ML 演習 第3回

おおいわ

June 13, 2000

今回の内容

- 例外の処理
- 逐次プログラミング
  - reference
  - 複文
  - 変更可能フィールド

例外 (1)

例1: ベクトルの正規化

```
# let normalize (x1, x2) =
    let n = sqrt (x1 *. x1 +. x2 *. x2)
    in (x1 /. n, x2 /. n);;
val normalize : float * float -> float * float = <fun>
# normalize (3.0, 4.0);;
- : float * float = 0.600000, 0.800000
```

→ (0, 0) が与えられたとき?

例外 (2)

- 2つのベクトルのなす角

```
# let angle v1 v2 =
    let ((x,y),(x ',y ')) = (normalize v1, normalize v2)
    in acos (x *. x '+. y *. y ');;
val angle : float * float -> float * float -> float = <fun>
# let degree_of_radian x = x *. 180.0 /. 3.1415926535897;;
val degree_of_radian = float -> float = <fun>
# let angleD v1 v2 = degree_of_radian (angle v1 v2);;
val angleD : float * float -> float * float -> float = <fun>
# angleD (1.0, 0.0) (0.0, 0.5);;
- : float = 90.000000
```

例外 (3)

方法1: エラーを示す値を決めておく

```
# type 'a err = Error | Good of 'a;;
type 'a err = Error | Good of 'a
# let normalize (x1, x2) =
    let n = sqrt (x1 *. x1 +. x2 *. x2) in
    if n = 0.0 then Error else Good(x1 /. n, x2 /. n);;
val normalize : float * float -> (float * float) err = <fun>
# normalize (0.0, 0.0);;
- : (float * float) err = Error
```

例外 (4)

- この方法の欠点: 使いづらい!!

```
# let angle v1 v2 =
    match  (normalize v1, normalize v2) with
     (Error, _) -> Error
   | (_, Error) -> Error
   | (Good(x,y), Good(x',y')) ->
                        Good(acos(x *. x' +. y *. y'));;
val angle : float * float -> float * float -> float err = <fun>
# let angleD v1 v2 = match angle v1 v2 with
   Error -> Error | Good x -> Good (degree_of_radian x);;
```

例外の送出 (1)

例外の定義と送出 (1)

```
# exception ZeroVector;;
exception ZeroVector
# raise ZeroVector;;
Uncaught exception: ZeroVector.

# exception BadArg of float;;
exception BadArg of float
# raise (BadArg -5.0);;
Uncaught exception: BadArg 5.000000.
```

例外の送出 (2)

例外の利用

```
# let normalize (x1, x2) =
    let n = sqrt (x1 *. x1 +. x2 *. x2) in
    if n = 0.0 then raise ZeroVector else (x1 /. n, x2 /. n);;
val normalize : float * float -> float * float = <fun>
# normalize (3.0, 4.0);;
- : float * float = 0.600000, 0.800000
# normalize (0.0, 0.0);;
Uncaught exception: ZeroVector.
```

例外の送出 (3)

```
# let angle v1 v2 =
    let ((x,y),(x ',y )) = (normalize v1, normalize v2)
    in acos (x *. x '+. y *. y );;
val angle : float * float -> float * float -> float = <fun>
# let angleD v1 v2 = degree_of_radian (angle v1 v2);;
val angleD : float * float -> float * float -> float = <fun>

# angleD (1.0, 0.0) (0.0, 0.5);;
- : float = 90.000000
# angleD (0.0, 0.0) (0.0, 0.5);;
Uncaught exception: ZeroVector.
```

例外の処理 (1)

- 発生した例外を処理する

```
# let angle_str v1 v2 = try
      “Angle is ” ^ string_of_float (angleD v1 v2)
  with ZeroVector -> “Not defined. ”;;
val angle_str : float * float -> float * float -> string = <fun>

# angle_str (1.0, 0.5) (2.0, 3.0);;
- : string = "Angle is 29.7448812969"
# angle_str (1.0, 0.5) (0.0, 0.0);;
- : string - “Not defined. ”
```

例外の処理 (2)

- 応用例

```
# exception Zero;;
# let prod l =
    let rec f = function [] -> 1
      | hd::tl -> if hd = 0 then raise Zero else hd * f tl
    in try f l with Zero -> 0;;
val prod : int list -> list
# prod [1;2;3;4;5;0;7;8;0];;
- : int = 0
```

逐次プログラミング

- **副作用に依存したプログラミングのサポート**
  - reference (**破壊的代入**)
  - **複文**

reference

- **変更可能なセル**

```
# val a = ref 0;;
val a : int ref = {contents=0}
# !a;;
- : int = 0
# a := 5;;
- : unit = ()
# !a;;
- : int = 5
```

変更可能なレコード

```
# type mutable_point = { mutable x:int; mutable y:int };;
type mutable_point = { mutable x : int; mutable y : int; }
# let p1 = { x = 5; y = 3; };;
val p1 : mutable_point = {x=5; y=3}
# p1.x <- 6;;
- : unit = ()
# p1;;
- : mutable_point = {x=6; y=3}
```

(cf.)　type 'a ref = { mutable contents : 'a }

複文

```
# let increment x a = (x := !x + a ; !x);;
val increment : int ref -> int -> int = <fun>
# let a = ref 0;;
val a : int ref = {contents=0}
# increment a 5;;
- : int = 5
# increment a 5;;
- : int = 10
```

(参考) unit 型

- () が唯一の値

```
# ();;
- : unit = ()
```

- 用途
  - 副作用以外に意味のない関数の返り値
  - 引数の不要な関数に与えるダミー値

課題1

- 暗証番号と残高を持ち、暗証が一致したときに残高を増減できる「銀行口座」を作成する関数を実装せよ。
  - 内部データ構造は自由に設計してよい。
  - 内部データ構造によっては暗証番号が見えてしまうが今回それは気にしなくてよい。
  - システム中に複数の銀行口座を作れるように設計すること。

課題1（例）

```
# let ac = make_account “password ”10000;;
val ac : account = ...
# get ac “password ”3000;;
- : int = 7000
# get ac “password ”3000;;
Uncaught exception: Invalid_password.
# put ac 10000;;
- : int = 17000
# get ac “password ”20000;;
Uncaught exception: Not_enough_money.
# tell ac;;
- : int = 17000
```

課題1 (ヒント)

- 内部データ構造に何を使うか？
  - レコード型のオブジェクト
  - 関数オブジェクト

```
# ac “password ”(-12000);;
- : int = 5000
# ac “”5000;;
- : int = 10000
```

課題2

- 課題1 の銀行口座システムを改良し、これまで作成した全口座を記憶して、それらに1割の利息を一斉に付加する関数を実装せよ。

課題2 (例)

```
# let ac1 = make_account "password" 10000
  and ac2 = make_account "foobar" 5000;;
val ac1 : account = ...
val ac2 : account = ...
# add_interests_for_all ();;
- : unit = ()
# tell ac1, tell ac2;;
- : int * int = 11000, 5500
```

提出方法

- **〆切**: 2000年6月26日 (月) 24:00
- **提出先**: ml-report@yl.is.s.u-tokyo.ac.jp
- **題名**: “Report 3 (**学生証番号**) ”