関数型言語（OCaml 演習）(5)

いろいろな再帰関数

浅井 健一

お茶の水女子大学

再帰関数に対するデザインレシピ

デザインレシピ

データ定義	入出力データの型を定義する。再帰的データ型なら自己参照部分を確認する。
目的	関数の目的を考え、ヘッダを作成する。
例	関数の入出力の例を作成する。
テンプレート	構造データをばらす `match` 文を書く。可能な再帰呼び出しをコメントに書く。
本体	関数本体を作成する。再帰呼び出しの意味を目的から理解する。
テスト	作った関数の動作を確認する。

今回、扱う問題

問題 1

整数のリストを受け取ったら、その接頭語のリストを返す関数 `prefix` を定義せよ。

問題 2

学生データのリスト（`gakusei_t list` 型）を受け取ったら、その中の最高点を返す関数 `max_score` を定義せよ。

テストを先に作ると、関数の型を考えやすい

```
(* 目的:受け取った lst の接頭語のリストを返す *)
(* prefix : int list -> (int list) list *)
let rec prefix lst = []



(* テスト *)
let test8 =
  prefix [1; 2; 3; 4] =
  [[1]; [1; 2]; [1; 2; 3]; [1; 2; 3; 4]]
```

整数のリストを受け取ったら、その接頭語のリストを返す関数 prefix を定義せよ。

テンプレートは機械的に入れる

```
(* 目的:受け取った lst の接頭語のリストを返す *)
(* prefix : int list -> (int list) list *)
let rec prefix lst = match lst with
    [] -> []
  | first :: rest -> [] (* prefix rest *)

(* テスト *)
let test8 =
  prefix [1; 2; 3; 4] =
  [[1]; [1; 2]; [1; 2; 3]; [1; 2; 3; 4]]
```

整数のリストを受け取ったら、その接頭語のリストを返す関数 prefix を定義せよ。

本体は、例を使って考える

```
let rec prefix lst = match lst with
    [] -> []
  | first :: rest ->
    [] (* prefix rest *)

(* テスト *)
let test8 =
  prefix [1; 2; 3; 4] =
  [[1]; [1; 2]; [1; 2; 3]; [1; 2; 3; 4]]
```

整数のリストを受け取ったら、その接頭語のリストを返す関数 prefix を定義せよ。

本体は、例を使って考える

```
let rec prefix lst = match lst with
    [] -> []
  | first :: rest ->
    [first] :: [] (* prefix rest *)

(* テスト *)
let test8 =
  prefix [1; 2; 3; 4] =
  [1] :: [[1; 2]; [1; 2; 3]; [1; 2; 3; 4]]
```

整数のリストを受け取ったら、その接頭語のリストを返す関数 prefix を定義せよ。

rest 部分は再帰の結果と結びつける

```
let rec prefix lst = match lst with
    [] -> []
  | first :: rest ->
    [first] :: [] (* prefix rest
       = [   [2];    [2; 3];    [2; 3; 4]] *)
(* テスト *)
let test8 =
  prefix [1; 2; 3; 4] =
  [1] :: [[1; 2]; [1; 2; 3]; [1; 2; 3; 4]]
```

整数のリストを受け取ったら、その接頭語のリストを返す関数 prefix を定義せよ。

自明でない部分問題には補助関数を作る

```
let rec prefix lst = match lst with
    [] -> []
  | first :: rest ->
    [first] :: add_to_each first (prefix rest)

(* テスト *)
let test4 =
add_to_each 1 [   [2];    [2; 3];    [2; 3; 4]]
            = [[1; 2]; [1; 2; 3]; [1; 2; 3; 4]]
```

整数のリストを受け取ったら、その接頭語のリストを返す関数 prefix を定義せよ。

補助関数作成もデザインレシピに従う

```
(* 目的:lst のそれぞれの先頭に n をくっつける *)
(* add_to_each
   : int -> int list list -> int list list *)
let rec add_to_each n lst = []

(* テスト *)
let test4 =
add_to_each 1 [   [2];     [2; 3];     [2; 3; 4]]
            = [[1; 2]; [1; 2; 3]; [1; 2; 3; 4]]
```

n と lst を受け取ったら、lst のそれぞれの先頭に n をつけたリストを返す関数 add_to_each を定義せよ。

テンプレートはいつも機械的に入れる

```
(* 目的:lst のそれぞれの先頭に n をくっつける *)
(* add_to_each
   : int -> int list list -> int list list *)
let rec add_to_each n lst = match lst with
    [] -> []
  | first :: rest ->
    [] (* add_to_each n rest *)
```

n と lst を受け取ったら、lst のそれぞれの先頭に n をつけたリストを返す関数 add_to_each を定義せよ。

本体は例を使って考える

```
(* 目的:lst のそれぞれの先頭に n をくっつける *)
(* add_to_each
   : int -> int list list -> int list list *)
let rec add_to_each n lst = match lst with
    [] -> []
  | first :: rest ->
    (n :: first) :: add_to_each n rest
```

n と lst を受け取ったら、lst のそれぞれの先頭に n をつけたリストを返す関数 add_to_each を定義せよ。

忘れずにテストする

```
# #use "prefix.ml" ;;
val add_to_each : 'a -> 'a list list -> 'a list
val test2 : bool = true
val test3 : bool = true
val test4 : bool = true
val prefix : 'a list -> 'a list list = <fun>
val test6 : bool = true
val test7 : bool = true
val test8 : bool = true
```

整数のリストを受け取ったら、その接頭語のリストを返す関数 prefix を定義せよ。

今回、扱う問題

問題 1

整数のリストを受け取ったら、その接頭語のリストを返す関数 `prefix` を定義せよ。

問題 2

学生データのリスト（`gakusei_t list` 型）を受け取ったら、その中の最高点を返す関数 `max_score` を定義せよ。

いつもデザインレシピ通りに作る

```
(* 学生ひとり分のデータを表す型 *)
type gakusei_t = {
  namae : string;    (* 名前 *)
  tensuu : int;      (* 点数 *)
  seiseki : string; (* 成績 *)
}

(* 学生データの例 *)
let g1 = ...
```

学生のリスト（gakusei_t list 型）を受け取ったら、その中の最高点を返す関数 max_score を定義せよ。

目的、テスト、そしてテンプレート

```
(* 目的:学生リストを受け取ったら、最高点を返す *)
(* max_score : gakusei_t list -> int *)
let rec max_score lst = match lst with
    [] -> 0
  | first :: rest -> 0 (* max_score rest *)

(* テスト *)
let test2 = max_score [g1; g2] = 90
let test3 = max_score [g1; g2; g3] = 90
```

学生のリスト（gakusei_t list 型）を受け取ったら、その中の最高点を返す関数 max_score を定義せよ。

さらにテンプレート

```
(* 目的:学生リストを受け取ったら、最高点を返す *)
(* max_score : gakusei_t list -> int *)
let rec max_score lst = match lst with
    [] -> 0
  | first :: rest -> match first with
    {namae=n; tensuu=t; seiseki=s} ->
    0 (* max_score rest *)
```

学生のリスト（gakusei_t list 型）を受け取ったら、その中の最高点を返す関数 max_score を定義せよ。

あるいはテンプレートの複合

```
(* 目的:学生リストを受け取ったら、最高点を返す *)
(* max_score : gakusei_t list -> int *)
let rec max_score lst = match lst with
    [] -> 0
  | {namae=n; tensuu=t; seiseki=s} :: rest ->
    0 (* max_score rest *)
```

学生のリスト（gakusei_t list 型）を受け取ったら、その中の最高点を返す関数 max_score を定義せよ。

例を見ながら本体を作る

```
(* 目的:学生リストを受け取ったら、最高点を返す *)
(* max_score : gakusei_t list -> int *)
let rec max_score lst = match lst with
    [] -> ???
  | {namae=n; tensuu=t; seiseki=s} :: rest ->
    if t > max_score rest then t
                          else max_score rest
```

学生のリスト（gakusei_t list 型）を受け取ったら、その中の最高点を返す関数 max_score を定義せよ。

本当は例外処理（１８章）を使うのが正しい

```
(* 目的:学生リストを受け取ったら、最高点を返す *)
(* max_score : gakusei_t list -> int *)
let rec max_score lst = match lst with
    [] -> -1
  | {namae=n; tensuu=t; seiseki=s} :: rest ->
    if t > max_score rest then t
                          else max_score rest
```

学生のリスト（gakusei_t list 型）を受け取ったら、その中の最高点を返す関数 max_score を定義せよ。

局所変数定義（let ... in 文）

let 変数 = 式$_1$ in 式$_2$

式$_1$ を実行し、その結果に変数という一時的な名前をつける。次に式$_2$ を実行する。その実行の間（のみ）、ここで定義した名前を使うことができる。

```
# let x = 2 + 3 in
  x + x * 2 ;;
- : int = 15
# x ;;
Error: Unbound value x
```

同じ計算は 1 度のみ行う

```
(* 目的:学生リストを受け取ったら、最高点を返す *)
(* max_score : gakusei_t list -> int *)
let rec max_score lst = match lst with
    [] -> -1
  | {namae=n; tensuu=t; seiseki=s} :: rest ->
    let t_max = max_score rest in
    if t > t_max then t
                 else t_max
```

学生のリスト（gakusei_t list 型）を受け取ったら、その中の最高点を返す関数 max_score を定義せよ。

必ずテストする

```
# #use "max_score.ml" ;;
type gakusei_t = { namae : string; tensuu : int
val gakusei1 : gakusei_t = {namae = "asai"; ten
val gakusei2 : gakusei_t = {namae = "tanaka"; t
val gakusei3 : gakusei_t = {namae = "yamada"; t
val max_score : gakusei_t list -> int = <fun>
val test1 : bool = true
val test2 : bool = true
val test3 : bool = true
```

学生のリスト（gakusei_t list 型）を受け取ったら、その中の最高点を返す関数 max_score を定義せよ。

まとめ

- テストを先に作るとわかりやすい場合がある。
- テンプレートは機械的に入れる。
- 本体も例を見るとわかりやすい場合がある。
- 自明でない部分問題には補助関数を作る。
- 補助関数の作成もデザインレシピに従う。

- テンプレートの複合。（OCaml では、任意に深いパターンを書くことが出来る。）
- 局所的な変数定義。`let ... in` 文