# ソフトウェア技法: No.2 (関数と様々なデータ型)

亀山幸義 (筑波大学情報科学類)

## 1 複数の引数をもつ関数

これまでは、関数は 1 引数関数 (unary function) のみを扱ってきた。これでは実際には不便なので、2 引数以上の関数を定義してみよう。

```
(* 複数の引数を持つように見える ()関数 *)
let f x y z = x * y + z ;;
let r1 = f (1+2) (3*4) (5−6) ;;
let r2 = f 1 2 (f 3 4 5) ;;
let r3 = f r1 r2 r2 ;;
```

これから少しややこしい話をする。実は、OCaml では「複数の引数を持つ関数」というものはなく、上記のものも「1 引数関数」を何度も使って複数の引数を持つ関数であるように見せていてただけである。その証拠に、上の関数 f の型は、`(int * int * int) -> int` ではなく、`int -> (int -> (int -> int))` という全く違う型である、と表示されていることだろう。`int -> X` という型は、引数が 1 つだけで、int 型のものをもらって X 型を返す関数の型であるので、これは、上記の f が 1 引数関数であることを表している。この話題は当面はあまり問題ないので、深いりせず、後で型の話をするときにもう 1 度もどることにする。

次に関数の部分適用 (partial application) の話をしよう。これは、2 個以上の引数 (を取るように見える) 関数に、期待する個数の引数より少ない数の引数を渡すという機能である。

```
(* 部分適用: 引数が足りなくてもエラーにならない *)
let r4 = f 10 20 ;;

(* 部分適用をした関数に、追加で引数を渡すとちゃんと計算してくれる *)
let r5 = r4 30 ;;
let r6 = r4 40 ;;

(* 部分適用; 3引数関数に1つだけ実引数を渡す *)
let r7 = f 10 ;;
let r8 = r7 20 30 ;;
let r9 = r7 40 ;;
let r10 = r9 50 ;;
```

さて、足し算 `a + b` などは、関数でありながら、引数の間 (infix) に関数をあらわす記号がきている。このような関数を、前置 (prefix)、つまり、引数の前に書くための方法を述べる。

```
(* ところで + などの演算にかっこをつけると前置記法になる *)
let r11 = (+) 1 2 ;;
let r12 = (+) 5 8 ;;

(* (+) も部分適用ができる *)
let r13 = (+) 5 ;;
let r14 = r13 8 ;;

(* ほかの演算子も同様 *)
let r15 = (-) 5 3 ;;
let r16 = (-) 5 ;;
let r17 = r16 3 ;;

(* ただし、* は コメント記号と間違えないよう、スペースがいる *)
let r18 = ( * ) 5 3 ;;
```

さらに、新しく infix の関数を定義することもできる。

```
let (+%) x y = x + y * 100 ;;
10 +% 20 ;;    (* +% を infix 関数として使える *)
```

## 2 再帰関数

いよいよ再帰関数 (recursive function) である。この授業は再帰関数を学習するためにあるといっても過言ではないかもしれない[*1]。

```
(* 自然数の総和 *)
let rec f x =
   if x = 0 then 0
   else x + f (x - 1)
in
   (f 10) + (f 20) ;;

(* フィボナッチ関数 *)
let rec fib x =
   if x <= 2 then 1
   else (fib (x - 2)) + (fib (x - 1))
in
   fib 10 ;;
```

[*1] 「さすがに、それは過言である」という指摘が昨年の学生らあった。

```
(* 最大公約数を求めるユークッドの互除法の実装のつ 1 *)
let rec gcd x y =
   if x = 0 then y
   else if y = 0 then x
   else if x > y then gcd (x−y) y
   else gcd (y−x) x ;;

let r19a = gcd 10 3 ;;
let r19b = gcd 384 93 ;;
```

## 3 いろいろな型と演算: bool, int, float, char, string

OCaml を起動したときに (特にライブラリを読みこんでいなくても)、定義されている型や演算がある。これらは Pervasives モジュールにはいっている。Pervasives の詳細は、以下の URL に掲載されている。幸い英語で書かれている[*2]。

http://caml.inria.fr/pub/docs/manual-ocaml/libref/Pervasives.html

```
let r20 = (10 > 20) && (30 <> 40) || (50 < 60) ;;

(* 整数の最大値 *)
let r21 = max_int ;;

(* 整数の最小値 *)
let r22 = min_int ;;
```

```
(* 負の数 *)
let r23 = (−30) ;;

(* 整数の割り算と余りの演算 *)
let r24 = (10 / 3) * 100000 + (10 mod 3) ;;

(* 浮動小数点型の最大値 *)
let r30 = max_float ;;

(* 浮動小数点型の演算は、整数型と違い、ピリオドがつく *)
let r31 = 3.14 −. (2.78 *. 4.56) /. 2.3 ;;

(* 型の間の変換 *)
let r32 = truncate (sin (float 13) *. 100.0 ) ;;
```

[*2] 何が「幸い」だとおもう人もいるだろうが、初期の OCaml のマニュアルはフランス語で書かれていたことを思うと多くの人にとって大変幸いであろう。

```
(* 文字型 *)
let r33 = 'z' ;;
(* 整数へ変換 *)
let r34 = Char.code r33 ;;
(* 整数からもどす *)
let r35 = Char.chr (r34 + 3) ;;
```

```
(* 文字列の連結 *)
let r40 = "abc" ^ "defgh" ^ "" ;;

(* 文字列へ *)
let r41 = (string_of_int 135) ^ (string_of_bool (10 <> 20)) ;;
let r42 = string_of_float (sqrt 2.0) ;;
```

```
(* String モジュールの関数を使うといろいろできる *)
let r44 = String.length r41 ;;
(* 文字列の比較 *)
let r46 = String.compare r41 r42 ;;
(* 文字列のn文字目を取り出す *)
let r47 = String.get r41 3 ;;
(* 上記と同じことを簡単に書く *)
let r48 = r41.[3] ;;
(* 文字列を作る *)
let r49 = String.make 10 'x' ;;
```

**2 章の練習問題.**

- 正の整数 $n$ が与えられたとき、整数の範囲での $n$ の平方根 ($m^2 \leq n$ となる整数 $m$ の最大値) を計算する関数 isqrt を書きなさい。(ヒント: float 型に対する平方根を計算する関数は sqrt である。)

考え方: sqrt を使う。再帰は (この段階では) 知らないものとする。

(1) OCaml では、float 型を引数にとる関数である sqrt を、int 型の要素 (整数) に適用することはできない。そこで、整数 $n$ を float 側に変換して、sqrt を適用して、それを int 型に戻す、という方法が考えられる。

```
let isqrt n =
  truncate (sqrt (float n))

(* 関数適用が何重にも重なると、かっこが多くなる。そこで、@@ という便利なものがある.
   @@ の機能は、関数適用と同じだが、結合力（優先順位）が違うので、少ない数の括弧で
   済むことがある。*)
let isqrt2 n =
  truncate @@ sqrt @@ float n ;;
```

(2) ところで、上記の実装は、ちょっとだけ不安がある。なぜなら、整数を浮動小数点型に直したときに、ほんのわずかに誤差が出るかもしれないからである。たとえば、(1.0 /. 3.0 ) *. 3.0 は必ず 1.0 であろうか？誤差が出るとしたら、(truncate は

「切り捨て」の関数なので) 答えが「1」だけ違ってしまう可能性がある。(本来 N が答えなのに、N-1 が答えになってしまう可能性がある。) このあたりは浮動小数点型がどういう仕様なのかによるので調べればわかることであるが、ここでは、そういう事を調べずに「プログラミング力」だけでカバーすることを考える。

すなわち、上記の答えを「検算」する関数を作ろう。このための関数は、以下のように書ける。

```
(* m^2 <= n であるかどうか、真理値で返す *)
let le_sqrt m n =
   m * m <= n

(* m = isqrt n であるかどうか、真理値で返す *)
let is_isqrt m n =
   (le_sqrt m n) && not (le_sqrt (m+1) n)
```

後者の関数は、「$m^2 \leq n$ かつ $m^2 > n$」を計算するので、この結果が true であるとき、$m =$`isqrt` $n$ となっていることがわかる。つまり、この `is_isqrt` 関数を使えば、`isqrt` の検算することができる。

もとの例題では、「誤差」が出るとしてもせいぜい 1 なので、求めた答えを N とすると、N-1,N,N+1 あたりを検算すれば、どれが正解かが確実にわかるはずである。

**2 章の演習問題.**

- 例題を検算関数と組み合わせて、「浮動小数点への変換において誤差があっても、必ず正しい isqrt の値を返す関数」を作りなさい。ただし、誤差の影響は、せいぜい「1」であることを仮定してよい。
  ここで「正しい」とは、上記の検算関数にかけると「true」が返ってくるものである。
- 自分が使っている OCaml 処理系の整数型が、おおよそ何ビットであらわされているかを調べる関数を作りなさい。すなわち、`max_int - min_int`$\leq 2^n$ となる $n$ の最小値を求めなさい。ただし、$30 \leq n \leq 35$ または $60 \leq n \leq 65$ であることは既知としてよい。また、`**` は、float 型のべき乗関数であり、使ってもよい。
  [注意. OCaml では、整数型の演算で、桁をあふれたときもエラーもなにもでないので、プログラマが自分で注意する必要がある。本問では、`max_int - min_int` を計算してはいけない。]
- (発展課題) 前問と同じことを、浮動小数点型に対してやってみよ。ただし、浮動小数点の場合は、前問と違う点がいくつかあるので、自分でいろいろ調べる必要がある。
  [ヒント] `min_float` は浮動小数点の最小値ではなく、「0 より大きい最小値」であった。したがって、0 に非常に近い数である。
  また、浮動小数点の場合は、内部的には $a \times 2^n$ の形の浮動小数点数を $a$ と $n$ の 2 つの値で表しているので、前問と同様な手法だけでは解くことはできない。仮に $a$ と $n$ で 4 ビットずつ使うとしたら、(正負の符号で 1 ビットつかうことを考えると、4 ビットでは -7 から 8 までの数を表すことがができるので最大値は 8 であり)、$a \times 2^n$ の最大値は $8 \times 2^8 = 2048$ となる。これをそのまま $log_2$ をとってしまうと、11 となり、11 ビット使っているように思ってしまうが、実際には、たった 8 ビットでこの範囲の数をあらわしているのである。つまり、「最大値の $log_2$ を取る」という計算では、うまくいかない。
  しかし、内部的に $a \times 2^n$ で表していることを知れば、$n$ の方のおおよその値は、$log_2$ を取ることでわかる。つまり、最大値を $M$ としたとき、$log_2 M$ は $n \cdot log_2 a$ となるので、(もし $log_2 a$ の分を無視できれば) $log_2(log_2 M)$ は $n$ を表現するために必要な bit 数の概算を与えるものとなる。この問題はこれを答えればよいことにする。
  この話の詳細は、もはや OCaml の話ではない。数値計算をする人にとっては大事な話になるが、OCaml の float は「IEEE 標準の double」と同じなので、こちらを調べるとよい。(IEEE 754 標準とも呼ばれる。)