関数・論理型プログラミング実験
ML演習第2回

松田 一孝

TA: 武田広太郎 寺尾拓

講義のサポートページ

- http://www-kb.is.s.u-tokyo.ac.jp/~kztk/cgi-bin/m/
  - 講義資料等が用意される
  - レポートの提出先
  - 利用にはアカウントが必要
    - アカウントを用意するので，
      自分の名前と**学籍番号**を書いたメールを
      kztk@is.s.u-tokyo.ac.jp
      までメールすること

今日の内容

- 型多相性
  - Parametric Polymorphism
- ユーザ定義型
  - レコード型
  - ヴァリアント型
  - 多相データ型
- 副作用

型多相性

型多相性とは

- 異なる型や式などを
  どうにかしてまとめて扱う仕組み

型多相性のない世界

- リストの先頭要素
  - hd_int : int list -> int
    - 実装 let hd_int (a::_) = a
  - hd_bool : bool list -> bool
    - 実装 let hd_bool (a::_) = a
  - hd_i2i : (int -> int) list -> int -> int
    - 実装 let hd_i2i (a::_) = a

異なる型のリスト毎に関数を定義
→ 面倒なだけじゃなく，バグの元

解決：型でパラメトライズ

- hd[α] : α list -> α
  - hd[int] : int list -> int
  - hd[bool] : bool list -> bool
  - hd[int->int] :
    (int -> int) list -> int -> int

```
let hd[α] (a::_) = a
```

注：OCamlでは型パラメタを（式に）書かない
（型推論が自動的に補う）

注の注：実は3.12以降は書ける

Parametric Polymorphism

- 型をパラメータにすることで，
  本質的に同一なものをまとめる方法
  型によらず同じ動作

  - cf. subtype polymorphism
    - OOPの継承
  - cf. ad-hoc polymorphism
    - 関数のオーバロード
      - Haskellのtype class
      - OCamlの比較演算

多相関数の例

- 恒等関数

OCamlでは型パラメタを'aや'bで表す

```
# let id x = x;;
val id : 'a -> 'a = <fun>
# id 1;;
- : int = 1
# id true;;
- : bool = true
# id (fun x -> x+1);;
- : int -> int = <fun>
```

多相関数の例

- 組の要素の抽出（射影）

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

多相関数の例

- 前回の課題より
  - fold_left :
    ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
  - fold_right :
    ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
  - append : 'a list -> 'a list -> 'a list
  - last : 'a list -> 'a
  - map : ('a -> 'b) -> 'a list -> 'b list
  - fix : (('a->'b)->('a->'b))->'a->'b

型の明示

- (パターン : 型) もしくは (式 : 型)

```
# let id (x : int) = x;;
val id : int -> int = <fun>
# let id x = (x : int);;
val id : int -> int = <fun>
```

ユーザ定義型

レコード・ヴァリアント

ユーザの定義型

- レコード (record)
  - ≒ 組の各要素に名前がついたもの
  - Cのstructに相当
- ヴァリアント (variant)
  - 何種類かのうち一つをとる値
  - Cのstruct, unionの組み合わせに相当
  - 操作の安全性が型検査により保証

レコード型

```
# type complex = { re : float ; im : float };;
type complex = { re : float; im : float; }
# let t = { re = 1.0; im = 1.0 };;
val t : complex = {re = 1.; im = 1.}
# t.re;;
- : float = 1.
# let abs { re = r; im = i } = sqrt ( r*.r +. i*.i );;
val abs : complex -> float = <fun>
# abs t;;
- : float = 1.4142135623730951
```

ヴァリアント型

- type 型名 = タグ1 of 型1
  | タグ2 of 型2
  - 例：boolと同等な型
    - type mybool = False | True
  - 例：簡単なインタプリタの「値」
    - type value = VInt of int
      | VBool of bool

注意：型名の最初は必ず小文字，
タグ名の最初は必ず大文字

ヴァリアント型の構成

- タグ 式
  - 例：mybool

```
# True;;
- : mybool = True
# False;;
- : mybool = False
```

```
type mybool =
  True | False
```

  - 例：value

```
# VInt 1;;
- : value = VInt 1
# VBool true;;
- : value = VBool true
```

```
type value =
  VInt of int
| VBool of bool
```

ヴァイリアント型の値の分解

- match ... with

```
# let not x =
    match x with
        True  -> False
      | False -> True;;
val not : mybool -> mybool = <fun>
# let string_of_value x =
    match x with
      VInt  i -> string_of_int i
    | VBool b -> string_of_bool b;;
val string_of_value : value -> string = <fun>
```

ヴァリアント型の再帰的定義

- 例：整数のリスト
  - ```
    type int_list =
          Nil
        | Cons of int * int_list
    ```
- 例：ノードに整数の値を持つ二分木
  - ```
    type int_tree =
          Leaf
        | Node of int * int_tree * int_tree
    ```

相互再帰的な定義

- type ... and ... and ...
  - 例：intとboolが交互に出現するリスト
    - ```
      type ilist = INil
                 | ICons of int * blist
      and  blist = BNil
                 | BCons of bool * ilist
      ```
      ```

多相的なヴァリアント型

- 例：ノードに値を持つ二分木
  - type 'a tree =
      Leaf
    | Node of 'a * 'a tree * 'a tree

```
# Node (5, Leaf, Leaf);;
- : int tree = Node (5, Leaf, Leaf)
# Node ("a",Leaf,Leaf);;
- : string tree = Node ("a", Leaf, Leaf)
# Leaf;;
- : 'a tree = Leaf
```

その上の関数

```
# let rec size x =
    match x with
      | Leaf -> 0
      | Node (_,l,r) ->
          1 + size l + size r;;
val size : 'a tree -> int = <fun>
```

複数の型パラメタ

- type ('a,'b,…) 型名 = …

```
type ('a,'b) either
     = L of 'a
     | R of 'b
```

例外

例外処理

- エラーが発生したとき，
  現在の計算を打ち切ってエラー処理用のコードにジャンプする機構
  - エラー
    - 0除算
    - hd []
    - …
  - C++，Javaにも同様の機構が

例外の生成と捕捉

- 生成：raise 式
- 捕捉：try … with パターン$_1$ -> 式$_1$
  | パターン$_2$ -> 式$_2$

```
# let div x y =
    if y = 0 then raise Division_by_zero
    else          x/y;;
...
# div 8 0;;
Exception: Division_by_zero
# let f x y =
    try Some (div x y) with Division_by_zero -> None;;
...
# f 8 2;;
- : int option = Some 4
# f 8 0;;
- : int option = None
```

ユーザ定義例外

- exception 例外名 of 型

```
# exception My_exception;;
exception My_exception
# raise My_exception;;
Exception: My_exception.
# exception My_str_exception of string;;
exception My_str_exception of string
# raise (My_str_exception "Hello");;
Exception: My_str_exception "Hello".
# try
    raise (My_str_exception "Hello")
  with
    My_str_exception s -> s;;
- : string = "Hello"
```

副作用

副作用

- ある式の評価が参照等価でないとき
  その式は副作用を持つ，という
  - ある式が参照等価とは，
    大雑把には，いつどこで評価しても
    評価結果が同じこと
  - ようするに「関数でない」
- 副作用はどうしても必要でなければ
  使うべきでない
  - コードが読みづらく
  - バグの元

入出力

- print_string : string -> unit
  - unit 副作用があることを明示
    - 型 unit の値は () のみ

- read_line : unit -> string

```
# read_line ();;
Hello World
- : string = "Hello World"
```

参照（reference）

- 中身を変更可能な「入れもの」
  - Cなどの変数のように代入できる
- mutable recordの特殊例
  - マニュアルを参照のこと

```
# let r = ref 0;;
val r : int ref = {contents = 0}
# !r ;;
- : int = 0
# r := 1;;
- : unit = ()
# !r ;;
- : int = 1
```

複数の式の逐次実行

- 式1 ; 式2 ; … ; 式n
  - 順番に式を評価し，
    最後に評価した式の結果を返す

```
# let t = print_string "> ";read_line ();;
>
```

副作用と型多相

- let t = ref []
  - tの型は？

```
# let sum () =  fold_right (+) (!t) 0;;
...
# sum ();;
- : int = 0
# t := [true]
- : unit = ()
# sum ()
???
```

もし，t: 'a list ref
だとすると
型安全性が破壊

OCamlの解決策

- 「未決定な単相型」
  - '_aや'_bなど

```
# let t = ref [];;
val t : '_a list ref = { content = [] }
# let sum () =  fold_right (+) (!t) 0;;
…
# t;;
- : int list ref = { content = [] }
# t := [true];;
Error: …
```

'_aがintに決定

問題は参照型以外にも

- 式f ()の型は？
  - fはunit -> 'a -> 'a だが…

```
let f () =
  let r = ref None in
    fun x ->
      let old = match !r with
                  None -> x
                | Some y -> y in
          (r := Some x; old)
```

OCamlの最終的な解決策

- Value Restriction
  - 副作用がないと確実にわかる「値」にのみ多相的な型を与える
    - 「値」は評価済み
      - 値である式：定数，fun式，それらの組…
      - 値でない式：参照，let式，関数適用…

```
# let id x = x;;
val id : 'a -> 'a = <fun>
# let id2 = id id;;
val id2 : '_a -> '_a = <fun>
# let id2' x = id id x;;
val id2' : 'a -> 'a = <fun>
```

第2回レポート課題
締切 2週間後の13:00

注意

- 問題を良くよみましょう
- 実行例はかならず付けましょう
- 考察不要と書いていない限り
  考察（着想含む）は必ずつけましょう

- 「レポートの心得」を確認のこと

問1

- 自然数を表す型natを以下で定義する

```
type nat = Z | S of nat
```

  - 以下の関数を実装せよ
    - 加算 add : nat -> nat -> nat
    - 減算 sub : nat -> nat -> nat
    - 乗算 mul : nat -> nat -> nat
    - 累乗 pow : nat -> nat -> nat
    - n2i : nat -> int
    - i2n : int -> nat
      - 必要に応じて例外を投げる

問2

- 自然数の二進表現を表す型 bnat を適当に与え問1と同じ動作をする関数を実装せよ
  - 加算 add : bnat -> bnat -> bnat
  - 減算 sub : bnat -> bnat -> bnat
  - 乗算 mul : bnat -> bnat -> bnat
  - 累乗 pow : bnat -> bnat -> bnat
  - n2i : bnat -> int
  - i2n : int -> bnat
    - 必要に応じて例外を投げる

問3

- 二分木を受けとり，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を定義せよ
  - 行きがけ順 (preorder)
  - 通りがけ順 (inorder)
  - 帰りがけ順 (postorder)
- 二分木の定義は以下を用いよ
  - ```
    type 'a tree =
        Leaf
      | Node of 'a * 'a tree * 'a tree
    ```
    ```

問4

- 二分木を受けとり，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を定義せよ
  - レベル順（level-order）
    - 幅優先探索の順番
- 二分木の定義は問1と同じ

問5（1/2）

- intとbool型の値を計算する単純な式Eの構文を以下で定める

```
E ::= 0 | 1 | 2 | …
    | E + E | E - E | E * E | E / E
    | true | false
    | E = E | E < E
    | if E then E else E
```

- この式の評価結果を表す型を以下で定める

```
type value = VInt of int | VBool of bool
```

問5（2/2）

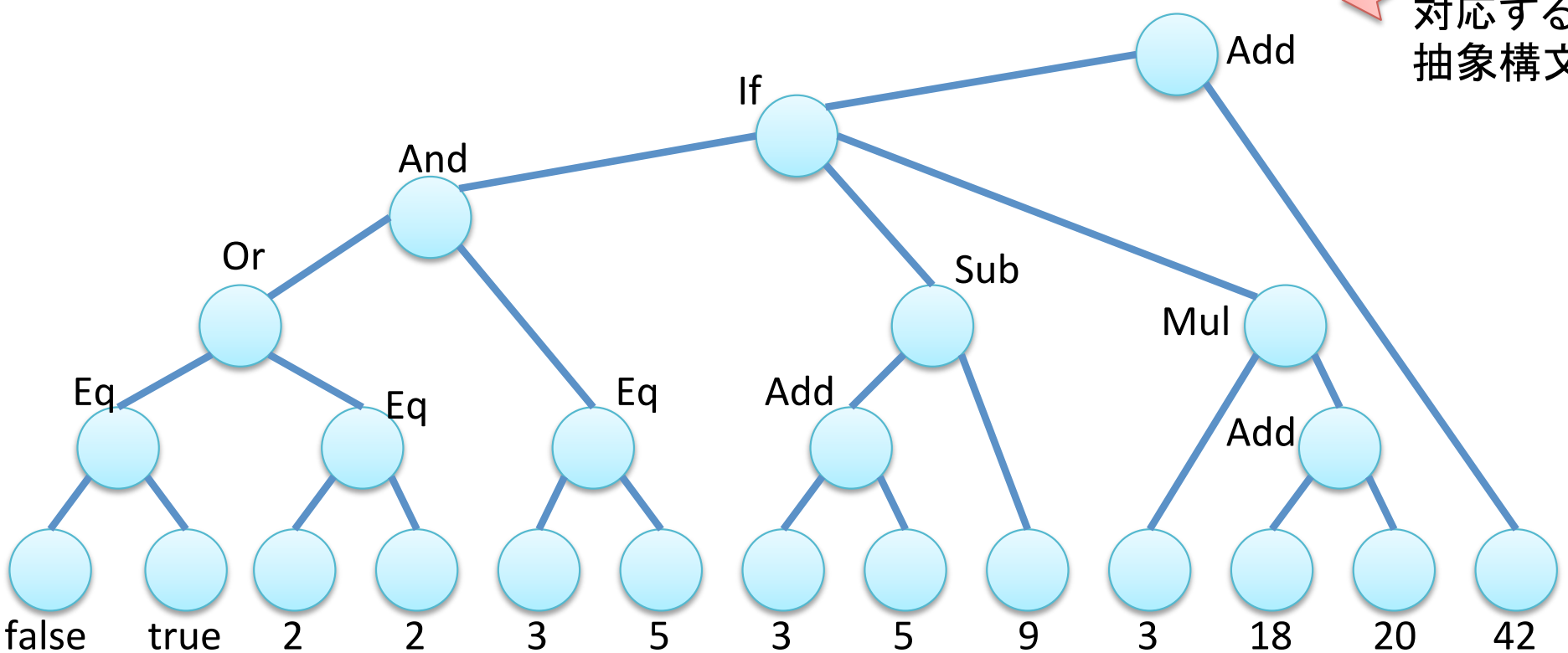
- このとき，Eの抽象構文木に対応する型 expr を定義せよ
  - 以下を完成させればよい

```
type expr = EConst of value
          | EAdd   of expr * expr
          | ESub   of expr * expr
...
```

補足：抽象構文木の例

```
(if ((false = true) || (2 = 2)) && (3 = 5)
 then (3 + 5) - 9 else 3 * (18 + 20)) + 42
```

対応する
抽象構文木



問6

- 前問で定義した式を評価する関数
  eval : expr -> value
  を定義せよ
  - valueは前問を参照
  - exprは前問の定義

- 例外 Eval_errorを定義し，
  boolをintを足す等の評価時エラーが
  発生した場合にはEval_errorを投げる
  こと

発展1（church数）

- 型 church を以下で定義する

```
type church =
    { t : 'a. ('a -> 'a) -> 'a -> 'a }
```

  - 関数c2iとi2cを
    c2i (i2c n) = n if n >= 0
    となるように実装せよ
  - 問1，2と同様にadd，mul，pow，subを定義せよ
    * 型はchurch -> church -> churchとなる
    * ただし，c2iやi2cを使ってはならない

発展2 (Curry-Howard対応)

- 型 false_t, not_t, and_t, or_t を以下とする
  - type false_t = { any : 'a.'a }
    - 型 false_t を持つ値は存在しないことに注意
  - type 'a not_t = 'a -> false_t
  - type ('a,'b) and_t = 'a * 'b
  - type ('a,'b) or_t = L of 'a | R of 'b

発展2（2/3）

- 次ページの型について，
  それぞれの型を持つ式を…
  - 再帰，例外，副作用を用いずに定義できるか？できないときは理由を述べよ
  - 例外と副作用を許す場合はどうか？

  - ただし，以下を満たすこと
    * 定義する式の中で発生する例外は，
      その式中で捕捉すること
      - ただし，発生しないと保証できれば許す
    * 標準ライブラリの関数は使用禁止

発展2（3/3）

1. ('a->'b)->('b->'c)->('a->'c)
2. ('a, ('b,'c) and_t) or_t ->
   (('a,'b) or_t, ('a,'c) or_t) and_t
3. (('a,'b) or_t, ('a,'c) or_t) and_t
   -> ('a, ('b,'c) and_t) or_t
4. ('a, 'a not_t) or_t
   - もしくは ('a ->'c) -> ('a not_t ->'c) -> c'
5. ('a, 'a not_t) and_t
   - もしくは ('a -> 'a not_t -> 'c) -> 'c
6. (('p -> 'q) -> 'p) -> 'p
   - cf. Peirce's law

補足

- 部分関数（停止しなかったり，例外を投げたりする関数）を使えば簡単

```
# let rec f x = f x
val f : 'a -> 'b = <fun>
# let f x = let Some y = None in y
val f : 'a -> 'b = <fun>
# let f x = failwith "Hey!"
val f : 'a -> 'b = <fun>
```

補足

- call/ccは6の関数の1つ

```
# #use "test.ml";;
...
val callcc : (('p -> 'q) -> 'p) -> 'p = <fun>
...
# callcc (fun jump -> 1 + 2);;
- : int = 3
# callcc (fun jump -> 1 + jump 2);;
- : int = 2
# callcc (fun jump -> "a" ^ jump "b");;
- : string = "b"
# callcc (fun jump -> "a" ^ "b");;
- : string = "ab"
```