# プログラミング言語論 #03

## Functions in Haskell

2018-04-23

前回 Haskell における変数の定義と宣言の方法を見た．

これを基に，関数の定義，関数の型宣言，関数の利用を学ぶ．
また，仮引数と実引数の違いを理解する．

# 簡単な関数

関数とは，引数を取り，値を返すもの（返値）

## 定義

関数 f という定義例

```
f a = a + 9
```

- 関数名[1]
- 仮引数名
- =
- 右辺（計算式が一つだけ書ける）

## 宣言

関数 f に対する宣言

```
f :: Int -> Int
```

- 関数名
- ::
- 仮引数の型[2]
- ->
- 返値の型

## 実行

```
1 f 1
```

~~宣言を書いてから次の行に定義を書く~~

[1] 小文字で始まる；大文字で始めてよいのは型名のみ

[2] 型名は大文字で始まる

# 複数の引数を取る関数（一般の関数）

関数とは，引数を（複数）取り，返値を 1 つ返すもの

## 定義

```
f2 a bb = (a + 9) < bb
```

- 関数名
- 仮引数名 1
- 仮引数名 2
- ...
- =
- 右辺（計算式が一つ）

->の個数と引数の個数は常に一致

## 宣言

```
f2 :: Int -> Int -> Bool
```

- 関数名
- ::
- 仮引数 1 の型 ->
- 仮引数 2 の型 ->
- ...
- 返値の型

**注意**：関数 f2 の型は Int -> Int -> Bool. Bool は関数の型ではなく返値の型（2 つは別物）.

実行例は後述

# 型の等価性，宣言の単一性

## 定義を見れば宣言が決まる

**ある関数定義**

```
f2 a bb = (a + 9) < bb
```

がエラーでないなら

**バリエーション**

```
f3 a bb = (div a 2) < bb
f4 a bb = a > (2 * bb)
```

もエラーでない（部分式の同型変換）

(1) 型エラーがどこにもなく，(2) 以下になることがわかる：

### f2 に関する型制約の全列挙

- a は (+) で使われているので Int でなければならない
- (+) の結果は Int になる
- (<) の両辺は同じ型でなければならない
- bb は (<) で使われているので Int でなければならない
- (<) の計算結果は Bool になる
- (=) で結ばれているので f2 の返値の型は右辺の型と同じでなければならない

以上を連立？させて解くと，

**論理的帰結としての型宣言**

```
f2 :: Int -> Int -> Bool
```

# 定義から関数の型が（一意に）決まる

## 第 2 回で説明した型しか存在しないものとする

```
g1 x     = x == "10"
g2 x y   = x ++ y
g3 x y   = (x ++ y) == y
g4 x y   = x && (y < 10)
g5 x y z = (x && y) <= z
```

- 全然わからない！ → 第 2 回の型，関数，演算子全て暗記すること
- 半分くらいわかる → これは本当にイントロ．半分行ったあたりで半数が挫折，，，

# 関数の実行（呼出し）または**関数適用**

## Haskell 言語の場合

- 関数名および引数を全てスペースで区切る
- 括弧およびコンマは不要（構文エラーの誘因）

**インタプリタでの関数適用例**

```
1  f 3                          -- 括弧もカンマも不要 f :: Int -> Int
2  f2 5 8                       -- f2は2引数関数だが括弧もカンマも不要
3  f (-1)                  -- この括弧は引数だからではなく，負数だから必要
4  f (1 + 4)                    -- 順番の指示 ←→ f 1 + 4 との違い
5  f2 (1 + 2 * 3) (f 4)    -- 各括弧対はどこまでが1つの引数かを指示
```

最後の例では f2 の第 2 引数として別の関数適用が使われている．下式にほぼ匹敵する．[3]

**ワンステップ化する前のプログラム（意味は変わらないはず）**

```
1  tmp = f 4                    -- 1引数関数fの関数適用を先にやってから
2  f2 (1 + 2 * 3) tmp           -- その結果を使ってf2の関数適用
```

[3] 純関数型言語ならではの評価戦略があるので本当は違うのだがこの件に深く立ち入らないことにして式評価の一般論としてこう説明する．

# プログラミング言語論専門用語

3

## parameter, 仮引数，パラメータ

関数を定義する時に引数にとりあえず付ける名前

## argument, actual argument, 実引数

関数適用（＝実行時）において引数の位置に置かれるもの

- f 3 : 値 3 が実引数，そして 3 が関数 f に渡される
- f x : 変数 x が実引数，そして **x の値**が関数 f に渡される
- f (3 + 4) : (3 + 4) が実引数，そして**計算結果の 7** が関数 f に渡される
- f (h1 a < h2 5 b) の場合も全部計算した**最終結果**が関数 f に渡される (そのためには h1, h2 の関数適用もまず必要)

このように「引数で書いたもの」と「実際に渡されるもの」は同一ではない．そして仮引数名と実引数名は一致させる必要は全くない． ただし，言語によっては全く違う機構を使うものもある．これは値呼びという機構の説明． C 言語が採用しているので用いた． Haskell は call-by-need.

# 余談：C 言語における表面的理解 2 例

- 関数 f を定義せよ
- 関数 f を関数 main の中で使え（呼出せ）

**関数定義と関数適用の混乱**

```
1  int main (void) {
2
3    int f (int x)   // これしか思いつかない
4    { ...
```

**実引数と仮引数の混乱**

```
1  int f (int x) {   // ★だから仮引数名をxにしなければいけない
2    ...
3  int main (void){  // 以下に合わせて関数fを定義せよ
4    int a, x, y, z;
5    a = f(x);       // これが要求された★
6    a = f(y);       // 絶句
7    a = f(f(z)+1);  // パニック => int f (int f(z) + 1) {
```