Jacques Garrigue, 2012 年 10 月 3 日

OCaml プログラミングの基礎

1 OCaml とは

OCaml 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 70 年代に Edinburgh LCF という証明器のために Robin Milner[1]が考案した言語である．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

近年，OCaml に関する日本語の書籍が増えて来た．

- OCaml-Nagoya 著，入門 OCaml・プログラミングの基礎と実践理解，毎日コミュニケーションズ， 2007 年
- 五十嵐 淳著，プログラミング in OCaml，技術評論社，2007 年
- 浅井 健一著，プログラミングの基礎 (Computer Science Library 3)，サイエンス社，2007 年

また，インターネット上の情報も豊富にある．

`http://www.math.nagoya-u.ac.jp/~garrigue/lecture/2012_AW/`
この講義に関する資料．

`http://caml.inria.fr/`
OCaml 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．マニュアルの英語版と日本語版も置いたる．

`http://www.ocaml-lang.org/`
OCaml コミュニティのサイト．チュートリアルなども豊富．

`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/`
過去の講義資料．

[1]Robin Milner (1934–2010) イギリスの計算機科学者．並行計算の研究でも知られる．1991 年度 Turing 賞受賞．

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

プログラムの編集や実行には Emacs が便利である．使い方について別紙を参照．

2.3　OCamlBrowser の使い方

もう一つの便利なツールとしては `ocamlbrowser` がある．シェルで

```
$ ocamlbrowser &
```

と入力すると新しいウィンドウが開く．メニューで File – Shell を選ぶとインタープリタが使える．

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 = 12                      (* 新しい関数なのでは新しい y を使います *)
```

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	$-2^{30} \sim 2^{30}-1$ (64 ビットだと 2^{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，定数倍と足算を計算する関数を定義せよ．

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

そこで使う関数は `*.` (実数の掛け算), `+.` (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．

ベクトルの定数倍も，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 mult x v = map (fun x' -> x *. x') v ;;
val mult : 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/2010_AW/ 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 *)
```

- 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．

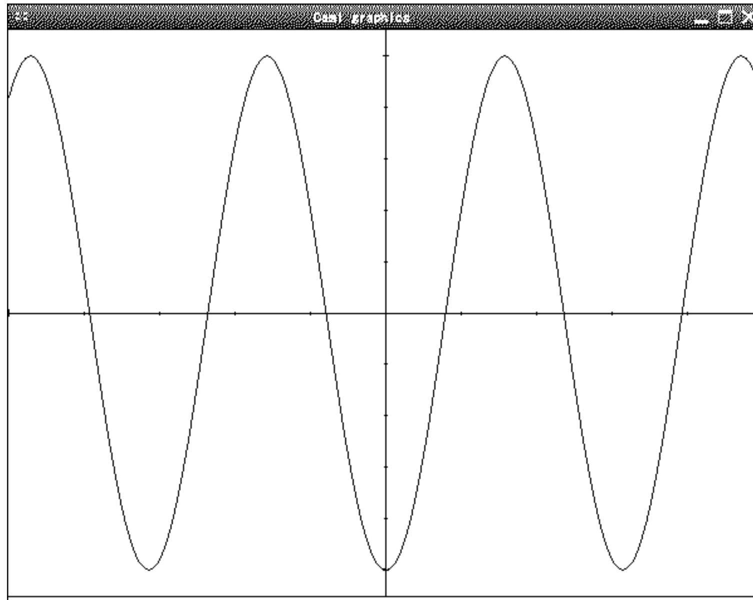


図 1: Plot を使った例

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 反復と再帰

再帰関数の定義

自明でないプログラムを書くにはループ (反復) 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．

```
# let rec fact n =                                    (* let rec を使う *)
    if n = 0 then 1 else n * fact (n-1) ;;
val fact : int -> int = <fun>
# fact 5;;
- : int = 120
```

let rec というキーワードは再帰的な定義を表している．定義の中で定義しようとする関数を使ってもいい．分岐の中に再帰呼び出しを入れれば，実行の回数が制御できるわけだ．

#trace というコマンドで実際の呼び出しを見ることができる．

```
# #trace fact;;
fact is now traced.
# fact 3;;
fact <-- 3
fact <-- 2
fact <-- 1
fact <-- 0
fact --> 1
fact --> 1
fact --> 2
fact --> 6
- : int = 6
# #untrace fact;;
fact is no longer traced.
```

ちなみに，定義しようとする関数が見えているので，**and** を使うと相互再帰になる．

```
# let rec even n = if n = 0 then true else odd (n-1)
  and odd n = if n = 0 then false else even (n-1) ;;
val even : int -> bool = <fun>
val odd : int -> bool = <fun>
# even 3;;
- : bool = false
# odd (-1);;                                     (* 無限ループに陥いる *)
^CInterrupted.                                   (* キーボードで Ctrl-C *)
```

練習問題 6.1 以下のものを計算する再帰関数を定義せよ

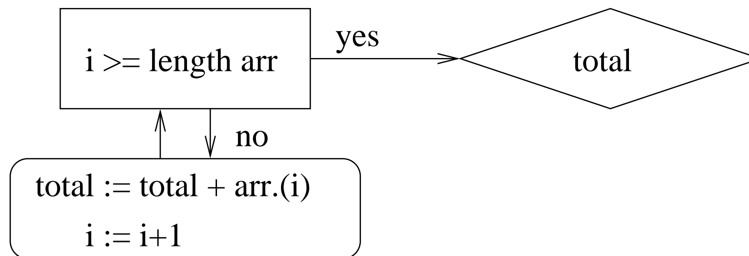
1. 1 から n までの合計 $(\sum_{i=1}^{n} i)$
2. 高速乗算法を行う関数．x と n が与えられたとき，もしも n が偶数だったら，$x^{2n} = x^n \cdot x^n$，奇数だったら $x^{2n+1} = x^n \cdot x^n \cdot x$．プログラムでは x は float で，n は int にする．

反復の再帰への翻訳

さて，今まで for ループで定義していたものは再帰でも定義できるか？最初は前に見た配列の合計からやって見よう．

```
# let sum4 (arr : int array) =
    let rec loop i total =                              (* 元の sum のループの直訳 *)
      if i >= Array.length arr then total
      else loop (i+1) (total + arr.(i))
    in loop 0 0 ;;
val sum4 : int array -> int = <fun>
# sum4 [|1;2;3|];;
- : int = 6
```

内部で変更する全ての値を引数とする再帰関数 `loop` を定義すればいい．その値を変えてループを続けるときは `loop` を新しい値で呼び，終るときは最終的に返したい値だけを返せばいい．以下の図ではその処理を示す．



for ループや while ループで表現されるのは図の左の部分だけですが，再帰関数では結果を返すところまで含まれる．

さらに，再帰関数だと，戻りながらも計算できる．

```
# let sum5 (arr : int array) =
    let rec loop i =                                            (* 引数が i のみ *)
      if i < 0 then 0 else loop (i-1) + arr.(i)
    in loop (Array.length arr - 1) ;;
val sum5 : int array -> int = <fun>
# sum5 [|1;2;3|];;
- : int = 6
```

こちらのプログラムを for ループで表現することが難しい．`loop i` の結果は配列の `i` 番目までの合計だが，処理として一旦 `i` を `Array.length arr - 1` から`-1` まで下げて行って，今度は戻りながら配列の要素を結果に足していく構造だ．

末尾再帰

`sum4` が一対一でループを使ったプログラムに対応しているのに，`sum5` はそういう対応ができない．こういうときは，`sum4` の `loop` 関数を末尾再帰といい，`sum5` のは末尾再帰でないという．区別する方法は，再帰呼び出しの後に計算が残っているかどうかだ．計算が残らない場合は OCaml が再帰呼び出しを末尾呼び出しと認識し，ジャンプとしてコンパイルされるので，結果的に for ループと全く同じコードが生成される．効率の面で末尾再帰が有利なときが多いが，そのためにプログラムが分かりにくくなったり，他のところで非効率になることもありうるので，常に末尾再帰にこだわる必要がない．

反復の抽象化

`iter` は配列の各要素にある関数を適用することで反復を抽象化しているが，`iter` を使うときは可変な変数を使わなければならない．下記のように，可変な変数を全く使わない汎関数も定義できる．

```
# let fold f init arr =
    let len = Array.length arr in
    let rec loop i r =
      if i >= len then r else loop (i+1) (f r arr.(i))
    in loop 0 init ;;
val fold : ('a -> 'b -> 'a) -> 'a -> 'b array -> 'a = <fun>
# let sum6 = fold (+) 0 ;;
val sum6 : int array -> int = <fun>
# sum6 [|1;2;3|];;
- : int = 6
```

練習問題 6.2 *1.* fold を使って，配列の中の最大値を求めよ．

2. fold と ^ を使って文字列の配列を空白で区切られた文字列に変えよ．
`concat [|"Little"; "brown"; "fox"|] --> "Little brown fox"`

再帰関数と帰納法

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.

練習問題 6.3 sum5 が正しく配列の中身の合計を計算していることを証明せよ．

線形でない再帰

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 6.4 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．
ヒント：補助関数が fib(n) と fib$(n-1)$ の組みを返せばいい．

7 リストと構造的帰納法

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 7.1 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 *)
val hd : 'a list -> 'a
# let tl l =
    match l with
      []     -> failwith "List.tl"
    | _ :: t -> t ;;                                      (* 後部を返す *)
val tl : 'a list -> 'a list
```

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                                    (* 空リストの長さは 0 *)
    | _ :: l' -> 1 + length l' ;;
val length : 'a list -> int
# let rec append l1 l2 =
    match l1 with
      []      -> l2                           (* l1 が空リストなら l2 を返す *)
    | a :: l' -> a :: append l' l2 ;;
val append : 'a list -> 'a list -> 'a list
# let rec iter (f : 'a -> unit) (l : 'a list) =
    match l with                           (* リストの先頭から f を適用する *)
      []      -> ()
    | a :: l' -> f a; iter f l' ;;
val iter : ('a -> unit) -> 'a list -> unit
# let rec map (f : 'a -> 'b) (l : 'a list) =
    match l with
      []      -> []           (* 頭に f を適用してからリストの残りに map をかける *)
    | a :: l' -> let b = f a in b :: map f l' ;;
val map : ('a -> 'b) -> 'a list -> 'b list
```

練習問題 7.2 1. リストの和を計算する関数を再帰関数として書きなさい.

```
val sum : int list -> int
```

2. ある条件を満している要素のリストを返す関数 filter を書きなさい.

```
val filter : ('a -> bool) -> 'a list -> 'a list
```

3. 配列に対して定義した fold をリストに対する再帰関数として書きなさい.

```
val fold : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

上記の List.fold_left に当る.

4. 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 *)
```

構造的帰納法

上の 3 つの関数では, 再帰呼び出しがリストの尻尾に対して行われている. こんなように, 再帰呼び出しが引数の一部分に対して行われている帰納法を**構造的帰納法**という. 自然数に対する帰納法も構造的帰納法の一例である.

```
# type nat = unit list ;;                            (* 型の略称を定義する *)
type nat = unit list
# let rec nat_of_int n : nat =                         (* 結果を nat にする *)
  if n <= 0 then [] else () :: nat_of_int (n-1) ;;
val nat_of_int : int -> nat
# let int_of_nat : nat -> int = length ;;           (* length が変換になる *)
val int_of_nat : nat -> int
# let add_nat : nat -> nat -> nat = append ;;             (* append が足算 *)
val add_nat : nat -> nat -> nat
# int_of_nat (add_nat (nat_of_int 2) (nat_of_int 3));;
- : int = 5
```

多くの関数は直接な構造的帰納法で書けるが，効率がよくない場合もある．たとえば，リストを逆順にする rev 何も考えずに書くとこんな感じになる．

```
let rec rev l =
  match l with
    [] -> []
  | a :: l' -> append (rev l') [a] ;;
```

しかし，これで append が呼ばれる回数は l の長さの 2 乗に比例する．補助関数を使うと長さに対して線形な関数が書ける．

```
let rec rev_append l1 l2 =
  match l1 with
    []     -> l2
  | a :: l -> rev_append l (a :: l2)
;;
let rev l = rev_append l [] ;;
```