関数型言語（OCaml 演習）(12)

モジュールと抽象データ型

浅井 健一

お茶の水女子大学

２分探索木

```
(* キーが 'a 型、値が 'b 型の木の型 *)
type ('a, 'b) t = Empty
          | Node of ('a, 'b) t * 'a * 'b * ('a, 'b) t

(* empty : ('a, 'b) t, 空の木 *)
let empty = Empty

(* 目的：木にキーと値を挿入した木を返す *)
(* insert : ('a, 'b) t -> 'a -> 'b -> ('a, 'b) t *)
let rec insert tree k v = ...

(* 目的：木の中からキーに対応する値を探して返す *)
(* なければ例外 Not_found を起こす *)
(* search : ('a, 'b) t -> 'a -> 'b *)
let rec search tree k = ...
```

２分探索木

```
module Tree = struct
  (* キーが 'a 型、値が 'b 型の木の型 *)
  type ('a, 'b) t = Empty
                  | Node of ('a, 'b) t * 'a * 'b * ('a, 'b) t

  (* empty : ('a, 'b) t, 空の木 *)
  let empty = Empty

  (* 目的：木にキーと値を挿入した木を返す *)
  (* insert : ('a, 'b) t -> 'a -> 'b -> ('a, 'b) t *)
  let rec insert tree k v = ...

  (* 目的：木の中からキーに対応する値を探して返す *)
  (* なければ例外 Not_found を起こす *)
  (* search : ('a, 'b) t -> 'a -> 'b *)
  let rec search tree k = ...
end
```

モジュール

意味的につながりのあるプログラムのまとまり。

`module モジュール名 = struct ... end`

モジュールを定義する。

モジュールの中身には

- 型定義
- 変数定義
- 関数定義

など、およそ何でも書くことができる。

OCaml のモジュール名は、大文字で始まらなくてはならない。

モジュールの中身へのアクセス

モジュール名 . 変数名（型名ほか）

モジュール中の「変数」「型」ほかにアクセスする。

```
(* 目的：学生リストの情報を持つ木を作る *)
(* inserts : gakusei_t list ->
                (string, int * string) Tree.t *)
let rec inserts lst = match lst with
    [] -> Tree.empty
  | {namae=n; tensuu=t; seiseki=s} :: rest ->
      Tree.insert (inserts rest) n (t, s)
```

gakusei_t 型のリストを受け取ったら、名前をキー、点数と成績のペアを値とする木を作成する。

モジュールの型：シグネチャ

モジュール内で宣言されているものの型を集めたもの。

例えば Tree モジュールなら以下。

```
sig
  type ('a, 'b) t = Empty
        | Node of ('a, 'b) t * 'a * 'b * ('a, 'b) t
  val empty  : ('a, 'b) t
  val insert : ('a, 'b) t -> 'a -> 'b -> ('a, 'b) t
  val search : ('a, 'b) t -> 'a -> 'b
end
```

明示的にシグネチャを書かない限りモジュール内の全てが公開される（＝ドット記法で外からアクセス可能）

モジュール内の一部を非公開にする

非公開にしたい理由：

- モジュール内専用の補助関数は公開したくない。
- 型の中身を非公開にしたい。→ 抽象データ型

シグネチャに現れているもの（のみ）が、モジュール外からドット記法でアクセス可能。

自分でシグネチャを書き、モジュール定義時に指定をすれば、一部を非公開にすることができる。

`module type シグネチャ名 = sig ... end`

シグネチャを宣言する。

型の中身を非公開にする

```
module type Tree_t = sig
  type ('a, 'b) t    (* 型の中身（構成子）は非公開 *)
  val empty  : ('a, 'b) t
  val insert : ('a, 'b) t -> 'a -> 'b -> ('a, 'b) t
  val search : ('a, 'b) t -> 'a -> 'b
end
```

- empty, insert, search は使える。
- ('a, 'b) t 型は存在のみ公開されている。
- その構成子は非公開。(外部からは使用不可。)

構成子が使えないので

- 木を作るには empty か insert を使うしかない。
- 木の中身を見るには search を使うしかない。

抽象データ型

データ型を可能な操作の集まりで定義したもの。

内部でどのように実現されているかには関知しない。

木とは、('a, 'b) t 型の（何らかの）値で、
empty, insert, search を使えるもの。

- モジュールの中身を入れ替えることができる。
- モジュールの作成と使用が分離される。
  - プログラムの作成を分業できるようになる。
  - 分割コンパイルができる。
- シグネチャさえ理解すれば、モジュール内部のことは知らなくてもプログラミングをできる。
- シグネチャが良いドキュメントになっている。

シグネチャの書き方

```
module type Tree_t = sig
  type ('a, 'b) t  (* キーが 'a 型、値が 'b 型の木の型 *)

  val empty : ('a, 'b) t
  (* 使い方：empty *)
  (* 空の木を表す *)

  val insert : ('a, 'b) t -> 'a -> 'b -> ('a, 'b) t
  (* 使い方：insert tree key value *)
  (* 木 tree にキー key と値 value を挿入した木を返す *)
  (* キーが既に存在していたら新しい値に置き換える *)

  val search : ('a, 'b) t -> 'a -> 'b
  (* 使い方：search tree key *)
  (* 木 tree の中からキー key に対応する値を探して返す *)
  (* なければ例外 Not_found を起こす *)
end
```

シグネチャの指定の仕方

```
module Tree : Tree_t = struct
  (* キーが 'a 型、値が 'b 型の木の型 *)
  type ('a, 'b) t = Empty
                  | Node of ('a, 'b) t * 'a * 'b * ('a, 'b) t

  (* empty : ('a, 'b) t, 空の木 *)
  let empty = Empty

  (* 目的：木にキーと値を挿入した木を返す *)
  (* insert : ('a, 'b) t -> 'a -> 'b -> ('a, 'b) t *)
  let rec insert tree k v = ...

  (* 目的：木の中からキーに対応する値を探して返す *)
  (* なければ例外 Not_found を起こす *)
  (* search : ('a, 'b) t -> 'a -> 'b *)
  let rec search tree k = ...
end
```

まとめ

- モジュール： `module 名 = struct ... end`
- モジュール名は大文字で始まる。
- モジュールアクセスにはドット記法。
- シグネチャ： `module type 名 = sig ... end`

抽象データ型

データ型を可能な操作の集まりで定義したもの。
型の定義はモジュール内に隠蔽され非公開。

- モジュールの作成と使用が分離。
- シグネチャのみでモジュールの理解が可能。
- シグネチャは良いドキュメント。