはどれ？
  - (*) x = (*x)
  - (+) x = (x+)
  - (-) x = (-x)

例題：平方根の計算sqrt

- 仕様

次の仕様を満たすsqrt関数を定義する。

任意整数xに対して
sqrt x >=0 かつ abs((sqrt x)^2-x) <= ε

例題：平方根の計算sqrt (cont)

- Newton法

Y(n)をxの平方根の近似値とすると
y(n+1) = (y(n) + x / y(n)) / 2
によるy(n+1)はy(n)よりもよい近似値である.

例： 2の平方根の計算

y(0)	= 2	improve
y(1) = (2+2/2)/2	= 1.5	
y(2) = (1.5+2/1.5)/2	= 1.4167	
y(3) = (1.4167+2/1.4167)/2	= 1.4142157	
…	終止条件：satisfy	

例題：平方根の計算sqrt (cont)

- メーン関数

sqrt x = until (satisfy x) (improve x) x

abs(y^2-x)<eps

y(0)

y(n) → y(n+1)

ある条件が真になるまで初期値に関数を繰り返して適用する

improve 関数

- 近似値yから新しい近似値を生成する関数

```
improve :: Float → Float → Float
improve x y = (y + x/y) / 2
```

括弧を明示的に表すと

```
improve :: Float → (Float → Float)
(improve x) y = (y + x/y) / 2
```

になる.

satisfy 関数

- 終了条件を判定する関数

```
satisfy x y = abs (y^2 – x) < eps
```

Q: satisfy関数の型は？

until 関数

- ある条件pが真になるまで初期値に関するfを繰り返し適用する関数

```
until p f x | p x = x
            | otherwise = until p f (f x)
```

高階関数：関数引数を取る関数

Q：untilの型は？

プログラム sqrt.hs

```
sqrt1 x = until (satisfy x) (improve x) x
  where
    improve x y = (y + x/y) / 2
    satisfy x y = abs (y^2 – x) < eps
    eps = 0.0001
```

単純な関数の組み合わせ ➔ 修正しやすくなる
（教科書 p.24, 練習問題 2.1.7）

Newton法の一般的な解法

- f(x) = 0の根を求める手法

yが関数fの根の近似値であるならば,

$$y - f(y) / f'(y)$$

がよりよい根の近似値である.

```
newton f  x = until satis improve x
   where satis y = abs(f y) < eps
            improve y = y – (f y / deriv f y)
            deriv f x = (f(x+dx) – f x) / dx
            dx = 0.00001
            eps = 0.0001

sqrt x = newton f x
    where f y = y^2-x
```

論理型 Bool

True, False :: **Bool**

述語：論理値を返す関数

- E.g. even :: Int → Bool
- 比較演算子
  - == 等しい 1==1
  - /= 等しくない True /= False
  - < より小さい
  - > より大きい
  - <= より小さいかまたは等しい
  - >= より大きいかまたは等しい
- 論理演算子
  - && 論理積
  - || 論理和
  - not 論理否定

例題：閏年の判定

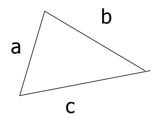
- 閏年とは、4で割り切れる年であるが、100で割り切れるならば400でも割り切れなくてはならない。

```
leap y = y `mod` 4 == 0    &&
         (y `mod` 100 /= 0 || y `mod` 400 == 0)
```

または

```
leap y | y `mod` 100 == 0   = y `mod` 400 ==  0
       | otherwise          = y `mod` 4 ==  0
```

その他の例題



a<=b<=c

```
analysis a b c
 | a+b<=c                    = 0  -- 三角形ではない
 | a==b && b==c              = 1  -- 正三角形
 | a/=c && (a==b || b==c)    = 2  -- 二等辺三角形
 | a<b && b<c                = 3  -- 一般三角形
```

文字型

'a' '7' ' ' :: **Char**

- 基本関数
  - ord :: Char → Int
  - chr :: Int → Char

例: ord 'b' ➔ 98
chr 98 ➔ 'b'
chr (ord 'b' + 1) ➔ 'c'

例題

- 文字が数字であることを判定する関数
  isDigit x = '0' <= x && x <= '9'

- 小文字を大文字に変える関数

```
capitalise x
  | isLower x   = chr (offset + ord x)
  | otherwise   = x
where offset = ord 'A' – ord 'a'
```

文字列型

"a" "hello" : : **String**

- 文字列の比較は通常の辞書式順に従う
  "hello" > "hallo"
  "Jo" < "Joanna"
- 関数
  - show :: a → String
    show 100 ➔ "100"
    show True ➔ "True"
    show (show 100) ➔ ""100""
  - 文字列をつなぐ連接演算子 ++
    "hello" ++ " " ++ "world" ➔ "hello world"

組型

- (T1,T2,…,Tn)
  - (17.3,'+') :: (Float,Char)
  - (3,6) :: (Int,Int)
- 順序：辞書式順序
  - ("s",4) < ("s",5)
- 関数
  - fst (x,y) = x
  - snd (x,y) = y

例題1：2次方程式

- 2次方程式の根を求める関数

a x^2 + b x + c = 0 (a,b,c)

⇩ ↓

let d = b^2 – 4ac. (r1,r2)
If d >=0 then
r1 = (-b + sqrt d) / 2a
r2 = (-b – sqrt d) / 2a

```
roots :: (Float,Float,Float) → (Float,Float)
roots (a,b,c) | d>=0 = (r1,r2)
  where
    r1 = (-b+r) / (2*a)
    r2 = (-b-r) / (2*a)
    r  = sqrt d
    d  = b^2 - 4*a*c
```

例題2：有理数

- 有理数の表現：対
  x/y ➔ (x,y)
- 問題：
  - 有理数の正規化　(18,16) → (9,8)
  - 有理数の四則演算
  - 有理数の比較
  - 有理数の表示

有理数の正規化

$$\frac{x}{y} = sign(x*y)\frac{|x| / \gcd(|x|,|y|)}{|y| / \gcd(|x|,|y|)}$$

```
norm (x,y)
     | y /= 0  =  (s * (u `div` d), v `div` d)
  where u = abs x
        v = abs y
        d = gcd u v
        s = sign (x*y)

    sign x | x>0  = 1
           | x==0 = 0
           | x<0  = -1
```

$$\frac{x}{y} = sign(x*y)\frac{|x| / \gcd(|x|,|y|)}{|y| / \gcd(|x|,|y|)}$$

有理数上の四則演算

```
radd (x,y) (u,v) = norm (x*v+u*y,y*v)
rsub (x,y) (u,v) = norm (x*v-u*y,y*v)
rmul (x,y) (u,v) = norm (x*u,y*v)
rdiv (x,y) (u,v) = norm (x*v,y*u)
```

有理数の比較

```
compare' op (x,y) (u,v) = op (x*v) (y*u)

requals  = compare' (==)
rless    = compare' (<)
rgreater = compare' (>)
```

有理数の表示

```
showrat (x,y)
   = if v==1 then show u
       else show u ++ "/" ++ show v
  where (u,v) = norm (x,y)
```

パターン

- 等式の左側にパターンを用いて関数を定義することができる。
  - 論理値パターン

```
cond True x y = x
cond False x y = y

cond p x y | p == True   = x
           | p == False  = y
```

- 自然数（負でない整数）パターン

```
pred 0 = 0
pred (n+1) = n

count 0 = 0
count 1 = 1
count (n+2) = 2
```

関数

- 関数はあらゆる型の値を引数にとりうるし、あらゆる種類の値を結果として返すことができる。
  - 例：高階関数
    - 引数として関数をとる、あるいは
    - 結果として関数を返す

    微分演算子： 引数 – 関数

    結果 – 導関数

    ⇨ 関数の性質

関数合成

- 二つの関数を合成する演算子 .

  (.) :: (b→c)→(a→b)→(a→c)

  (f . g) x = f (g x)

- 結合的

  (f . g) . h = f . (g . h)

演算子と関数

- 二項演算子は関数とよく似ている。異なる点は2つの引数の前に置くのではなく間に書くということだけある。

  ⊗ ⊕ ♣ ♦ ♥ ♠

  - セクション： 演算子 ➔ 関数

    2 + 3 ➔ (+) 2 3 ➔ (2+) 3 ➔ (+3) 2

  - バッククオート： 二引数関数 ➔ 演算子

    div 5 3 ➔ 5 \`div\` 3

逆関数

- 単射関数
  - ∀x,y :: A. f x = f y ➔ x=y
- 全射関数
  - ∀y in B, ∃x in A. f x = y
- 逆関数
  - f^{-1}(f x) = x
  - 例 f x = (sign x, abs x)

    f^{-1}(s,a) = s*a

正格関数と非正格関数

- 正格関数
  - 定義： f ⊥ = ⊥
  - 例： square (1/0) = ⊥
- 非正格関数：正格でない関数
  - 例： three x = 3

    > three (1/0)

    3

非正格な意味論の利点

- 相等性に関する議論しやすい

  2 + three x = 5

  →単純で統一的な置換操作

  →プログラムの正当性を議論しやすい
- 関数を定義して、新たら制御構造を定義することができる。

```
cond p x y | p          = x
           | otherwise = y
recip x = cond (x==0) 0 (1/x)
```

正格な意味論では recip 0 = ⊥

非正格な意味論では recip 0 = 0

簡約戦略

f x1 x2 … xnの評価

- 先行評価
  - 引数優先評価戦略

    X1,x2,…,xnを評価したらfを評価する。
- 遅延評価
  - (外側の)関数優先評価戦略

    fをまず評価する。

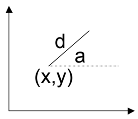
型の同義名

- 距離、角度、位置を引数にとり、角度と距離で示される新しい位置に場所を移動する関数 move：

```
move :: Float→Float→(Float,Float)→ (Float,Float)
move d a (x,y) = (x+d*cos a,y+d*sin a)

type Position = (Float,Float)
type Angle = Float
type Distance = Float

move :: Distance→Angle→Position→Position
```



型推論

- 適用規則

$$\frac{f\ x :: t}{\exists t'.\ x :: t',\ f :: t' \to t}$$

- 相等性規則

$$\frac{x::t,\ x::t'}{t=t'}$$

- 関数の規則

$$\frac{t \to u = t' \to u'}{t=t',\ u=u'}$$

(.) f g x = f (g x)

引数名と結果に型を割り当てる

f :: t1

g :: t2

x :: t3

f (g x) :: t4

(.) ::t1→t2→t3→t4

適用規則

g x :: t5

f :: t5→t4

適用規則

x :: t6

g :: t6→t5

相等性規則

t1 = t5→t4

t2 = t6→t5

t3 = t6

(.) :: (t5→t4)→(t6→t5)→t6→t4

f x y = fst x + fst y

仮定

fst :: (a,b)→a
(+) :: Int→Int→Int

x :: t1
y :: t2
fst1 x + fst2 y :: t3
f :: t1→t2→t3

fst2 y :: t4
(+) (fst1 x) :: t4→t3

y :: t5
fst2 :: t5→t3

fst1 x :: t6
(+) :: t6→t4→t3

x :: t7
fst1 :: t7→t6

相等性規則、関数規則 ...

f :: (Int,u1) → (Int,u2) → Int

fix f = f (fix f)

f :: t1
f (fix f) :: t2
fix :: t1→t2

fix f :: t3
f :: t3→ t2

f :: t4
fix :: t4→t3

t1 = t3→t2 = t4
t1 = t4
t2 = t3

fix :: (t2→t2)→t2

selfapply f = f f

f :: t1
f f : t2
Selfapply :: t1→ t2

f :: t3
f :: t3→t2

t3 = t3 → t2

上の等式はt3に対する解をもたないので、型エラー