自分で型を定義する

- OCaml では，組込みの型（tuple, list を含む）を使っている限り型定義は不要（cf. C の typedef）
- ただし，型に名前をつけて，それを使って（let で定義される）名前の型を宣言することもできる

```
# type complex = float * float;;
# let c:complex = (3.0,-1.5);;
```

  - つまり，型を処理系に推論させることも，自分で表明して処理系に検査させることもできる
- 型に名前をつけなければならない場合もある
  - 例：型どうしの直和を作る場合 (次頁)

直和型 (Variant型)

例１： いろいろな形の図形を表す型

コンストラクタ (タグ)
(必ず大文字で始める)

```
# type shape =
   | Point
   | Circle of float
   | Rectangle of float * float;;

# let c = Circle 5.0;;
val c : shape = Circle 5.
# let r = Rectangle (10.0,7.0);;
val r : shape = Rectangle (10., 7.)
# let p = Point;;
val p : shape = Point
```

Point		
Circle	5.0	
Rectangle	10.0	7.0

直和型を引数にとる関数

- 図形の面積を求める

```
# let area s = match s with
  | Point -> 0.0
  | Circle r -> r *. r *. 4.0 *. atan(1.0)
  | Rectangle (w, h) -> w *. h;;
val area : shape -> float = <fun>
```

- このように，直和型を引数にとる関数は，match 式を使ってコンストラクタによって場合分けすることが多い．

再帰的な直和型

例２： 整数を要素とするリスト

```
# type intlist =
    | Nil
    | Cons of int * intlist;;
```

◆ 要素の型をパラメタ化することもできる

```
# type 'a mylist =
    | Nil
    | Cons of 'a * 'a mylist;;
```

- 組込みの型 `list` は，上の定義のコンストラクタ `Nil` と `Cons` を特別の記法（`[]`と `::`）で書くようにしたもの．

直和型 (Variant型)

例３： 列挙型も直和型の一種

```
# type grade = Aplus | A | B | C | F
```

例４： 組込みで用意されている option 型

```
# type 'a option = None | Some of 'a
```

- 結果があるかどうかわからない関数（探索など）の結果型として使うと便利（次頁）

連想リスト (associtation list) と Option 型

- 連想リスト = 「キーと値の組」のリスト
- 多くの応用をもつ KVS (key-value store) の原型
  - 例： [(2,"two"); (3,"three"); (5,"five")]

- 連想検索

```
# let rec assoc x = function
  | [] -> ???
  | (a,b)::t -> if a=x then b else assoc x t;;
```

  - 検索キーに対応するエントリがなかったらどうするか？

```
# assoc 4 [(2,"two"); (3,"three"); (5,"five")];;
```

連想リスト (assocition list) と Option 型

- 対応するエントリがなかったらどうするか？

```
# assoc 4 [(2,"two"); (3,"three"); (5,"five")];;
```

  - 案１： 例外を投げる（OCaml standard library の assoc）
    ➔ 制御の流れが関数的でなくなる
  - 案２： 答があるかないかわからない演算の結果は, option 型の値として返すのが現代的な方法

```
# let rec assoc x = function
  | [] -> None
  | (a,b)::t -> if a = x then Some b else assoc x t;;
```

連想リスト (associtation list) の変種

- オリジナルの連想リストはキーをソートしない
  - $(key, value)$ 対の挿入はリストの先頭に追加するだけ
  - 検索は最初に見つかったキーに対応する値を返す

```
[(3,"trois"); (2,"two"); (3,"three"); (5,"five")];;
```

- 代案１：既存の $(key, value)$ 対の更新を行う
- 代案２：既存の $(key, value)$ 対を活かしたまま新たな対の追加を行う
  - 例：書籍の索引
  - いずれにせよキーの索引管理が重要となる（次頁）

再帰的な直和型

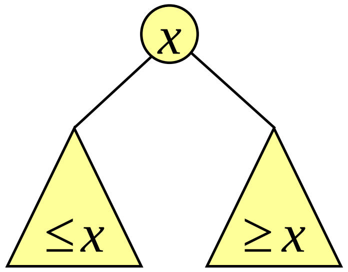
例５： 'a 型の要素を中間節点（NonTerminal node）にもつ二分木

```
# type 'a tree =
    | T
    | N of 'a tree * 'a * 'a tree;;

# N(N(T,2,T),3,N(N(T,5,T),7,T));;
- : int tree = N(N(T,2,T),3,N(N(T,5,T),7,T))
```

- 型の定義は関数の定義と異なり，再帰的であっても `type rec ...` と書く必要はない（cf. `let rec`）

二分木の操作の例




```
# let rec ins_key a = function
  | T -> N (T, a, T)
  | N (l, c, r) ->
      if a<=c then N ((ins_key a l), c, r)
      else N (l, c, (ins_key a r));;
val ins_key : 'a -> 'a tree -> 'a tree = <fun>

# let rec tree_of_list = function
  | []   -> T
  | h::t -> ins_key h (tree_of_list t);;
val tree_of_list : 'a list -> 'a tree = <fun>

# tree_of_list [3;1;4;1;5;9];;
- : int tree =
   (N(N(N(T,1,T),1,N(N(T,3,T),4,T)),5,T),9,T)
```

二分木の操作の例


```
# let rec list_of_tree = function
  | T -> []
  | N (l, c, r) ->
      list_of_tree l @ (c :: list_of_tree r);;
val list_of_tree : 'a tree -> 'a list = <fun>

# let sort a =
  | list_of_tree (tree_of_list a);;

# sort [3;1;4;1;5;9];;
- : int list = [1; 1; 3; 4; 5; 9]
```

数式の記号微分

- 数式を表現するデータ型を定義する

```
# type exp =
  | Num of int
  | Var of string
  | Add of exp * exp
  | Mul of exp * exp;;
```

- 例： $x \times (y + 3)$ を上の方式で表すと

```
Mul (Var "x", Add (Var "y", Num 3))
```

数式の記号微分

- deriv は（どの変数で微分したいかを指定する）変数と数式とをもらって，偏導関数を求める

```
# let make_sum a1 a2 = Add (a1,a2);;
# let make_product m1 m2 = Mul (m1,m2);;
# let rec deriv v = function
  | Num n -> Num 0
  | Var x -> if (Var x)=v then Num 1 else Num 0
  | Add (x, y) ->
      make_sum (deriv v x) (deriv v y)
  | Mul (x, y) ->
      make_sum
        (make_product x (deriv v y))
        (make_product (deriv v x) y);;
```

数式の記号微分

- 使ってみる

```
# let x = Var "x" and y = Var "y";;
# let t1 = Add (x, Num 3);;
# let t2 = Mul (x, y);;
# let t3 = Mul (t2, t1);;
# deriv x t1;;
# deriv x t2;;
...
```