Jacques Garrigue, 2007 年 8 月 27-8 日

関数型言語 Objective Caml 入門

1 Objective Caml とは

Objective Caml は ML 系の関数型言語である．関数型言語には二つの意味がある．第一の意味は関数が値のように扱える言語，第二の意味は計算が数学的な関数のように行われ，結果が引数にしかよらない言語である．ML の場合には，第一の意味が当っているが，第二の意味は部分的にしか当て嵌らない．次の講義で見る Haskell は第二の意味を追求している．

ML と Haskell に共通の特徴として，多相型付ラムダ計算に基いた強力な型システムが上げられる．実行時型エラーを完全に防ぐのと同時に，関数の再利用を可能にしている．

元の ML は 80 年代の初期に Ediburgh LCF という証明器のために開発された言語である．Caml はそういう ML を拡張して，オブジェクトシステムや省略可能引数を備えている．多くのライブラリも提供されている．

OCaml 関連リソース　日本語の本はないが，ウェブで使える資料が多くある．

`http://www.math.nagoya-u.ac.jp/~garrigue/lecture/tsukuba07/`
　この講義に関する資料.

`http://caml.inria.fr/`
　Objective Caml の開発元．処理系がダウンロードできる．英語での情報も多い.

`http://ocaml.jp/`
　日本語での情報を集めたサイト.

`http://ocaml.jp/archive/ocaml-manual-3.06-ja/`
　マニュアルの日本語版.

`http://wwwfun.kurims.kyoto-u.ac.jp/soft/ocaml/htmlman/`
　マニュアルの英語版.

`http://www.math.nagoya-u.ac.jp/~garrigue/papers/jssst-ocaml.html`
　PPL サマースクールの資料．中級者向け.

`http://www.math.nagoya-u.ac.jp/~garrigue/lecture/2006_AW/`
　名古屋での講義資料．色々なアルゴリズムを扱う.

連絡　質問と課題の答を `garrigue@math.nagoya-u.ac.jp` に送って下さい.

2 インタープリタの使い方

2.1 起動と終了

インタープリタを直接に使うには，Terminal など仮想端末を開けばいい．その中で `ocaml` を起動する.

```
$ ocaml
        Objective Caml version 3.09.2

#
```

この種類のインタープリタをトップレベルと言うが，対話的に定義や評価する式が入力できる．たとえば 1+2;;<ret>を入力する．

```
# 1+2;;
- : int = 3
```

;; は OCaml に入力が終わったことを知らせる．それがないと，<ret>が 無視され，改行しても入力が続く．

- : int = 3 が OCaml の出力した答である．-は，OCaml が入力された式の値を計算し，それをそのまま答えたことを表す．int は入力された式が整数型だったことを表す．3 は実行の結果である．

```
# let x = 1 ;;                                          (* let は定義 *)
val x : int = 1
```

出力が val で始まるそき，定義された名前とその型・値が表示される．入力の中で (* と *) で囲まれた部分は註釈で，無視される．

トップレベルから抜けるために#quit を使う．

```
# #quit;;                                              (* コマンドが#で始まる *)
$
```

2.2 Emacs での使い方

設定ファイルの編集

以下の 6 行をホームの.emacs に追加し，Emacs を再起動する．

```
(setq auto-mode-alist
      (cons '("\\.ml[iylp]?$" . caml-mode) auto-mode-alist))
(autoload 'caml-mode "caml" "Major mode for editing Caml code." t)
(autoload 'run-caml "inf-caml" "Run an inferior Caml process." t)
(if window-system (require 'caml-font))
(setq inferior-caml-program "/usr/local/bin/ocaml")
```

註釈　Preview で PDF のファイルから文字をコピーするには，ツールを **A**(文字モード) に変えてから，普通に選択・コピーをすればいい．しかし，‘’’ (クオート) と ‘\’ (バックスラッシュ) が全角になってしまうので手で修正しないといけない．

OCaml を Emacs の中で実行

Emacs の中で ocaml を実行するために，以下でキー列ををを入力する．

```
<M-x>run-caml<ret><ret>
```

これで新しいバッファの中で以下の内容が表われる．

```
        Objective Caml version 3.09.2

#
```

の後にプログラムを入れると，そのまま実行される．

```
# let x = 2+2;;<ret>
val x : int = 4
```

このモードで使える主なキー列は以下のとおりである．

<M-p>	以前の入力文を編集する
<C-c><C-c>	実行を途中で中断させる
<C-c><C-d>	ocaml 自体を終わらせる

ocaml を直接にシェルで起動することもできるが，そうすると編集機能が使えない．

プログラムを編集する

まず，名前が.ml で終わるファイルを作る．

```
<C-x><C-f>test.ml<ret>
```

そのバッファの中でプログラムを書くと，<tab>を押すだけでインデントが自動的に行われる．(構文によって，行を書いてから<tab>を押さないといけない．) また，emacs 21 ではキーワードに色が付く．

編集中のプログラムを一段落ずつ ocaml に実行させることもできる．まず，ocaml を前面に持ってくる．

```
<C-c><C-s>
```

そして，例えば以下の行を書いたら (_はカーソルの位置を表す)

```
let x = 3 * 5;;_
```

今度は次のキー列を入力する (先頭の<C-a>はプログラムの中に戻るため)

```
<C-a><C-c><C-e>
```

そうする実行の結果が ocaml のバッファに表れる．(実行したコードがそちらで表示されないので，先頭の # だけが見える)

```
# val x : int = 15
```

もしもプログラムにエラーがあれば，カーソルがその位置に移る．

2.3 ファイルからプログラムを読み込む

Emacs のバッファからの評価は中々便利であるが，ファイルを丸ごと読み込むこともできる．これはトップレベルの機能であり，Emacs を使わなくてもできる．そのとき，Caml は，ファイルの内容があたかも入力ループで入力されたように動作する．

ファイル test.ml の中身は次の通りだとする．

```
let double x = x * 2;;      (* double は引数の 2 倍を計算する *)
let y = 10;;                (* y を適当な値に *)
y + double y;;              (* これで 3 倍だ！ *)
```

test.ml を読み込む.

```
# #use "test.ml";;
val double : int -> int = <fun>
val y : int = 10
- : int = 30
```

このようにファイルからプログラムを読み込む場合は，入力ループで #use "ファイル名";; のように入力すればよい．答は，読み込んだ入力によるものである．#use は結果を出さない.

3 定義と型

値と関数の定義

```
# let x = "hello" ;;                                    (* let は定義 *)
val x : string = "hello"
# let x = 1 ;;                                          (* 新しい定義 *)
val x : int = 1
# x = 2 ;;
- : bool = false                                        (* '=' だけだと等号になる *)
# let x = 3 in x+2;;                                    (* 局所的な定義 *)
- : int = 5
# x;;
- : int = 1                                            (* 元の定義に影響がない *)
# let x = 3 and y = x+2 ;;                              (* 同時定義 *)
val x : int = 3
val y : int = 3                                        (* 定義の前の x が使われる *)
# let f (z : int) =                                    (* 関数の定義 *)
    y + z ;;
val f : int -> int = <fun>                             (* '->' は関数の型を表す *)
# f 0;;
- : int = 3
# let y = 12 ;;
val y : int = 12
# f 0;;
- : int = 3                                            (* 値を再定義しても影響はない *)
# let f z = y+z ;;                                     (* 型を書かなくても大丈夫 *)
val f : int -> int = <fun>
# let f = fun z -> y+z ;;                              (* 関数のもう一つの書き方 *)
val f : int -> int = <fun>
# (fun z -> y+z) 0;;                                   (* 定義がなくても使える *)
- : int = 3
```

多引数の関数

```
# let p (x : int) (y : int) =                          (* 引数を並べるだけ *)
    2 * x - y * y ;;
val p : int -> int -> int = <fun>                      (* '->' が増える *)
# p 3 4;;                                              (* 適用のときも並べるだけ *)
- : int = -10
# let p = fun x y -> 2 * x - y * y ;;                  (* 同じ定義 *)
val p : int -> int -> int = <fun>
# let p = fun x -> fun y -> 2 * x - y * y ;;           (* これも同じ *)
val p : int -> int -> int = <fun>
```

```
# let q = p 3 ;;                                    (* 最初の引数だけを渡す *)
val q : int -> int = <fun>
# q 4 ;;                                            (* 残りの引数を渡す *)
- : int = -10
```

引数の一部を渡すことを「部分適用」と言う.

実数と演算子

```
# let pi = 3.1416 ;;
val pi : float = 3.1416
# let twopi = 2 * pi;;
This expression has type float but is here used with type int
# let twopi = 2 *. pi;;                          (* 実数の演算子は整数と違う! *)
This expression has type int but is here used with type float
# let twopi = 2. *. pi ;;                          (* 整数は実数ではない! *)
val twopi : float = 6.2832
```

上の例で分かるように，式を実行するために，型が完全に一致しないといけない.

```
val ( *. ) : float -> float -> float
val pi : float
val float : int -> float                            (* 整数を実数に変換 *)
val truncate : float -> int                         (* 実数を整数に変換 *)
```

例えば，pi の倍数を整数に戻す関数を以下のように定義できる.

```
# let npi (n : int) = truncate ((float n) *. pi);;
val npi : int -> int = <fun>
# npi 8;;
- : int = 25
```

必要な型に応じて関数を選ぶという方法は有効である.

様々な値

```
# true || false;;
- : bool = true
# 'A';;
- : char = 'A'
# "Hello" ^ " everybody";;
- : string = "Hello everybody"
# ();;
- : unit = ()
# (1, "one", 1.0);;
- : int * string * float = (1, "one", 1.)
# [| "little"; "brown"; "fox" |];;
- : string array = [|"little"; "brown"; "fox"|]
# Array.init 5 (fun i -> i*i);;
- : int array = [|0; 1; 4; 9; 16|]
# [1; 2; 3; 4];;
- : int list = [1; 2; 3; 4]
```

型の種類　よく使われる型のまとめ

`int`	整数	$-2 \cdot 10^{30} \sim 2 \cdot 10^{30} - 1$ (64 ビットだと 10^{62})
`bool`	真偽値	`true` または `false`
`char`	文字	8 ビット文字
`string`	文字列	8 ビット文字の列
`unit`	単位型	`()` だけ．C 言語の `void` に似ている
`float`	実数	C 言語の `double` と同じ
t `->` u	関数型	t 型から u 型への関数
t_1 `->` ... `->` t_n `->` u	多引数関数型	$t_1, \ldots, t_n$ 型から u 型への関数
t_1 `*` ... `*` t_n	組型	$t_1, \ldots, t_n$ の値の組
t `array`	配列	t 型の値を要素とする配列
t `list`	リスト	t 型の値を要素とするリスト
t `ref`	変数 (参照型)	t 型の値への参照 (変更可能)
`'a`, `'b`,...	型変数	関数適用時に具体的な型が選べる

型を書くとき `*` が `->` より結合力が強いが，型パラメーター (t `array` など) より弱い．

変換関数

```
Char.code       : char -> int
Char.chr        : int -> char
string_of_int   : int -> string
int_of_string   : string -> int
float           : int -> float
truncate        : float -> int
string_of_float : float -> string
float_of_string : string -> float
Array.of_list   : 'a list -> 'a array
Array.to_list   : 'a array -> 'a list
```

`Char.code` という書き方は `Char` というモジュールの中の関数 `code` を指している．異なるモジュールでは同じ名前の関数があってもいい．

構成・読み出し関数

```
String.get      : string -> int -> char              (* s.[i] とも書く *)
String.length   : string -> int                        (* 文字列の長さ *)
String.make     : int -> char -> string
                                          (* 同じ文字を n 回繰り返した文字列 *)
Array.get       : 'a array -> int -> 'a              (* a.(i) とも書く *)
Array.length    : 'a array -> int                          (* 配列の長さ *)
Array.init      : int -> (int -> 'a) -> 'a array
                                          (* 0...n-1 に対して関数が呼ばれる *)
```

数値演算子

```
+ - * / mod     : int -> int -> int
+. -. *. /. **  : float -> float -> float
-               : int -> int
-.              : float -> float
```

数値演算子は整数用と実数用が別々に定義されている．ただし，引数の型が実数だと入力時にわかれば，整数用のものは自動的に実数用に変換される．

比較演算子

```
=  <>  <  > <= >=   : 'a -> 'a -> bool
== !=               : 'a -> 'a -> bool
```

型の 'a はどの値でも使えるということを意味する．同じ型どうしの値ならどの値でも比較できる．2 行目は値のメモリ上の物理的な位置を比較する．

真偽演算子

```
&&  ||          : bool -> bool -> bool
```

式 1 && 式 2 は両方の式が真のときのみ真を返し，式 1 || 式 2 は両方の式の少なくとも一方が真のときのみ真を返す．ただし，実際には次のように実行される．式 1 && 式 2 は式 1 の結果が偽だったら式 2 を実行しない．式 1 || 式 2 は式 1 の結果が真だったら式 2 を実行しない．

結合演算子

```
^               : string -> string -> string
Array.append    : 'a array -> 'a array -> 'a array    (* 演算子ではない *)
@               : 'a list -> 'a list -> 'a list
```

"^" は文字列にも使える．たとえば，"hello" ^ " world"は "hello world" になる．

練習問題 3.1 *1.* この章の例を *ocaml* のトップレベルに入力して，慣れて見る．

2. 二つの実数の平均を取る関数を定義せよ．

```
val heikin : float -> float -> float
```

3. 配列をベクトルと見做して，スカラー積と内積を計算する関数を定義せよ．

```
val scalar : float -> float array -> float array
val plus   : float array -> float array -> float array
```

そこで使う関数は +. *(*実数の足算*)*，Array.length，Array.init と Array.get である．

4 多相型と汎関数

型推論

ML では型が自動的に推論されるので，関数を定義するときでも型を書かなくてもいい．

```
# let f x y = (x+1, y.[0]);;
val f : int -> string -> int * char = <fun>
```

x は + で使われたために int 型になる．y は String.get で使われたために string 型になる．

多相型

前記のような制約が現れないとき，型変数が使われる．

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

ここでは x と y の型を特定するものがないので，それぞれ型変数 'a と 'b が付けられる．結果が x なので，結果の型が 'a になる．型変数を含む型を多相型と言う．厳密には，決まらない型の多相型への変換は let による定義の度に行なわれる．

多相型を持った関数を使うとき，型変数に対する型が自由に決められる．しかも，それが何回も，異なる型でもできる．

```
# fst ("France", 33) ;;
- : string = "France"
# fst (5.0, 2.3) ;;
- : float = 5.
```

一つ目の例では fst の型が string * int -> string になる．二つ目の例では float * float -> float になる．

今まで見た関数の型の中に，二種類の多相型が見られる．Array.of_list, Array.get, Array.init などは引数と結果の両方に同じ型変数'a がある．こういう場合では結果の型が引数の型により決まると思えばいい．しかし Array.length では型変数が引数の型にしか現れない．そうい場合，配列の中身が結果に現れないことになる．

第一種の場合でも，関数に多相型が付けられると，関数が使われるときに動きが型変数に対する実際の型に依存しない．型の一部が変数であるということは，それに対する値の一部が関数の中で処理されていないことを保証する．

参照型

第 3 章では，同じ名前に対する新しい定義を追加しても，前の定義が隠れるだけで，その定義を参照していた他の定義に影響がないことを見た．ML の定義は C のような変数ではなく，定数である．

ただ，プログラムを書くときに，変更可能な変数が欲しい場合もある．ML では参照型として提供される．厳密にいうと，参照型は新しい種類の定義ではなく，データ構造である．

```
# let x = ref 1 ;;                                    (* 変数の定義 *)
val x : int ref = {contents = 1}
# !x ;;                                               (* 変数の読み出し *)
- : int = 1
# x := 2 ;;                                           (* 変数の書き込み *)
- : unit = ()
# !x ;;
- : int = 2
```

参照型以外に，実は文字列と配列も変更可能である．文字列の場合，それを使わない方がいいが，配列の場合は様々なアルゴリズムで役に立つ．

```
# let arr = [| 2; 3; 4 |];;
val arr : int array = [|2; 3; 4|]
# arr.(0) <- 5;;                                      (* 配列への書き込み *)
- : unit = ()
# arr;;
- : int array = [|5; 3; 4|]
```

参照型は for ループを使うときに役に立つ．

```
# let sum (arr : int array) =
    let r = ref 0 in                                  (* 変数を作る *)
    for i = 0 to Array.length arr - 1 do
      r := !r + arr.(i)
```

```
    done;                                    (* ';' は逐次的に実行する式を並べる *)
    !r;;                                     (* 変数の値を返す *)
val sum : int array -> int = <fun>
```

可変な変数を使うと，プログラムの意味が各変数の状態に依存してしまうので，解釈が複雑になる．そのために，可変な変数を最低限に抑えるべきだ．

参照型や配列が可変であるために，多相型が少し制限されている．

```
# let r = ref [];;
val r : '_a list ref = {contents = []}
# r := [3];;
- : unit = ()
# r;;
- : int list ref = contents = [3]
# let single () x = [x];;
val single : unit -> 'a -> 'a list = <fun>
# let safe = single ();;
val safe : '_a -> '_a list = <fun>
```

上記の下線で始まる型変数は通常の型変数と違い，多相型ではない．始めて使われたときに型が決まる．一旦 r に整数のリストを入れると，r の型自体が int list ref に変わってしまう．

しかし，この現象は ref を使ったときだけではない．例えば，Array.init を部分適用したときでも同じことが起きる．OCaml では，関数文の中にない関数適用を含む定義に関しては，結果の型変数は多相型にしない．厳密には，関数型の引数の方または ref か array(可変な型) の引数として表われる型変数だけが多相型にされない．

```
# single () [];;
- : 'a list list = [[]]                  (* 制限の対象にならない *)
```

汎関数

汎関数とは，関数を引数とする関数である．今まで見たものの中に Array.init は汎関数である．汎関数は普通の関数と全く同様に定義できる．ここに sum のもう一つの定義を与える．

```
# let iter (f : 'a -> unit) (arr : 'a array) =
    for i = 0 to Array.length arr - 1 do
      f arr.(i)
    done;;
val iter : ('a -> unit) -> 'a array -> unit = <fun>
# let sum2 arr =
    let r = ref 0 in
    iter (fun x -> r := !r + x) arr;
    !r ;;
val sum2 : int array -> int = <fun>
```

iter はある配列の各要素に対して引数の関数を適用する．手で for ループを書くより，配列の長さを気にする必要がないという利点がある．iter を利用して sum を書きなおすと，プログラムが短かくなる．

iter は一般的な汎関数で，Array.iter という名前で最初から定義されている．しかし，自分の好みに合わせて汎関数を定義するのもいい．汎関数が大好きなら，以下のコードも書ける．

```
# let local x0 (f : 'a ref -> unit) =
    let r = ref x0 in f r; !r;;
val local : 'a -> ('a ref -> unit) -> 'a = <fun>
# let sum3 (arr : int array) =
```

```
    local 0 (fun r -> iter (fun x -> r := !r + x) arr);;
val sum3 : int array -> int = <fun>
```

一見読みにくいが，各汎関数の働きを正しく理解していれば，実は元の sum の定義より分かりやすくて，間違いの機会が少ない．

ベクトルのスカラー積も，以下の汎関数 map を使えば簡単に定義できる．

```
# let map f arr = Array.init (Array.length arr) (fun i -> f arr.(i)) ;;
val map : ('a -> 'b) -> 'a array -> 'b array = <fun>
# let scalar x v = map (fun x' -> x *. x') v ;;
val scalar : float -> float array -> float array = <fun>
```

ここでも，配列の長さや個別の要素の読み出しを気にしなくていいのがメリットである．

汎関数の働きをより分かりやすくするために，プログラムの等価性を考えるといい．以下の 2 つの展開規則を使う．

$$\textbf{let } f\ x_1 \ldots x_n = E$$

の元で，次の展開ができる

$$f\ E_1 \ldots E_n \Longrightarrow \textbf{let } x_1 = E_1 \textbf{ and } \ldots \textbf{ and } x_n = E_n \textbf{ in } E \tag{1}$$

x の定義である v が値か名前 ((1) と (2) が使えない式) なら，そして x および v(名前なら) が E の中で再定義されていない場合，それを E の中に代入することもできる (E の中の x を全て v に変える)

$$\textbf{let } x = v \textbf{ in } E \Longrightarrow E\{x := v\} \tag{2}$$

では，展開してみよう．

```
sum3 arr =
local 0 (fun r -> iter (fun x -> r := !r + x) arr)
```
⇓ (1)
```
let x0 = 0 and f r = iter (fun x -> r := !r + x) arr in
let r = ref x0 in f r; !r
```
⇓ (2)
```
sum2 arr =
let r = ref 0 in
(let r = r in iter (fun x -> r := !r + x) arr);
!r
```
⇓ (1)
```
let r = ref 0 in
let f x = r := !r + x and arr = arr in
for i = 0 to Array.length arr - 1 do f arr.(i) done; !r
```
⇓ (2)
```
let r = ref 0 in
for i = 0 to Array.length arr - 1 do r := !r + arr.(i) done; !r
= sum arr
```

すなわち sum3，sum2 と sum は等価なプログラムである．

関数の型を読む

コンパイルが推論した型が多くのことを教えてくれる．たとえば，map の型を見よう．

```
val map : ('a -> 'b) -> 'a array -> 'b array
```

- map は二つの引数を取る．

- その 1 つ目は関数である．
- 'b array の要素は'a array からその関数を使って計算される．なぜかというと，'a array 以外に'a 型の値はなく、他に'b 型の値を作る方法もない．

さらに，推論された型を見ると定義の中のバグが発見できる．

```
val map_array : ('a -> 'a) -> 'a array -> 'a array
```

多相性不足．多分，入力の要素をそのまま出力に使っている．

```
val map_array : ('a -> 'b) -> 'a array -> 'c array
```

多相性過剰．'c 型の値が作れないので，結果がからっぽ．

練習問題 4.1 *1.* 逆順の配列を返す関数を定義せよ．

```
val rev_array : 'a array -> 'a array
```

2. f，ϵ，x が与えられたとき，以下の $f'(x)$ を計算する関数 derive を定義せよ．

$$f'(x) = \frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon}$$

```
val derive : (float -> float) -> float -> float -> float
```

3. 行列を作る関数を定義せよ．Array.init を二重に使えばいい．

```
val init_matrix : int -> int -> (int -> int -> 'a) -> 'a array array
# init_matrix 2 3 (fun i j -> 3*i+j);;
- : int array array = [|[|0; 1; 2|]; [|3; 4; 5|]|]
```

4. 台形式によって関数 f の積分を計算する関数 integ を定義せよ． 式は

$$Int(f, N, x, x') = \frac{x' - x}{N} \sum_{k=0}^{N-1} \frac{f(x_k) + f(x_{k+1})}{2} \quad where\ x_k = \frac{(N-k)x + kx'}{N}$$

```
val integ : (float -> float) -> int -> float -> float -> float
```

5 応用 関数グラフの描画

準備

http://www.math.nagoya-u.ac.jp/~garrigue/lecture/tsukuba07/ からライブラリーのソースをダウンロードする．二つのファイルを使う．

- plot.mli はライブラリーのインターフェースであり，使える関数の型を与えている．日本語に翻訳すると以下が内容である．

```
(* plot.mli *)
val adjust_size : (float * float) list -> unit
    (* リストに含まれる点が全て表示できるように縮尺を合わせる *)
val create_curve : (float -> float) -> (float * float) list
    (* 渡された関数の現在表示可能な範囲でのグラフを作る *)
val draw_curve : (float * float) list -> unit
    (* グラフを描画する *)
```

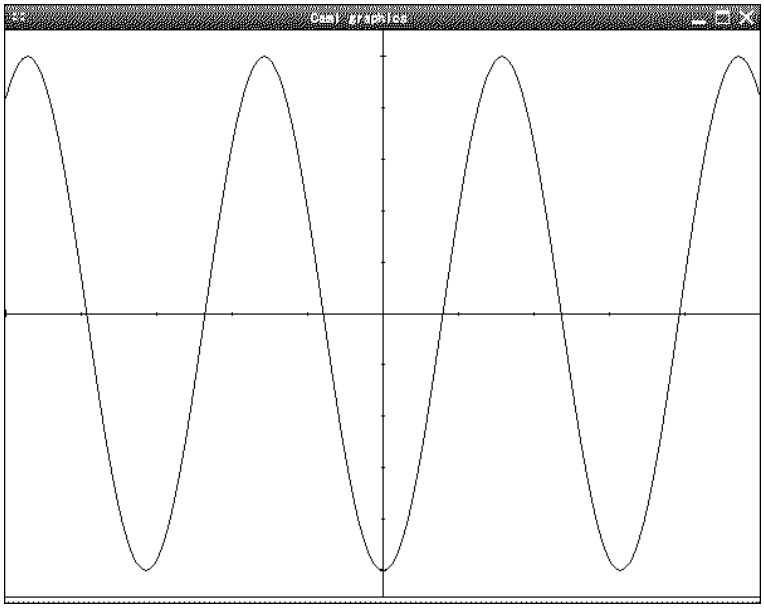

図 1: Plot を使った例

- plot.ml はライブラリーの実装である．Graphics という OCaml に添付されているグラフィックスライブラリーを使って plot.mli の関数を実装している．使い方はインターフェースで決まっている，このファイルの中身を見る必要はない．

アプリケーションに入っている X11 も起動しなければならない．

ファイルのコンパイル

まずは，plot.mli と plot.ml をそれぞれコンパイルしなければならない．順番は重要である．

```
$ ocamlc -c plot.mli                                  (* plot.mli -> plot.cmi *)
$ ocamlc -c plot.ml                                    (* plot.ml -> plot.cmo *)
```

ライブラリーの利用

```
#load "graphics.cma" ;;                  (* OCaml のグラフィックスライブラリー *)
#load "plot.cmo" ;;                                    (* plot.cmo をロード *)
open Plot ;;                                           (* plot.cmi をロード *)
adjust_size [-5., 0.; 5., 0.] ;;
let curve = create_curve (fun x -> (sin x) ** 2. -. 0.5) ;;
adjust_size curve ;;
draw_curve curve ;;
```

このプログラムの実行結果は図 1 に示してある．

練習問題 5.1　*1.* 様々な関数のグラフを描画して見よ．

2. 関数のリストと x 軸の範囲を与えらるとそれぞれの関数を同時に描いたグラフを表示する関数を定義せよ．

```
val draw_functions : (float -> float) list -> float -> float -> unit
```

その関数を定義するのに，以下の二つの関数が役に立つ．

```
List.map     : ('a -> 'b) -> 'a list -> 'b list     (* Array.map と同じ *)
```

```
List.iter    : ('a -> unit) -> 'a list -> unit     (* Array.iter と同じ *)
List.flatten : 'a list list -> 'a list
                                   (* 入れ子のリストを一つの長いリストに変換 *)
```

3. 実は draw_curve は点を直線でつなぐ関数である．draw_curve を使って任意の正多角形を描画する関数を定義せよ．

6 再帰関数とリスト

再帰関数

自明でないプログラムを書くにはループが必要である．しかし，for ループを使うときには，可変な変数が必要になり，プログラムの (理論的な) 解釈が複雑になる．再帰関数はもう一つのループの書き方を与えてくれる．

まず，最大公約数の計算を見る．

```
# let rec gcd m n =                                  (* let rec は再帰的な定義 *)
    if n = 0 then m else gcd n (m mod n) ;;
val gcd : int -> int -> int = <fun>
# gcd 15 70;;
- : int = 5
```

もしも同じプログラムを while ループで書いたら，大分長くなる．

```
# let gcd2 m n =
    let m = ref m and n = ref n in
    while !n <> 0 do                                      (* while ループ *)
      let n' = !m mod !n in                          (* 名前は n' でもいい! *)
      m := !n; n := n'                               (* 同時代入ができない *)
    done;
    !m ;;
val gcd2 : int -> int -> int = <fun>
```

しかし，再帰の最大のメリットは短かさではなく，証明しやすさである．再帰関数は帰納法と同じ構造をしているので，帰納法により証明が簡単にできる．gcd の場合では ($m, n \geq 0$ と仮定して)

- $n = 0$ ならば，0 はどんな自然数でも割れるが，m を割る最大の自然数は m 自身である．ゆえに m と n の最大公約数は 0 である．
- $n > 0$ ならば，任意の $k < n$ と任意の m に対して，gcd m k が m と k の最大公約数であると仮定する．
  m と n の最大公約数は m **mod** n を割らなければならない．逆に，q が n と m **mod** n を割っていれば，m も q で割れる．ゆえに m と n 最大公約数は n と m **mod** n の最大公約数でもある．$0 \leq m$ **mod** $n < n$ がなりたつので，帰納法の仮定が適用できて，(m と n の最大公約数) = (n と m **mod** n の最大公約数) = gcd n (m **mod** n) = gcd m n.

再帰では，単純なループで表現できない計算の構造も表現できる．

```
# let rec fib n = if n < 2 then 1 else fib (n-1) + fib (n-2) ;;
val fib : int -> int = <fun>
# fib 5;;
- : int = 8
```

しかし，こういうものは必ずしも効率よくない．

```
# #trace fib;;                                          (* 呼び出しを表示させる *)
fib is now traced.
# fib 4;;
fib <-- 4
fib <-- 2
fib <-- 0
fib --> 1
fib <-- 1                                              (* fib 1 が呼ばれる *)
fib --> 1
fib --> 2
fib <-- 3
fib <-- 1                                                  (* ここも *)
fib --> 1
fib <-- 2
fib <-- 0
fib --> 1
fib <-- 1                                                  (* ここも *)
fib --> 1
fib --> 2
fib --> 3
fib --> 5
- : int = 5
```

練習問題 6.1 fib のもっと賢い書き方があるはず．補助定義を使ってもいい．

リスト

```
# [1; 2; 3];;                                      (* リストは配列のように書く *)
- : int list = [1; 2; 3]
# 1 :: [2; 3];;                          (* cons(コンス) という構成子で作られる *)
- : int list = [1; 2; 3]
# 1 :: (2 :: (3 :: []));;                          (* これが本当の内部構造 *)
- : int list = [1; 2; 3]
# List.hd [1;2;3];;                                      (* リストの頭 *)
- : int = 1
# List.tl [1;2;3];;                                    (* リストの尻尾 *)
- : int list = [2; 3]
```

リストは配列のようなデータ構造だが，簡単に頭に値を追加・削除できるので重宝される．
リストに対して多くの関数が定義されている．

```
List.length     : 'a list -> int
List.hd         : 'a list -> 'a
List.tl         : 'a list -> 'a list
List.nth        : 'a list -> int -> 'a
List.rev        : 'a list -> 'a list
List.append     : 'a list -> 'a list -> 'a list        (* l1 @ l2 とも書く *)
List.flatten    : 'a list list -> 'a list
List.iter       : ('a -> unit) -> 'a list -> unit
List.map        : ('a -> 'b) -> 'a list -> 'b list
List.fold_left  : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
List.fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
List.for_all    : ('a -> bool) -> 'a list -> bool
List.exists     : ('a -> bool) -> 'a list -> bool
List.mem        : 'a -> 'a list -> bool
```

```
List.filter     : ('a -> bool) -> 'a list -> 'a list
List.split      : ('a * 'b) list -> 'a list * 'b list
List.combine    : 'a list -> 'b list -> ('a * 'b) list
List.assoc      : 'a -> ('a * 'b) list -> 'b
List.mem_assoc  : 'a -> ('a * 'b) list -> bool
List.remove_assoc : 'a -> ('a * 'b) list -> ('a * 'b) list
...
```

練習問題 6.2 型と名前から，各関数の働きを推理せよ．試しに値を渡してもいい．

パターン・マッチング

```
let hd l =
  match l with
    []     -> failwith "List.hd"                    (* エラーを起こす *)
  | a :: _ -> a ;;                                  (* 頭部を返す *)
let tl l =
  match l with
    []     -> failwith "List.tl"
  | _ :: t -> t ;;                                  (* 後部を返す *)
```

パターンマッチングは以下の構造の計算式である．

```
match 計算式 with
  パターン_1 ->
    計算式_1
| ...
| パターン_n ->
    計算式_n
```

パターンは構成子と名前だけでできた式である．パターンを順番に見て，マッチする値がパターン $_i$ と同じ形をしていれば，パターン $_i$ の中の名前をマッチする値の対応する部分に束縛して ('_' は束縛されない特別な名前)，計算式 $_i$ の結果を返す．

リストと再帰 リストに対する関数のほとんどは再帰的に定義されている．

```
let rec length l =
  match l with
    []      -> 0
  | _ :: l' -> 1 + length l'
;;
let rec rev_append l1 l2 =
  match l1 with
    []     -> l2
  | a :: l -> rev_append l (a :: l2)
;;
let rev l = rev_append l [] ;;
```

練習問題 6.3 *1.* 二つのリストの連結を行う append を再帰関数として定義せよ．

2. rev はなぜ直接に再帰関数として定義されていないのか？

3. リストを多項式と見做し *(*頭が定数*)*，ある点で多項式の値を計算する関数を定義せよ．

```
val eval_poly : int list -> int -> int
```

```
# eval_poly [1; 0; 3] 2 ;;
- : int = 13                                          (* 1 + 0*2 + 3*4 *)
```

検索問題　再帰的な関数は検索アルゴリズムに向いている．例えば，有名な SEND + MORE = MONEY 問題は以下のように解ける．(S と M は 1〜9 の数字, E, N, D, M, O, R, Y は 0〜9 の数字に対応しており，SEND と MORE が対応している 10 進の数の足し算が MONEY になっている．)

```
(* 文字と数字の対応が条件を満しいるか判定する関数 *)
val check : (char * int) list -> bool = <fun>

(* 解のリストを作る再帰関数 *)
# let rec search dict letters numbers =
  match letters with
    [] -> if check dict then [dict] else []
  | a :: letters ->                          (* letters <- List.tl letters *)
      let rec choose tried numbers =
        match numbers with
          [] -> []
        | n :: numbers ->                    (* numbers <- List.tl numbers *)
            let sols = search ((a,n)::dict) letters (tried @ numbers) in
            sols @ choose (n::tried) numbers
      in
      choose [] numbers ;;
val search :
  (char * int) list -> char list -> int list -> (char * int) list list =
  <fun>
# let rec interval m n =
    if m > n then [] else m :: interval (m+1) n ;;
val interval : int -> int -> int list = <fun>
# let solve () =
    search [] ['S';'E';'N';'D';'M';'O';'R';'Y'] (interval 0 9) ;;
val solve : unit -> (char * int) list list = <fun>
# solve () ;;
- : (char * int) list list = [[('Y',2); ...]]
```

練習問題 6.4　上記の check という関数を定義せよ．List.map, List.assoc と前問の eval_poly を使うといい．

解を見付けるのに，check は何回呼ばれたか？search を少し変えてその回数が減らせる．

7　再帰データ型

再帰データ型が型付関数型言語の最大の特徴であると言える．このデータ構造が他の言語では見当らないのに，関数型言語の全てのプログラムに使われている程に便利なものである．リストは再帰データ構造の一例である．

型定義

再帰データ型の定義は以下のように行われる

```
type 型変数 型名 =
    構成子 1 [of 引数型 1]
  | ...
```

| 構成子 $_n$ [of 引数型 $_n$]

型変数は要らないときに省略する.

リストは最初から定義されるが，自分で定義しようと思えば，以下のようになる.

```
# type 'a mylist =
      Nil                                              (* '[]' に当る *)
    | Cons of 'a * 'a mylist ;;                        (* '::' に当る *)
type 'a mylist = Nil | Cons of 'a * 'a mylist
# let rec mylist_of_list l =
    match l with
      [] -> Nil
    | a :: t -> Cons (a, mylist_of_list t) ;;
val mylist_of_list : 'a list -> 'a mylist = <fun>
# mylist_of_list [1;2;3];;
- : int mylist = Cons (1, Cons (2, Cons (3, Nil)))
# let my_hd l =
    match l with                              (* 当然パターンマッチングが使える *)
      Nil       -> failwith "my_hd"
    | Cons(a,_) -> a ;;
val my_hd : 'a mylist -> 'a = <fun>
```

再帰データ型が将来のコードの変更を安全にしてくれる.

```
type 'a mylist = Nil | Cons of 'a * 'a mylist | One of 'a
                                                  (* mylist の定義を変更 *)
# let my_hd l =                                    (* 元の定義をそのまま *)
    match l with
      Nil       -> failwith "my_hd"
    | Cons(a,_) -> a ;;
Warning P: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
One _
val my_hd : 'a mylist -> 'a = <fun>
```

間違ったプログラムがコンパイルできるものの，問題箇所を指摘したワーニングが出力される.
(エラーにもできる)

抽象構文と評価

以下のような言語を扱う処理系を作りたい.

$$\text{式} ::= \text{整数} \mid \text{変数} \mid \text{式} + \text{式} \mid \text{式} \times \text{式} \mid (\text{式})$$

式の例

$$5 \times 2 + 3 \qquad (3 + y) \times 12$$

抽象構文を表す型をまず定義する.

```
type expr =
    Num of int
  | Var of string
  | Plus of expr * expr
  | Mult of expr * expr
```

それに対して基本操作を定義する.

```
# let map_expr f e =                                    (* 再帰的でない map *)
    match e with
    | Num _ | Var _ -> e
    | Plus (e1, e2) -> Plus (f e1, f e2)
    | Mult (e1, e2) -> Mult (f e1, f e2)
val map_expr : (expr -> expr) -> expr -> expr
# let rec subst env e =                                 (* 変数の代入 *)
    match e with
    | Var x when List.mem_assoc x env -> Num (List.assoc x env)
    | e -> map_expr (subst env) e
val subst : (string * int) list -> expr -> expr
```

元の map_expr は再帰的でないが，subst の中では再帰的に使われている．汎関数は役に立つ．

可能な限り式を評価する関数を定義する．

```
# let rec eval e =
    match map_expr eval e with
    | Plus (Num x, Num y) -> Num (x + y)
    | Mult (Num x, Num y) -> Num (x * y)
    | e' -> e'
val eval : expr -> expr = <fun>
# let e = subst ["x", 3; "z", 2]
                (Plus (Var "y", Mult (Var "x", Var "z")));;
val e : expr = Plus (Var "y", Mult (Num 3, Num 2))
# let e' = eval e;;
val e' : expr = Plus (Var "y", Num 6)
```

練習問題 7.1　*1.* expr に関するコードを入力し，例に対して動かす．

2. 式の構文に ‘−式’ を追加し，それに合わせて定義を修正せよ．

3. expr を文字列に変える関数を定義せよ．括弧を多く使ってもいい．

8　ストリーム・パーザとプリティ・プリンター

実際に処理系を作るには，構文に合わせて文字入力を解釈し，評価した結果をユーザーに見せる必要がある．

ストリーム・パーザ　OCaml に附属している Camlp4 では簡単な字句解析を含めたパーザライブラリーが提供されている．

```
# #load"camlp4o.cma";;
        Camlp4 Parsing version 3.09.2

# open Genlex;;
# let lexer = Genlex.make_lexer ["+";"*";"(";")"] ;;
val lexer : char Stream.t -> Genlex.token Stream.t = <fun>
# let s = lexer (Stream.of_string "1 2 3 4");;
val s : Genlex.token Stream.t = <abstr>
# (parser [< ' x >] -> x) s ;;                        (* 任意のトークンを読む *)
- : Genlex.token = Int 1
# (parser [< 'Int 1 >] -> "ok") s ;;                    (* Int 1 だけを読む *)
Exception: Stream.Failure.                             (* 消されているので失敗 *)
# (parser [< 'Int 1 >] -> "one" | [< 'Int 2 >] -> "two") s ;;
```

```
- : string = "two"                                    (* 次のトークンは Int 2だった *)
```

ストリームパーザも汎関数と相性がいい.

```
(* リストをパーズしながら結果を蓄積 *)
# let rec accumulate parse accu = parser
    | [< e = parse accu; s >] -> accumulate parse e s
    | [< >] -> accu;;
val accumulate : ('a -> 'b Stream.t -> 'a) -> 'a -> 'b Stream.t -> 'a = <fun>
```

'e = parse accu' のようなパターンは，入力ストリーム s に対して関数適用 parse accu s を実行し，その結果を e に束縛する．もしも parse が Stream.Failure を起こしたら，次の場合に移る．しかし，Stream.Failure を起した関数がストリームパターンの先頭でなければならない.

```
(* 左結合の演算子を定義 *)
# let left_assoc parse op wrap =
    let parse' accu =
      parser [< 'Kwd k when k = op; s >] -> wrap accu (parse s) in
    parser [< e1 = parse; e2 = accumulate parse' e1 >] -> e2;;
val left_assoc :
  (Genlex.token Stream.t -> 'a) ->
  string -> ('a -> 'a -> 'a) -> Genlex.token Stream.t -> 'a = <fun>
```

'パターン when 条件' は条件が true になったときだけ選ばれる．通常のパターンマッチングでも使える.

練習問題 8.1 同じ優先順位の演算子が同時に使えるように left_assoc を書き換えよ.

汎関数を使うとパーザがとても短かく書ける.

```
# let rec parse_simple = parser
    | [< 'Int n >] -> Num n
    | [< 'Ident x >] -> Var x
    | [< 'Kwd"("; e = parse_expr; 'Kwd")" >] -> e
  and parse_mult s =
    left_assoc parse_simple "*" (fun e1 e2 -> Mult(e1,e2)) s
  and parse_expr s =
    left_assoc parse_mult "+" (fun e1 e2 -> Plus(e1,e2)) s ;;
val parse_simple : Genlex.token Stream.t -> expr = <fun>
val parse_mult : Genlex.token Stream.t -> expr = <fun>
val parse_expr : Genlex.token Stream.t -> expr = <fun>

# let parse_string s =
    match lexer (Stream.of_string s) with parser
      [< e = parse_expr; _ = Stream.empty >] -> e;;
val parse_string : string -> expr = <fun>
# let e = parse_string "5+x*(4+x)";;
val e : expr = Plus (Num 5, Mult (Var "x", Plus (Num 4, Var "x")))
# eval (subst ["x", 3] e);;
- : expr = Num 26
```

プリティ・プリンター

Format モジュールを使えば，式を綺麗に出力できる.

```
# let rec print_expr prio ppf e =
  let printf fmt = Format.fprintf ppf fmt in
```

```
  match e with
  | Num x -> printf "%d" x
  | Var x -> printf "%s" x
  | Mult (e1, e2) ->
      printf "@[%a *@ %a@]" (print_expr 1) e1 (print_expr 1) e2
  | Plus (e1, e2) as e ->
      if prio > 0 then printf "(%a)" (print_expr 0) e else
      printf "@[%a +@ %a@]" (print_expr 0) e1 (print_expr 0) e2;;
val print_expr : int -> Format.formatter -> expr -> unit
```

プリティ・プリンターをトップレベルで使うことができる.

```
# let print_expr0 = print_expr 0;;
val print_expr0 : Format.formatter -> expr -> unit = <fun>
# #install_printer print_expr0;;
# Plus(Num 3, Var "a");;
- : expr = 3 + a
```

read-eval-print ループ

簡単なインタープリタを作るには，パーザ・評価・プリンターを組み合わせればいい.

```
# let toplevel () =
  while true do
    try                                              (* エラー (例外) から守る *)
      print_string "? ";                                    (* ? を出力 *)
      let s = read_line() in                              (* 一行を読む *)
      let e = parse_string s in                          (* expr に変換 *)
      let e' = eval e in                                        (* 評価 *)
      Format.printf "=> %a@." (print_expr 0) e'                (* 出力 *)
    with                                          (* 例外のパターンマッチング *)
      Stream.Failure -> ()
    | Stream.Error _ -> Format.printf "Syntax error!@."
  done
val toplevel : unit -> unit = <fun>
# toplevel ();;
? 5+3*2
=> 11
? 5+3*a
=> 5 + 3 * a
? a+3*2
=> a + 6
? <C-c><C-c>Interrupted.
```

練習問題 8.2 *1.* 式の構文に –式 を追加し，パーザとプリンターも修正せよ.

2. 同様に 'let 名前 = 式 in 式'という構文を追加せよ．そのために lexer を以下のように定義しなければならない.

```
let lexer = Genlex.make_lexer ["+";"*";"(";")";"=";"let";"in"] ;;
```

subst や eval の定義にも気を付けよ．トップレベルでの動作は以下のようになる.

```
# toplevel ();;
? let x = 1 + 1 in x + (let y = 1+2 in y*y)
=> 11
```