# 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)

[2017/7/12 誤植修正] 最後の演習問題で、$n \cdot log_2 a$ となっていたのは、$n + log_2 a$ の誤りなので修正しました。

## 1 複数の引数をもつ関数

引数が 2 個以上ある関数についてもう少し考察しよう。

```
(* 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 ;;
```

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るだけである。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))` と表示される。`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。

ただし、この `f` が返すものの型は、`int -> (int -> int)` であるので、返すものが (`f` とは別の) 関数である。この型の関数にさらに整数型のデータを渡すと、関数が返ってきて、さらにそれに整数型のデータを渡すと、最終的に整数が返ってくる。つまり、もともとの `f` は「3 つの整数を同時にもらって整数を返す関数」ではなく、「3 つの整数を順番に 1 つずつもらって、最終的に整数を返す関数」であった。

この話は、関数型言語の理論的基礎であるラムダ計算において「カリー化、非カリー化」という名称で出てくる話であり、本授業でも後で型の話をするときに詳細に説明するが、ここではあまり深いりせず、現実的に、上記の「1 つずつもらう」機能を使ってみることにしよう。

関数の部分適用 (partial application) とは、2 個以上の引数を取る (ように見える) 関数に対して、少ない個数の引数を渡すという機能である。

```
(* 部分適用: 引数の個数が不足し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 ;;
```

このように、OCaml では「3 引数 (に見える) 関数」に、引数を 1 個や 2 個だけ渡して結果を得て、その結果 (これも関数である) に残りの引数を渡して、最終結果を得るということが可能であり、これを部分適用という。現時点では、部分適用の機能を積極的に使いたい例はないが、うっかり引数の個数が少なすぎたときにエラーにならなくて首をかしげることがあるとおもうので、ここで説明した。

## 2 中置と前置

足し算の式 `a + b` において + という記号は、関数でありながら、引数の間に関数をあらわす記号がきている。これは中置 (infix) 記法という。このような関数を、普通の関数のように、前置 (prefix)、つまり、引数の前に書くための方法を述べる。

```
(* + などの中置演算子に括弧をつけると前置記法になる *)
let r11 = (+) 1 2 ;;
let r12 = (+) 5 8 ;;

(* (+) も普通の関数と同様に部分適用ができる *)
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、* はそのまま書くとコメント記号になってしまうので、* のまわ
   りにスペースをいれる必要がある *)
let r18 = ( * ) 5 3 ;;
```

逆に、新しく中置の関数を定義する場合、括弧をつけて前置の関数として定義すればよい。

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

なお、OCaml では中置にできる関数の名前は **+%** などの特殊記号の列だけであり、`foo` などの名前を中置にすることはできない。

```
(* これはエラー *)
let (foo) x y = 10 ;;
```

## 3 少しだけ再帰関数

再帰関数 (recursive function) は、この授業の中心テーマであるが、詳細は次回に譲ることにして、ここでは定義のしかただけを見ておこう。

```
(* 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 ;;
```

```
(* 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 ;;
```

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

OCaml を起動したときに、(ライブラリを新たに読みこんでいなくても)、もともと定義されている型や演算がある。これらは Pervasives と呼ばれる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[*1]。

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 ;;
```

[*1] 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大変幸いである。

```
(* 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) ;;

(* 浮動小数点数の最大値 *)
let r30 = max_float ;;

(* 浮動小数点数の演算は、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 ) ;;
```

```
(* 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、整数を浮動小数点数に直したときに、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数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 $(m+1)^2 > n$」の真理値を計算するので、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、浮動小数点数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、桁があふれたときはエラーが出ないので、プログラマが自分で注意する必要がある。たとえば、`max_int - min_int` を計算するとオーバーフローして意味のない値が返ってくる。
- (発展課題) 前問と同じことを、浮動小数点数に対してやってみよ。
  ただし、浮動小数点数の場合は、内部的には $a \times 2^n$ の形で表現されているので、前問のような単純な方法で、a と n に何ビットずつが使われているかを正確に推定するのは難しい。また、`min_float` は浮動小数点数の最小値 (マイナスの大きな値) ではなく、「0 より大きい最小値」である。したがって、0 に非常に近い正の数である。

内部的に $a \times 2^n$ で表していることがわかれば、$n$ 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 $n + log_2 a$ ([2017/7/12 誤植修正] もとは $n \cdot log_2 a$ となっていました) 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。
では、$a$ の方のおおよその値はどうやったらわかるだろうか？
参考: OCaml の float は IEEE 標準の double と同じであるので、自分が推定した結果が正しいかどうかは、その標準を見ればよい。(IEEE 754 標準とも呼ばれる。)