関数型言語（OCaml 演習）(10)

再帰的なデータ構造

浅井 健一

お茶の水女子大学

バリアント型

「どれかひとつ」を表す型

バリアント型の宣言

```
type 型名 = 構成子 of 引数の型 | ...
```

引数を持たないときは「of 引数の型」を省略する。

```
(* 時刻を表す型 *)                    (* 引数 *)
type jikoku_t = Gozen of int (* 整数ひとつ *)
              | Gogo of int  (* 整数ひとつ *)
              | Noon         (* なし *)
              | Midnight     (* なし *)
```

OCaml の構成子は大文字で始まらなくてはならない。

バリアント型

```
# Gozen (10) ;;
- : jikoku_t = Gozen 10
# Gogo (3) ;;
- : jikoku_t = Gogo 3
# Noon ;;
- : jikoku_t = Noon
# Midnight ;;
- : jikoku_t = Midnight
```

構成子の引数は、必ず括弧でくくるようにすると間違いが減る。

バリアント型を扱う関数

バリアント型の値をばらす際にも match 文を使う

```
(* 目的：24 時間表示の時刻を返す *)
(* let jikoku24 : jikoku_t -> int *)
let jikoku24 jikoku = match jikoku with
    Gozen (n) -> n
  | Gogo  (n) -> n + 12
  | Noon      -> 12
  | Midnight  -> 0

# jikoku24 (Gogo (3)) ;;
- : int = 15
```

青字部分が jikoku_t 型のテンプレート

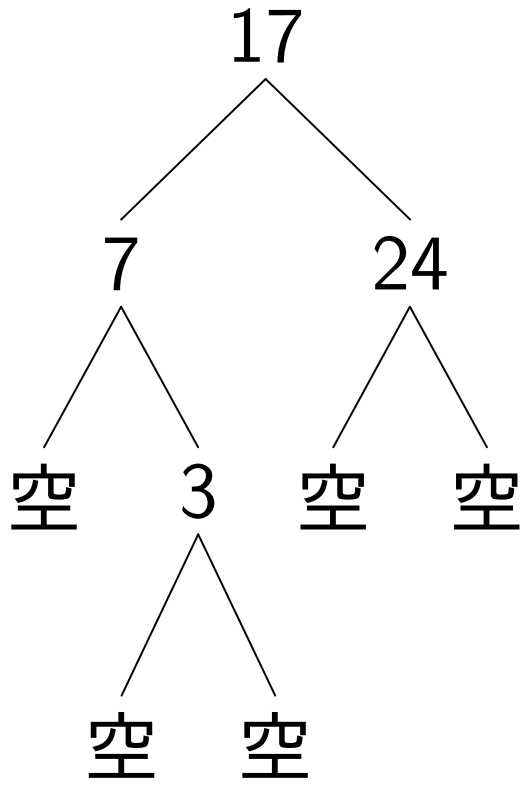
木

空の木または節からなるデータ構造。

節は

- 木を子供に持つ。（循環構造）
- 値を持つことが多い。

節の子供の数がいつも 2 である木のことを２分木と呼ぶ。



２分木

２分木は、以下のいずれかである。

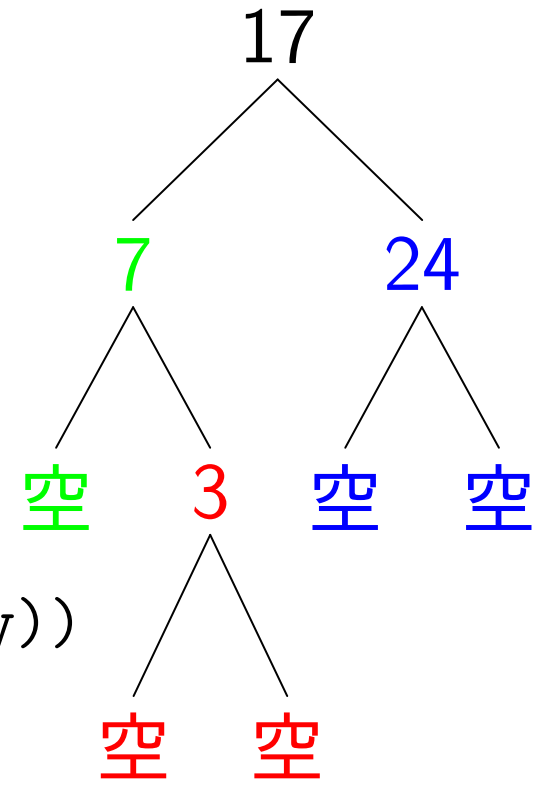
`Empty`	空の木、あるいは
`Node (l, n, r)`	節。ここで `n` はこの節の持つ値、`l, r` は左右の子供（２分木）。

```
(* ２分木を表す型 *)
type tree_t = Empty
            | Node of tree_t * int * tree_t
```

リストとは違い２ヶ所で自己参照している。

２分木の例

```
# Empty ;;
- : tree_t = Empty
# let t1 = Node (Empty, 3, Empty) ;;
val t1 : tree_t =
  Node (Empty, 3, Empty)
# let t2 = Node (Empty, 7, t1) ;;
val t2 : tree_t =
  Node (Empty, 7, Node (Empty, 3, Empty))
# let t3 = Node (Empty, 24, Empty) ;;
val t3 : tree_t =
  Node (Empty, 24, Empty)
# let t4 = Node (t2, 17, t3) ;;
val t4 : tree_t =
  Node (Node (Empty, 7, Node (Empty, 3, Empty)),
        17,
        Node (Empty, 24, Empty))
```



２分木を処理する関数

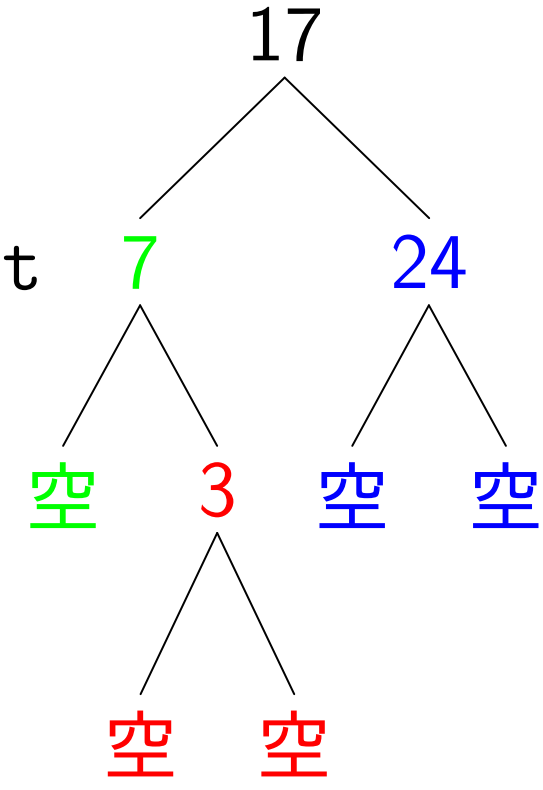
- 基本的にリストを処理する関数と全く同じ。
- テンプレートの形が２分木用に変わるだけ。

問題

`tree_t` 型の木を受け取ったら、その中の整数の合計を返す関数 `sum_tree` を定義せよ。

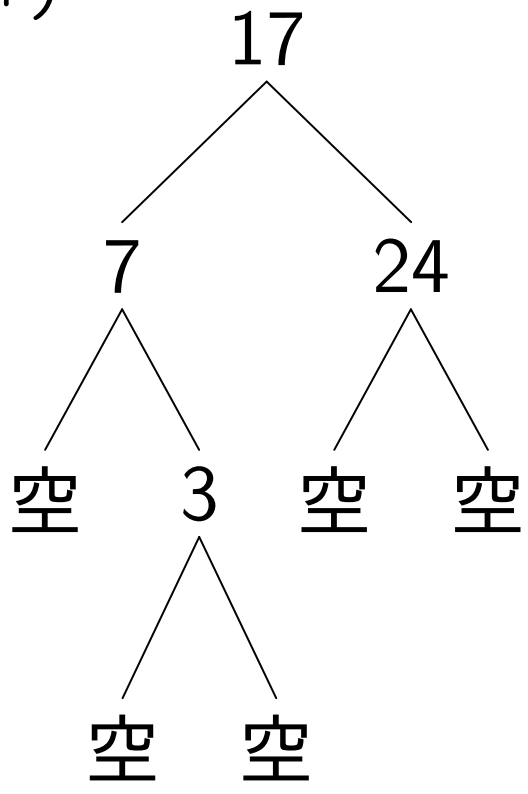
データ定義

```
(* ２分木を表す型 *)
type tree_t =
    Empty
  | Node of tree_t * int * tree_t

(* ２分木の例 *)
let t0 = Empty
let t1 = Node (Empty, 3, Empty)
let t2 = Node (Empty, 7, t1)
let t3 = Node (Empty, 24, Empty)
let t4 = Node (t2, 17, t3)
```



目的と例

```
(* 目的：木の中の整数の合計を返す *)
(* sum_tree : tree_t -> int *)
let rec sum_tree tree = 0

(* テスト *)
let test0 = sum_tree t0 = 0
let test1 = sum_tree t1 = 3
let test2 = sum_tree t2 = 10
let test3 = sum_tree t3 = 24
let test4 = sum_tree t4 = 51
```



テンプレート

```
(* 目的：木の中の整数の合計を返す *)
(* sum_tree : tree_t -> int *)
let rec sum_tree tree = match tree with
    Empty -> 0
  | Node (l, n, r) -> 0
                      (* sum_tree l *)
                      (* sum_tree r *)
```

テンプレート

```
(* 目的：木の中の整数の合計を返す *)
(* sum_tree : tree_t -> int *)
let rec sum_tree tree = match tree with
    Empty -> 0
  | Node (l, n, r) -> 0
                      (* sum_tree l *)
                      (* sum_tree r *)
```

- sum_tree l は左の子 l の合計
- sum_tree r は右の子 r の合計

本体

```
(* 目的：木の中の整数の合計を返す *)
(* sum_tree : tree_t -> int *)
let rec sum_tree tree = match tree with
    Empty -> 0
  | Node (l, n, r) -> sum_tree l
                      + n
                      + sum_tree r
```

- sum_tree l は左の子 l の合計
- sum_tree r は右の子 r の合計

２分探索木

２分木で、各節のデータが順に並んでいるもの

- 左の子のデータは自分より小さい
- 右の子のデータは自分より大きい

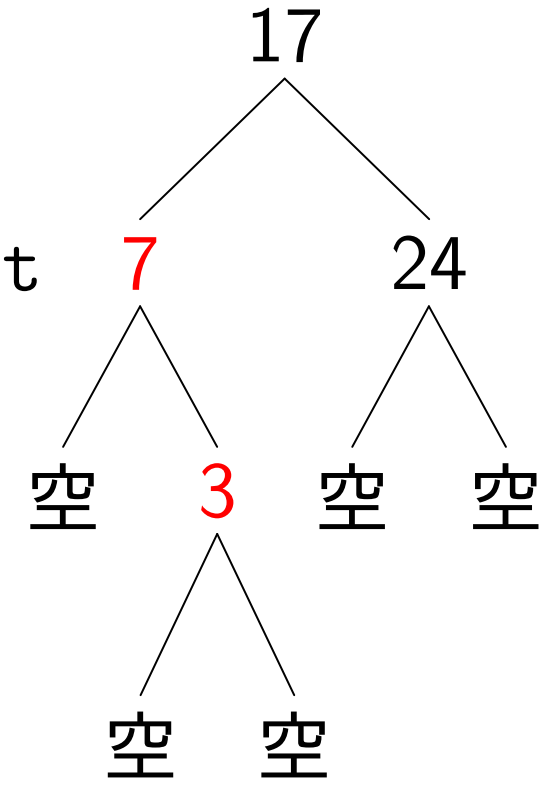
木がバランスしていれば、対数時間でデータにアクセスできる。

問題

`tree_t` 型の２分探索木の中に、受け取った数字 `m` が存在するかを判定する関数 `search` を定義せよ。

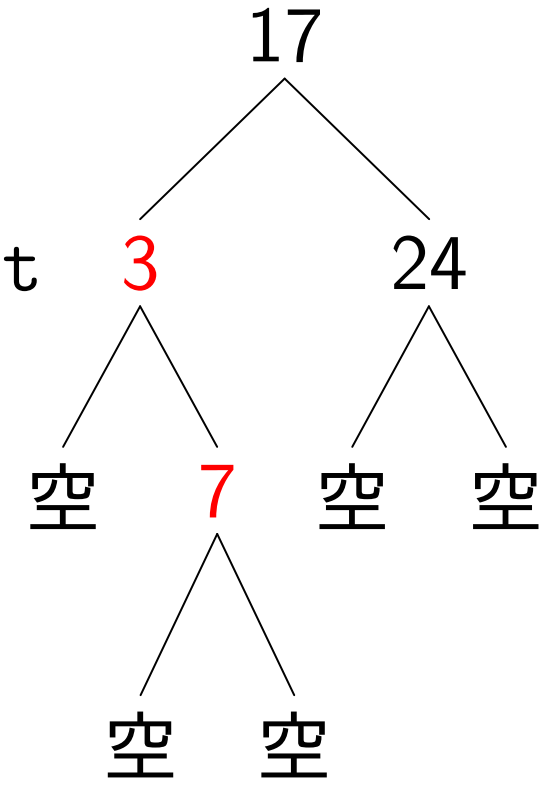
データ定義

```
(* ２分木を表す型 *)
type tree_t =
    Empty
  | Node of tree_t * int * tree_t

(* ２分木の例 *)
let t0 = Empty
let t1 = Node (Empty, 3, Empty)
let t2 = Node (Empty, 7, t1)
let t3 = Node (Empty, 24, Empty)
let t4 = Node (t2, 17, t3)
```



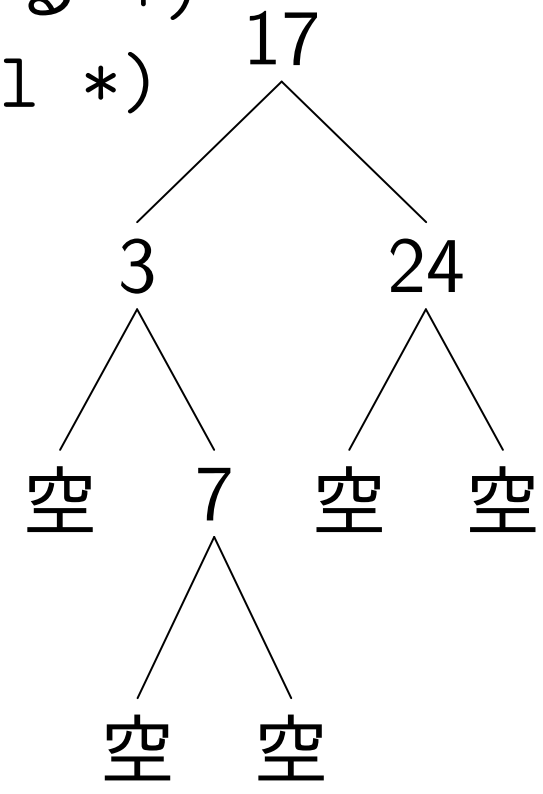
データ定義

```
(* ２分木を表す型 *)
type tree_t =
    Empty
  | Node of tree_t * int * tree_t

(* ２分探索木の例 *)
let t0 = Empty
let t1 = Node (Empty, 7, Empty)
let t2 = Node (Empty, 3, t1)
let t3 = Node (Empty, 24, Empty)
let t4 = Node (t2, 17, t3)
```



目的と例

```
(* 目的：木の中に m があるか判定する *)
(* search : tree_t -> int -> bool *)
let rec search tree m = false

(* テスト *)
let test0 = search t0 0 = false
let test1 = search t4 17 = true
let test2 = search t4 7 = true
let test3 = search t4 5 = false
let test4 = search t4 20 = false
```



テンプレート

```
(* 目的：木の中に m があるか判定する *)
(* search : tree_t -> int -> bool *)
let rec search tree m = match tree with
    Empty -> false
  | Node (l, n, r) -> false
                      (* search l m *)
                      (* search r m *)
```

- search l m は l に m が含まれているか
- search r m は r に m が含まれているか

本体

```
(* 目的：木の中に m があるか判定する *)
(* search : tree_t -> int -> bool *)
let rec search tree m = match tree with
    Empty -> false
  | Node (l, n, r) ->
      if m < n then search l m
      else if n < m then search r m
      else (* m = n *) true
```

- search l m は l に m が含まれているか
- search r m は r に m が含まれているか

まとめ

- バリアント型。構成子（大文字で始める）。
- ばらすには `match` 文（＝この型のテンプレート）。
- 木、２分木、２分探索木。

再帰的データ構造はリストの一般化

リストに対する手法をそのまま使うことができる。

- デザインレシピに従ってプログラムを作成。
- テンプレートは再帰的データ構造に合わせる。
- 再帰は自己参照の回数だけ行うことができる。
- この再帰は必ず停止する。（もとのデータの一部に対してのみ再帰を行っているため。）