プログラミング演習B
ML編　第4回

2011/5/31（コミ）
2011/6/1（情報・知能）

住井

http://www.kb.ecei.tohoku.ac.jp/~sumii/class/proenb2011/ml4/

今日のポイント

1. 匿名関数、部分適用
2. レコードとレコード型
3. バリアントとバリアント型
   - パターンマッチング

レポートについて

電気・情報系内のマシンから

http://130.34.188.208/ （情報・知能）

http://130.34.188.209/ （コミ）

にアクセスし、画面にしたがって提出せよ。締め切りは一週間後厳守。

- 初回は画面にしたがい自分のアカウントを作成すること。
- 「プログラム」のテキストボックスがある課題では、プログラムとしてsmlに入力した文字列のみを過不足なく正確にコピー＆ペーストして提出せよ。（smlの出力は「プログラム」ではなく考察に含めて書くこと。）
- プログラムの課題でも必ず考察を書くこと。
- 提出したレポートやプログラムの実行結果は「提出状況」から確認できる。
  - 質問はml-enshu@kb.ecei.tohoku.ac.jpにメールせよ。
  - レポートの不正は試験の不正と同様に処置する。

復習：高階関数

- 「関数を引数として受け取る」あるいは「関数を結果として返す」ような関数
  - 「関数」自体を値として受け取ったり返したりできる

```
fun diff f =
  let
    fun f' x =
      (f (x + 0.001) - f x) / 0.001
  in
    f'
  end
```

匿名関数

letやfunで名前をつけて関数を定義するのが面倒なとき、

```
fn 引数名 => 式
```

という式により、名前をつけずに関数そのものを表すことができる。

例

```
- fun diff f =
=     fn x =>
=         (f (x + 0.001) - f x)
=                         / 0.001 ;
val diff = fn : (real ->
  real) -> real -> real
- (diff Math.exp) 1.0 ;
val it = 2.71964142253 : real
```

ちなみに…

- `fun succ x = x + 1`
  は
  `val succ = fn x => x + 1`
  と同じ
- `fun sum n =`
  `  if n=0 then 0 else sum(n-1)+n`
  は
  `val rec sum = fn n =>`
  `  if n=0 then 0 else sum(n-1)+n`
  と同じ
  - 再帰関数を`val`で定義するときは`rec`が必要

部分適用

「複数の引数をとる」関数に、
一部の引数だけ与えて使う

```
- fun distance x y =
=   Math.sqrt (x * x + y * y) ;
val distance = fn : real -> real -> real
- val distance1 = distance 1.0 ;
val distance1 = fn : real -> real
- distance1 1.0 ;
val it = 1.41421356237 : real
- distance1 2.0 ;
val it = 2.2360679775 : real
```

ちなみに…

- ```
  fun distance x y =
    Math.sqrt (x * x + y * y)
  ```
  は
  ```
  val distance =
    fn x =>
      fn y =>
        Math.sqrt (x * x + y * y)
  ```
  と同じ
  - MLでは、厳密には「複数の引数をとる関数」は存在せず、「関数を返す関数」の一つとして表現されている
  ```

以下はすべて同じ

- ```
  fun diff f =
      let fun f' x =
          (f (x + 0.001) - f x) / 0.001
      in f' end
  ```
- ```
  fun diff f =
      fn x =>
          (f (x + 0.001) - f x) / 0.001
  ```
- ```
  fun diff f x =
      (f (x + 0.001) - f x) / 0.001
  ```
- ```
  val diff = fn f => fn x =>
      (f (x + 0.001) - f x) / 0.001
  ```

課題４．１

1. 前回の課題３．３の関数deltaを、letを使わずfnで書け。
2. 前回の例題の関数composeを、letを使わずfnで書け。
3. 前回の課題３．６の関数のそれぞれを、funを使わずvalとfnで書け。

基本データ型と構造データ型

int, real, bool, stringなどの
「基本データ型」だけでは
複雑なデータを表現できない

⇒　基本データ型を合成した
「構造データ型」を利用する

- 「XおよびY」のような「組み合わせ」を表す**レコード型**（C言語の構造体に相当）
- 「XまたはY」のような「場合わけ」を表す**バリアント型**（C言語の共用体にほぼ相当）

レコードとレコード型

- 「レコード」：1つ以上の値にラベルをつけて組み合わせた値
- 「レコード型」：レコードの型（1つ以上の型にラベルをつけて組み合わせた型）
  - 組み合わせる値や型の順序は影響しない

```
- val r = { surname = "Sumii",
=            given = "Eijiro", age = 20 } ;
val r =
  {age=20,given="Eijiro",surname="Sumii"}
  : {age:int, given:string, surname:string}
- #age r ;
val it = 20 : int
```

構文

- レコード

  { ラベル$_1$ = 式$_1$, …, ラベル$_n$ = 式$_n$ }

- レコード型

  { ラベル$_1$: 型$_1$, …, ラベル$_n$: 型$_n$ }

- フィールドの取り出し

  #ラベル 式

  - 「フィールド」：ラベルをつけてレコードとして組み合わされた値

課題4．2

1. 先の例のように、自分の苗字と名前と年齢を組み合わせて、レコードとして表せ。
2. そのレコードからフィールドgivenとsurnameを取り出し、文字列連結の二項演算子^を用いて、

   "Eijiro Sumii"

   のように「名前スペース苗字」という形の文字列を作る、という式を書け。

ちょっと微妙な注意...

- SMLでは、たとえば先の例のようなレコードxから、フィールドageを取り出す関数fを定義しようとすると、

```
- fun f x = #age x ;
stdIn:17.1-17.17 Error: unresolved flex record
   (can't tell what fields there are besides #age)
```

というエラーになる。このような場合は

```
- fun f (x : {surname:string,given:string,age:int}) =
=   #age x ;
val f = fn : {age:int, given:string, surname:string}
   -> int
```

のようにxの型を指定しなければならない。

- ちなみにOCamlでは、レコードやレコード型はあらかじめ定義する必要があるので、上述のような問題はない。

余談

- SMLを拡張したSML#という言語には、
  「レコード多相」(record polymorphism)
  という機能があり、前述のような問題はない。
  (http://www.pllab.riec.tohoku.ac.jp/smlsharp/ja/)

```
eiw01 % smlsharp
SML# 0.30 (2007-07-03 16:16:12 JST)
# fun f x = #age x ;
val f = fn : ['a,'b#{age:'a}.'b  -> 'a]
# f { surname = "Sumii", given = "Eijiro",
>     age = 20 } ;
val it = 20 : int
```

バリアントとバリアント型

- 「バリアント」：いくつかの値のうち、いずれか一つをとるような値
  - どの一つであるか、「コンストラクタ」という名札（タグ）で区別する
- 「バリアント型」：バリアントの型

例1：曜日を表すバリアント

```
- datatype day =
=    Sun | Mon | Tue | Wed | Thu
= | Fri | Sat ;
datatype day = Fri | Mon | Sat | Sun |
  Thu | Tue | Wed
- Sun ;
val it = Sun : day
- Mon ;
val it = Mon : day
- Sat ;
val it = Sat : day
```

例2：二者択一を表すバリアント

```
- datatype binary = Yes | No ;
datatype binary = No | Yes
- Yes ;
val it = Yes : binary
- No ;
val it = No : binary
```

注: bool型の値true, falseも

```
datatype bool = true | false
```

のようなバリアントとみなせる

例3：「整数またはエラー」を表すバリアント

```
- datatype int_or_error =
=    Int of int | Error ;
datatype int_or_error = Error | Int
  of int
- Int 123 ;
val it = Int 123 : int_or_error
- Int 45 ;
val it = Int 45 : int_or_error
- Error ;
val it = Error : int_or_error
```

例3：「整数またはエラー」を表すバリアント（続き）

```
- fun my_div x y =
=    if y = 0 then Error else
=    Int (x div y) ;
val my_div = fn : int -> int ->
  int_or_error
- my_div 10 3 ;
val it = Int 3 : int_or_error
- my_div 10 0 ;
val it = Error : int_or_error
```

例4：整数を葉とする木を表すバリアント

```
- datatype itree =
=   ILeaf of int
= | INode of { left : itree,
=              right : itree } ;
datatype itree = ILeaf of int | INode of
  {left:itree, right:itree}
- ILeaf 3 ;
val it = ILeaf 3 : itree
- INode { left = it, right = it } ;
val it = INode {left=ILeaf 3,right=ILeaf
  3} : itree
- INode { left = it, right = ILeaf 7 } ;
val it = INode {left=INode {left=ILeaf
  #,right=ILeaf #},right=ILeaf 7} : itree
```

構文

バリアント型の定義：

```
datatype 型の名前 =
    コンストラクタ1 of 引数の型1
  | コンストラクタ2 of 引数の型2
  | …
  | コンストラクタn of 引数の型n
```

- 引数をとらないコンストラクタは of以降を省略する

構文（続き）

バリアント型の値を作る式：

コンストラクタ 引数

- 関数適用と同じく並べて書く
  - というか、SMLではコンストラクタも関数の一種とみなされる
- 定義のときにof以降を省略したコンストラクタは引数をとらない

課題4．3

1. my_divにならって、整数の割り算の余りを求める関数my_modを書け。除数が0のときはErrorを返すこと。
2. 文字列を葉とする木を表すバリアント型streeを定義し、木の例をいくつか作れ。

パターンマッチング

バリアントがどの値であるか という「場合わけ」

例題：先の「曜日を表すバリアント」を受け取って、休日だったらtrue、平日だったらfalseを返す関数holidayを書け。

例題の解答

```
- fun holiday x = case x of
=     Sun => true | Mon => false
=   | Tue => false | Wed => false
=   | Thu => false | Fri => false
=   | Sat => true ;
val holiday = fn : day -> bool
- holiday Sun ;
val it = true : bool
- holiday Mon ;
val it = false : bool
- holiday Fri ;
val it = false : bool
- holiday Sat ;
val it = true : bool
```

例題2

先の「整数を葉とする木」を表すバリアントを受け取り、すべての葉の整数の合計を返す関数isumを書け。

例題２の解答

```
- fun isum t = case t of
=   ILeaf i => i
= | INode r => isum (#left r) +
=              isum (#right r) ;
val isum = fn : itree -> int
- isum (ILeaf 3) ;
val it = 3 : int
- isum (INode { left = ILeaf 3,
=                right = ILeaf 7 }) ;
val it = 10 : int
```

構文

case 式 of

　　コンストラクタ$_1$ 引数$_1$ => 式$_1$

| コンストラクタ$_2$ 引数$_2$ => 式$_2$

…

| コンストラクタ$_n$ 引数$_n$ => 式$_n$

↑

コンストラクタが引数をとらない場合は省略

課題４．４

先の「二者択一を表すバリアント」を受け取って、Yesに対してはNoを、Noに対してはYesを返す関数hinekureを書け。また、その関数の働きを実際に確かめよ。

課題4．5

次の考え方に基づいて、先の「整数を葉とする木を表すバリアント」tを受け取って、その葉の個数を返す関数sizeを書け。

- tがILeaf iの形だったら1を返す
- tがINode rの形だったら、左の木の葉の個数であるsize(#left r)と、右の木の葉の個数であるsize(#right r)の和を返す

課題4.6 (optional)

次の考え方に基づいて、先の「整数を葉とする木を表すバリアント」tを受け取って、その木の高さを返す関数heightを書け。

- tがILeaf iの形だったら0を返す
- tがINode rの形だったら、左の木の高さと、右の木の高さのうち、より小さくないほうに1を加えて返す
  - 無駄な計算をしないように注意せよ