関数型言語（OCaml 演習）(7)

関数の一般化と map

浅井 健一

お茶の水女子大学

よく似たリスト上の再帰関数

```
let add1 n = n + 1
let square n = n * n

(* 目的:リストの各要素に 1 を加えたリストを返す *)
let rec g1 lst = match lst with
    [] -> []
  | first :: rest -> add1 first :: g1 rest

(* 目的:リストの各要素を 2 乗したリストを返す *)
let rec g2 lst = match lst with
    [] -> []
  | first :: rest -> square first :: g2 rest
```

関数の一般化

似たような関数を複数、書くのは避けるべき

変更し忘れなど引き起こし、保守性が低下する。

似たような関数があったら一般化することを考える。

1. 異なる部分を特定する。
2. 異なる部分を引数として受け取る
   一般化した関数を定義する。
3. 元の関数を一般化した関数で定義し直す。
4. 元の関数のテストが通ることを確認する。

異なる部分

```
let add1 n = n + 1
let square n = n * n

(* 目的:リストの各要素に 1 を加えたリストを返す *)
let rec g1 lst = match lst with
    [] -> []
  | first :: rest -> add1 first :: g1 rest

(* 目的:リストの各要素を 2 乗したリストを返す *)
let rec g2 lst = match lst with
    [] -> []
  | first :: rest -> square first :: g2 rest
```

異なる部分

```
let add1 n = n + 1
let square n = n * n

(* 目的:リストの各要素に 1 を加えたリストを返す *)
let rec g1 lst = match lst with
    [] -> []
  | first :: rest -> add1 first :: g1 rest

(* 目的:リストの各要素を 2 乗したリストを返す *)
let rec g2 lst = match lst with
    [] -> []
  | first :: rest -> square first :: g2 rest
```

異なる部分をパラメータ化

```
(* 目的:リストの各要素に f を施したリストを返す *)
let rec map f lst = match lst with
    [] -> []
  | first :: rest -> f first :: map f rest
```

- map は g1 と g2 を一般化した関数。
- f は first に施す処理を表す関数。

map は引数として「関数」を受け取っている。

高階関数 (higher-order function)

関数を引数として受け取るような関数

元の関数を map を使って再定義

```
(* 目的:リストの各要素に f を施したリストを返す *)
let rec map f lst = match lst with
    [] -> []
  | first :: rest -> f first :: map f rest

(* 目的:リストの各要素に 1 を加えたリストを返す *)
let g1 lst = map add1 lst

(* 目的:リストの各要素を 2 乗したリストを返す *)
let g2 lst = map square lst
```

関数は first-class の値である

関数型言語では、関数を普通の値として他の関数に渡したり関数の結果として返したりすることができる。

局所関数定義

```
(* 目的:リストの各要素に 1 を加えたリストを返す *)
let g1 lst =
  let add1 n = n + 1 in
  map add1 lst

(* 目的:リストの各要素を 2 乗したリストを返す *)
let g2 lst =
  let square n = n * n in
  map square lst
```

let 関数 引数 ... = 式$_1$ in 式$_2$

引数を受け取ったら式$_1$を実行するような関数を一時的に定義する。次に式$_2$を実行する。その実行の間(のみ)、ここで定義した関数を使うことができる。

map の活用

```
(* 目的:lst の各要素を x との関係文字列にする *)
(* hikaku : int -> int list -> string list *)
let hikaku x lst = []
```

整数 x と整数のリストを受け取ったら、リスト中の各要素について x より小さいものは "小"、等しいものは "等"、大きいものは "大" にした文字列のリストを返す関数 hikaku を定義せよ。

map の活用

```
(* 目的:lst の各要素を x との関係文字列にする *)
(* hikaku : int -> int list -> string list *)
let hikaku x lst = []
```

```
(* テスト *)
let test1 = hikaku 3 [4; 1; 3; 2]
          = ["大"; "小"; "等"; "小"]
```

map の活用

```
(* 目的:lst の各要素を x との関係文字列にする *)
(* hikaku : int -> int list -> string list *)
let hikaku x lst =
  (* 目的：x との大小を表す文字列を返す *)
  (* hikaku1 : int -> string *)
  let hikaku1 y = ""


  in map hikaku1 lst

(* テスト *)
let test1 = hikaku 3 [4; 1; 3; 2]
            = ["大"; "小"; "等"; "小"]
```

map の活用

```
(* 目的:lst の各要素を x との関係文字列にする *)
(* hikaku : int -> int list -> string list *)
let hikaku x lst =
  (* 目的：x との大小を表す文字列を返す *)
  (* hikaku1 : int -> string *)
  let hikaku1 y = if y < x then "小"
                  else if x < y then "大"
                  else "等"
  in map hikaku1 lst

(* テスト *)
let test1 = hikaku 3 [4; 1; 3; 2]
            = ["大"; "小"; "等"; "小"]
```

関数（変数）のスコープ

- 関数の引数　：関数の本体のみで使用可。
- let ... in 文：in に続く文のみで使用可。
- let　　文：この let 文以降のみで使用可。

```
let hikaku x lst =                          ↓ x, lst 使用可
    let hikaku1 y =  if y < x then "小"
                     else if x < y then "大"
                     else "等"
                                          y 使用可↑
    in  map hikaku1 lst  ←hikaku1 使用可
```

これ以降 hikaku 使用可

多相型と多相関数

```
            add1 : int -> int
            lst  : int list
   map add1 lst : int list

          hikaku1 : int -> string
          lst     : int list
map hikaku1 lst : string list
```

map は多相関数。つまり任意の型変数 'a, 'b について

```
map : ('a -> 'b) -> 'a list -> 'b list
```

多相型と多相関数

```
          add1 : int -> int          'a = int
          lst  : int list            'b = int
   map add1 lst : int list

        hikaku1 : int -> string      'a = int
        lst     : int list           'b = string
map hikaku1 lst : string list
```

map は多相関数。つまり任意の型変数 'a, 'b について

```
map : ('a -> 'b) -> 'a list -> 'b list
```

まとめ

- 関数の一般化：異なる部分を引数にする。
- 関数を引数に受け取る高階関数。
- 関数型言語では、関数は first-class の値。
- 局所関数定義。
- スコープ：変数の使える範囲。
- 多相型と多相関数。

```
map : ('a -> 'b) -> 'a list -> 'b list
```

リストの各要素に、処理を施したリストを返す。

`map` は `List.map` という名前で OCaml にあらかじめ定義されている。