



ML 演習 第 3 回

おおいわ
おおやま ながた
April 22, 2003

今回の内容

- 例外の処理
- 副作用のサポート
 - reference
 - 変更可能フィールド
 - 複文
- OCaml の等値演算子と比較演算子

例外 (1)

■ 例1: ベクトルの正規化

```
# let normalize (x, y) =  
  let n = sqrt (x *. x +. y *. y) in (x /. n, y /. n);;  
val normalize : float * float → float * float = <fun>  
# normalize (3.0, 4.0);;  
- : float * float = (0.6, 0.8)
```

→ (0, 0) が与えられたときの処理は?

例外 (2)

■ (例1続き) 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 angle_deg v1 v2 = degree_of_radian (angle v1 v2);;  
val angleD : float * float → float * float → float = <fun>  
# angle_deg (3.0, 1.0) (1.0, 2.0);;  
- : float = 45.
```

例外 (3)

■ 方法1: エラーを示す値を決めておく

```
# type 'a option = None | Some of 'a;
```

```
type 'a option = None | Some of 'a
```

```
# let normalize (x1, x2) =
```

```
  let n = sqrt (x1 *. x1 +. x2 *. x2) in
```

```
  if n = 0.0 then None else Some(x1 /. n, x2 /. n);;
```

```
val normalize : float * float -> (float * float) option = <fun>
```

```
# normalize (0.0, 0.0);;
```

```
- : (float * float) option = None
```

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  
    (None, _)  
  | (_, None) → None  
  | (Some(x,y), Some(x',y')) → Some(acos(x *. x' +. y *. y'));;  
val angle : float * float -> float * float -> float option = <fun>  
# let angle_deg v1 v2 =  
  match angle v1 v2 with  
    None → None  
  | Some x → Some (degree_of_radian x);;
```

例外の送出 (1)

■ 方法2: 例外機構を使う

```
# exception ZeroVector;;          (* 例外を定義 *)
```

```
exception ZeroVector
```

```
# raise ZeroVector;;             (* 例外を送出 *)
```

```
Exception: ZeroVector.
```

```
# exception BadArg of float;;    (* 引数を持つ例外 *)
```

```
exception BadArg of float
```

```
# raise (BadArg 5.0);;
```

```
Exception: BadArg 5..
```

例外の送出 (2)

■ 例題プログラムの改良

```
# let normalize (x1, x2) =  
  let n = sqrt (x1 *. x1 +. x2 *. x2) in  
  if n = 0.0 then raise ZeroVector (* 例外を送出 *)  
  else (x1 /. n, x2 /. n);;  
  
val normalize : float * float → float * float = <fun>  
# normalize (3.0, 4.0);;  
- : float * float = 0.6, 0.8  
# normalize (0.0, 0.0);;  
Exception: ZeroVector.
```


例外の送出 (3)

```
# let angle v1 v2 = (* 例外処理は行わない *)  
    let ((x,y),(x',y')) = (normalize v1, normalize v2)  
    in acos (x *. x' +. y *. y');;  
val angle : float * float → float * float → float = <fun>  
# let angle_deg v1 v2 = degree_of_radian (angle v1 v2);;  
val angleD : float * float → float * float → float = <fun>
```

```
# angle_deg (3.0, 1.0) (1.0, 2.0);;
```

```
- : float = 45.
```

```
# angle_deg (0.0, 0.0) (0.0, 0.5);;
```

```
Exception: ZeroVector.
```

- normalize で発生した例外が伝播されて返ってくる

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 (angle_deg v1 v2)
with ZeroVector -> "Not defined.;;

val angle_str : float * float -> float * float -> string = <fun>

# angle_str (3.0, 1.0) (1.0, 2.0);;
- : string = "Angle is 45."
# angle_str (1.0, 0.5) (0.0, 0.0);;
- : string - "Not defined."
```

例外の処理 (2)

■ 応用例: 大域脱出への応用

```
# 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
```

- 0だとわかった時点で無駄な掛け算をせずに返ってくる

Imperative Features

- 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)
 - 変更可能フィールド
 - 複文

Reference

■ 変更可能なセル (「箱」)

```
# let 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
```

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
 - C++ の void 型に相当

Value restriction (1)

- 型多相と reference は相性が悪い
 - (実際には存在しない) 例

```
# let r1 = ref [];;
```

```
val r1 : 'a list ref = { contents = <fun> }
```

```
# let f () = List.map not (!r1);;
```

```
val f : unit → bool = <fun>
```

```
# r1 := [5];;
```

```
- : unit = ()
```

```
# f ();;
```

- どこがおかしい？

Value restriction (2)

- 'a list ref に int list を代入?

- 根本的な原因ではあるが
代入禁止だけでは防げない

```
# let r1 = ref (fun x → x);;
```

```
# let set_r1 f x = r1 := f; !r1 x;;
```

```
val set_r1 : ('a → 'a) → 'a → 'a = <fun>
```

```
# let twice f x = f (f x);;
```

```
val func1 : ('a → 'a) → 'a → 'a = <fun>
```

- 同じ型に見える...

Value restriction (3)

- (続き)

```
# twice (fun x → x + 1) 5;; (* (A) *)
```

```
- : int = 7
```

```
# set_r1 (fun x → x) 5;; (* これは通らないと困る *)
```

```
- : int = 5
```

```
# set_r1 (fun x → x + 1) 5;; (* (A) と同じ使い方 *)
```

```
- : int = 6 ... だけど同じ問題を引き起こす
```

- 結局、ML のシステムと整合を取って参照に多相型を与えるのは難しい

Value restriction (4)

- とりあえずの解決: reference には「未決定の単相型」を与える

```
# let r1 = ref [];;
```

```
val r1 : 'a list ref = { contents = <fun> }
```

```
# let f1 () = List.map not !r1;;
```

```
val f1 : unit -> bool list = <fun>
```

```
# let _ = !r1 @ [5];;
```

This expression has type int but is here used with type bool

```
# !r1;;
```

```
- : bool list = []
```

Value restriction (5)

- 更なる問題: $\text{unit} \rightarrow 'a \rightarrow 'a$ 型の値に
() を apply した結果の型は?
 - 自然な $\text{fun } () \rightarrow (\text{fun } x \rightarrow x)$ の場合を考えると
 $'a \rightarrow 'a$ としたい
 - 次の f の場合単相型関数 $'_a \rightarrow '_a$ しかだめ
let f () = let r = ref None in
fun x →
let old = match !r with
None → x | Some y → y
in r := Some x; old

Value restriction (6)

- MLでの解決: 副作用がないと確実にわかる値にのみ多相型を与える
 - OK: 定数, fun 式, それらの tuple, それらからなる変更不可データ構造
 - NG: reference, let 式, 関数適用 etc...

```
# (fun () x -> x) ();;
```

```
- : '_a → '_a = <fun>
```

Value restriction (7)

■ 注意点:

- 部分適用が単相型になることがある

```
# let f = List.map (fun x -> (x, x));;
```

```
val f : 'a list -> ('a * 'a) list = <fun>
```

- 解決: η 展開

```
# let f xs = List.map (fun x -> (x, x)) xs;;
```

```
val f : 'a list -> ('a * 'a) list = <fun>
```

Value restriction (8)

■ 参考文献

- 最初に Value restriction を提案した論文
 - Andrew K. Wright, Matthias Felleisen.
A Syntactic Approach to Type Soundness.
 - 読みやすいので学部生にもおすすめ
 - Value restriction 以外の解決法との得失比較あり
- 最近提案されている OCaml の拡張
 - Jacques Garrigue. Relaxing Value Restriction.
 - OCaml 3.07 で採用予定

等値演算子 (1)

■ 2つの等値演算子

- =: 「構造的な一致」 (否定: <>)
 - 複合データの中身まで探索する
 - Scheme の equal? に相当
- ==: 「物理的な一致」 (否定: !=)
 - 「アドレス」しか見ない
 - Scheme の eq? に相当
- == の方が識別力が強い

等値演算子 (2)

```
# let test x y = (x = y, x == y)
```

```
val test : 'a → 'a → bool * bool
```

```
# test 1 1;;
```

```
- : bool * bool = true, true
```

```
# test 1.0 1.0;; (* 一般的には = じゃないとまずい *)
```

```
- : bool * bool = true, false
```

```
# test "string" "string";; (* 同上 *)
```

```
- : bool * bool = true, false
```

等値演算子 (3)

test (ref 1) (ref 1);; (* Reference では == も重要 *)

- : bool * bool = true, false

let r = (ref 1) in test r r;;

- : bool * bool = true, true

(fun x → x) = (fun x → x);;

Exception: Invalid_argument "equal: functional value".

(fun x → x) == (fun x → x);;

- : bool = false

let f = (fun x → x) in test x x;;

- : bool = true, true

比較演算子

- 比較演算子 $<$, $>$, $<=$, $>=$
 - 型: $\alpha \rightarrow \alpha \rightarrow \text{bool}$
 - $=$ と対応した比較演算子
 - 整数・実数 \rightarrow 数値比較
 - 文字列・文字 \rightarrow 辞書順比較
 - その他のオブジェクト \rightarrow 実装依存
 - 基本的には要素の辞書順
 - 注: 循環参照を持つデータでは止まらないことがある

課題1

- “Turtle” のモデルとして、現在の位置と頭の向きを記憶するデータ型 turtle を作り、次の4つの動作を実現する関数群を定義せよ。
 - (0.0, 0.0) に新しい turtle を生成する
 - turtle を n 歩前に進める
 - turtle を左に k 度回転させる
 - 現在の turtle の位置を返す

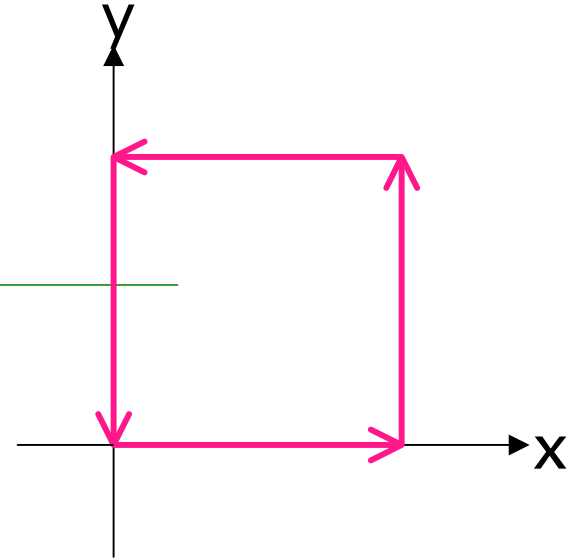
課題1 (仕様)

- `type turtle = (any specification)`
- `val new_turtle = unit → turtle`
 - 新しいタートルを生成
- `val advance = turtle → float → unit`
- `val rotate = turtle → float → unit`
 - タートルの位置を移動させる
- `val locate = turtle → float * float`
 - 現在の位置を (x, y) のペアで報告

課題1 (例)

```
# let t1 = new_turtle ();;
val t1 : turtle = { ... }
# advance t1 1.0; locate t1;;
- : float * float = (1., 0.)
# rotate t1 90.0; advance t1 1.0; locate t1;;
- : float * float = (1., 1.)
# rotate t1 90.0; advance t1 1.0; locate t1;;
- : float * float = (0., 1.)
# rotate t1 90.0; advance t1 1.0; locate t1;;
- : float * float = (0., 0.)
```

(*計算誤差で値が正確に0にならないのは気にしなくて良い*)



課題1 (ヒント)

- 三角関数
 - `sin, cos, ... : float → float` (弧度法)
 - `let pi = atan 1.0 *. 4.0` (必要なら...)
- データ型
 - 変更可能なレコードが最適か?

課題2

- Stack のデータ構造を表現する多相型を定義し、次の操作を実装せよ。
 - `new_stack: unit → 'a stack`
新しい stack の作成
 - `push: 'a stack → 'a → unit`
要素を頭に追加
 - `pop: 'a stack → 'a`
先頭要素を取り出す
 - 空 stack に対する pop は例外を送出

課題2 (例)

```
# let s = new_stack ();;  
val t1 : 'a stack = .....  
# push s 1;;  
- : unit = ()  
# push s 2;;  
- : unit = ()  
# pop s;;  
- : int = 2  
# pop s;;  
- : int = 1  
# pop s;;  
Uncaught exception: EmptyStack.
```

課題2 (ヒント)

- データ型の実際の定義は?
 - 実はシンプルに

```
type 'a stack = { mutable s : 'a list }
```

でよい...
 - push: リストの先頭に要素を追加
 - pop: リストの head を取り出す、
取り出した後の stack はリストの tail

課題3 (optional)

- 課題2と同様に Queue を作れ。但し、各操作は定数ステップで完了すること。
 - new_queue: 新しい queue の作成
 - add: 新しい要素を末尾に追加
 - take: 先頭の要素を取り出す
- Stack と似てますが、ずっと難しいです。
- 単純なリストでは取り出しか追加のどちらかが $O(1)$ 操作になりません。

提出方法

- 〆切: 2003年5月13日 (火) 13:00
- 提出先: ml-report@yl.is.s.u-tokyo.ac.jp
- 題名: "Report 3 xxxxx" (学生証番号)