

プログラミング演習B

ML編 第4回

2010/6/29 (コミ)

2010/6/30 (情報・知能)

住井

[http://www.kb.ecei.tohoku.ac.jp/
~sumii/class/proenb2010/ml4/](http://www.kb.ecei.tohoku.ac.jp/~sumii/class/proenb2010/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. バリエーションとバリエーション型
 - 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m1-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

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**バリエーション型**（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せられた値

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、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
```

バリエーションとバリエーション型

- 「バリエーション」：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
- 「バリエーション型」：バリエーションの型

例 1 : 曜日を表すバリエーション

```
- 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エانت

```
- 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
のようなバリエانتとみなせる

例3：「整数またはエラー」を表すバリエーション

```
- 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 バリエーション（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 バリエーション

```
- 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

バリエーション型の定義：

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）

バリエーション型の値を作る式：

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. 「浮動小数点数またはエラー」を表すバリエーション型`real_or_error`を定義し、浮動小数点数の平方根を求める関数`my_sqrt`と、自然対数を求める関数`my_ln`を書け。引数が定義域外の場合はErrorを返すこと。
3. 文字列を葉とする木を表すバリエーション型`stree`を定義し、木の例をいくつか作れ。

パターンマッチング

バリエーションがどの値であるか
という「場合わけ」

例題：先の「曜日を表すバリエーション」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
バリエーションを受け取り、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エーション」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エーション」`t`を受け取って、その葉の数を返す関数 `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、左の木の葉の数である `size(#left r)` と、右の木の葉の数である `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エーション」`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、より小さいほうに1を加えて返す