情報システム科学実習II第8回
モジュールとバッチコンパイル
システムプログラミング入門

担当: 山口 和紀・五十嵐 淳

2001年12月5日

1　モジュールとは

モジュール(*module*) とは英語で部品のことであり，プログラミング言語では「プログラムの部品」のことを指す．ひとつのプログラムを部分に切り分けることをモジュール化ということもある．ネジなどの部品が様々な製品に使えるのと同様に，プログラムモジュールも「他のプログラムのために再利用可能なプログラム断片」として分割されるのがよいとされている．そのため，通常は，なんらかのデータとそれを操作する定義の集合を提供することが多い．例えば，OCaml ライブラリではリスト操作を行う関数は`List`というモジュールにまとめられている．

プログラム全体を分割し，必要なモジュールを設計するのは (特に大規模な) プログラム構築において重要な技術である．なぜなら，既存のモジュールの再利用によるコストの軽減がはかれ，またモジュールごとの分割開発が可能になるからである．また，モジュールが階層的な名前空間 (関数などの名前がインターネットにおけるドメイン名のように階層的な構造を持てる) を提供することも見逃せない利点の一つだろう．

OCaml は他言語に比べてもかなり強力なモジュール機能を備えている．この実習では2週間にわたって，その機能をみていく．

2　ライブラリモジュールの使い方

まずは，ライブラリの使用法を学びながら，既存のモジュールの使い方をみてゆく．OCaml のライブラリは，`List`, `Array`, `Sort` モジュールなどのデータ構造に関するもの，`Printf` などの入出力に関連するもの，`Sys` モジュールなどの，OSやOCaml処理系とのインターフェースをとるためのもの等が豊富に用意されている．ひとつひとつの詳しい機能などはマニュアルの19章をみてほしい．ここでは `List` モジュールと `Queue` モジュールを題材にする．

モジュール内の関数は，⟨ モジュール名 ⟩.⟨ 関数名 ⟩ と言う形で呼び出すことができる．

```
# List.length [5; 6; 8];;
- : int = 3
# List.concat [[4; 35; 2]; [1]; [9; -4]];;
- : int list = [4; 35; 2; 1; 9; -4]
```

以前にみてきたリスト操作のための関数 rev, append, map, fold_left, fold_right 等は，ほとんど List モジュールに定義されている．

Queue モジュールは，いわゆる待ち行列のデータ構造を実装したもので，リストのように同種のデータをまとめて格納するのに用いる．add, take という要素を追加，取り出す関数が用意されているが，特徴は「先入れ先出し」であり，add した順番でしか take できない．Queue モジュールはモジュール内で独自の型を定義している．この場合には，その型にもモジュール名がついた形 'a Queue.t で表される．

```
# let q = Queue.create ();;
val q : '_a Queue.t = <abstr>
# Queue.add 1 q; Queue.add 2 q;;
- : unit = ()
# Queue.take q;;
- : int = 1
# Queue.take q;;
- : int = 2
# Queue.take q;;
Uncaught exception: Queue.Empty.
```

最後に発生した例外 Empty は Queue モジュール内で定義されたものである．

同じモジュールを使い続けると，いちいちモジュール名をつけるのが面倒になってくる．open 宣言は文字通りモジュールを「開く」もので，モジュール内の定義がモジュール名なしでアクセスできるようになる．

```
# open List;;

# length [3; 9; 10];;
- : int = 3
```

この open 宣言は，open した時点ですでに宣言されている同名の定義を隠してしまうので，同名の定義を提供する複数のモジュールを開くときには順番に注意した方がよい．隠されてしまった名前は，結局モジュール名つきの記法でアクセスすることになる．

3 モジュール宣言

さて，次に自前のモジュールを宣言してみよう．以下は，以前にみた，木構造に関する定義を集めたモジュール Tree の宣言である．

```
# module Tree =
#   struct
#     type 'a t = Lf | Br of 'a * 'a t * 'a t
#
#     let rec size = function
#         Lf -> 0
#       | Br (_, left, right) -> 1 + size left + size right
#
#     let rec depth = function
#         Lf -> 0
#       | Br (_, left, right) -> 1 + max (depth left) (depth right)
#   end;;
module Tree :
  sig
    type 'a t = Lf | Br of 'a * 'a t * 'a t
    val size : 'a t -> int
    val depth : 'a t -> int
  end
```

モジュール宣言は，上に見るように module に続けて，(大文字で始まる) モジュール名，= の右辺に struct, end ではさまれた宣言群を並べる．宣言群はこれまでに見た，let による値，関数宣言，type による型宣言，exception による例外宣言，open 宣言だけでなく module 宣言を入れ子にすることもできる．基本的には，コンパイラのプロンプトに打ち込んでいたものは書けると思ってよい．(例外は，ただの式で，これは宣言ではないので書くことができない．) また，各宣言の後には ;; が省略されていることに注意．

コンパイラからは，モジュール内の宣言の型情報が返ってきている．module Tree : sig や，end を除けば，各宣言を個別に打ち込んだときと同じものが返ってきていることがわかるだろう．sig, end で囲まれた部分全体を **シグネチャ**(*signature*) と呼び，モジュール全体の型のようなものを表す．

さて，もうひとつ，辞書またはテーブルと呼ばれるデータ検索のためのデータ構造のモジュールを定義してみよう．テーブルは，

- 見出し (キーと呼ばれる) とその内容の組の集合で，
- empty は空のテーブルを表現し，
- add 関数で，キーと内容の組を追加
- retrieve で，キーから内容の検索
- 発見できない場合には，例外 Not_found を発行する

ものとする．

もっとも単純な実装は，テーブルをキーと内容の組のリストとして表現することであろう．キーと内容は固定せずに多相的なテーブルを定義する．

```
# module DictionaryL =
#   struct
#     type ('a, 'b) t = ('a * 'b) list
#
#     exception Not_found
#
#     let empty = []
#
#     let add key_contents table = key_contents :: table
#
#     let rec retrieve key = function
#         [] -> raise Not_found
#       | (key', contents) :: rest ->
#         if key = key' then contents else retrieve key rest
#
#     let size = List.length
#   end;;
module DictionaryL :
  sig
    type ('a, 'b) t = ('a * 'b) list
    exception Not_found
    val empty : 'a list
    val add : 'a -> 'a list -> 'a list
    val retrieve : 'a -> ('a * 'b) list -> 'b
    val size : 'a list -> int
  end
```

4 シグネチャと情報隠蔽，抽象データ型

シグネチャはいわばモジュールの型を表すわけだが，ただ単に，定義したモジュールの型をコンパイラから受け取って (プログラマが) 読み取るだけのものではない．シグネチャはモジュールと外界 (つまりモジュールを使う側) とのインターフェースを規定するものとして積極的に活用することができる．

まず，プログラマは明示的にシグネチャを宣言し，モジュールに与えることによって，定義の隠蔽を行うことができる．この定義の隠蔽には，モジュール内でのみ使用される補助的な定義の隠蔽だけでなく，型定義の詳細，例えば辞書がリストを使って実装されていることなど，の隠蔽が含まれている．このような情報隠蔽のことを**カプセル化**(*encapsulation*) と呼ぶこともある．

また，トップダウンな見方をしてみよう．プログラムの開発は，まず部品の仕様を決定し，部品を実装することが一般的であるが，この方式のメリットは，いったん仕様が固まれば実装は独立に行えることにある．これをモジュールの言葉に置換えると,「シグネチャを確定すればその実装であるモジュールは独立して開発可能 (テストは可能でないかもしれないが) である」ことを意味する．また，シグネチャで要請される機能さえ満たせばモジュールを入れ

替えることも容易になる．

ここではひとまずシグネチャによる情報隠蔽の観点から辞書モジュールを再吟味してみよう．辞書の仕様をみてみると，辞書を表す型がリストであることは特に要請されていないわけである．つまり「なんらかの型 (ただしキーと要素型に関してパラメータ化されている)」があって，add は，キーと内容の組と,「なんらかの型」をもつ辞書データから，新たな辞書データを生成すればよい．これを表現するようなシグネチャを実際に宣言してみよう．

```
# module type DICT =
#   sig
#     type ('a, 'b) t
#     exception Not_found
#     val empty : ('a, 'b) t
#     val add : ('a * 'b) -> ('a, 'b) t -> ('a, 'b) t
#     val retrieve : 'a -> ('a, 'b) t -> 'b
#   end;;
module type DICT =
  sig
    type ('a, 'b) t
    exception Not_found
    val empty : ('a, 'b) t
    val add : 'a * 'b -> ('a, 'b) t -> ('a, 'b) t
    val retrieve : 'a -> ('a, 'b) t -> 'b
  end
```

シグネチャは module type 宣言で宣言する．DICT はシグネチャの名前 (慣習としてすべて大文字の名前を用いることが多い)，= の右辺の sig, end で囲まれた部分に，モジュール内の定義の「仕様」を記述する．仕様は基本的には，コンパイラの応答にみられる表現が使われる．とりあえず，このシグネチャを読めば，DICT シグネチャを実装するモジュールは型 t, 例外 Not_found，値 empty, add, retrieve を宣言しなくてはいけないのだな，ということがわかる．

DICT の定義を DictionaryL の定義時にコンパイラの出力したシグネチャと比べると，まず，size が入っていないことがわかる．また type の部分も異なっている．モジュール宣言時にシグネチャを指定するのは，値の型を宣言時に指定するのと似ていて，

```
# module DictionaryAbs : DICT = DictionaryL;;
module DictionaryAbs : DICT
```

のように : の後にシグネチャを指定する．ここでは宣言済のシグネチャ DICT を使ったが sig end を直接書くこともできる．また，モジュール宣言の = の右側に定義済のモジュールを記述することができる．

さて，実体は同じ定義である DictionaryL と DictionaryAbs は実際何が違うのであろうか．まず，DictionaryAbs からは size にアクセスすることはできない．

```
# DictionaryL.size;;
- : 'a list -> int = <fun>
# DictionaryAbs.size;;
```

```
Characters 0-18:
Unbound value DictionaryAbs.size
```

type の部分の違いはどう現れるのであろうか．DictionaryAbs では t の定義の中身が見えないので，辞書データがリストであるということは,「見えなく」なっている．このことは例えば以下のことで確かめられる．

```
# let emp = DictionaryAbs.empty;;
val emp : ('a, 'b) DictionaryAbs.t = <abstr>
# DictionaryL.add ("A", "the first letter of the English alphabet") [];;
- : (string * string) list =
["A", "the first letter of the English alphabet"]
# DictionaryAbs.add ("A", "the first letter of the English alphabet") emp;;
- : (string, string) DictionaryAbs.t = <abstr>
# DictionaryAbs.add ("A", "the first letter of the English alphabet") [];;
Characters 68-70:
This expression has type 'a list but is here used with type
  ('b, 'c) DictionaryAbs.t
```

最後の式でわかるように [] と emp はもはや同じものとしてモジュールの外では扱われていない．結果として，この辞書データについて操作できる関数は DictionaryAbs モジュールの add と retrieve だけに限定される．このような，抽象化されたデータ型がそれに関する操作の集合とともに宣言されたものを**抽象データ型**(*abstract data type*) という．抽象データ型はモジュールを実装する側から言えば，データの内部表現を勝手にいじらせないための境界線を設けられるという意味で重要な抽象化の手段の一つである．

このように，シグネチャは，モジュール外部に見せたくない関数や，型定義の詳細 (データの実装方法を表現していることが多い) を隠蔽して抽象データ型を作成するのに用いることができる．

シグネチャマッチング　モジュールをシグネチャなしに宣言したときにコンパイラが応答として返すシグネチャは,「もっとも一般的な」ものであり，モジュール内の情報をなにも隠していないものである．モジュールをシグネチャ付で宣言した際には，この「もっとも一般的な」シグネチャと比較することで，モジュールが与えられたシグネチャ(仕様) をみたすことが検査される．

与えられたシグネチャに以下の操作を行って，モジュールの「もっとも一般的な」シグネチャが得られるとき，満たすといわれる．

- 仕様 type... や val... などの追加
- type t を type t = ... と定義を明らかにする．
- type t = ⟨型⟩ があったら，シグネチャ内の以降の t の出現を ⟨型⟩ で置換える．

3番目の操作は，t の定義内容をそれが参照されているところに伝播していく意味をもつ．例えば，DICT に val size : ... を加え，type 'a t を type ('a, 'b) t = ('a, 'b) list

にかえて隠蔽された定義を明らかにしても，まだ正確には DictionaryL のシグネチャとは一致していない．

```
val retrieve : 'a -> ('a, 'b) t -> 'b
```

を

```
val retrieve : 'a -> ('a, 'b) list -> 'b
```

として始めて一致する．

このようにして，与えられたシグネチャを満たすことの検査を**シグネチャ マッチング**(*signature matching*) ということがある．

Exercise 8.1 以下は，(副作用を伴わないで実装される) 待ち行列のシグネチャである．多相型 $\langle t \rangle$ Queue.t が要素が $\langle t \rangle$ であるような待ち行列を表現する．先に紹介したライブラリのモジュールと違い，副作用を伴わないので，各操作は新しい待ち行列を生成する関数として表現される．

```
# module type QUEUE =
#   sig
#     type 'a t
#     exception Empty
#     val empty: 'a t
#     val enqueue: 'a t -> 'a -> 'a t
#     val dequeue: 'a t -> 'a t
#     val hd: 'a t -> 'a
#     val null: 'a t -> bool
#   end;;
module type QUEUE =
  sig
    type 'a t
    exception Empty
    val empty : 'a t
    val enqueue : 'a t -> 'a -> 'a t
    val dequeue : 'a t -> 'a t
    val hd : 'a t -> 'a
    val null : 'a t -> bool
  end
```

値の意味は

- 空の待ち行列 empty
- 待ち行列の一番後ろに要素を追加する enqueue
- 一番先頭の要素を除去する dequeue
- 一番先頭の要素を (除去せずに) 取出す hd

- 待ち行列が空か判定する null

である．これの実装を二種類せよ．ひとつめのモジュール Queue1 は，待ち行列をリストで表現する．例えば，[1; 5; 4; 2; 3] は待ち行列 1，5，4，3，2 を表現する．もうひとつの Queue2 は，待ち行列をリストのペアで表現する．たとえば，([1; 5; 4], [2; 3]) が待ち行列 1，5，4，3，2 を表現する．(同じ待ち行列を表現するリストのペアは複数あり得る．) ただし，待ち行列が空でない限り一つ目のリストが空でないように工夫せよ．以下に，モジュール定義の一部を示すので，完成させた上で，うまく動くことを確かめよ．

```
module Queue1 : QUEUE =
  struct
    type 'a t = 'a list
    ...
    let hd = function [] -> raise Empty | x :: rest -> x
  end

module Queue2 : QUEUE =
  struct
    type 'a t = Queue of ('a list * 'a list)
    ...
    let hd = function (Queue ([],_) -> raise Empty | Queue (x :: _, _)) -> x
  end
```

Exercise 8.2 次の抽象データ型を表す，ふたつのシグネチャをもつモジュールは，どちらもあまり実用上意味がない．なぜか? (ヒント: empty のない DICT モジュールや，add のない DICT モジュールを考えてみよ．)

```
module type BOGUS1 =
  sig
    type t
    val f: t -> int -> t
  end
module type BOGUS2 =
  sig
    type t
    val e: t
  end
```

5 バッチコンパイラと分割コンパイル

さて，ここでインタラクティブコンパイラからいったん離れて，(プログラムファイルから実行形式ファイルを生成する) バッチコンパイラ ocamlc の使用法と，分割コンパイルの方法を学ぶ．

もっとも単純な ocamlc の使用法は，OCaml の宣言を書いたファイル (拡張子は .ml) を用意して，シェルのコマンドラインから

ocamlc -o ⟨出力ファイル名⟩ ⟨*OCaml* ソースファイル名⟩

としてコンパイラを 起動すると，⟨出力ファイル名⟩という実行可能なファイルが生成される． -o オプションを省略すると a.out という名前で生成される．

```
igarashi@zither:text> cat hello.ml
let _ = print_string "Hello, World!\n"
igarashi@zither:text> ocamlc hello.ml
igarashi@zither:text> a.out
Hello, World!
igarashi@zither:text> cat fact.ml
let rec fact n =
  if n = 0 then 1 else n * fact (n - 1)
let _ = print_int (fact 10)
igarashi@zither:text> ocamlc -o fact10 fact.ml
igarashi@zither:text> ./fact10
3628800igarashi@zither:text>
```

バッチコンパイルされるファイルの中には宣言の並びだけが許され，インタラクティブコンパイラで見たような，式だけからなるものははじかれるので，let _ = のような，式を評価して結果を捨てる宣言として記述している．(C の main 関数に相当するものがなく，ただ単に上から評価を行っていく．)

さて，ソースファイルが複数のファイルに分割された場合を考えよう． OCaml システムでは，一つ一つのソースファイルがモジュールに対応し，UNIX 上でのファイル名の先頭を大文字にしたものが，OCaml でのモジュール名になる．例えば，foo.ml のソースファイル中の宣言は，他のファイルからは，モジュール Foo にあるものとしてアクセスされる．下は，fact.ml と main.ml である．

```
igarashi@zither:samples> cat fact.ml
let rec fact n =
  if n = 0 then 1 else n * fact (n - 1)
igarashi@zither:samples> cat main.ml
(* main.ml *)
let _ =
  print_int (Fact.fact 10);
  print_newline();
```

main.ml では Fact モジュール内の fact 関数を Fact.fact という名前で使用している．コンパイラには，二つのファイル名を，依存されているものから順に並べる．

```
igarashi@zither:samples> ocamlc -o fact10 fact.ml main.ml
igarashi@zither:samples> fact10
3628800
```

また，-c オプションを用いると，各モジュールを個別にコンパイルすることができる．中間的なオブジェクトファイルとして，.cmi という拡張子を持つモジュールのシグネチャをコンパイルしたファイルと，.cmo という拡張子を持つモジュール自体をコンパイルしたファ

イルが生成される．.cmi ファイルは，そのモジュールを使用するファイルがコンパイルされるときに必要であり，(.cmo 自体は必要ではない．) 下の例で，main.ml を先にコンパイルすることはできない．そして，.cmo はあとでリンクして実行可能ファイルを生成することができる．

```
igarashi@zither:samples> ocamlc -c fact.ml
igarashi@zither:samples> ocamlc -c main.ml
igarashi@zither:samples> ocamlc -o fact10 fact.cmo main.cmo
```

各モジュールのシグネチャは -i オプションで標準出力に書き出すことができる．

```
igarashi@zither:samples> ocamlc -i -c fact.ml
val fact : int -> int
```

プログラマはこれを見て，関数に意図通りの型が与えられていることを確認することができる．

さて，ここまで見てきたのは，あらかじめすべてのソースファイルがそろっている場合のコンパイルの方法であった．各モジュールのシグネチャを明示的に**あらかじめ**記述しておくことで，一部のモジュールを記述することなく，それに依存するモジュールをコンパイルすることができる．シグネチャは .mli という拡張子のファイルに記述し，中身としては -i オプションで得られるのと同じようなもの (module type 宣言で sig...end の間に記述したもの) を並べておく．.mli を -c オプション付きでコンパイルすると .cmi が生成される．以下は，fact.ml なしに，main.ml をコンパイルした例である．

```
igarashi@zither:samples> ls fact.ml
ls: fact.ml: そのようなファイルやディレクトリはありません
igarashi@zither:samples> ocamlc -c fact.mli
igarashi@zither:samples> ocamlc -c main.ml
```

また，あらかじめシグネチャを用意しない場合でも，.mli ファイルは先に見た DICT の例のように，モジュール内の宣言の一部を隠蔽するのに使うことができる．この場合，既に実装したモジュールから -i オプションで，シグネチャを得て，適宜外に見せたくない定義を削る，という作業が行われる．

このバッチコンパイラを使ったプログラミングは，インタラクティブコンパイラであれば，次の宣言列に対応する．

```
module Fact : sig (* contents of fact.mli *) end
  = struct (* contents of fact.ml *) end;;
module Main
  = struct (* contents of main.ml *) end;;
```

シグネチャファイル main.mli がないので，Main モジュールのシグネチャは指定されない．

6　モジュール，シグネチャの文法

ここまでに出てきたモジュール関連の文法をまとめると，図 1となる．詳しくはマニュアルを参照してもらいたい．

シグネチャ宣言

`module type` ⟨ シグネチャ名 ⟩ `=` ⟨ モジュール型 ⟩

⟨ モジュール型 ⟩ ::= ⟨ シグネチャ名 ⟩
| `sig` ⟨*spec*⟩ の並び `end`
⟨*spec*⟩ ::= `val` ⟨ 名前 ⟩ `:` ⟨ 型 ⟩
| ⟨ 型宣言 ⟩
| ⟨ 例外宣言 ⟩
| `open` ⟨ モジュール名 ⟩

モジュール宣言

`module` ⟨ モジュール名 ⟩ `[:` ⟨ モジュール型 ⟩`]` `=` ⟨ モジュール式 ⟩

⟨ モジュール式 ⟩ ::= ⟨ モジュール名 ⟩
| `struct` ⟨*decl*⟩ の並び `end`
⟨*decl*⟩ ::= ⟨`let` 宣言 ⟩
| ⟨ 型宣言 ⟩
| ⟨ 例外宣言 ⟩
| `open` ⟨ モジュール名 ⟩

図 1: モジュールの文法

7　システム周りのモジュール

その他の演習に必要な関数・モジュールを紹介しておこう.

exit 関数　この関数は実行終了時の実行終了コードを決定する関数である．以下は，UNIX の true, false コマンドを実装したプログラムである.

```
igarashi@zither:samples> cat true.ml
let _ = exit 0
igarashi@zither:samples> cat false.ml
let _ = exit 1
```

コマンドライン引数　コマンドラインで与えられる引数は Sys モジュールの argv 変数 (string array 型) に格納されている．下のプログラムは，コマンドラインのコマンド名も含めた各引数を出力する.

```
# let _ =
#   for i = 0 to Array.length Sys.argv - 1 do
#     print_endline Sys.argv.(i)
#   done;;
/usr/local/bin/ocaml
- : unit = ()
```

Sys モジュールには他にも，ファイルの有無を調べる file_exists, ファイルの消去，名前換えを行う remove, rename，環境変数の値をとってくる getenv などが用意されている.

Printf モジュール　このモジュールには C の printf に相当する関数が用意されている．使い方はほぼ同じである.

```
# let x = 110;;
val x : int = 110
# Printf.printf "decimal: %d hexadecimal: %x string: %s\n" x x "foo";;
decimal: 110 hexadecimal: 6e string: foo
- : unit = ()
```

その他，出力チャネル名を第一引数として取る fprintf, 標準エラー出力に出力する eprintf, 文字列を返す sprintf 関数が用意されている.

```
# Printf.fprintf stdout "decimal: %d hexadecimal: %x string: %s\n" x x "foo";;
decimal: 110 hexadecimal: 6e string: foo
- : unit = ()
# Printf.sprintf "decimal: %d hexadecimal: %x string: %s" x x "foo";;
- : string = "decimal: 110 hexadecimal: 6e string: foo"
```

Arg モジュールとコマンドラインオプション　Arg モジュールはコマンドラインのオプションを解析するのに便利なライブラリである．コマンドラインオプションは ⟨ キーワード ⟩ の形で与えられ，整数，文字列，実数の引数を扱うことができる．プログラム内部では，キーワードの名前と，(あれば) その引数の型，そのオプションが与えられたときの挙動，キーワードの説明文を (string * spec * string) list 型の値で指定する．最初の string がキーワード名，最後の string が説明文である．spec 型は以下で定義されるヴァリアント型で，キーワードに対する挙動を関数を使って記述する．

```
type spec =
    Unit of (unit -> unit)  (* 引数無しキーワード *)
  | Set of bool ref         (* 引数無しキーワード:
                               与えられた参照に true を代入する *)
  | Clear of bool ref       (* 引数無しキーワード:
                               与えられた参照に false を代入する *)
  | String of (string -> unit) (* 文字列引数のキーワード *)
  | Int of (int -> unit)       (* 整数引数のキーワード *)
  | Float of (float -> unit)   (* 実数引数のキーワード *)
  | Rest of (string -> unit)   (* -- 以後の引数の解釈 *)
```

実際のコマンドライン解釈は，Arg.parse 関数で行う．

```
Arg.parse [("-n", Set display_num, "Display line number")]
          (fun s -> filenames := s :: !filenames)
          "Usage: cat [-n] filename ..."
```

第一引数 [...] は各オプション (この場合は -n ひとつ) の挙動を決めている．第二引数 fun s -> ... はオプションでない引数の処理を行う関数で，ここでは filenames という文字列リストの参照へ次々に名前を追加している．典型的には，参照を使って後のプログラムの挙動に影響を与えるフラグ操作をしたりすることが多い．最後の文字列はエラーが起きたときなどに表示するメッセージである．

例えば，演習問題に出てくる UNIX の cat コマンドのプログラムの引数処理は以下のように書ける．

```
let version = "0.1"
let display_linenum = ref false
let filenames = ref []

let spec = [("-n", Arg.Set display_linenum, "Display line number");
            ("-version",
             Arg.Unit
               (fun () -> Printf.printf "cat in OCaml version: %s\n" version),
             "Display version number")]

let _ =
  Arg.parse spec
    (fun s -> filenames := s :: !filenames)
    "Usage: cat [-n] [-help] [-version] filename ...";
```

```
if !display_linenum then print_endline "-n was turned on";
List.iter (fun s -> Printf.printf "filename specified is: %s\n" s)
  (List.rev !filenames)
```

実行結果は以下のようになる．Arg.parse 関数を使うと -help オプションがデフォルトで装備され，各オプションの説明を表示してくれる．

```
igarashi@zither:samples> ./cat -n a b c
-n was turned on
specified filename is: a
specified filename is: b
specified filename is: c
igarashi@zither:samples> ./cat -version
cat in OCaml version: 0.1
igarashi@zither:samples> ./cat -help
Usage: cat [-n] [-help] [-version] filename ...
  -n Display line number
  -version Display version number
  -help  display this list of options
```

Exercise 8.3 UNIX の cat コマンドを実装せよ．オプションに関しては多く実装すればするほどよいが，-n オプションは少なくとも実装すること．挙動に関しては man ページ参照のこと．

Exercise 8.4 UNIX の wc コマンドを実装せよ．オプションに関しては多く実装すればするほどよい．細かい挙動に関しては man ページ参照のこと．

Exercise 8.5 UNIX の fold コマンドを実装せよ．オプションに関しては多く実装すればするほどよいが -w オプションは少なくとも実装すること．細かい挙動に関しては man ページ参照のこと．

A　レポートその 3: 締切 12 月 19 日

必修課題:　6.2, 6.3, 6.6, 6.8, 6.9, 7.3, 7.4, 7.6, 7.8, 8.1, 8.2, 8.3–5 の 3 問の中から 2 問．

オプション課題:　第 6–8 回の資料の残り全部の問題．

レポート形式について　8.3 以降の問題に関しては，バッチコンパイラで実行可能ファイルをつくる関係上，ソースファイル (群) はレポートに埋め込まず，ソースを tar コマンドでアーカイブして送付することが望ましい．また，説明はソースファイルにコメントとして埋め込むこと．複数のファイルにソースがわかれる場合，コンパイル方法をレポート本体に書いてあるとよい．