ML 演習 第 3 回

新井淳也、中村宇佑、前田俊行

2011/04/26

今回の内容

- OCaml のモジュールシステムについて
  - Structure
  - Signature
  - Functor
- OCaml コンパイラの使い方
  - 分割コンパイルなど

※今日使うソースは
演習ホームページに置いておきます

The Module System

モジュールシステム

大規模ソフトウェアのプログラミングは難しい

- 人間が記憶できるプログラムの量には限界があるから
  - 例 1: 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
  - 例 2: 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

ではどうするか?

- 答: 複数人でプログラミングする
  - 10 人でやれば 1 人あたりの量は 10 分の 1 に
  - 100 人でやれば 100 分の 1 に
  - 1000000 人でやれば 1000000 分の 1 に...
  - ...

ならない

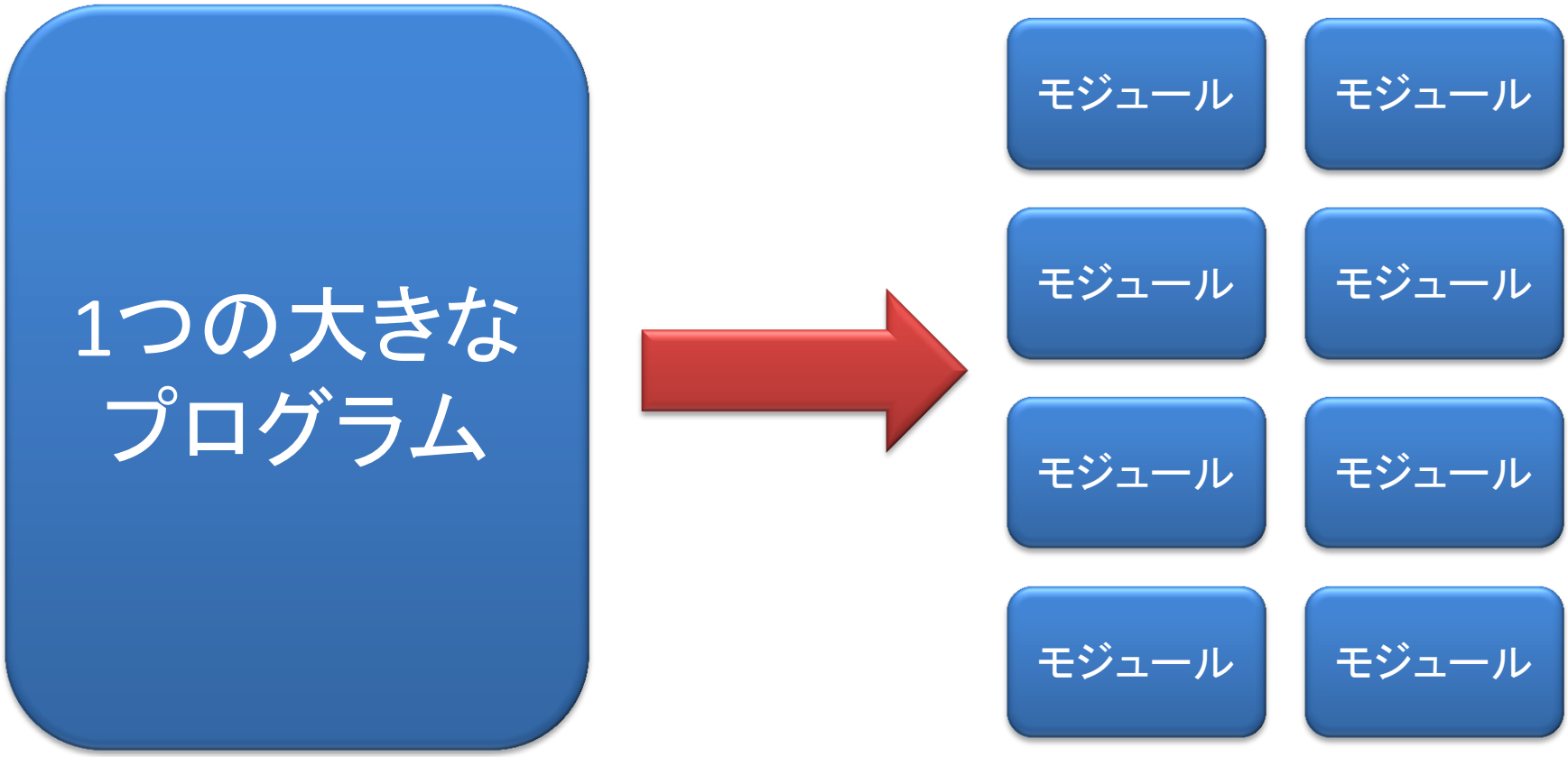
複数人でのプログラミング
最悪のシナリオ

- 似たようなプログラムが大量にできてしまう
  - 他人の書いたプログラムは読みにくい
  - 自分で書いた方が早い
- プログラムの改善・修正が難しくなってしまう
  - 似たようなプログラムを全て修正しないといけない
  - 修正が及ぼす影響が予測できない

最悪のシナリオを避けるには?

- プログラムを「モジュール化」する
  - プログラムを幾つかの部分 (モジュール) に分ける
  - モジュールの「仕様」と「実装」を切り分ける

プログラムをモジュールに分ける



- これだけなら簡単

モジュールの「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を表すもの
- 実装
  - 仕様を実現するデータ・プログラムなど

モジュール

「仕様」と「実装」を分けると何がうれしいか?

- モジュールの外からの利用が容易になる
  - モジュールの「仕様」だけ見ればよいので
    - 「実装」は基本的には気にしなくてよい
- モジュールの「実装」の修正が容易になる
  - モジュールの「仕様」さえ守っていればよいので
    - 「仕様」以外の使われ方を気にせず修正できる

OCaml の提供する モジュールシステム

- Structure
  - モジュールの実装と名前空間を提供する
    - 型や関数の実装を一つの名前空間にまとめてくれる
- Signature
  - Structure の仕様を定義する
    - Structure の外から使える型や関数を定義する
      - Structure の型や関数の実装 (定義) を隠蔽できる
- Functor
  - Structure から別の structure を作る関数のようなもの

Structure

- モジュールの実装を定義する

- 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

Structure の例: 多重集合

example4-1.ml

```
module Multiset =
  struct
    type 'a t = 'a list
    let empty_set = []
    let add elem set = elem :: set
    let rec remove elem = function
      | [] -> []
      | hd :: tl when hd = elem -> tl
      | hd :: tl -> hd :: (remove elem tl)
    let rec countsub elem n = function
      | [] -> n
      | hd :: tl when hd = elem ->
          countsub elem (n+1) tl
      | _ :: tl -> countsub elem n tl
    let count elem set = countsub elem 0 set
  end
```

Structure の使い方

- Structure 内部の変数や型を使うには
  ドット表記を使う
- 構文: (*「モジュール名」.「変数名 or 型名」*)

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

モジュール名の省略

- Structure を open することで
  モジュール名を省略できる

```
(* open「モジュール名」*)
# open Multiset;;
# let s = add 5 empty_set;;
val s : int list = [5]
# let s = add 5 s;;
val s : int list = [5; 5]
# count 5 s;;
- : int = 2
```

OCaml の組込みのモジュール

```
# List.length ["a"; "b"; "c"];;
- : int = 3
# String.sub "1234567" 2 3;;
- : string = "345"
# Printf.printf "%d %s¥n" 123 "abc";;
123 abc
- : unit = ()
```

- 他にもいろいろある
  - 詳しくはマニュアルの Part IV を参照

Signature

- モジュールのインタフェースを与える
  - Signature に書いた型や関数だけが
    モジュールの外から利用できる

- 構文:

  `module type`「シグネチャ名」`= sig`「内容」`end`

  - 「内容」の部分に型の宣言や関数の型を書く
    - シグネチャ名の先頭は (慣習的に) 大文字

Signature の例: 集合

example4-1.ml

```
module type MULTISET =
  sig
    type 'a t
    val empty_set : '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 にあてはめる
  - 構文

    module「モジュール名」:「シグネチャ」=「モジュール」

    または

    module「モジュール名」= (「モジュール」:「シグネチャ」)

  - 実体は元のモジュールと同じ
  - ただしモジュール外からは signature で示された型や関数しか使えない

Signature の適用の例

```
# module AbstMultiset : MULTISET = Multiset;;
module AbstMultiset : MULTISET

# AbstMultiset.empty_set;;
- : 'a AbstMultiset.t = <abstr>
# AbstMultiset.add 0 AbstMultiset.empty_set;;
- : int AbstMultiset.t = <abstr>
```

集合の実体が list であることが
外部からは分からない

Signature の適用の例 (続き)

```
# AbstMultiset.countsub;;
Unbound value AbstMultiset.countsub
```

countsub は MULTISET にはないので
外からはアクセスできない

```
# AbstMultiset.add 0 Multiset.empty_set;;
This expression has type 'a list but is here
used with type int AbstMultiset.t
```

実体は同じでも違う型と見なされる

補足: Signature の適用

- 構文

  module「モジュール名」:「シグネチャ」=「モジュール」

  または

  module「モジュール名」= (「モジュール」:「シグネチャ」)

– 「シグネチャ」や「モジュール」の部分に直接シグネチャやモジュールの定義を書くこともできる

```
module Foo : sig ... end = struct ... end
```

Functor

- モジュールを受け取ってモジュールを返す関数のようなもの
  - Functor を作る構文:

    `functor (「仮引数」:「シグネチャ」) -> 「モジュール」`

Functor の例: 多重集合再び

example4-2.ml

```
type comparison = Less | Equal | Greater
module type ORDERED_TYPE = sig
    type t
    val compare : t -> t -> comparison
  end

module Multiset2 =
  functor (Elem : ORDERED_TYPE) -> struct
    type t = Elem.t list
    let eq x y = Elem.compare x y = Equal
    let rec remove elem = function
      | [] -> []
      | hd :: tl when eq hd elem -> tl
      | hd :: tl -> hd :: (remove elem tl)
    … (* その他 *) …
  end
```

Functor からモジュールを作るには

- Functor にモジュールを渡すことで
  functor が定義しているモジュールが得られる
  - 構文:
    「functor」 (「モジュール」)

括弧は必須

- 例:

```
module StringMultiset =
    Multiset2(OrderedString)
```

example4-2.ml

Functor に対する signature

- Functor にも signature が作れる
  - Signature の functor を作る構文:
    `functor (「仮引数」:「シグネチャ」) ->「シグネチャ」`
- Signature の functor の定義の例:

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

example4-2.ml

Recursive Module

- 相互再帰的なモジュールも定義できる
  - 構文:

```
module rec「モジュール名1」:「signature1」=「struct1」
       and「モジュール名2」:「signature2」=「struct2」…
```

- 例:

```
# 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 156;;
- : bool = true
```

How to Use Compilers

コンパイラの使い方

OCaml のコンパイラ

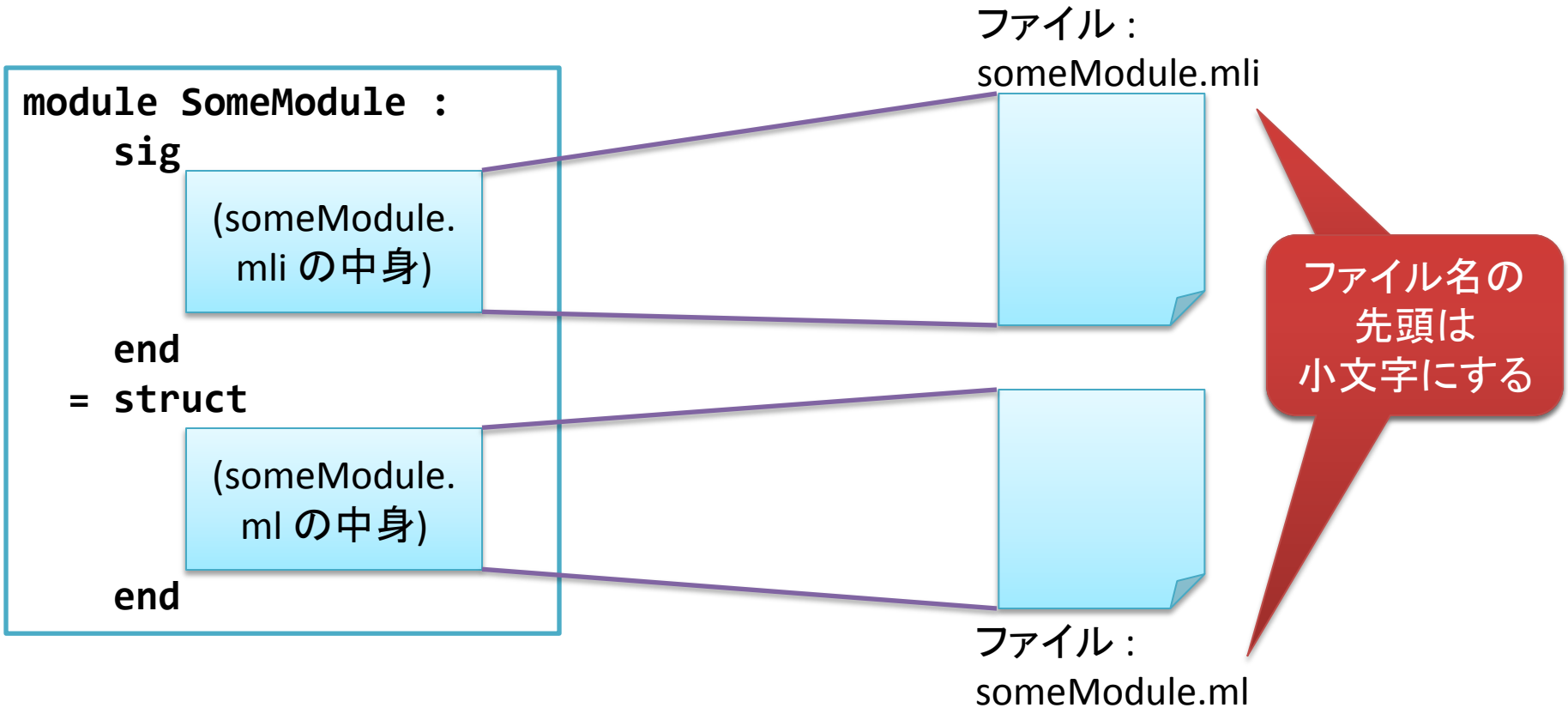
- 実行可能形式ファイルを生成してくれる
- 二種類のバックエンドがある
  - ocamlc: バイトコードコンパイラ
    - バイトコードを生成
      - バイトコードインタプリタ (ocamlrun) 上で実行される
  - ocamlopt: ネイティブコードコンパイラ
    - x86 や SPARC などの機械語を生成
- モジュール単位での
  分割コンパイルをサポートしている

OCaml のコンパイラが扱うファイルの種類

- ソースファイル
  - .ml　　モジュールの実装
  - .mli　　モジュールのシグネチャ
- オブジェクトファイル
  - .cmo　実装のバイトコード
  - .cmi　　インタフェースのバイトコード
  - .o　　実装のネイティブコード
  - .cmx　実装のネイティブコードの付加情報
  - .a, .cma, .cmxa　　ライブラリ

モジュールと分割コンパイルの関係

- モジュールの signature と structure を
  別々のファイルとして分割コンパイルできる



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 が生成される
  - .mli があれば .cmi を用いて型チェックしてくれる
- .ml ファイルを ocamlopt でコンパイル
  → .cmx と .o が生成される
  - .mli があれば .cmi を用いて型チェックしてくれる

.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 モジュールを用いてソートを行うプログラム本体

分割コンパイルの例

```
$ 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 strSet.cmo sort.cmo
$ ls -F sort
sort*
```

順序が重要:
sort.ml の中で StrSet を使っているので
sort.cmo を strSet.cmo より後に書く必要がある

sort の実行例

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

.cmo をインタプリタで利用する

- #load でバイトコードファイルを読み込み可能

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

OCamlMakefile を使う

- Makefile 中で OCamlMakefile を使うと OCaml プログラムの分割コンパイルが簡単になる
  - Makefile:
    プログラムなどの生成手順を記述したファイル
- OCamlMakefile の入手方法
  - パッケージ

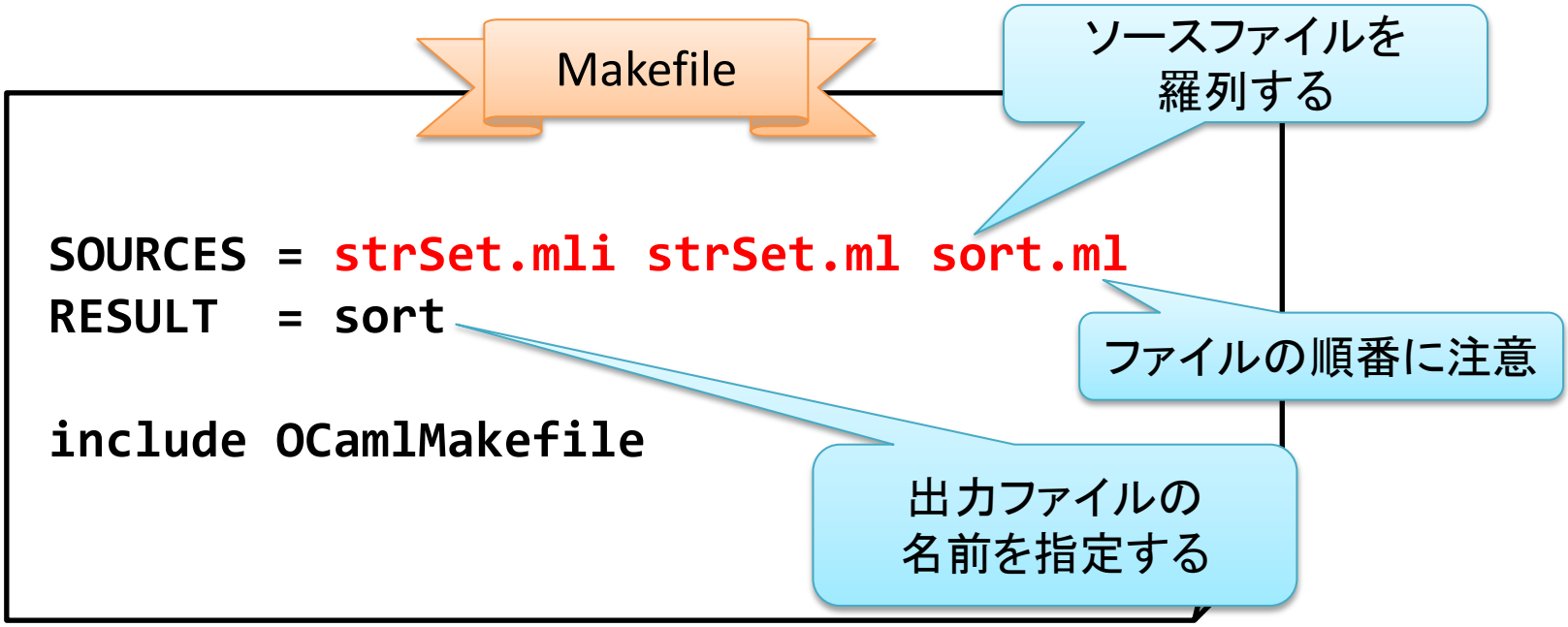
```
$ sudo apt-get install ocamlmakefile
```

  - 直接ダウンロード

http://www.ocaml.info/home/ocaml_sources.html#OCamlMakefile

    - 詳しい使い方は同梱の README.txt を参照

Makefile の書き方の例



ビルド (make) の実行例

パッケージで導入した場合はこの場所にある

このコピーは一回だけで十分
(ビルド毎にコピーする必要はない)

```
$ cp /usr/share/ocamlmakefile/OCamlMakefile ./
$ ls
Makefile  OCamlMakefile  sort.ml  strSet.ml  strSet.mli
$ make
make[1]: ディレクトリ `/tmp/sort' に入ります
ocamldep strSet.mli > ._bcdi/strSet.di
( … 省略 … )
$ ls
Makefile        sort       sort.cmo  strSet.cmi  strSet.ml
OCamlMakefile   sort.cmi   sort.ml   strSet.cmo  strSet.mli
```

第 3 回 課題

締切: 5/10 13:00 (日本標準時)

課題 1 (10 点)

- sort の例を自分で試してみよ
  - 例に従って実行ファイル生成し、実行してみよ
  - .cmo ファイルをインタプリタで利用してみよ
  - .mli をコンパイルしないとどうなるか試してみよ
  - 最後のリンク時にモジュールの順番を変えるとどうなるか試してみよ
  - OCamlMakefile を用いてみよ
  - その他いろいろ試してみよ

※ 今後課題で OCamlMakefile を用いても構わない

課題 2 (5 点)

- 前回 (第 2 回) の課題 2 で作ったスタックをモジュール化せよ
  - シグネチャも与えて
    内部の実装を適切に抽象化すること

課題 3 (5 点)

- 前回 (第 2 回) の課題 4 (または課題 9) で作ったキューをモジュール化せよ
  - シグネチャも与えて
    内部の実装を適切に抽象化すること

課題 4 (15 点)

- リスト以外のデータ構造を使って
  signature MULTISET2 に対する
  別の実装を与えよ
  - ただし、add, remove は平均時間計算量 O(log *n*)
    となるようにすること

課題 5 (20 点)

- 課題 4 での別の実装が
  元の実装と「同じ」であることを証明せよ
  - 「同じ」の定義は自分で与えること

課題 6 (15 点)

- ORDERED_TYPE で表現される型の key と任意の型の値についての連想配列 (マップ) を作る functor を作成せよ
  - シグネチャも与えて内部の実装を適切に抽象化すること
  - 必要ならば組込みの例外 Not_found を用いること
    - 標準ライブラリの Map モジュールを用いないこと

課題 6 (例 1)

```
# module NCStringMap = MyMap(NoCaseString);;
module NCStringMap :
  sig
    type key = NoCaseString.t
    type 'a t = 'a MyMap(NoCaseString).t
    val empty : 'a t
    val add : key -> 'a -> 'a t -> 'a t
    val remove : key -> 'a t -> 'a t
    val get : key -> 'a t -> 'a
  end
```

課題 6 (例 2)

```
# open NCStringMap;;
# let sa = add "C" "/* */" empty;;
val sa : string NCStringMap.t = <abstr>
# let sa = add "OCaml" "(* *)" sa;;
val sa : string NCStringMap.t = <abstr>
# let sa = add "Perl" "#" sa;;
val sa : string NCStringMap.t = <abstr>
# get "ocaml" sa;;
- : string = "(* *)"
# get "ruby" sa;;
Exception: Not_found.
```

課題 7 (15 点)

- とある木の型を以下のように定義する

```
type 'a t =
  | Leaf
  | Node of 'a * 'a t t
```

- このとき、与えられた関数を木のノードの各要素に適用した木を返す関数 map を定義せよ

```
map : ('a -> 'b) -> 'a t -> 'b t
```

課題 8 (15 点)

- 以下のような signature を持つ module EQ を定義せよ
  - ただし、各関数は呼び出されれば必ず停止し
    例外が発生しないようにすること

```
module 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
```

課題 9 (15 点)

- 前回 (第2回) の課題6の値と式の定義を
  課題8の結果を用いて以下のように定義したとする:

```
type 'a value =
  | Bool of (bool, 'a) EQ.equal * bool
  | Int of (int, 'a) EQ.equal * int;;

type 'a expr =
  | Const of 'a value
  | Add of (int, 'a) EQ.equal * (int expr) * (int expr);;
```

- このとき前回の課題7と同様に
  式を評価して値を返す関数 **eval** を定義せよ

```
val eval : 'a expr -> 'a value
```