ML 演習 第 1 回

新井淳也、中村宇佑、前田俊行

2011/04/05

今日から ML 演習

- 担当者
  - 新井淳也 (石川研)、中村宇佑 (萩谷研)、前田俊行
  - 質問アドレス: ml-query-2011@yl.is.s.u-tokyo.ac.jp
  - 課題提出アドレス: ml-report-2011@yl.is.s.u-tokyo.ac.jp

演習の内容

- 第 1 回～第 3 回
  - ML 言語を学ぼう
    - 本演習では ML の一派である Objective Caml (OCaml) を使用
- 第 4 回～第 7 回
  - ML インタプリタを作ってみよう
- 第 8 回
  - 最終課題: リバーシ思考ルーチンの実装 (予定)

演習資料について

- ML演習ホームページ

  http://ukulele.is.s.u-tokyo.ac.jp/wiki/lectures/CamlEnshu2011.ja

  – 講義資料(PDF, PowerPoint)
  – 演習で使用するソースファイル
  – 参考資料へのリンク

参考資料

- OCaml の本家サイト http://caml.inria.fr/
  - マニュアル・紹介など
    - 第 1 章のチュートリアルは簡潔でよい
  - 処理系のダウンロード (Windows 版など)
- Developing Applications With Objective Caml
  - Online pre-release
  - http://caml.inria.fr/pub/docs/oreilly-book/

日本語の参考資料

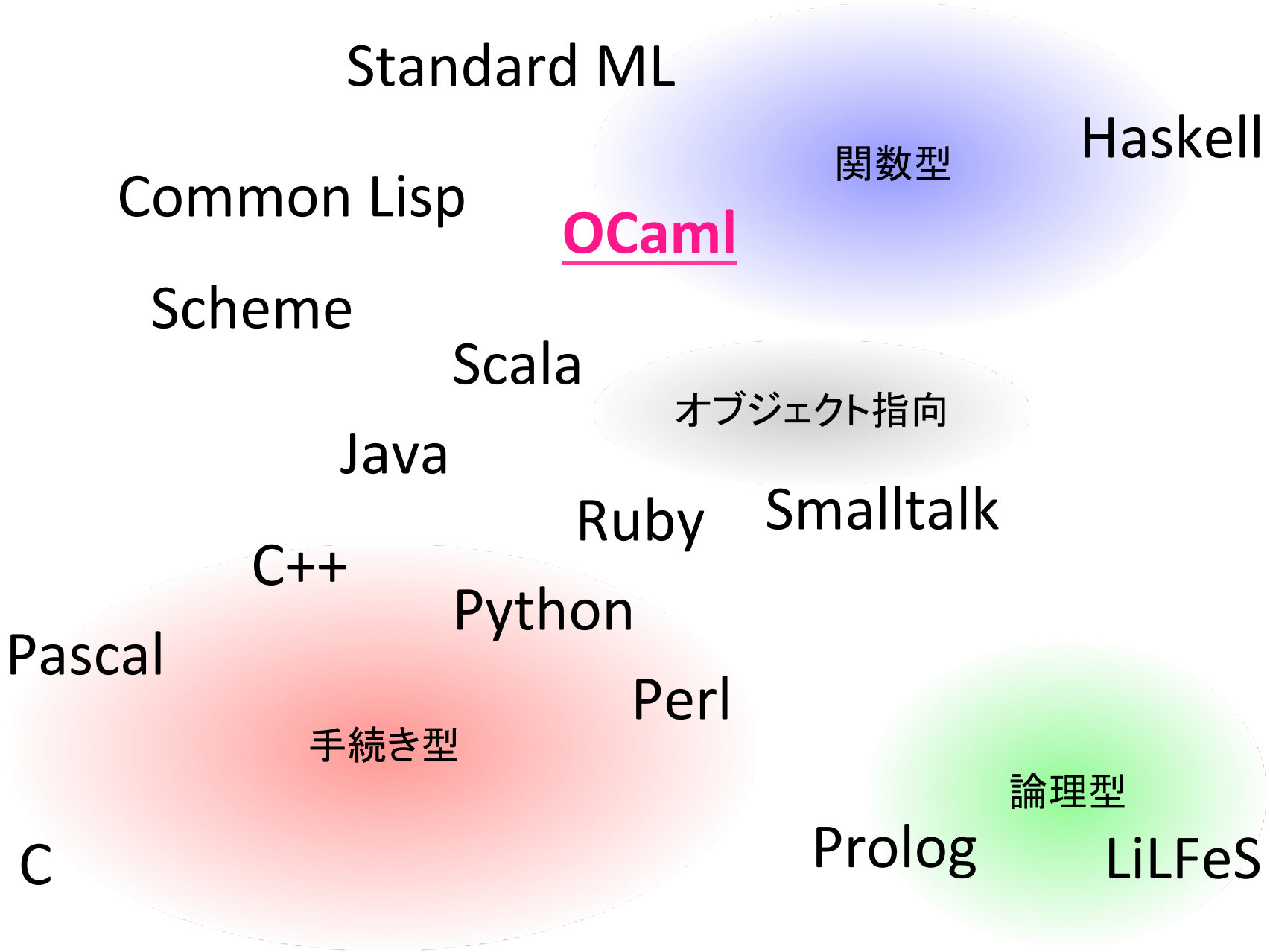
- プログラミング in OCaml (技術評論社)
  - 五十嵐 淳 著
  - ISBN 978-4774132648
- プログラミングの基礎（サイエンス社）
  - 浅井 健一 著
  - ISBN 978-4781911609
- 入門 OCaml プログラミング基礎と実践理解
  - OCaml-Nagoya 著
  - ISBN 978-4839923112

今日の内容

- Objective Caml とは?
- インタプリタの使い方
- 基本的な式の構文(文法)
- パターンマッチング
- レポートについて
- 課題

Objective Caml とは?

- 関数型言語 ML の一流派
  - 強力な型システム
  - 柔軟なデータ型定義とパターンマッチ
  - 強力なモジュールシステム
  - オブジェクト指向プログラミングのサポート



(Original figure courtesy of E. Sumii. Slightly modified by authors)

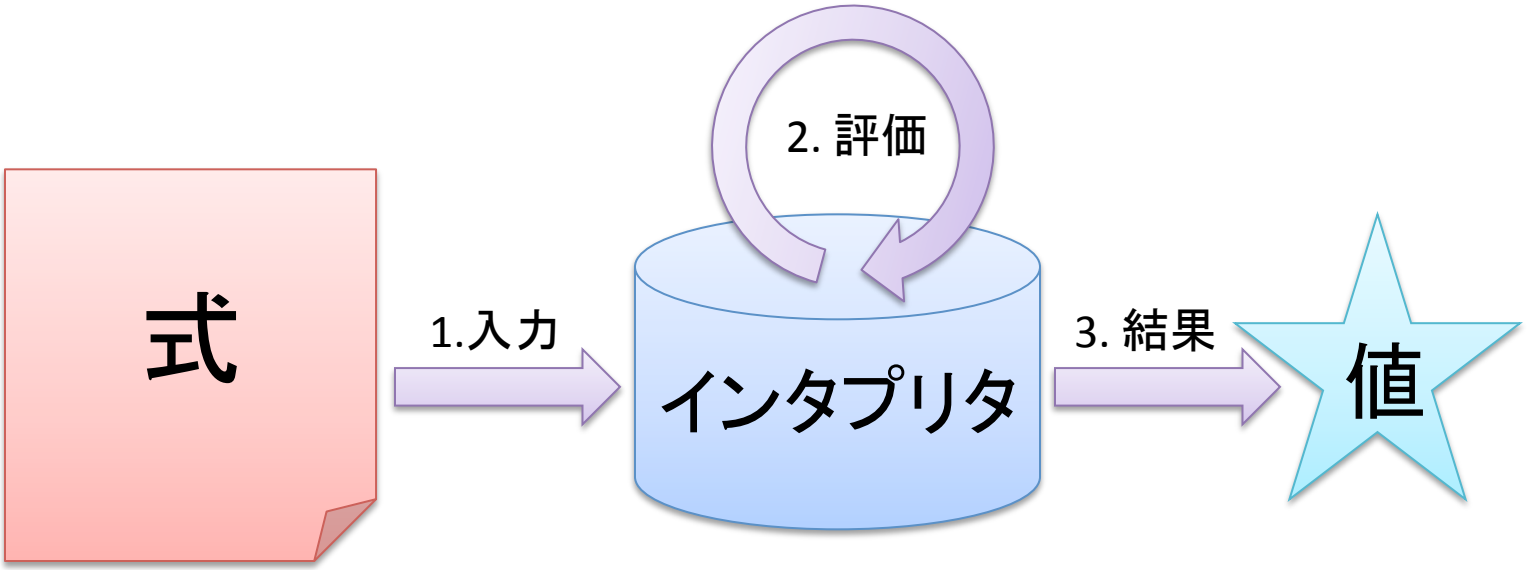
MLの型システムの特徴

- 強い静的な型付け (Strong Static Typing)
  - 「強い」= 型整合を強制する
    - メモリエラーなどが絶対に生じない
      - Cf. 弱い型付け (e.g. C, C++)
  - 「静的な」= コンパイル時に型をチェックする
    - 実行時のオーバーヘッドがない
      - Cf. 動的な型付け (e.g. Scheme, Perl)
- 型推論
  - プログラマは変数等の型を書かなくてよい
- 型多相 (型の多態性: Parametric Polymorphism)
  - 第 2 回で解説

インタプリタの使い方

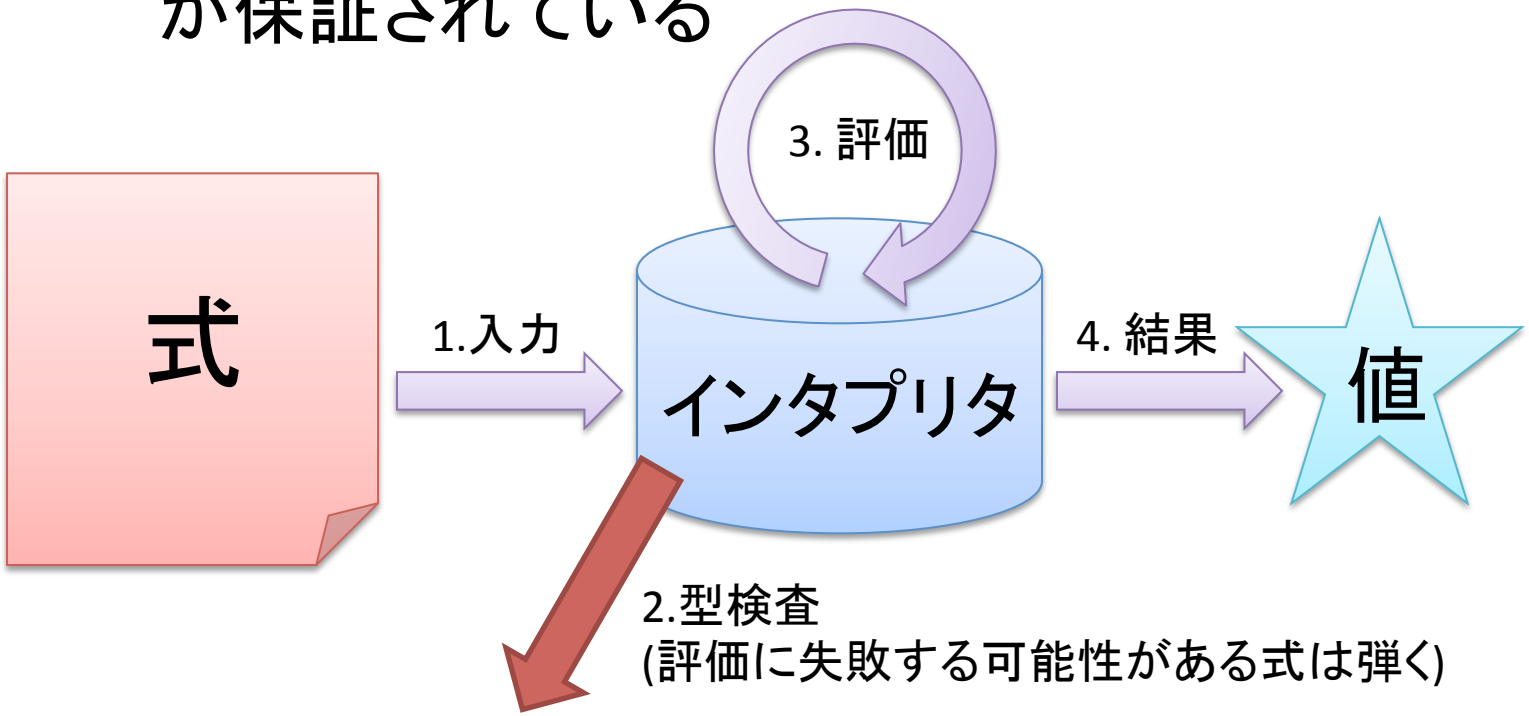
インタプリタとは

- 「式」を入力として受け取り
  その式を「評価」して
  その評価の結果の「値」を返すプログラム



インタプリタと型検査

- O'Caml のインタプリタは
  式を評価する前に「型検査」を行う
  - 型検査にパスした式の評価は失敗しないことが保証されている

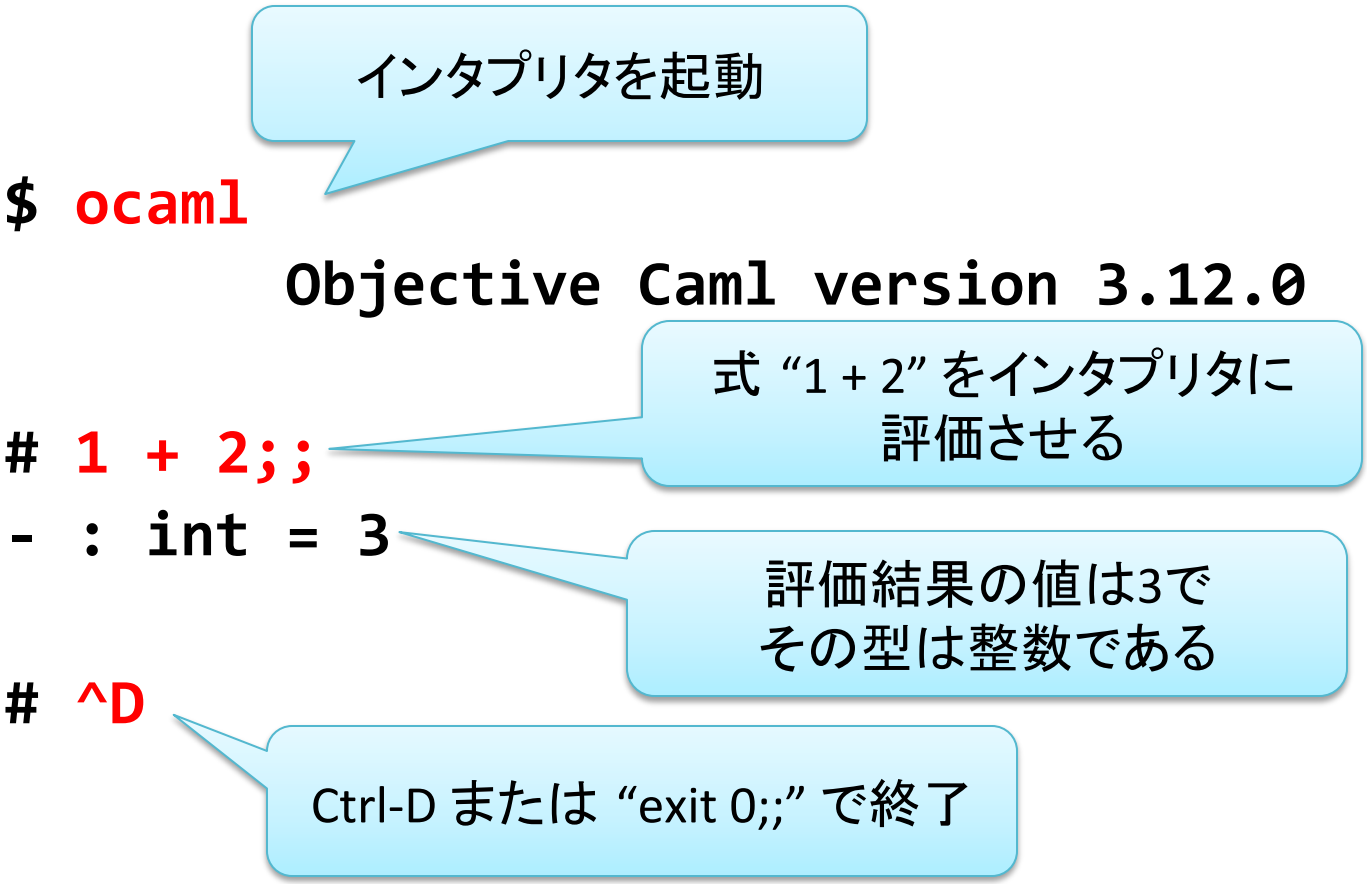


インタプリタの準備

- 配布ノートPCにはインストール済み
- 地下マシン (csc, csp) にもインストール済み

- その他のマシンを使う場合は各自インストール
  - Unix
    - パッケージを利用する (Ubuntu なら apt など)
    - ソースからコンパイルする
  - Windows
    - 配布されているバイナリを利用する
    - Cygwin ならインストーラを使えば入れられる
    - ソースからコンパイル (VC, mingw32)

インタプリタの使い方


インタプリタを起動
$ ocaml
Objective Caml version 3.12.0
式 “1 + 2” をインタプリタに
評価させる
1 + 2;;
- : int = 3
評価結果の値は3で
その型は整数である
^D
Ctrl-D または “exit 0;;” で終了

ファイルに書いたプログラムの利用

- 直接式を打つ代わりに
  テキストファイルの内容を
  読み込ませることができる

```
# #use "test.ml";;
- : int = 3
- : float = 7.
```

“test.ml” の中身:

```
1 + 2;;
3.0 +. 4.0;;
```

基本的な式の構文(文法)

- コメント, 整数, 浮動小数点数, ブール演算, 比較演算, 変数, 関数, 条件分岐, 文字列, タプル, リスト

コメントの書き方

- (* と *) の間はコメントとして無視される

```
# (* comment *) 2 + 3;;
- : int = 5
```

- なんと入れ子にもできる (C ではできない)

```
# 1 (* + 2 + (* 3 + *) 4 *) + 2;;
- : int = 3
```

整数値を表す式

```
# 3;; (* 整数値3を表す式を評価すると… *)
- : int = 3 (* 整数値3が返される *)
# 12345678;;
- : int = 12345678
# 0x12345678;; (* 16進表現も使える *)
- : int = 305419896
# 0b101010;; (* 2進表現も使える *)
- : int = 42
# 0o1234567;; (* 8進表現も使える *)
- : int = 342391
```

整数演算を表す式

```
# 13 + 4;; (* 和：「式」+「式」*)
- : int = 17
# 13 - 4;; (* 差：「式」-「式」*)
- : int = 9
# 13 * 4;; (* 積：「式」*「式」*)
- : int = 52
# 13 / 4;; (* 商：「式」/「式」*)
- : int = 3
# 13 mod 4;; (* 余：「式」mod「式」*)
- : int = 1
# -(13 + 4);; (* 符号反転：-「式」*)
- : -17
# (3 + 5) * 8 / -4;; (* 括弧で結合順の変更可 *)
- : int = -16
```

浮動小数点数値を表す式

```
# 3.1415;; (* 値3.1415を表す式を評価すると… *)
- : float = 3.1415 (* 値3.1415が返される *)
# 3.1415e-10;; (* 指数表現も使える *)
- : float = 3.1415e-10
# infinity;; (* 無限大 *)
- : float = infinity
# nan;; (* Not a number *)
- : float = nan
# epsilon_float;;
- : float = 2.22044604925031308e-16
```

浮動小数点数演算を表す式

- 加減乗除は整数と異なる演算子を用いる

```
# 13.0 +. 4.0;; (* 和：「式」+.「式」*)
- : float = 17.
# 13.0 -. 4.0;; (* 差：「式」-.「式」*)
- : float = 9.
# 13.0 *. 4.0;; (* 積：「式」*.「式」*)
- : float = 52.
# 13.0 /. 4.0;; (* 商：「式> /「式」*)
- : float = 3.25
# 1.41421356 ** 2.0;; (* 累乗：「式」**「式」*)
- : float = 1.9999999328787381
# 13.0 + 4.0;; (* 整数とは混ぜられない *)
Error: This expression has type float but is here
  used with type int
# 13 +. 4.0;;
Error: This expression has type float but is here
  used with type int
```

ブール値を表す式

```
# true;; (* ブール値trueを表す式を評価すると… *)
- : bool = true (* ブール値trueが返される *)
# false;; (* ブール値falseを表す式を評価すると… *)
- : bool = false (* ブール値falseが返される *)
```

比較演算を表す式

```
# 2 = 3;; (* 等号：「式」=「式」*)
- : bool = false
# 2.0 <> 3.0;; (* 等号の否定：「式」<>「式」*)
- : bool = true
# 2 < 3;; (* 不等号：「式」<「式」*)
- : bool = true
# 5.0 <= 5.0 ;; (* 不等号：「式」<=「式」*)
- : bool = true
# 5.0 > 5.0;; (* 不等号：「式」>「式」*)
- : bool = false
# 2.5 >= 3.5;; (* 不等号：「式」>=「式」*)
- : bool = false
```

ブール演算を表す式

```
# 2 < 3 && 2.0 >= 3.0;; (* 論理積：「式」&&「式」*)
- : bool = false
# 2 < 3 || 2.0 >= 3.0;; (* 論理和：「式」||「式」*)
- : bool = true
# not false;; (* 否定：not「式」*)
- : bool = true
```

変数を定義する式

- トップレベルに変数を定義する

```
# let alice = 3;; (* let「変数」=「式」*)
val alice : int = 3
(*↑「式」を評価した結果の値に「変数」を束縛 *)
# let bob = 1.41421356 ** 2.0;;
val bob : float = 1.9999999328787381
# let charlie = nan = nan;;
val charlie : bool = false
```

変数を用いた式

```
# let root_2 = 1.41421356;;
val root_2 : float = 1.41421356
# root_2 ** 2.0;; (* 変数「root_2」を用いた式を評価 *)
- : float = 1.99999999328787381
# let r = root_2 ** 2.0;; (* さらに変数束縛できる *)
val r : float = 1.99999999328787381
```

局所的に変数を定義する式

- 使える範囲が限定された変数を定義する

```
(* let「変数」=「式1」in「式2」
   「式1」を評価した結果に「変数」を束縛して
   「式2」を評価*)
# let x = 3 in x + x ;;
- : int = 6
# x;;
Error: Unbound value x
# let x = 3 in (let x = 5 in let x =
    let x = x in x + x in x + x) + x;;
- : int = ?
```

変数「x」は
この範囲内でのみ
利用可能

関数 (closure) を表す式

```
(* 関数定義: fun「変数(引数)」->「式(本体)」*)
# fun x -> x + 2;;
- : int -> int = <fun>
# fun x y -> x + y;; (* fun x -> fun y -> x + y *)
- : int -> int -> int = <fun>
# let pi = 3.1415;;
val pi : float = 3.1415
# fun r -> pi *. r *. r;;
- : float -> float = <fun>
```

let 式を用いた関数の定義

```
(* 関数定義:
   let「変数(関数名)」「変数(引数)」… =「式(本体)」*)
# let add x y = x + y;;
val add : int -> int -> int = <fun>
(*↑let add = fun x -> fun y -> x + y;;
  と等しい(カリー化)*)
# let y = 2;;
val y : int = 2
# let add2 x = x + y;;
val add2 : int -> int = <fun>
```

関数適用(呼出)を表す式

```
(* 関数適用(呼出):「式1」「式2」… *)
# (fun x -> x + 2) 40;;
- : int = 42
# let add x y = x + y;;
val add : int -> int -> int = <fun>
# add 40 2;;
- : int = 42
# let f = add 40;; (*「部分適用」*)
val f : int -> int = <fun>
# f 2;;
- : int = 42
```

条件分岐を表す式

```
(* 条件分岐: if「式」then「式1」else「式2」*)
# if 42 > 156 then 1 else -1;;
- : int = -1
# let abs x =
    if x < 0 then -x else x;;
val abs : int -> int = <fun>
# abs 10;;
- : int = 10
# abs (-10);;
- : int = 10
```

再帰関数を扱う式

```
(* 再帰関数定義(本体の中で自分自身を使う関数の定義):
   let rec 「変数(関数名)」「変数(引数)」… = 「式(本体)」 *)
# let rec pow x n =
    if n = 0 then 1
    else x * pow x (n - 1);;
val pow : int -> int -> int = <fun>
# pow 3 10;;
- : int = 59049
# pow 3 (-1);;
Stack overflow during evaluation (looping
 recursion?).
```

末尾再帰形式にすると…

```
# let pow x n =
    let rec powsub v n =
      if n = 0 then v
      else powsub (v * x) (n - 1)
    in powsub 1 n;;
val pow : int -> int -> int = <fun>
# pow 3 10;;
- : int = 59049
# pow 3 (-1);;
^CInterrupted.
(*↑止まらないので Ctrl-C を入力した *)
```

相互再帰関数を扱う式

```
(* let rec「定義」and「定義」… *)
# let rec even x = if x = 0 then true
                   else odd  (x - 1)
      and odd  x = if x = 0 then false
                   else even (x - 1);;
val even : int -> bool = <fun>
val odd : int -> bool = <fun>
# odd 5423;;
- : bool = true
```

文字列値を表す式

```
# "O'Caml";; (* 「"」で囲まれた部分の文字列を表す *)
- : string = "O'Caml"
# "Hello, world!";;
- : string = "Hello, world!"
# "Hello, world!\n";; (* 改行文字「\n」も使用可 *)
- : string = "Hello, world!\n"
# "Hello, \"O'Caml\"";; (* 引用符「\"」も使用可 *)
- : string = "Hello, \"O'Caml\""
```

文字列演算等を表す式

```
# "Str" ^ "ing";; (* 文字列連結：「式」^「式」*)
- : string = "String"
# print_string;; (* 文字列を表示する関数 *)
- : string -> unit = <fun>
# print_string "Hello, world!";;
Hello, world!- : unit = ()
# print_string "Hello, world!\n";;
Hello, world!
- : unit = ()
```

タプル (組) 値の生成を表す式

```
(* タプルの生成：(「式1」,「式2」,…) *)
# (3 + 5, 5.0 -. 1.0);;
- : int * float = (8, 4.)
# (3, true, "A");;
- : int * bool * string = (3, true, "A")
# "Answer", 42;;
- : string * int = ("Answer", 42)
```

要素をカンマで区切る

実はまわりの括弧はなくてもよい

タプル値の分解操作を表す式

```
# let v = (3, true, "A");;
val v : int * bool * string = (3, true, "A")
(* タプルの分解: let (「変数1」,「変数2」,…) = 「式」… *)
# let (x, y, z) = v;;
val x : int = 3
val y : bool = true
val z : string = "A"
# let (_, w, _) = v in if w then 42 else 156;;
- : int = 42
# fst (3, 2.0);;
- : int = 3
# snd (3, 2.0);;
- : float = 2.
```

使わない要素は「_」で無視できる

2 要素タプルは
fst, snd 関数でも
中身を取り出せる

リスト値の生成を表す式

- リストを表す式: [「式1」; 「式2」; 「式3」; …]

```
# [];; (* 空リスト *)
- : 'a list = []
# [2; 3; 5; 7; 11];;
- : int list = [2; 3; 5; 7; 11]
```

この「'a」については次回以降説明します

- Consセルを表す式:「式1」::「式2」

```
# 1 :: [];;
- : int list = [1]
# 1 :: (2 :: (3 :: []));;
- : int list = [1; 2; 3]
```

括弧はなくてもよい

- リストの連結を表す式:「式1」@「式2」

```
# [1; 2] @ [3; 4; 5];;
- : int list = [1; 2; 3; 4; 5]
```

リストに関する注意

- リストの要素はすべて同じ型で
  なければならない

```
# [1; 2; 3];;
- : int list = [1; 2; 3]
# [1; true; "hoge"];;
This expression has type bool but is
here used with type int.
```

パターンマッチング

パターンマッチングとは?

- 値の「パターン」で処理を分岐させること

```
(* match「式」with |「パターン1」->「式1」
                   |「パターン2」->「式2」| … *)
```

```
# let rec fib n =
    match n with
    | 0 -> 1
    | 1 -> 1
    | x -> fib (x - 1) + fib (x - 2);;
val fib : int -> int = <fun>
```

最初の縦棒「|」は省略してもよい

- 順番にパターンにマッチするかを判定して
  マッチしたら対応する式を評価する

パターンの種類

- 定数パターン
- 変数束縛パターン
- OR パターン
- タプルパターン
- リストパターン
- ワイルドカード

定数パターン & 変数束縛パターン

- 定数パターン
  - 定数と比較 (一致すれば OK)
- 変数束縛パターン
  - 任意の値にマッチ
  - マッチした値は本体中で使用可能

```
# let rec pow x n =
    match n with
    | 0 -> 1
    | m -> x * pow x (m - 1);;
val pow : int -> int -> int = <fun>
```

ORパターン

- 複数のパターンのうち
  一つでもマッチすれば OK

```
# let rec fib n =
    match n with
    | 0 | 1 -> 1
    | x     -> fib (x - 1) + fib (x - 2);;
val fib : int -> int = <fun>
```

タプルパターン

- タプルの要素それぞれがマッチすれば OK
  - 変数束縛パターンと組合わせて
    要素の取出しが可能

```
# let scalar n p =
    match p with
    (x, y) -> (n * x, n * y);;
val scalar : int -> int * int -> int * int
# scalar 3 (2, 3);;
- : int * int = (6, 9)
```

リストパターン

- 空リスト [] と Cons セル :: の二つのパターンがある

```
# let rec sum l =
    match l with
      [] -> 0
    | hd :: tl -> hd + sum tl;;
val sum : int list -> int = <fun>
# sum [1; 2; 3; 4; 5];;
- : int = 15
```

ワイルドカード (_)

- 任意の値にマッチ
  - マッチした値が使われないことを明示している

```
# let has_single_element lst =
    match lst with
    | [_] -> true
    | _ -> false;;
val has_single_element : 'a list -> bool = <fun>
```

ガード

- パターンに条件を追加する仕組み

```
# let is_diag p =
    match p with
    | (x, y) when x = y -> true
    | _ -> false;;
val is_diag : 'a * 'a -> bool = <fun>
# is_diag (3, 3);;
- : bool = true
# is_diag (3, 4);;
- : bool = false
```

関数定義におけるパターンマッチ

- let (rec) や fun の仮引数部分にはパターンを書ける

```
# let thd (_, _, z) = z;;
val thd : 'a * 'b * 'c -> 'c = <fun>
# thd (3, true, "hoge");;
- : string = "hoge"
# fun n (x, y) -> (n * x, n * y);;
- : int -> int * int -> int * int = <fun>
```

function 式

- 引数を 1 個受け取り
  パターンマッチを行う関数を作る

```
# let rec fib = function
    | 0 | 1 -> 1
    | x     -> fib (x - 1) + fib (x - 2);;
val fib : int -> int = <fun>
# let rec pow x = function
    0 -> 1 | n -> x * pow x (n - 1);;
val pow : int -> int -> int = <fun>
```

どれにもマッチしない値がある場合

```
# let flip x = match x with
    0 -> 1 | 1 -> 0;;
Warning P: this pattern-matching is not
exhaustive. Here is an example of a value that
is not matched:
2
val flip : int -> int = <fun>
# flip 2;;
Exception: Match_failure ("", 1, 13).
```

2 が来たときに対応するパターンがない

一応定義はされる

実行すると例外が出る

パターンを書く上での注意

- 1 個のパターン中に同じ変数を
  2 回以上使うことはできない

```
# let is_diag p =
    match p with
      (x, x) -> true | _ -> false;;
This variable is bound several times in this
matching.
```

x を 2 回使用しようとしている

- 前述のようにガードを使えば OK

```
# let is_diag p =
    match p with
      (x, y) when x = y -> true | _ -> false
```

パターンを書く上での注意

- OR でつながったそれぞれのパターンでは束縛する変数が一致しなければならない

```
# let f p =
    match p with
      (1, x, y) | (2, x, _) -> x;;
Variable y must occur on both sides of this |
pattern.
```

| の右側で
y が束縛されていない

パターンを書く上での注意

- 定義された変数を
  定数パターンとして使うことはできない
  - 変数束縛パターンとなってしまうので注意

```
# let x = 0;;
val x : int = 0
# match (1, 2) with
   | (x, _) -> true
   | _ -> false;;
Warning U: this match case is unused.
- : bool = true
```

レポートについて

- 出題後 2 週間以内に提出
- 締め切り厳守
- 質問メールは以下のアドレスまで:
  ml-query-2011@yl.is.s.u-tokyo.ac.jp

単位取得条件

- 毎回の課題を50点以上得点すること

レポート提出上の注意 (1)

- 提出方法: 電子メール
  - 宛先: ml-report-2011@yl.is.s.u-tokyo.ac.jp
    - 受領通知が返送されるはずなので確認のこと
- Subject を

  Report <レポート番号> <学生証番号>

  とすること
  - 例えば今回の場合は

    `Report 1 110xx`
  - 間に一つずつスペースを入れること

レポート提出上の注意 (2)

- レポートに含めるべきもの
  - 氏名、学生証番号
  - ソース
    - コメントを適宜補い、各関数の動作を説明すること
  - 動作例
    - プログラムが正しく動作することを示すのに
      ふさわしい例を考えること
  - 考察
    - 考察不要と指定されている場合を除き、必ず入れる

第1回 課題

締切: 4/19 13:00 (日本標準時)

課題 1 (考察不要：12点)

a. 正の整数 n を受け取って n の階乗を求める関数 `fact : int -> int` を書け
b. 二つの正の整数の最大公約数を求める関数 `gcd : int -> int -> int` を書け
c. 正の整数 n が素数かどうかを判定する関数 `isprime : int -> bool` を書け

課題 2 (考察不要：8点)

a. 関数 f を受け取って
   f を 2 個合成する関数を返す関数
   double を書け

b. 関数 f と整数 n を受け取って
   f を n 個合成する関数を返す関数
   repeat を書け

課題 2 の例

```
# let add1 x = x + 1;;
val add1 : int -> int = <fun>
# let add2 = double add1;;
val add2 : int -> int = <fun>
# add2 3;;
- : int = 5
# let add5 = repeat add1 5;;
val add5 : int -> int = <fun>
# add5 3;;
- : int = 8
```

課題 3 (5点)

- 関数 f を受け取って f を再帰的に無限回合成する関数を返す関数 fix を書け
  - f は「関数を受け取って関数を返す関数」と仮定してよい
  - 以下のようなイメージ

```
fix f ≡ f (f (f (f (f (f (f ...))))))
```

  - 使用例

```
# let fib = fix (fun g n ->
                 if n <= 2 then 1
                 else g (n - 1) + g (n - 2));;
val fib : int -> int = <fun>
# fib 10;;
- : int = 55
```

  - let rec を使ってよい

課題 4 (20点)

- 任意の関数 f について
  課題3の関数 fix が
  関数 f の不動点を返すことを証明せよ
  - f は「関数を受け取って関数を返す関数」
    と仮定してよい

課題 5 (8点)

a. 2 変数関数 f と値 z とリスト $[a_1; a_2; \ldots; a_n]$ を受け取り

$$f\ (f\ (\ldots\ (f\ z\ a_1)\ \ldots)\ a_{n-1})\ a_n$$

を返す関数

```
fold_left: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

を書け

b. 2 変数関数 f とリスト $[a_1; a_2; \ldots; a_n]$ と値 z を受け取り

$$f\ a_1\ (\ldots\ (f\ a_{n-1}\ (f\ a_n\ z)\ )\ \ldots)$$

を返す関数

```
fold_right: ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

を書け

課題 5 (例)

```
# fold_left (-) 0 [1; 2; 3; 4; 5];;
- : int = -15
# fold_right (-) [1; 2; 3; 4; 5] 0;;
- : int = 3
# let flatten x = fold_left (@) [] x;;
flatten : 'a list list -> 'a list
# flatten [[1; 2]; [3; 4]; [5; 6; 7]];;
- : int list = [1; 2; 3; 4; 5; 6; 7]
```

- (注)中置演算子は記号を括弧で囲むことで関数として使える
  - 自分で定義することもできる

```
# let (%) x y = x mod y;;
val ( % ) : int -> int -> int
# 5 % 3;;
- : int = 2
```

課題 6 (12点)

a. 二つのリストを受け取りそれらを連結する関数
   `append : 'a list -> 'a list -> 'a list`
   を書け
   - ただし、@ を使ってはいけない

b. 判定関数とリストを受け取り元のリストの要素のうち
   判定条件を満たす要素だけからなるリストを生成する関数
   `filter : ('a -> bool) -> 'a list -> 'a list`
   を書け

c. ペア(二要素のタプル)のリストを
   リストのペアに分割する関数
   `split : ('a * 'b) list -> 'a list * 'b list`
   を書け

課題 6 (例)

```
# append [1; 2; 3; 4] [5; 6; 7];;
- : int list = [1; 2; 3; 4; 5; 6; 7]
# filter isprime [1; 2; 3; 4; 5; 6; 7; 8];;
- : int list = [2; 3; 5; 7]
# split [(1, 2); (3, 4); (5, 6)];;
- : int list * int list = ([1; 3; 5], [2; 4; 6])
```

課題 7 (10点)

- 課題5、課題6で定義した関数が
  長さ10000000以上のリストを
  正しく扱えるか確認せよ
  - 正しく扱えない場合は
    正しく扱えるように修正せよ

課題 8 (5点)

- リスト **lst** と整数 **n** を受け取り
  lst から要素を **n** 個取り出す組み合わせを
  リストにして返す関数

  comb : 'a list -> int -> 'a list list
  を定義せよ

```
# comb [1; 2; 3] 2;;
- : int list list = [[1; 2]; [1; 3]; [2; 3]]
# comb [1; 2; 3] 0;;
- : int list list = [[]]
# comb [] 3;;
- : 'a list list = []
```

課題 9 (20点)

a. f を m 個合成する関数と f を n 個合成する関数を受け取って、f を m + n 個合成する関数を返す関数 add を書け
b. f を m 個合成する関数と f を n 個合成する関数を受け取って、f を m * n 個合成する関数を返す関数 mul を書け
c. f を m 個合成する関数と f を n 個合成する関数を受け取って、f を m - n 個合成する関数を返す関数 sub を書け