関数型言語（OCaml 演習）(3)

構造データとテンプレート

浅井 健一

お茶の水女子大学

組

(データ$_1$, データ$_2$, ...)

複数のデータをコンマで並べたもの。

```
# (1, 4) ;;
- : int * int = (1, 4)
# ("January", 1, 31) ;;
- : string * int * int = ("January", 1, 31)
# ("January", (1, 31)) ;;
- : string * (int * int) = ("January", (1, 31))
# let point1 = (3.0, 4.0) ;;
val point1 : float * float = (3., 4.)
# point1 ;;
- : float * float = (3., 4.)
```

パターンマッチ（match 文）

match 式$_1$ with
**　パターン -> 式$_2$**

式$_1$ を実行し、その結果をパターンと照合した上で
式$_2$ を実行する。

```
# match (3.0, 4.0) with
    (a, b) -> sqrt (a *. a +. b *. b) ;;
- : float = 5.
```

(a, b) ２つ組のパターン

(a, b, c) ３つ組のパターン

パターンマッチ（match 文）

match 式$_1$ with
**　パターン -> 式$_2$**

式$_1$ を実行し、その結果をパターンと照合した上で
式$_2$ を実行する。

```
# let point1 = (3.0, 4.0) ;;
val point1 : float * float = (3., 4.)
# match point1 with
    (a, b) -> sqrt (a *. a +. b *. b) ;;
- : float = 5.
```

パターンマッチ（match 文）

match 式$_1$ with
**　パターン -> 式$_2$**

式$_1$ を実行し、その結果をパターンと照合した上で
式$_2$ を実行する。

```
# let kyori pair = match pair with
    (a, b) -> sqrt (a *. a +. b *. b) ;;
val kyori : float * float -> float = <fun>
# kyori (3.0, 4.0) ;;
- : float = 5.
# kyori (5.0, 12.0) ;;
- : float = 13.
```

構造データのためのデザインレシピ

デザインレシピ

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。
本体	関数本体を作成する。
テスト	作った関数の動作を確認する。

構造データのためのデザインレシピ

デザインレシピ

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。
本体	関数本体を作成する。
テスト	作った関数の動作を確認する。

問題

実数のペアで表された座標を受け取ったら、原点からその座標までの距離を返す関数 `kyori` を定義せよ。

目的

関数の目的を考え、ヘッダを作成する。

```
(* 目的：原点から受け取った座標 pair までの
        距離を求める *)
(* kyori : float * float -> float *)
let kyori pair = 0.0
```

問題

実数のペアで表された座標を受け取ったら、原点からその座標までの距離を返す関数 kyori を定義せよ。

例

関数の入出力の例を作成する。

```
(* テスト *)
let test1 = kyori (3.0,  4.0) =  5.0
let test2 = kyori (5.0, 12.0) = 13.0
let test3 = kyori (8.0, 15.0) = 17.0
```

問題

実数のペアで表された座標を受け取ったら、原点からその座標までの距離を返す関数 `kyori` を定義せよ。

テンプレート

構造データをばらす match 文を書く。

```
(* 目的：原点から受け取った座標 pair までの
        距離を求める *)
(* kyori : float * float -> float *)
let kyori pair = match pair with
  (a, b) -> 0.0
```

問題

実数のペアで表された座標を受け取ったら、原点からその座標までの距離を返す関数 kyori を定義せよ。

テンプレート

構造データをばらす match 文を書く。

```
(* 目的：原点から受け取った座標 pair までの
        距離を求める *)
(* kyori : float * float -> float *)
let kyori pair = match pair with
  (a, b) -> 0.0
```

２つ組のテンプレート： `match ... with (a, b) -> ...`

テンプレート＝入力データの型から必然的に決まるプログラムの形、ひな形。

本体

関数本体を作成する。

```
(* 目的：原点から受け取った座標 pair までの
        距離を求める *)
(* kyori : float * float -> float *)
let kyori pair = match pair with
  (a, b) -> sqrt (a *. a +. b *. b)
```

問題

実数のペアで表された座標を受け取ったら、原点からその座標までの距離を返す関数 kyori を定義せよ。

テスト

作った関数の動作を確認する。

```
# #use "kyori.ml" ;;
val kyori : float * float -> float = <fun>
val test1 : bool = true
val test2 : bool = true
val test3 : bool = true
```

問題

実数のペアで表された座標を受け取ったら、原点からその座標までの距離を返す関数 `kyori` を定義せよ。

レコード

**{フィールド名$_1$ = データ$_1$;
フィールド名$_2$ = データ$_2$; ...}**

名前付きデータの集まり。

```
# ("asai", 70, "B") ;;
- : string * int * string = ("asai", 70, "B")
# {namae="asai"; tensuu=70; seiseki="B"} ;;
- : gakusei_t = {namae = "asai"; tensuu = 70;
                                  seiseki = "B"}
# {tensuu=70; seiseki="B"; namae="asai"} ;;
- : gakusei_t = {namae = "asai"; tensuu = 70;
                                  seiseki = "B"}
```

型定義（type 文）

type 型 = その定義

新しい型を宣言する。

```
# type gakusei_t = {
    namae : string;
    tensuu : int;
    seiseki : string;
  } ;;
type gakusei_t = { namae : string;
                   tensuu : int; seiseki : string; }
```

フィールド名は、プログラム中で全て互いに異ならなくてはならない。

データ定義のためのデザインレシピ

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。
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。
本体	関数本体を作成する。
テスト	作った関数の動作を確認する。

データ定義のためのデザインレシピ

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。
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。
本体	関数本体を作成する。
テスト	作った関数の動作を確認する。

問題

学生の成績データを受け取ったら、成績通知文を返す関数 `tsuuchi` を定義せよ。

データ定義

入出力データの型を定義する。

```
(* データ定義 *)
type gakusei_t = {
  namae : string;    (* 名前 *)
  tensuu : int;      (* 点数 *)
  seiseki : string; (* 成績 *)
}
```

問題

学生の成績データを受け取ったら、成績通知文を返す関数 `tsuuchi` を定義せよ。

データ定義

入出力データの型を定義する。

```
(* データ定義 *)
type gakusei_t = {
  namae : string;    (* 名前 *)
  tensuu : int;      (* 点数 *)
  seiseki : string; (* 成績 *)
}

(* 学生データの例 *)
let g1 = {namae="asai";   tensuu=70; seiseki="B"}
let g2 = {namae="tanaka"; tensuu=90; seiseki="A"}
let g3 = {namae="yamada"; tensuu=60; seiseki="C"}
```

目的

関数の目的を考え、ヘッダを作成する。

```
(* 目的：学生の成績データを受け取ったら、
          成績通知文を返す *)
(* tsuuchi : gakusei_t -> string *)
let tsuuchi gakusei = ""
```

問題

学生の成績データを受け取ったら、成績通知文を返す関数 `tsuuchi` を定義せよ。

例

関数の入出力の例を作成する。

```
(* テスト *)
let test1 = tsuuchi g1 = "asai さんは B です"
let test2 = tsuuchi g2 = "tanaka さんは A です"
let test3 = tsuuchi g3 = "yamada さんは C です"
```

問題

学生の成績データを受け取ったら、成績通知文を返す関数 `tsuuchi` を定義せよ。

テンプレート

構造データをばらす match 文を書く。

```
(* 目的：学生の成績データを受け取ったら、
        成績通知文を返す *)
(* tsuuchi : gakusei_t -> string *)
let tsuuchi gakusei = match gakusei with
  {namae=n; tensuu=t; seiseki=s} -> ""
```

問題

学生の成績データを受け取ったら、成績通知文を返す関数 tsuuchi を定義せよ。

テンプレート

構造データをばらす match 文を書く。

```
(* 目的：学生の成績データを受け取ったら、
        成績通知文を返す *)
(* tsuuchi : gakusei_t -> string *)
let tsuuchi gakusei = match gakusei with
  {namae=n; tensuu=t; seiseki=s} -> ""
```

レコードのテンプレート：

```
match ... with
  {namae=n; ...} -> ...
```

本体

関数本体を作成する。

```
(* 目的：学生の成績データを受け取ったら、
          成績通知文を返す *)
(* tsuuchi : gakusei_t -> string *)
let tsuuchi gakusei = match gakusei with
  {namae=n; tensuu=t; seiseki=s} ->
    n ^ " さんは " ^ s ^ " です"
```

問題

学生の成績データを受け取ったら、成績通知文を返す関数 `tsuuchi` を定義せよ。

テスト

作った関数の動作を確認する。

```
# #use "tsuuchi.ml" ;;
type gakusei_t = { namae : string; tensuu ...
val tsuuchi : gakusei_t -> string = <fun>
val test1 : bool = true
val test2 : bool = true
val test3 : bool = true
```

問題

学生の成績データを受け取ったら、成績通知文を返す関数 tsuuchi を定義せよ。

まとめ

- 組、レコード。(1, 2) {namae="asai"; ...}
- パターンマッチ。match 式 with パターン -> 式
- 型定義。type 型 = {namae:string; ...}
- テンプレート＝入力データの型から必然的に決まるプログラムの形、ひな形。

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。
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。
本体	関数本体を作成する。
テスト	作った関数の動作を確認する。