ML演習 第２回

2007/06/12

飯塚 大輔, 後藤 哲志, 前田 俊行

今日の内容

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

今日の内容

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

パターンマッチとは?

- 値の「パターン」で処理を分岐させること

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

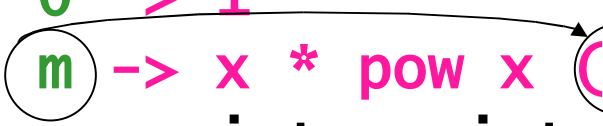
パターンの種類

- 定数パターン
- 変数束縛パターン
- OR パターン
- ペアパターン
- コンストラクタパターン (リストなど (後述))
- ワイルドカード

パターンの種類

- 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
  - マッチした値は本体中で使用可能

```
# let rec pow x n =
    match n with
      0 -> 1
    | m -> x * pow x (m - 1);;
val pow : int -> int -> int = <fun>
```



パターンの種類

- ORパターン
  - 複数のパターンのうち一つでもマッチすれば OK

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

パターンの種類

- Tuple パターン
  - Tuple の要素それぞれがマッチすれば OK
  - 変数束縛パターンと組合わせて要素の取出しが可能

```
# let scalar n p =
    match p with
    (x, y) -> (n * x, n * y);;
val scalar : int -> int * int -> int * int
# scalar 3 (2, 3);;
- : int * int = (6, 9)
```

パターンの種類

- ワイルドカード (_)
  - 任意の値にマッチ
    - マッチした値が計算に使われないことを明示している

```
# let is_vowel x =
    match x with
      'a' | 'e' | 'i' | 'o' | 'u' -> true
    | _ -> false;;
val is_vowel : char -> bool = <fun>
```

ガード

- パターンに条件を追加する仕組み

```
# let is_diag p =
    match p with
      (x, y) when x = y -> true
    | _ -> false;;
val is_diag : 'a * 'a -> bool = <fun>
# is_diag (3, 3);;
- : bool = true
# is_diag (3, 4);;
- : bool = false
```

関数定義におけるパターンマッチ

- let (rec) や fun の仮引数部分にはパターンを書ける

```
# let thd (_, _, z) = z;;
val thd : 'a * 'b * 'c -> 'c = <fun>
# thd (3, true, "hoge");;
- : string = "hoge"
# fun n (x, y) -> (n * x, n * y);;
- : int -> int * int -> int * int = <fun>
```

関数定義におけるパターンマッチ

- function式: 引数を 1 個受け取り パターンマッチを行う関数を作る

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

マッチできない値があるとき

- 警告が出る

2が来たときに対応するパターンがない

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

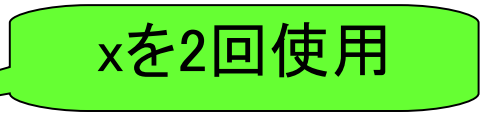
一応定義はされる

パターンマッチに失敗

パターンを書く上での注意

- 1 つのパターン中に同じ変数を
  2 回以上使うことはできない

```
# let is_diag p =
    match p with
      (x, x) -> true | _ -> false;;
This variable is bound several times in this
matching.
```



パターンを書く上での注意

- OR でつながったそれぞれのパターンでは束縛する変数が一致しなければならない

```
# let f p =
    match p with
      (1, x, y) | (2, x, _) -> x;;
Variable y must occur on both sides of this |
pattern.
```

| の右側で
yが束縛されていない

今日の内容

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

リストの基本要素

- 空リスト ([])

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

- Cons セル (::)

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

型に関する注意

- リストの要素はすべて同じ型でなければならない

```
# [1; 2; 3];;
- : int list = [1; 2; 3]
# [1; true; "hoge"];;
Characters 4-8:
  [1; true; "hoge"];;
      ^^^^
This expression has type bool but is
here used with type int.
```

リストの操作

- cons (::) と append (@) を使ったリスト操作

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

リストの分解

- パターンマッチを使って行う

```
# 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
```

今日の内容

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

型多相の必要性

- 例: リストの先頭要素を得る関数 hd
  - int_hd : int list -> int
  - bool_hd : bool list -> bool
  - string_hd : string list -> string
  - intpair_hd : (int * int) list -> int * int
  - …
    - 操作は `let hd (h::t) = h` のように型によらず共通なのでまとめてしまいたい

型毎に異なる関数を作らなければならない

型多相の必要性

- 解決法: 型をパラメタにしてしまう
  - hd[α] : α list -> α
    - hd[int] : int list -> int
    - hd[bool] : bool list -> bool
    - hd[int * int] : (int * int) list -> int * int
    - …

全部 hd[α] のインスタンスになっている

  - 実際にはαをプログラマが明示する必要はない
    - O'Caml の型推論が自動的に解決する

Parametric Polymorphism

- 型をパラメタにすることで
  型は異なるが本質的に同一なモノを
  ひとまとめにする方法
  - cf. オブジェクト指向の polymorphism
    (≒ subtyping polymorphism)
    - 例えば、Javaの継承
  - cf. overloading (ad-hoc polymorphism)
    - 例えば、OCamlの＝演算子

多相関数の例

- 恒等関数

型変数 α は
'a で表される

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

任意の型の値に
適用できる

多相関数の例

- ペアの要素を取り出す関数

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

多相関数の例

- 長さ n のリストを作る関数

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

多相関数の例

- 反転させたリストを返す関数

```
# let rec rev l =
    let rec iter s d =
      match s with
        [] -> d
      | hd :: tl -> iter tl (hd :: d)
    in iter l []
val rev : 'a list -> 'a list = <fun>
# rev [1; 2; 3];;
- : int list = [3; 2; 1]
```

多相関数の例

- リストの各要素に関数を適用して返す関数

```
# 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 [0; 1; 2; 3; 4];;
- : int list = [1; 1; 2; 3; 5]
```

型を明示的に指定する

- 多相型を持つ式，変数に対し型を明示できる

```
# 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 but...
```

今日の内容

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

独自データ型の定義

- レコード (record)
  - 複数の値を組にした値
    - C 言語の struct に相当
- バリアント (variant)
  - 何種類かの値のうち一つをとる値
    - C 言語の enum, union, cast などの組み合わせに相当
    - 操作の安全性が型検査により完全に保証される

レコード型

- 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.
```

フィールドのラベル名で
型を判別する

レコード型

- レコードに対するパターンマッチ

```
# 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

- 例: ノードに整数を持っている二分木

```
# type itree =
    Leaf
  | Node of int * itree * itree;;
type itree =
  Leaf | Node of int * itree * itree
# Leaf;;
- : itree = Leaf
# Node(1, Leaf, Leaf);;
- : itree = Node(1, Leaf, Leaf)
```

大文字で始まる識別子は
バリアント型のタグとみなされる

型の定義中にその型自身が
用いられるような型を
再帰型と呼ぶ

バリアント型

- 木のノードの値の合計を求める関数

```
# 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
```

今日の内容

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

多相データ型の必要性

- 二分木が持つ値は整数とは限らない
  - itree = Leaf | Node of int * itree * itree
  - btree = Leaf | Node of bool * btree * btree
  - ibtree = Leaf
    | Node of (int * bool) * ibtree * ibtree
  - ...
- データ構造としては同じなのでまとめたい
  - 多相関数と同じような考え方ができないか?

多相データ型の必要性

- 解決法: 再び「型のパラメタ化」
  - 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
    - ...

多相データ型の例

- ノードに要素を持つ二分木

```
# 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("str", Leaf, Leaf);;
- : string tree = Node("str", Leaf, Leaf)
```

多相データ型の例

- システム組み込みのバリアント型
  - `'a option = None | Some of 'a`
    - 「存在しないかもしれない値」を表す型
  - `'a list = [] | (::) of 'a * 'a list`
    - 構文がちょっと特殊

option 型の例

- 整数の割り算

```
# let div x y =
    if y = 0 then None
    else Some (x / y);;
val div : int -> int -> int option = <fun>
# div 4 3;;
- : int option = Some 1
# div 4 0;;
- : int option = None
```

多相データ型と多相関数

- 木の深さを返す関数

```
# 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>
```

2つ以上の型変数を含む多相データ型の定義

- 型の定義時に型変数をタプルの様に列挙する

```
# type ('a, 'b) pair = P of 'a * 'b;;
type ('a, 'b) pair = P of 'a * 'b
# P (42, "answer");;
- : (int, string) pair = P (42, "answer")
```

第2回 課題

締め切り: 6/26 13:00 (日本標準時)

課題1 (必須)

- 以下の関数を定義せよ
  - 引数の2リストを連結したリストを返す関数
    append: 'a list -> 'a list -> 'a list
    - ただし@をつかってはいけない
  - 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: ('a -> bool) -> 'a list -> 'a list

```
# filter isprime [1; 2; 3; 4; 5; 6; 7; 8; 9];;
- : int list = [2; 3; 5; 7];;
```

課題2 (必須)

- 2変数関数(⊕)と値 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
  $a_1 \oplus (a_2 \oplus (\ldots \oplus (a_n \oplus z)\ldots))$
  を返す関数
  `foldr: ('a -> 'b -> 'b) -> 'b → 'a list → 'b`
  を定義せよ。

課題2 (例)

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

- 中置演算子は括弧で記号を囲むことで関数として使える
  - 自分で定義することもできる

```
# let (%) x y = x mod y;;
val ( % ) : int -> int -> int
# 5 % 3;;
- : int = 2
```

課題3 (必須)

- リスト lst と整数 n を受け取り
  lst から要素をn個取り出す組み合わせを
  リストにして返す関数
  `comb : 'a list -> int -> 'a list list`
  を書け。

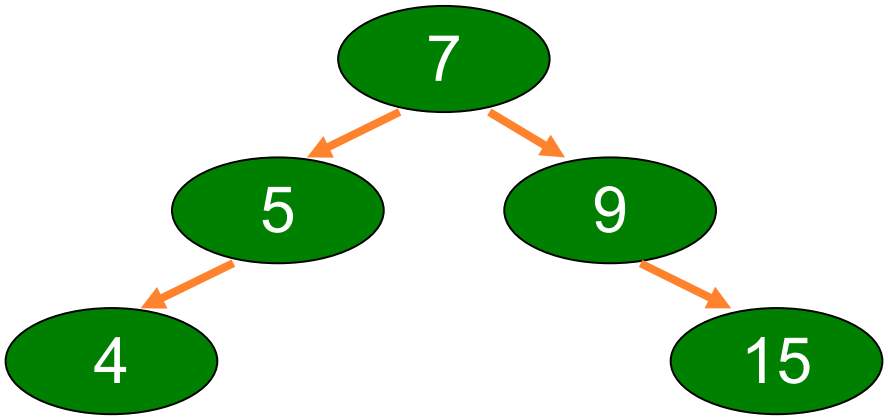
```
# comb [1; 2; 3] 2;;
- : int list list = [[1; 2]; [1; 3]; [2; 3]]
# comb [1; 2; 3] 0;;
- : int list list = [[]]
# comb [] 3;;
- : int list list = []
```

課題4 (必須)

- 木 (`'a tree`) を受け取り
  以下に挙げた探索順で全要素を並べたリストを生成する関数 `'a tree -> 'a list` を定義せよ
  - 行きがけ順 (preorder)
  - 通りがけ順 (inorder)
  - 帰りがけ順 (postorder)

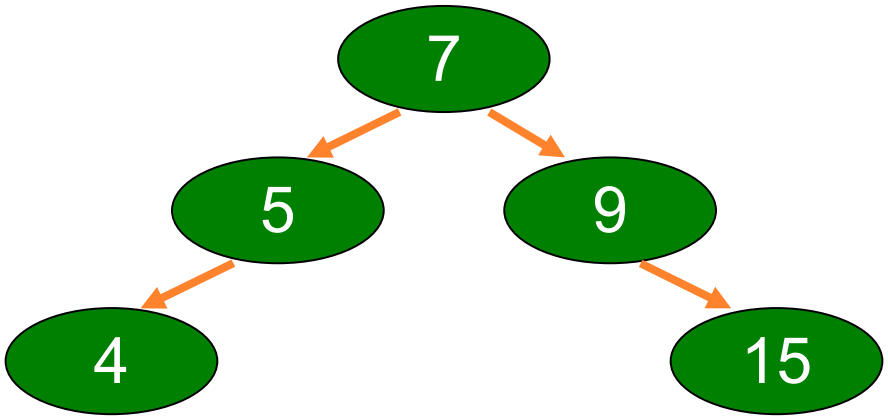
課題4 (例)

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



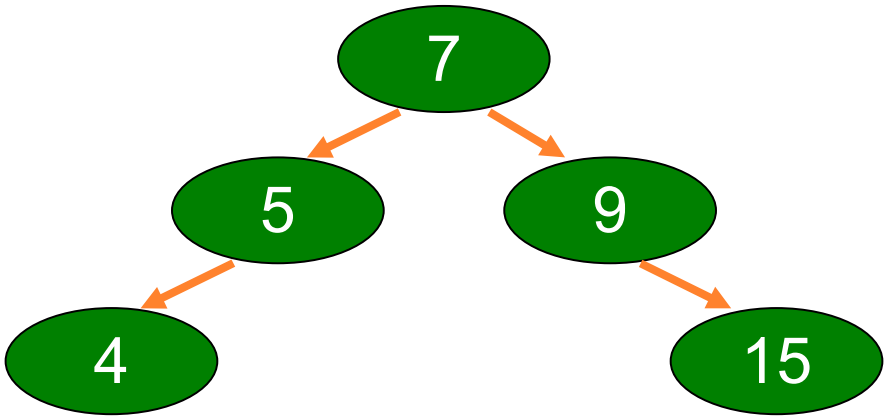
課題4 (例)

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



課題4 (例)

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



課題5 (Optional)

- 以下の条件を満たす2分探索木を実装してください
  - 文字列を格納できること
    - 入力方法は任意
    - 格納される文字列の最大長を仮定してはいけない
  - 削除機能を持つこと
  - 格納されているデータをアルファベット順にソートして表示する機能を持つこと
    - もちろん木のノードを通りがけ順に読み出せば自動的にソートされているはずである
  - ファイルにセーブする機能を持つこと
    - ファイルの構造は任意
    - 内容のみでなく構造も保存すること
  - 保存したファイルを読み込んで木を再構成する機能を持つこと

課題6 (Optional)

- 型 bot と型 not と型 or_type が以下のように定義されているとする

```
type bot = B of bot
type 'a not = 'a -> bot
type ('a, 'b) or_type = L of 'a | R of 'b
```

- このとき、次ページの1～6の型についてそれぞれの型を持つ値を let rec や再帰型を用いずに定義できるか?
  - 定義できる場合はその定義を示せ
  - let rec や再帰型を用いてよい場合はどうか?

課題6 (Optional)

1. `('a -> 'b) -> ('b -> 'c) -> ('a -> 'c)`
2. `('a, 'a not) or_type`
3. `('a, 'b * 'c) or_type`
   `-> ('a, 'b) or_type * ('a, 'c) or_type`
4. `('a, 'b) or_type * ('a, 'c) or_type`
   `-> ('a, 'b * 'c) or_type`
5. `('a * 'b) not`
   `-> ('a not, 'b not) or_type`
6. `('a not, 'b not) or_type`
   `-> ('a * 'b) not`