関数型言語（OCaml 演習）(4)

リストと再帰

浅井 健一

お茶の水女子大学

リスト（'a list 型）

任意個のデータを持つことのできる構造データ。

`[]`	空のリスト。
`first :: rest`	リスト rest の先頭に要素 first を付け加えてできたリスト。

```
# [] ;;
- : 'a list = []
# 1 :: [] ;;
- : int list = [1]
# 2 :: 1 :: [] ;;    これは 2 :: (1 :: []) の意
- : int list = [2; 1]
# 3 :: 2 :: 1 :: [] ;;    3 :: (2 :: (1 :: []))
- : int list = [3; 2; 1]
```

リスト（'a list 型）

リストの略記法

$[d_1;\ d_2;\ \ldots\ ;\ d_n] = d_1\ ::\ d_2\ ::\ \ldots\ ::\ d_n\ ::\ []$

```
# [] ;;
- : 'a list = []
# 1 :: [] ;;
- : int list = [1]
# 2 :: 1 :: [] ;;
- : int list = [2; 1]
# 3 :: 2 :: 1 :: [] ;;
- : int list = [3; 2; 1]
```

これは 2 :: (1 :: []) の意

3 :: (2 :: (1 :: []))

リストの型（'a list 型）

'a list 要素の型が 'a 型であるようなリストの型

```
# [3; 2; 1] ;;
- : int list = [3; 2; 1]
# true :: [false; true] ;;
- : bool list = [true; false; true]
# ("January", 1) :: ("February", 2) :: [] ;;
- : (string * int) list =
              [("January", 1); ("February", 2)]
# [] ;;
- : 'a list = []
```

リストの要素は、全て同じ型を持たなくてはならない。

リストのテンプレート（match 文）

```
match 式0 with
    [] -> 式1
  | first :: rest -> 式2
```

（'a list 型の）式$_0$ を実行する。その結果が

- [] だったら式$_1$ を実行する。
- first :: rest の形だったら、先頭の要素を first に、先頭を取り除いたリストを rest に入れた上で、式$_2$ を実行する。

```
# match [] with
    [] -> 0
  | first :: rest -> first ;;
- : int = 0
```

リストのテンプレート（match 文）

```
match 式0 with
    [] -> 式1
  | first :: rest -> 式2
```

（'a list 型の）式$_0$ を実行する。その結果が

- [] だったら式$_1$ を実行する。
- first :: rest の形だったら、先頭の要素を first に、先頭を取り除いたリストを rest に入れた上で、式$_2$ を実行する。

```
# match [1; 2; 3] (* = 1 :: [2; 3] *) with
    [] -> 0
  | first :: rest -> first ;;
- : int = 1
```

リストのテンプレート（match 文）

```
match 式0 with
    [] -> 式1
  | first :: rest -> 式2
```

（'a list 型の）式$_0$ を実行する。その結果が

- [] だったら式$_1$ を実行する。
- first :: rest の形だったら、先頭の要素を first に、先頭を取り除いたリストを rest に入れた上で、式$_2$ を実行する。

- 式$_0$ とパターンは、同じ型でなくてはならない。
- 矢印の右側は、みな同じ型でなくてはならない。

リスト：再帰的なデータ型

リストは、以下のいずれかである。

`[]` 空リスト、あるいは

`first :: rest` 長さ 1 以上のリスト
ここで `first` は先頭の要素、
`rest` は先頭を除いた残りのリスト

※ リストの定義中で、リストを自己参照している。

リスト：再帰的なデータ型

リストは、以下のいずれかである。

`[]` 空リスト、あるいは

`first :: rest` 長さ 1 以上のリスト
ここで `first` は先頭の要素、
`rest` は先頭を除いた残りのリスト

※ リストの定義中で、リストを自己参照している。

問題

整数のリストを受け取ったら、その中に 0 が含まれているかを判定する関数 `contain_zero` を定義せよ。

データ定義

入出力データの型を定義する。

```
(* int list は
   - []                空リスト、あるいは
   - first :: rest     最初の要素が first,
                       残りのリストが rest
                       (rest が自己参照のケース)
   という形 *)
```

問題

整数のリストを受け取ったら、その中に 0 が含まれているかを判定する関数 contain_zero 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：受け取ったリストに 0 があるかを調べる *)
(* contain_zero : int list -> bool *)
let contain_zero lst = false
```

問題

整数のリストを受け取ったら、その中に 0 が含まれているかを判定する関数 contain_zero を定義せよ。

例

関数の入出力の例を作成する。

```
(* テスト *)
let test1 = contain_zero []           = false
let test2 = contain_zero [0; 1; 2; 3] = true
let test3 = contain_zero [1; 2; 0; 3] = true
let test4 = contain_zero [1; 2; 3; 4] = false
```

問題

整数のリストを受け取ったら、その中に 0 が含まれているかを判定する関数 `contain_zero` 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:受け取ったリストに 0 があるかを調べる *)
(* contain_zero : int list -> bool *)
let contain_zero lst = match lst with
    [] -> false
  | first :: rest -> false
```

問題

整数のリストを受け取ったら、その中に 0 が含まれているかを判定する関数 contain_zero を定義せよ。

本体

関数本体を作成する。

```
(* 目的：受け取ったリストに 0 があるかを調べる *)
(* contain_zero : int list -> bool *)
let contain_zero lst = match lst with
    [] -> false
  | first :: rest ->
    if first = 0 then true
                 else false (* ??? *)
```

問題

整数のリストを受け取ったら、その中に 0 が含まれているかを判定する関数 contain_zero を定義せよ。

本体

関数本体を作成する。

```
(* 目的：受け取ったリストに 0 があるかを調べる *)
(* contain_zero : int list -> bool *)
let rec contain_zero lst = match lst with
    [] -> false
  | first :: rest ->
    if first = 0 then true
                 else contain_zero rest
```

問題

整数のリストを受け取ったら、その中に 0 が含まれているかを判定する関数 contain_zero を定義せよ。

テスト

作った関数の動作を確認する。

```
# #use "contain_zero.ml" ;;
val contain_zero : int list -> bool = <fun>
val test1 : bool = true
val test2 : bool = true
val test3 : bool = true
val test4 : bool = true
```

問題

整数のリストを受け取ったら、その中に 0 が含まれているかを判定する関数 contain_zero を定義せよ。

再帰関数に対するデザインレシピ

データ定義

入出力データの型を定義する。再帰的データ型の場合は、自己参照している部分がどこかを確認する。

テンプレート

構造データをばらす `match` 文を書く。自己参照している部分に対する再帰呼び出しもコメントとして書いておく。

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。

データ定義

入出力データの型を定義する。

```
(* int list は
    - []                  空リスト、あるいは
    - first :: rest       最初の要素が first,
                          残りのリストが rest
                           (rest が自己参照のケース)
    という形 *)
```

問題

整数のリストを受け取ったら、その合計を返す関数 sum 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：整数のリストを受け取ったら合計を返す *)
(* sum : int list -> int *)
let rec sum lst = 0
```

問題

整数のリストを受け取ったら、その合計を返す関数 sum を定義せよ。

例

関数の入出力の例を作成する。

```
(* テスト *)
let test1 = sum [] = 0
let test2 = sum [7] = 7
let test3 = sum [3; 2] = 5
let test4 = sum [1; 2; 3; 4; 5; 6; 7; 8] = 36
```

問題

整数のリストを受け取ったら、その合計を返す関数 sum 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：整数のリストを受け取ったら合計を返す *)
(* sum : int list -> int *)
let rec sum lst = match lst with
    [] -> 0
  | first :: rest -> 0 (* sum rest *)
```

問題

整数のリストを受け取ったら、その合計を返す関数 sum を定義せよ。

本体

関数本体を作成する。

```
(* 目的：整数のリストを受け取ったら合計を返す *)
(* sum : int list -> int *)
let rec sum lst = match lst with
    [] -> 0
  | first :: rest -> first + sum rest
```

問題

整数のリストを受け取ったら、その合計を返す関数 sum を定義せよ。

テスト

作った関数の動作を確認する。

```
# #use "sum.ml" ;;
val sum : int list -> int = <fun>
val test1 : bool = true
val test2 : bool = true
val test3 : bool = true
val test4 : bool = true
```

問題

整数のリストを受け取ったら、その合計を返す関数 sum を定義せよ。

sum の動きと数学的帰納法

```
(* 目的：整数のリストを受け取ったら合計を返す *)
(* sum : int list -> int *)
let rec sum lst = match lst with
    [] -> 0
  | first :: rest -> first + sum rest
```

```
sum [1; 2; 3] = 1 + sum [2; 3]
              = 1 + (2 + sum [3])
              = 1 + (2 + (3 + sum []))
              = 1 + (2 + (3 + 0))
              = 1 + (2 + 3)
              = 1 + 5
              = 6
```

sum の動きと数学的帰納法

```
(* 目的：整数のリストを受け取ったら合計を返す *)
(* sum : int list -> int *)
let rec sum lst = match lst with
    [] -> 0
  | first :: rest -> first + sum rest
```

```
sum [1; 2; 3] = 1 + sum [2; 3]
              = 1 + (2 + sum [3])
              = 1 + (2 + (3 + sum []))
              = 1 + (2 + (3 + 0))
              = 1 + (2 + 3)
              = 1 + 5
              = 6
```

まとめ

- リスト（'a list 型）。match 式 with
  [] -> 式 | first :: rest -> 式
- 自己参照と再帰、数学的帰納法。

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。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。