関数・論理型プログラミング実験
ML演習第3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

問題の訂正

- 第2回の発展2
  - 副作用・例外を用いた場合は
    intやfloat等の「基本型」に対して
    例外が発生しなければよいとします
    * 前回のcallccは'pが基本型でないと正しくない
  - 4，6番目は以下の型を持つ関数を
    副作用やユーザ定義型や再帰なしで実装したのでもよいことにする

```
4. ('a -> 'c) -> ('a not_t -> 'c)
     -> 'c not_t not_t
6. (('p -> 'q) -> 'p) -> 'p not_t not_t
```

今日の内容

- OCamlのモジュールシステム
  - Structure
  - Signature
  - Functor
- OCamlの（分割）コンパイル

大規模なソフトウェアのプログラミングは難しい

- 人の記憶できるプログラムの量には限度があるから
  - OCaml処理系のソースプログラム全てを記憶している人は（多分）いない
  - Linuxカーネルのソースプログラム全てを記憶している人は（多分）いない

Q：ではどうする？

- A：複数人でプログラムする
  - 10人やれば一人あたり1/10の作業量
  - 100人で1/100に
  - 1000人で1/1000に
  - 10000人で1/10000に
  - …

ならない

最悪のシナリオ

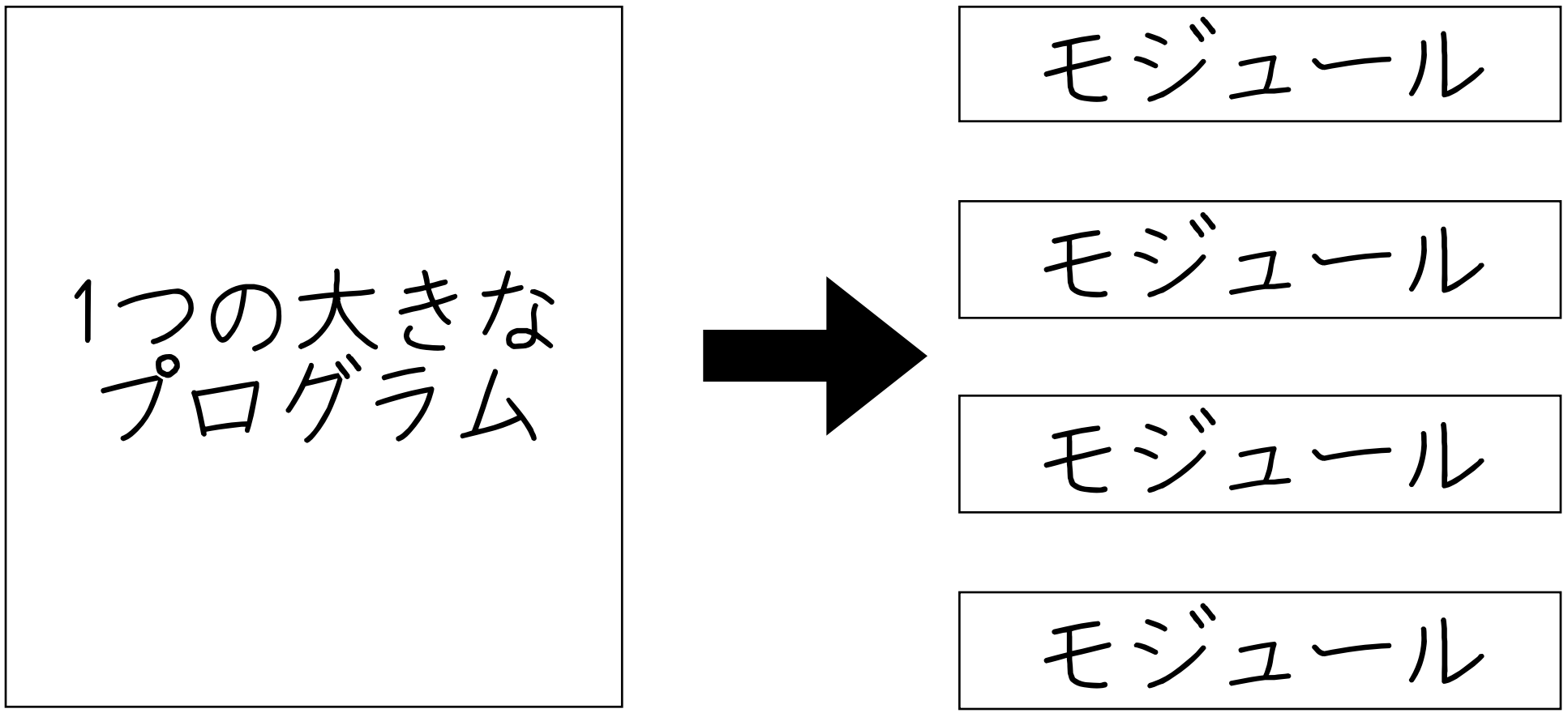
- 似たプログラムの大量の生成
  - 他人のコードなんぞ読めるか！
  - 自分で書いたほうが早い！

⇒プログラムの改善・修正が難しく
  - 似たプログラムを全て修正する必要
  - 修正が及ぼす影響は？

どう避ける？

- プログラムを「モジュール化」する
  - モジュール：
    再利用可能なプログラム部品
- モジュールの仕様と実装を切り分ける

モジュールに分ける



これだけでは不十分

仕様と実装を切り分ける

- 仕様
  - モジュールの外からの使われ方
  - どんな関数がある
  - それらの型は？
  - …
- 実装
  - 仕様の実現

なぜ仕様と実装を分離？

- モジュールの外からの利用が容易に
  - 利用者は仕様だけ見ればよい
- モジュールの実装の修正が容易に
  - モジュールの仕様さえ守ればよい

OCamlのモジュールシステム

- Structure
  - モジュールの実装
  - 名前空間の切り分け
- Signature
  - Structureの仕様
  - 関数名とその型，および提供する型

- Functor
  - StructureからStructureを作る関数のようなもの

Structure

- モジュールの実装を定義
- 構文

```
module モジュール名 =
    struct 内容 end
```

  - 内容の部分に型や関数の定義を書く
  - モジュール名の先頭は大文字

例：多重集合

```
module Multiset =
struct
  type 'a t = 'a list
  let empty = []
  let add a xs = a::xs
  let rec remove a xs =
    match xs with
        []    -> []
      | y::ys -> if a=y then ys else y::remove a ys
  let rec count_sub a xs k =
    match xs with
        []     -> k
      | y::ys  ->
          if a=y then count_sub a ys (k+1)
          else        count_sub a ys k
  let count a xs = count_sub a xs 0
end
```

Structureの使い方

- 中の型や関数を使うには：
  - 「モジュール名」.「型名 or 変数名」

```
# let e = Multiset.empty ;;
val e : 'a list = []
# let s = Multiset.add 5 e;;
val s : int list = [5]
# Multiset.count 5 s;;
- : int = 1
```

Module名の省略

- openすることでモジュール名を省略可
  - open モジュール名

```
# open Multiset ;;
# let s = add 5 empty;;
val s : int list = [5]
# let s = add 5 s;;
val s : int list = [5; 5]
# count 5 s;;
- : int = 2
```

標準ライブラリのモジュール

- List, String, Printf, …
  - 詳しくはマニュアルのPart IV参照

```
# List.length [1;2;3];;
- : int = 3
# String.sub "abcde" 2 3;;
- : string = "cde"
# Printf.printf "%04d %s\n" 12 "XXX";;
0012 XXX
- : unit = ()
```

Signature

- モジュールのインタフェース
  - Signatureに書いた型や関数のみが外部から利用可
  - モジュールの「型」

- 構文

```
module type シグニチャ名 =
    sig 内容 end
```

  - 内容部分に型の宣言や関数の型を書く
  - シグニチャ名の先頭は慣習的に大文字

例：多重集合

```
module type MULTISET =
sig
  type 'a t
  val empty : 'a t
  val add : 'a -> 'a t -> 'a t
  val remove : 'a -> 'a t -> 'a t
  val count : 'a -> 'a t -> int
end
```

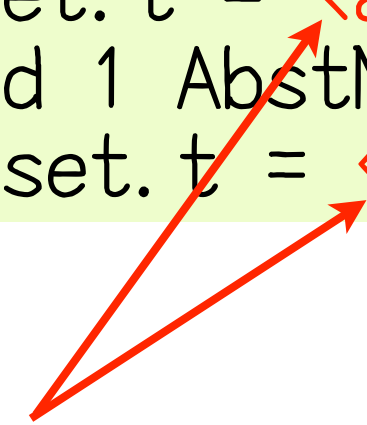
Signatureの適用

- Signatureをstructureに当て嵌める
  - 構文
    - module モジュール名:シグニチャ
      = 元モジュール
    - module モジュール名
      = (元モジュール:シグニチャ)

- 実体は元モジュールと同じ
- モジュール外からは signature で示された型や関数しか利用できない

例

```
# module AbstMultiset : MULTISET = Multiset;;
module AbstMultiset : MULTISET
# AbstMultiset.empty;;
- : 'a AbstMultiset.t = <abstr>
# AbstMultiset.add 1 AbstMultiset.empty;;
- : int AbstMultiset.t = <abstr>
```



実体がlistであることは外部からは隠蔽

例（つづき）

```
# AbstMultiset.count_sub;;
Error: Unbound value AbstMultiset.count_sub
```

count_subはMULTISETにないので利用不可

```
# AbstMultiset.add 0 Multiset.empty;;
Error: This expression has type 'a list
       but an expression was expected of
type int AbstMultiset.t
```

実体は同じでも違う型だと見なされる

Functor

- モジュールを受けとり，
  モジュールを返す関数のようなもの
  - 構文
    functor（仮引数 :シグニチャ）
    -> モジュール

例：多重集合

```
type order = LT | EQ | GT
module type ORDERED_TYPE =
sig
  type t
  val compare : t -> t -> order
end

module Multiset2 =
  functor (T : ORDERED_TYPE) -> struct
    type t = T.t list
    let rec remove a xs =
      match xs with
        | [] -> []
        | y :: ys ->
            (match T.compare a y with
               | EQ -> ys
               | _  -> y :: remove a ys)
    (* 以下略 *)
  end
```

Functorの適用

- Functorにモジュールを渡す
  - 構文
    ファンクタ（モジュール）
  - 括弧は必要

例

```
module OrderedString : ORDERED_TYPE = struct
  type t = string
  let compare x y = if x < y then      LT
                    else if x > y then GT
                    else               EQ
end

module StringMultiset =
   Multiset2 (OrderedString)
```

Functorに対するsignature

- Functorにもsignatureが作れる
  - functor (仮引数 : シグニチャ) -> シグニチャ

```
module type MULTISET2 =
  functor (T : ORDERED_TYPE) ->
    sig
      type t
      val empty : t
      val add    : T.t -> t -> t
      val remove : T.t -> t -> t
      val count  : T.t -> t -> int
    end
```

再帰モジュール

- 相互再帰的なモジュールも作成可
  - module rec モジュール名$_1$:シグニチャ$_1$ = モジュール
    and モジュール名$_2$:シグニチャ$_2$ = モジュール

```
# module rec Even : sig val f : int -> bool end =
    struct let f n = if n = 0 then true else Odd.f (n-1) end
  and Odd : sig val f : int -> bool end =
    struct let f n = if n = 0 then false else Even.f (n-1) end;;
module rec Even : sig val f : int -> bool end
and Odd : sig val f : int -> bool end
# Even.f 24;;
- : bool = true
# Odd.f 24;;
- : bool = false
```

OCamlコンパイラの使い方

OCamlのコンパイラ

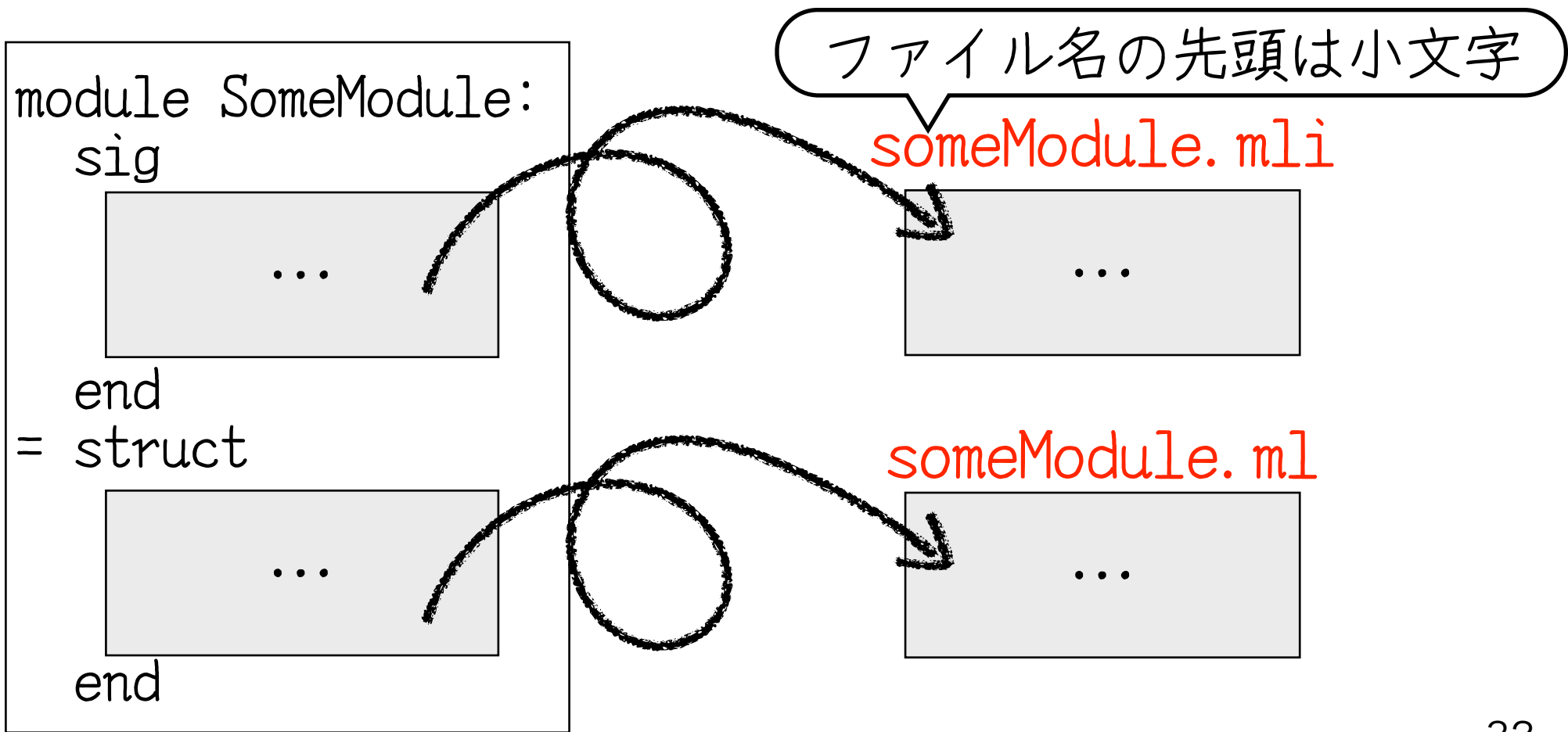
- 二種類
  - ocamlc: バイトコードコンパイラ
    - OCaml仮想マシン（ocamlバイトコードインタプリタ）用コードを生成
  - ocamlopt: ネイティブコードコンパイラ
    - x86など，実際のマシン用コードの生成
- モジュール単位での分割コンパイルをサポート

FILES

- ソースファイル
  - .ml　モジュールの実装
  - .mli　モジュールのシグニチャ
- オブジェクトファイル
  - .cmo　実装のバイトコード
  - .cmi　I/Fのバイトコード
  - .o　実装のネイティブコード
  - .cmx　上の付加情報
  - .a, .cma, .cmxa ライブラリ

モジュールと分割コンパイル

- モジュールのsignatureとstructureを別のファイルとして分割コンパイル可



モジュールの分割コンパイル

- .mliファイルをコンパイル
  - .cmiが生成される
- .mlファイルをocamlcでコンパイル
  - .cmoが生成
  - .mliがあれば.cmiを用いて型検査
- .mlファイルをocamloptでコンパイル
  - .cmxと.oが生成
  - .mliがあれば.cmiを用いて型検査

.mli,.mlによるモジュールの例

- strSet.ml, strSet.mli
  - 文字列の順序付き多重集合のモジュール StrSet の定義

- sort.ml
  - StrSetモジュールを利用しソートを行うプログラム本体

サポートページよりDL可

分割コンパイルの例

順番が大事：mliが先

```
$ ocamlc -c strSet.mli
$ ocamlc -c strSet.ml
$ ocamlc -c sort.ml
$ ls -F *.cm*
sort.cmi    sort.cmo    strSet.cmi    strSet.cmo
$ ocamlc -o sort srtSet.cmo sort.cmo
$ ls -F sort
sort*
```

順番が大事：sort.mlの中で，StrSetを使っているので，strSet.cmoを先に

sortの実行例

```
$ ./sort <<END
> bbb
> ccc
> aaa
> bbb
> END
aaa
bbb
bbb
ccc
```

.cmoをインタプリタで

- #load "SomeFile.cmo"

```
# #load "strSet.cmo";;
# StrSet.empty ;;
- : StrSet.t = <abstr>
# StrSet.count_sub
Unbound value StrSet.count_sub
# open StrSet;;
# add "abc" empty ;;
- : StrSet.t = <abstr>;;
```

注意

- レポート課題で，実行にコンパイルが必要な場合，ビルド方法の記述も提出すること
  - Makefileを用いてよい
    - OCamlMakefileを用いてもよい
  - がどちらの場合も「makeせよ」とは書くこと

第3回レポート課題
締切は2週間後の13:00

問1

- sortの例を自分で試せ
  - 例にそって実行ファイルを生成・実行せよ
  - .cmoをインタプリタで利用せよ
  - .mliをコンパイルしないとどうなるか？
  - 最後のリンク時にファイルの順番を変えるとどうなるか？
  - OCamlMakefileを用いてみよ
  - その他いろいろ試してみよ
    - ネイティブコードコンパイルなどなど

注意：今後課題でMakefileはもちろんOCamlMakefileを用いてもよい

問2

- スタックを扱うモジュールを実装せよ
  - 以下の関数をサポートすること
    - pop : 'a スタックの型
      -> ('a * 'a スタックの型)
    - push : 'a -> 'a スタックの型
      -> 'a スタックの型
    - empty : 'a スタックの型
    - size : 'a スタックの型 -> int
  - シグニチャを適切に与え抽象化せよ
    - スタックの実装を'a list から
      'a list * intに変えてもよいように

問3

- functorを用いて
  集合を扱うモジュールを作成せよ
  - 要素はORDERED_TYPEで表現される型
  - シグニチャを与え内部の実装を適切に抽象化すること
  - 集合の実装はただの二分探索木でよい
    * リストはだめ
  - 副作用は用いてはならない
  - 組み込みのSetは用いてはならない
  - 集合として利用するのに十分なだけの関数を用意すること
    * empty, add, remove, mem, sizeは実装

問4

- 問3を参考に連想配列を扱うモジュールを作成せよ
  - 連想配列のキーはORDERED_TYPEで表現される型とする
  - 副作用は用いてはならない
  - 利用するのに十分なだけの関数を用意すること
    - empty, add, remove, lookup等

問5

- functorを用いて，
  加算と乗算が定義された要素を含む
  行列とベクトルの演算を定義する
  モジュールを作成せよ
  - 加算がor，乗算がandな真偽値
  - 加算がmin，乗算が+な 整数∪{∞}

- 得られたモジュールを利用し様々な
  計算を行ってみよ

補足

- 半環$(R, 0, 1, +, \times)$に対し，
  行列$(R^{m,m}, I^{m,m}, \bullet)$はモノイド
  - 「matrix semiring」等で検索すれば
    多数の例が見つかる

例：最短路の長さ

- M_{ij}：ij要素が
  - もしi，j間に枝があれば枝の重み
  - なければ∞

- 半環$(R, \infty, 0, \min, +)$の上において$M^{|V|}$が全点間の最短路長を表現
  - M^n_{ij}：iからjまでにn点経由したときの最短路長

    * 注意：これだと$O(n^3 \log n)$かかる
    * もっと工夫するとFloyd-Warshall法になる

発展1（GADTs in OCaml）

- 以下のシグネチャEQを持つモジュールEqを定義せよ
  - ただし，各関数は呼ばれれば停止し，例外が発生しないようにすること

```
module type EQ = sig
  type ('a,'b) equal
  val refl : ('a,'a) equal
  val symm : ('a,'b) equal -> ('b,'a) equal
  val trans : ('a,'b) equal -> ('b,'c) equal -> ('a,'c) equal
  val apply : ('a,'b) equal -> 'a -> 'b
  module Lift : functor (F: sig type 'a t end) -> sig
    val f : ('a,'b) equal -> ('a F.t, 'b F.t) equal
  end
end
```

発展1（つづき）

- 前回の問5の簡単な言語の式と値をEqを用いて以下のように定義したとする

```
type 'a value =
  | VBool of (bool, 'a) Eq.equal * bool
  | VInt of (int, 'a) Eq.equal * int
type 'a expr =
  | EConst of 'a value
  | EAdd of (int,'a)  Eq.equal * (int expr) * (int expr)
  | EIf  of bool expr * 'a expr * 'a expr
  | EEq  of (bool,'a) Eq.equal * 'a expr * 'a expr
  ...
```

- このとき，この言語の式を評価する関数evalを定義せよ
  - eval : 'a expr -> 'a value

補足

- 実はOCaml 4.00以降ではこんなことをしなくても GADT が利用できる
  - マニュアルの"Language Extensions"を参照