

値の組 (tuple) を作る (cf. p.13)

- ◆ $(1, \text{"one"})$ は, 第 1 要素が 1 で第2要素が "one" であるような組 (tuple)
 - この組は $\text{int} * \text{string}$ 型 (直積型) に属する
 - 二つの値の組のことを対 (pair) と呼ぶ
- ◆ $(1, \text{"one"}, \text{"i chi"})$ のように 3 つ以上の値の組も作れる
 - OCaml では $(1, \text{"one"}, \text{"i chi"})$ と $(1, (\text{"one"}, \text{"i chi"}))$ は別物とされる. つまり
 $A * B * C$ と $A * (B * C)$ は別の型である

値の組 (tuple) を使う

- ◆ 関数から複数個の値を返すのに利用できる.

- 例: 複素数の積

```
let cm (x1, y1) (x2, y2) =  
  (x1*.x2-.y1*.y2, x1*.y2+.x2*.y1)
```

- ◆ 組込み関数 `fst`, `snd` が次のように定義されている

- `let fst (x, _) = x;;`

- `let snd (_, y) = y;;`

関数の型は？

- ◆ 複数個の引数をまとめてトレース出力を見やすくするのも使える

値の組 (tuple) の特別な場合

- ◆ 要素数 0 の組は, **unit 型**と呼ばれる.
 - 記法: `()`
 - cf. Java 等の `void`
 - 副作用 (effects) を伴うプログラミングでよく使われる
 - 例: `Printf.printf` 関数は `()` を返す
 - Cの0引数関数は実は `()` をもらっている
- ◆ 要素数 1 の組の型は OCaml には存在しない.

- ◆ let の等号の左辺にもパターンが書ける.

例 :

```
# let (x, y) = (1, 2);
```

```
# let (x, y) = (y, x);
```

```
# let (re, im) = cm (1.0, 2.0) (3.0, 4.0);;
```

- 「値を受け取るのにどこでもパターンマッチングが使える」が多くの関数型言語の基本設計思想.
 - let 定義の左辺, 関数の仮引数, 等

- ◆ 複数の名前を定義する 2 つの方法:

```
let [rec] <名前1> = <式1>  
    and <名前2> = <式2>  
    ...  
    and <名前n> = <式n>;;
```

p.12

```
let [rec] <名前1> = <式1>;;  
...  
let [rec] <名前n> = <式n>;;
```

- ◆ 前者は, $\langle \text{式}_1 \rangle \sim \langle \text{式}_n \rangle$ を一斉に評価してから
 $\langle \text{名前}_1 \rangle \sim \langle \text{名前}_n \rangle$ に一斉に結び付ける.
後者は, $\langle \text{式}_k \rangle$ の中で $\langle \text{名前}_1 \rangle \sim \langle \text{名前}_{k-1} \rangle$ に言及できる.
 - 補助関数を用いた関数定義や相互再帰関数の定義に前者の形式が役立つ.
- ◆ 以下を実行してみよう.
let x = 1 and y = 2;;
let x = y and y = x;;

- ◆ Java の局所変数宣言 (と初期値の設定) と同様に, OCaml でも let を使って, 値に一時的に名前をつけることができる.

`let [rec] <名前> = <式1> in <式2>`

- ◆ <式₁> を評価してその値を <名前> と結びつけたあとに <式₂> を評価. 全体が終わると <名前> が <式₁> の値を持っていたことは忘れ去られる.
- ◆ 例:
let f x = [x; x; x] in f (4.0 *. atan 1.0);;
let x = 4.0 *. atan 1.0 in [x; x; x];;

- ◆ 局所 let は, 無名関数に実引数を適用したものと本質的に同等. つまり

```
let x = 3*3 in [x; x; x]
```

は

```
(function x -> [x; x; x]) (3*3)
```

の syntactic sugar (構文上の便宜) にすぎない

- cf. Java のブロック

```
{ int x = 3*3; ... x ... x ... }
```


◆ match <式> with
| <パターン₁> -> <式₁>
| ...
| <パターン_n> -> <式_n>

は

```
(function  
| <パターン1> -> <式1>  
| ...  
| <パターンn> -> <式n>) <式>
```

と同じ.

◆ `if <式> then <式1> else <式2>`

は

```
(function  
| true  -> <式1>  
| false -> <式2>) <式>
```

と同じ．

◆ このように，OCaml の多くの重要機能が，無名関数とパターンマッチングを用いて説明できる．

- ◆ 局所的 let は、入れ子になっても良い。
では、次の式の値はそれぞれいくつか？

```
# let x=9 in  
  x * (let x = x/3 in x+x)
```

```
# let a=3 and b=4 in  
  let f x = a*x + b in  
    let a=5 and b=6 in f 2
```

- ◆ 無名関数 $\text{function } x \rightarrow a * x + b$ において
 - 最初の x は束縛する出現 (binding occurrence)
 - 2 個目の x は束縛された出現 (bound occurrence)
 - a と b は自由出現 (free occurrence)

という

- ◆ 無名関数 $\text{function } x \rightarrow a * x + b$ において
 - 最初の x は束縛する出現
 - 2 個目の x は束縛された出現
 - a と b は自由出現 という
- ◆ 束縛出現する名前は α 変換 で別の名前に変更可
 - ただし自由出現（上の例では a や b ）を束縛してはいけない。それ以外の名前には変更可
- ◆ 自由出現する名前の値は、無名関数の置かれた環境 (environment) に依存
 - 環境とは、名前と値とを対応付ける写像のこと

- ◆ リストの各要素に1引数関数 f を適用する高階 (higher-order) 関数のこと.

```
# let rec map f = function
  | [] -> []
  | h::t -> (f h) :: map f t;;
```

```
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

```
# map (fun x -> [x]) [1; 4; 7; 10];;
```

```
- : int list list = [[1]; [4]; [7]; [10]]
```

- ◆ 標準ライブラリの `List` モジュールに入っているので, 自分で定義しなくても `List.map` と書けば使える

◆ リストの和（積も同様）

```
# let rec sum a = match a with  
| [] -> 0  
| h::t -> h + sum t;;
```


◆ リストの連結

```
# let rec append a b = match a with  
| [] -> b  
| h::t -> h :: append t b;;
```

➔ パターンが非常に似ている（型は異なるが）

- ◆ sum と append の相違点をパラメタ化すると、
パターンの一般化ができる

```
# let rec fold_right f a unit t =  
  match a with  
  | [] -> unit  
  | h::t -> f h (fold_right f t unit);;
```



- ◆ $a = [a_1; a_2; \dots; a_n]$ と f と $unit$ をもらって
 $f\ a_1\ (f\ a_2\ (f\ a_3\ \dots\ (f\ a_n\ unit)\ \dots))$ を計算する関数になっている

畳み込み関数 fold_right

- ◆ $a = [a_1; a_2; \dots; a_n]$ と f と $unit$ をもらって

$$f\ a_1\ (f\ a_2\ (f\ a_3\ \dots\ (f\ a_n\ unit)\ \dots\))$$

を計算する関数になっている

- ◆ 中置記法で書くと

$$a_1\ f\ a_2\ f\ a_3\ \dots\ f\ a_n\ f\ unit$$

- ◆ 例

コメントと解釈されないための空白

- f が $(*)$ で $unit$ が 1 ならば数列の積
- $fold_right\ (-)\ [1; 2; 3; 4; 5]\ 0$ の値は？

◆ fold_right とは左右逆に

$f \dots (f (f (f \text{ unit } a_1) a_2) a_3) \dots a_n$

を計算する関数 fold_left は

```
# let rec fold_left f unit a =  
  match a with  
  | [] -> unit  
  | h::t -> fold_left f (f unit h) t;;
```

と書ける.

応用：多項式の評価（Horner 法）

- ◆ 多項式 $a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \cdots + a_1 x + a_0$ の値を, $O(n)$ 回の掛け算で求めたい. 係数は $[a_n; a_{n-1}; \dots; a_0]$ の形で与えられるものとする.
- ◆ Horner 法：
$$(\dots ((0x + a_n)x + a_{n-1})x + a_{n-2})x + \cdots + a_1)x + a_0$$
から規則性をみつける

応用：多項式の評価 (Horner 法)

◆ $(\dots(((0x + a_n)x + a_{n-1})x + a_{n-2})x + \dots + a_1)x + a_0$

```
# let horner coefficients x =
```

```
List.fold_left
```

```
(fun p a -> p * x + a)
```

```
0
```

```
coefficients;;
```

次の要素

中間結果

```
# let f = horner [1; 2; 3];;  $f(x) = x^2 + 2x + 3$ 
```

```
val f : int -> int = <fun>
```

- ◆ 2本のリストの対応する各要素を対にする

```
# let rec combine a1 a2 =  
  match (a1, a2) with  
  | ([], _) -> []  
  | (_, []) -> []  
  | (h1::t1, h2::t2) ->  
    (h1, h2) :: combine t1 t2;;
```

- ◆ `combine` を含めて, `List` モジュールの関数群の記述が <http://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html> にあるので見てみよう.