ML 演習 第 2 回

大岩
大山　永田
April 15, 2003

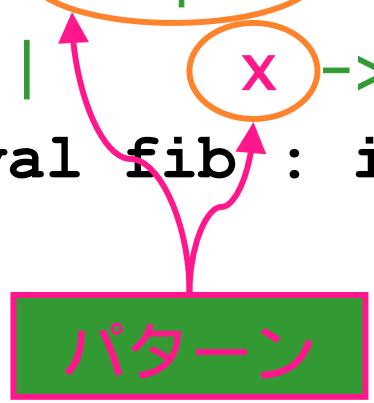
今回の内容

- パターンマッチ
- Parametric Polymorphism
  - リスト
  - 多相関数
  - ユーザ定義データ型
    - レコード型
    - バリアント型
  - 多相データ型と多相関数

パターンマッチ (1)

- 入力データとパターンを照合する

```
# let rec fib v = match v with
    0 | 1 -> 1
  |     x -> fib(x-1) + fib(x-2);;
val fib : int -> int = <fun>
```



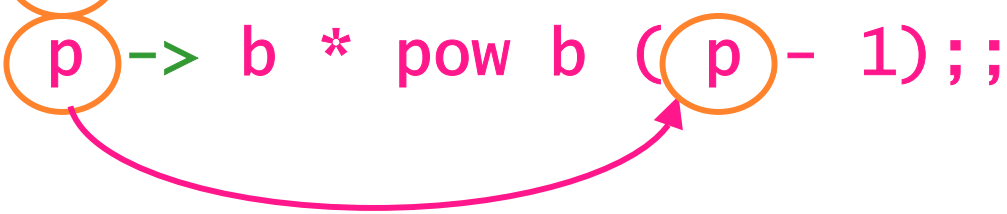
パターンマッチ (2)

- 様々なパターン
  - 定数 (整数、文字、文字列 etc.)
  - OR パターン
  - 変数束縛
  - ワイルドカード
  - ペア
  - コンストラクタ (リスト・ユーザ定義型など (後述)
  - これらの複雑な組み合わせ

パターンマッチ (3)

- 定数パターン
  - 定数と比較、一致すれば OK
- 変数束縛パターン
  - 任意の値にマッチ、対応する body 中でその値を使える

```
# let rec pow b p = match p with
    0 -> 1
  | p -> b * pow b (p - 1);;
```



パターンマッチ (4)

- OR パターン
  - 2つのパターンのどちらかにマッチすれば OK

```
# let rec fib v = match v with
    0 | 1 -> 1
  |     x -> fib(x-1) + fib(x-2);;
```

  - 変数束縛パターンと併用する場合、
    束縛する変数は同じでなければならない

パターンマッチ (5)

- pair パターン
  - 要素がそれぞれマッチすれば OK

```
# let f v = match v with
    0, y -> y
  | x, 0 -> x
  | x, y -> x * y;;
val f : int * int -> int = <fun>
```

パターンマッチ (6)

- ワイルドカード (_)
  - 値を捨てる
    - その値が計算に使われないことを明示できる

```
# let f x y = match x, y with
    0, y -> y
  | x, _ -> x;;
val f : int -> int -> int = <fun>
```

パターンマッチ (7)

- マッチできない値のある場合の警告

```
# let flip x = match x with
      0 -> 1 | 1 -> 0;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
2
val flip: int -> int
# flip 2;;
Uncaught exception: Match_failure ("", 13, 41).
```

パターンマッチ (8)

- 関数定義におけるパターンマッチ (1)
  - fun の引数部分もパターン
    - 引数は複数取れる
    - ただし選択肢は1つだけしか書けない

```
# let f (x, y) z = x + y * z;;
val f : int * int -> int = <fun>
# f (2, 3) 5;;
- : int = 17
```

パターンマッチ (9)

- 関数定義におけるパターンマッチ (2)
  - function: 1引数だが複数パターン

```
# let rec fib = function
    0 | 1 -> 1
  | x -> fib (x-1) + fib (x-2);;
val fib : int -> int = <fun>
# let rec pow b = function
      0 -> 1
    | x -> b * pow b (x - 1);;
val pow : int -> int -> int = <fun>
```

リスト (1)

- リストの基本要素
  - 空リスト

```
# [] ;;
- : 'a list = []
```

  - cons コンストラクタ

```
# 1 :: [] ;;
- : int list = [1]
# 1 :: (2 :: (3 :: [])) ;;
- : int list = [1; 2; 3]
```

リスト (2)

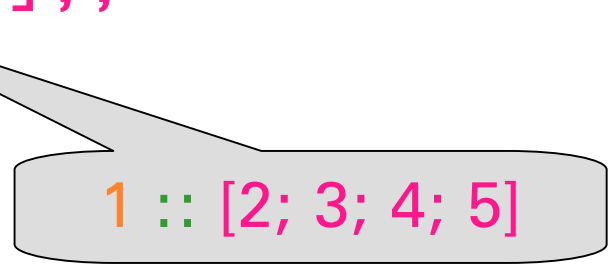
- リストの操作 (1): cons と append

```
# let l1 = [1; 2; 3; 4; 5];;
val l1 : int list = [1; 2; 3; 4; 5]
# -1 :: 0 :: l1;;
- : int list = [-1; 0; 1; 2; 3; 4; 5]
# l1 @ [6; 7];;
- : int list = [1; 2; 3; 4; 5; 6; 7]
```

リスト (3)

- リストの分解: パターンマッチを使う

```
# let rec sum l = match l with
          [] -> 0
        | hd :: tl -> hd + sum tl
val sum : int list -> int = <fun>
# sum [1; 2; 3; 4; 5];;
- : int = 15
```



型多相とは (1)

- 例1: リストの先頭要素を得る操作 hd
  - int_hd: int list -> int
  - bool_hd: bool list -> bool
  - string_hd: string list -> string
  - intpair_hd: (int * int) list -> int * int
  - etc…
    - 操作は共通: let hd (h::t) = h
      共通な定義は与えられない?

型多相とは (2)

- 解決: 「型」についてパラメータ化する
  - hd[α] : α list -> α
    - hd[int] : int list -> int
    - hd[string] : string list -> string
    - hd[int * bool] : (int * bool) list -> int * bool
  - ML では、[α] の部分は明示しなくてよい (型推論で自動的に解決)

多相関数 (1)

- 例1: 恒等関数

型変数 α などは
'a, 'b, ... と表記する

```
# let id x = x;;
id : 'a -> 'a = <fun>
# id 1;;
- : int = 1;;
# id [true; false];;
- : bool list = [true; false]
# id sum;;
- : int list -> int = <fun>
```

多相関数 (2)

- 例2: fst, snd : ペアの要素取り出し

```
# let fst (x, _) = x;;
val fst : 'a * 'b -> 'a = <fun>
# let snd (_, y) = y;;
val snd : 'a * 'b -> 'b = <fun>
```

多相関数 (3)

- 例3: mk_list

```
# let rec mk_list n v =
    if n = 0 then []
    else v :: mk_list (n-1) v;;
val mk_list: int -> 'a -> 'a list
# mk_list 3 "1";;
- : string list = ["1"; "1"; "1"]
# mk_list 2 true;;
- : bool list = [true; true]
```

多相関数 (4)

- 例3: rev : リストの反転

```
# let rev l =
    let rec iter s d =
      match s with
        [] -> d
      | (h::t) -> iter t (h::d)
    in iter l []
val rev: 'a list -> 'a list
# rev [1; 2; 3];;
- : int list = [3; 2; 1]
```

多相関数 (4')

- 例3: rev (解説)

```
  rev [1; 2; 3]
⇒ iter [1; 2; 3] []
⇒ iter [2; 3] [1]
⇒ iter [3] [2; 1]
⇒ iter [] [3; 2; 1]
⇒ [3; 2; 1]
```

```
let rev l =
  let rec iter s d =
    match s with
      []     -> d
    | (h::t) ->
        iter t (h::d)
  in iter l []
```

多相関数 (5)

- 例4: : map (高階関数)

```
# let rec map f l = match l with
      [] -> []
    | hd :: tl -> f hd :: map f tl;;
val map : ('a -> 'b) ->
            'a list -> 'b list = <fun>
# map fib [1; 2; 3; 4; 5];;
- : int list = [1; 1; 2; 3; 5]
```

多相関数 (6)

- 型の明示的な制限

```
# let f1 x = (x, x);;
val f1 : 'a -> 'a * 'a = <fun>
# let f2 (x:int) = (x, x);;
val f2 : int -> int * int = <fun>
# let f3 x = ((x, x):int * int);;
val f3 : int -> int * int = <fun>
# f3 "string"
This expression has type string ...
```

独自データ型の定義

- レコード (record)
  - 複数の値の組の型
    - C 言語の struct に相当
- バリアント (variant)
  - 複数の値の種類のうち1つを値とする型
    - C 言語の enum, union, cast などの組み合わせに相当
    - 操作の安全性をきちんと保証してくれる

レコード型 (1)

- 例: 複素数の直交座標表示

```
# type complex =
            {re : float; im : float};;
type complex =
      { re : float; im : float; }
# let c1 = {re = 5.0; im = 3.0};;
val c1 : complex
       = {re=5.; im=3.}
# c1.re;;
- : float = 5.
```

レコード型 (2)

- パターンマッチ

```
# let add_comp
        {re=r1; im=i1} {re=r2; im=i2}
    = {re = r1 +. r2; im = i1 +. i2};;
val add_comp = complex -> complex
                       -> complex = <fun>
# add_comp c1 c1;;
- : complex
          = {re=10.; im=6.}
```

バリアント型 (1)

- 例1: 整数をノードにもつ木

```
# type itree = Leaf
      | Node of int * itree * itree;;
type itree = Leaf
      | Node of int * itree * itree
# Leaf;;
- : itree = Leaf
# Node(5,Leaf,Leaf);;
- : itree = Node (5, Leaf, Leaf)
```

バリアント型 (2)

- 木のノードの値の合計を求める関数
  - パターンマッチで場合わけ

```
# let rec sum_itree = function
    Leaf -> 0
  | Node(a,t1,t2) ->
      a + sum_itree t1 + sum_itree t2;;
val sum_itree : itree -> int = <fun>
# sum_itree
  (Node(4,Node(5,Leaf,Leaf),Leaf));;
- : int = 9
```

多相データ型とは (1)

- 例2: 一般の「木」とは?
  - itree = Leaf | Node of int * itree * itree
  - btree = Leaf | Node of bool * btree * btree
  - ibtree = Leaf |
    Node of (int * bool) * ibtree * ibtree
  - 多相関数と同じような考え方が出来ないか?

多相データ型とは (2)

- 再び「型に関するパラメータ化」
  - 一般型
    - α tree = Leaf | Node of α * α tree * α tree
    - int tree = Leaf | Node of int * int tree * int tree
    - bool tree = Leaf |
      Node of bool * bool tree * bool tree
    - …

多相データ型 (1)

- 例2: 要素をノードにもつ木

```
# type 'a tree = Leaf | Node of
                 'a * 'a tree * 'a tree;;
  type 'a tree = Leaf | Node of
                   'a * 'a tree * 'a tree
# Node(5,Leaf,Leaf);;
- : int tree = Node (5, Leaf, Leaf)
# Node("ocaml",Leaf,Leaf);;
- : string tree =
          Node ("ocaml", Leaf, Leaf)
```

多相データ型 (2)

- システム組み込みの多相バリアント型
  - α option = None | Some of α
    - 「ないかもしれない」データを表す型
    - C では NULL ポインタの利用が一般的
  - α list = [] | :: of α * α list
    - 構文的にはちょっと特殊

多相データ型と多相関数

- (例) tree の深さを計算する関数

```
# let rec depth = function
    Leaf -> 0
  | Node(_,t1,t2) ->
    let d1 = depth t1 in
    let d2 = depth t2 in
    1 + max d1 d2;;
val depth : 'a tree -> int = <fun>
# depth(Node(5,Node(4,Leaf,Leaf),Leaf));;
- : int = 2
```

その他の構文 (1)

- コメント
  - (* と *) の間

```
# 1 + (* this is comment *) 2;;
- : int 3
```

  - 入れ子にできる

```
# 1 + (* 2 + (* 3 + *) 4 + *) 5;;
- : int = 6
```

その他の構文 (2)

- 中置演算子について
  - 実体は通常の2変数関数
  - ( ) で括ると通常の値として使える

```
# ( * );;
- : int -> int -> int = <fun>
# ( * ) 5 3;;
- : int = 15
```

課題1

- 引数の2リストを連結したリストを返す関数 append: α list → α list → α list を @ を使わずに定義せよ。

```
# append [1; 2] [3; 4; 5];;
- : int list = [1; 2; 3; 4; 5]
```

課題2

- 判定関数とリストを受け取り、元のリストの要素のうち条件を満たす要素だけからなるリストを生成する関数
  filter: (α → bool) → α list → α list
  を定義せよ。

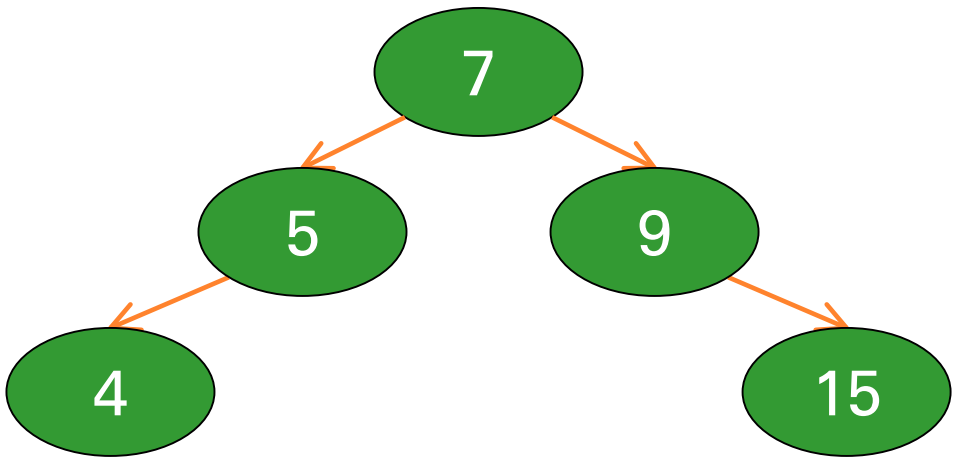
```
# filter odd [1; 2; 3; 4; 5];;
- : int list = [1; 3; 5]
```

課題3

- 木 (α tree) を受け取り深さ優先探索で、全要素を並べたリストを生成する関数 dfs: α tree → α list を定義せよ。
  - @ を使わなくてもできますが、
    この課題では使っても構いません。

課題3 (例)

```
# dfs(Node(7,
    Node(5,Node(4,Leaf,Leaf),Leaf),
    Node(9,Leaf,Node(15,Leaf,Leaf))));;
- : int list = [7; 5; 4; 9; 15]
```



課題4 (optional)

- 2項演算子 ⊕ と零元 z とリスト $[a_1; a_2; \ldots; a_n]$ を受け取り、右結合で結合させた結果 $a_1 \oplus (a_2 \oplus (\ldots \oplus (a_n \oplus z)\cdots))$ を返す関数 foldr: $(\alpha \to \beta \to \beta) \to \beta \to \alpha$ list $\to \beta$ を定義せよ。

```
# foldr (+) 0 [1; 2; 3; 4; 5];;
- : int = 15
```

課題4 (例)

```
# let flatten x = foldr (@) [] x;;
flatten : 'a list list -> 'a list
# flatten [[1;2]; [3;4]; [5;6;7]];;
- : int list = [1; 2; 3; 4; 5; 6; 7]
# let filter f x = foldr
   (fun x y -> if f x then x::y else y)
    [] y;;
filter : 'a list -> 'a list
```

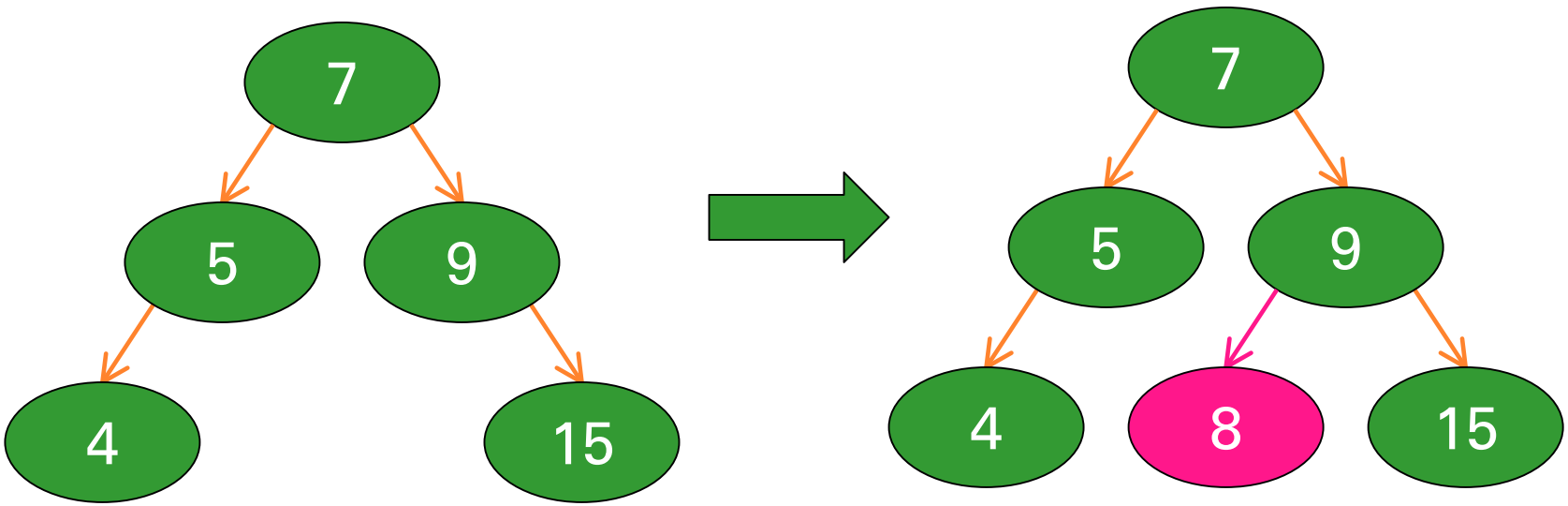
課題5 (optional)

- 課題3 (例) の木は、左の枝に含まれる要素はノードの値より小さく、右の枝に含まれる要素はノードの大きくなっている。このような木を2分探索木という。
- 2分探索木と新たな要素を与えられて、この要素を追加した木を返す関数 add_bst : α → α tree → α tree を定義せよ。

課題5 (例1)

```
# add_bst 8 (Node(7,
   Node(5,Node(4,Leaf,Leaf),Leaf),
   Node(9,Leaf,Node(15,Leaf,Leaf))));;
- : int tree =
Node
 (7, Node (5, Node (4, Leaf, Leaf),
              Leaf),
  Node (9, Node (8, Leaf, Leaf),
              Node (15, Leaf, Leaf)))
```

課題5 (例2)



課題の提出方法

- To: ml-report@yl.is.s.u-tokyo.ac.jp
- Subject: Report 2 *xxxxx* (学生証番号)
  - できるだけプログラムは
    本文に直接書いてください
- 〆切: 2002年4月29日(火) 24:00