関数型言語（OCaml 演習）(9)

一般の再帰

浅井 健一

お茶の水女子大学

前回まで：リストの構造に従った再帰

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

```
(* 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
```

一般的な問題の解き方（分割統治法）

1. 自明に答えが出るケースに対処する。
2. 自明に答えが出ないケースに対処する。

- 問題を部分問題に分割する。
- 各々の部分問題を解く。（再帰）
- 得られた解から全体の解を計算する。

一般的な問題の解き方（分割統治法）

1. 自明に答えが出るケースに対処する。
2. 自明に答えが出ないケースに対処する。

- 問題を部分問題に分割する。
- 各々の部分問題を解く。（再帰）
- 得られた解から全体の解を計算する。

```
(* 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
```

クイックソート

C. A. R. Hoare による分割統治法を使った整列アルゴリズム。多くの場合、とても高速。

入力の例：[5; 4; 9; 8; 2; 3]

1. 基準値を選ぶ。
   基準値：5、それ以外：[4; 9; 8; 2; 3]
2. 問題を基準値より小、基準値より大に分割する。
   小：[4; 2; 3]、大：[9; 8]
3. それぞれの部分問題を解く（整列する）。
   小：[2; 3; 4]、大：[8; 9]
4. そこから全体の解を計算する。
   解＝小 @ [基準値] @ 大＝ [2; 3; 4; 5; 8; 9]

クイックソート

```
(* 目的：受け取った整数のリストを昇順に並べる *)
(* quick_sort : int list -> int list *)
let rec quick_sort lst = []



(* テスト *)
let test1 = quick_sort [] = []
let test2 = quick_sort [4; 2; 3] = [2; 3; 4]
let test3 = quick_sort [5; 4; 9; 8; 2; 3]
                       = [2; 3; 4; 5; 8; 9]
```

クイックソート

```
(* 目的：受け取った整数のリストを昇順に並べる *)
(* quick_sort : int list -> int list *)
let rec quick_sort lst = match lst with
    [] -> []
  | first :: rest -> []


(* テスト *)
let test1 = quick_sort [] = []
let test2 = quick_sort [4; 2; 3] = [2; 3; 4]
let test3 = quick_sort [5; 4; 9; 8; 2; 3]
                       = [2; 3; 4; 5; 8; 9]
```

クイックソート

```
(* 目的：受け取った整数のリストを昇順に並べる *)
(* quick_sort : int list -> int list *)
let rec quick_sort lst = match lst with
    [] -> []
  | first :: rest -> []
```

```
take_less    5 [4; 9; 8; 2; 3] = [4; 2; 3]
take_greater 5 [4; 9; 8; 2; 3] = [9; 8]
```

クイックソート

```
(* 目的：受け取った整数のリストを昇順に並べる *)
(* quick_sort : int list -> int list *)
let rec quick_sort lst = match lst with
    [] -> []
  | first :: rest ->
                    take_less first rest

                      take_greater first rest

take_less    5 [4; 9; 8; 2; 3] = [4; 2; 3]
take_greater 5 [4; 9; 8; 2; 3] = [9; 8]
```

クイックソート

```
(* 目的：受け取った整数のリストを昇順に並べる *)
(* quick_sort : int list -> int list *)
let rec quick_sort lst = match lst with
    [] -> []
  | first :: rest ->
      quick_sort (take_less first rest)

        quick_sort (take_greater first rest)

take_less    5 [4; 9; 8; 2; 3] = [4; 2; 3]
take_greater 5 [4; 9; 8; 2; 3] = [9; 8]
```

クイックソート

```
(* 目的：受け取った整数のリストを昇順に並べる *)
(* quick_sort : int list -> int list *)
let rec quick_sort lst = match lst with
    [] -> []
  | first :: rest ->
      quick_sort (take_less first rest)
      @ [first]
      @ quick_sort (take_greater first rest)

take_less    5 [4; 9; 8; 2; 3] = [4; 2; 3]
take_greater 5 [4; 9; 8; 2; 3] = [9; 8]
lst1 @ lst2 で lst1 と lst2 をくっつける。
```

クイックソート

```
(* 目的：n より小さい要素を取り出す *)
(* take_less : int -> int list -> int list *)
let take_less n lst =
  let less x = x < n in
  List.filter less lst

(* 目的：n より大きい要素を取り出す *)
(* take_greater
                : int -> int list -> int list *)
let take_greater n lst =
  let greater x = x > n in
  List.filter greater lst
```

停止性

- 構造に従った再帰は停止性が保証されている。
- 一般の再帰では別途、停止性の証明が必要。

```
(* 目的：受け取った整数のリストを昇順に並べる *)
(* quick_sort : int list -> int list *)
let rec quick_sort lst = match lst with
    [] -> []
  | first :: rest ->
      quick_sort (take_less first rest)
      @ [first]
      @ quick_sort (take_greater first rest)
```

どのような `lst` が与えられても必ず停止するか？

停止性の証明

何らかの意味で入力が小さくなっていることを示す。

quick_sort の場合は、引数のリストの「長さ」が短くなっていることを示す。

1. quick_sort の入力：lst
2. rest の長さ ＝ lst の長さ − 1
3. take_less first rest の長さ $\leq$ rest の長さ
   （take_less は rest の中から条件を満たすものを抽出しているだけなので。）従って
4. take_less first rest の長さ $<$ lst の長さ

take_greater の場合も同様。

まとめ

一般的な問題の解き方（分割統治法）

1. 自明に答えが出るケースに対処する。
2. 自明に答えが出ないケースに対処する。

- 問題を部分問題に分割する。
- 各々の部分問題を解く。（再帰）
- 得られた解から全体の解を計算する。

停止性の証明

一般の再帰では別途、停止性の証明が必要。

何らかの意味で入力が小さくなっていることを示す。
構造に従った再帰は停止性が保証されている。