

2018年度 プログラミング B OCaml 入門

2018年 4月17日

情報理工学科 上田 和紀

- ◆ 起動法 (Linux, Mac OS X, Cygwin)
 - ターミナルウィンドウから `rlwrap ocaml`
- ◆ 脱出法
 - `exit 0;;` を入力（最後の `;;` は入力の終わりを表す）
- ◆ Emacs 内部から起動する場合は
 1. `M-x tuareg-mode`を実行，または適当なバッファでプログラム (*.ml) を作成
 2. `M-x run-ocaml` を実行
 3. 元のバッファのプログラムを読み込むには，元のバッファで `C-c C-b`（または `C-c C-e`）

(tuareg-mode 未インストールの場合は `M-x shell` を実行してシェルを起動して `ocaml` コマンドを実行)

```
$ rlwrap ocaml
```

```
OCaml version 4.02.3
```

```
# 1 + 23 / 4;;
```

```
- : int = 6
```

```
# 1 + 23.0 / 4;;
```

```
Characters 4-8:
```

```
1 + 23.0 / 4;;
```

```
^^^^
```

```
Error: This expression has type float but an expression  
was expected of type int
```

```
# 1.0 +. 23.0 /. 4.0;;
```

```
- : float = 6.75
```

read-eval-print loop (REPL)
が始まる

```
# let pi = 4.0 *. at an(1.0);;  
val pi : float = 3.14159265358979312  
# let area r = pi *. r *. r;;  
val area : float -> float = <fun>  
# area 2.0;;  
- : float = 12.5663706143591725  
# area (float_of_int 2);;  
- : float = 12.5663706143591725
```

結果の型

結果の値

何が定義されたか

- ◆ プログラムは**フレーズ (phrase)** の列
 - ユーザは, OCamlトoplevelから会話的にプログラムを入力していることになる
- ◆ 個々のフレーズは**式**か**定義**, **::** で終わる
 - 定義は, **名前** と **値や型など** とを関連づける
 - **値定義** `let pi = 4.0 *. atan(1.0);;`
`let area r = pi *. r *. r;;` ← **これも値定義!**
 - **型定義** `type complex = {re:float; im:float};;`
 - 式も定義もすぐ実行される

- ◆ OCaml トップレベルからはフレーズのほかに指令 (directive) も入力できる
 - `#quit;;` (これでも脱出できる)
 - `#use "file_name";;` (ファイル内容の読み込み)
 - `#cd "directory_name";;`
 - `#trace function_name;;`
 - `#untrace;;`
 - `#print_depth n;;`
 - `#print_length n;;`
- ◆ ファイル中のコメントは `(* *)` で囲む

- ◆ 小さな関数を定義し，テストし，組み合わせてゆく（関数合成！）のが関数型プログラミング
- ◆ ルーツは **λ 計算**（ラムダ計算，1930's）
 - 下記のメカニズムしかないが，これだけあれば原理的にはプログラムの表現に十分
 - 変数
 - 関数を定義する
 - 関数に引数を適用 (apply) する
 - 実用的な関数型言語は，基本型とその演算，宣言，再帰関数定義機能などを最初から提供

◆ int

- ~~max_int~~ min_int
- + - * / mod
- succ pred abs
- l and l or l xor l not
- l sl l sr asr

◆ float

- ~~max_float~~ min_float
- +. -. *. /. **
- sqrt
- exp log log10
- cos sin tan
- ceil floor

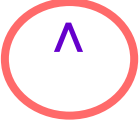
型変換はプログラマが明示する

- float_of_int int_of_float

◆ char

- 'a'
- '¥¥' '¥"' '¥' '
- '¥n' '¥r' '¥t'
- '¥b'
- '¥065' '¥x41'

◆ string

- "Hello! ¥n"
-  (文字列の連結)

- 関数名だけを与えると関数の型がわかる
- 中置演算子は (^) のように括弧で囲むと通常関数になる

◆ bool

- `false` `true`

- `not`

- `&&` `||`

- 右オペランドは評価
しないかもしれない

◆ unit

- `()` が唯一の値

- C や Java の `void` に
似た役割をもつ

- いろいろな式を実際に入力して
実験的に確かめよう！

◆ 比較演算子は (なぜか) 汎用 (generic)

- `=` `<` `>` `<=` `>=`
- `compare`
- `min` `max`
- (`=` と `!=` はこの講義では扱わない)

◆ 条件式

- `if b then e1 else e2`
- `if b then e1 else ()`
 - `b1 && b2`, `b1 || b2` は,
条件式と `true`, `false` を使って書き直せる

- ◆ `let 名前 = 式 and ... and 名前 = 式` が基本
- ◆ 名前の構文は
 - `名前 ::= (文字 | _) { 文字 | 0...9 | _ | ' }`
 - `文字 ::= A...Z | a...z`
- ◆ `let` は `C` の代入文とは異なる. 定数宣言に近い.
 - ただし同じ名前の再定義は可能で, 以前の定義が隠される.

```
# let twelve = 12;;  
val twelve : int = 12  
# let twelve = string_of_int twelve;;  
val twelve : string = "12"
```

関数も let で定義

引数の括弧は
書いてもよいが書かない

```
# let next x = x + 1;;
val next : int -> int = <fun>
# next (next (-5));;
- : int = -3
```

多引数関数は
Curry化して定義

```
# let f a b x = a * x + b;;
val f : int -> int -> int -> int = <fun>
# f 2 (-3) 5;;
- : int = 7
# let g = f 2 (-3);;
val g : int -> int = <fun>
# g 5;;
- : int = 7
```

let f (a, b, x) = a * x + b;;
と書いても良いが、
Curry化するのが OCaml流

- ◆ $f\ a\ b\ x$ は $((f\ a)\ b)\ x$ と同じ
 - つまり関数適用は左結合的 (left-associative)
- ◆ 一方, f の型である $int \rightarrow int \rightarrow int \rightarrow int$ は $int \rightarrow (int \rightarrow (int \rightarrow int))$ のこと
 - つまり関数の型を表す \rightarrow は右結合的 (right-associative)
- ◆ f は, あと三つ引数をもらうと int 型の値が返る関数.
 $f\ a\ b$ は, a と b が固定された $int \rightarrow int$ 型の関数
 - つまり多引数関数は, 引数を一つ与えるごとに, \rightarrow の数が一つ減った関数になってゆく

1. `sqrt 2.0*.2.0` の値はいくつか？

2. x と y をもらって $\sqrt{x^2 + y^2}$ を計算する関数 `distance` を書いてみよう

◆ その他：

- いろいろな式を入力して，評価結果の型と値をみてみよう.
- スライドに紹介した関数や自作の関数を入力して，型推論結果をみるとともに，いろいろな関数呼出しを実行してみよう.
- わざとまちがえて，エラーの出方を見ておこう.

```
# let rec fact n =
```

```
  if n=0 then 1
```

まずは簡単な場合 (base case) を書く

```
  else n * fact (n-1);;
```

次に再帰を伴う場合を書く

```
val fact : int -> int = <fun>
```

関数適用はどの中置演算子よりも優先順位が高いので、この括弧は省略不可

```
# fact 8;;
```

```
- : int = 40320
```

- 再帰呼出しをトレースしてみよう
- キーワード **rec** を忘れるとどうなるか？
 - ヒントは何枚か前のスライドにあり

再帰関数の定義

- ◆ 一般に, 再帰関数は以下の2つを書くことで定義する:
 - 1つ以上の base case (再帰せずに答えが求まる場合)
例: $0! = 1$
 - 1つ以上の再帰を伴う場合
例: $n! = n * (n - 1)! \quad (n > 0)$
- ◆ 再帰呼出しの連鎖は有限回で終わらなければならない.
例: $n! = (n+1)! / (n+1)$ は, 性質としては正しいが階乗の定義としては使えない.

再帰関数の定義 (べき乗とフィボナッチ)

```
# let rec power n x =  
    if n=0 then 1 else x * power (n-1) x;;  
val power : int -> int -> int = <f un>  
# (power 4) 3;;  
- : int = 81
```

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

注意:
とても良くない
アルゴリズム !!

- ◆ それぞれ, 計算量 (computational complexity)を改善できないか?

- ◆ 処理系は、入力した式の型を推論して表示する. 演算結果だけでなく型推論結果にも注目しよう.
- ◆ 中置演算子を（演算子でなく）普通の関数として使いたいときは、括弧で囲む.

```
# (mod) ; ;
```

```
- : int -> int -> int = <f un>
```

```
# (=) ; ;
```

```
- : 'a -> 'a -> bool = <f un>
```

- 'a の ' は型変数につける記号. 上の二つの 'a はどんな型でも良いが同じ型を表す.

無名関数 (anonymous function) と略記法

- ◆ 関数に名前をつけないで（再帰的でない）関数を書くことができる（最近は多くの言語が無名関数を備えるようになった）

```
# function x -> x*x*x;;
```

```
- : int -> int = <fun>
```

```
# (function x -> x*x*x) 5;;
```

```
- : int = 125
```

- ◆ `function x -> function y -> x*y;;` は `fun x y -> x*y;;` と略記できる

- ◆ さらに `let f = (fun x y -> x*y);;` は `let f x y = x*y;;` と略記できる

- ◆ OCaml では、同じ型の要素を 0 個以上並べたものをリストという。関数型言語では、リストは（手続き型言語の配列のように）不可欠なデータ構造。

- ["red"; "green"; "blue"]

- ["singleton"]

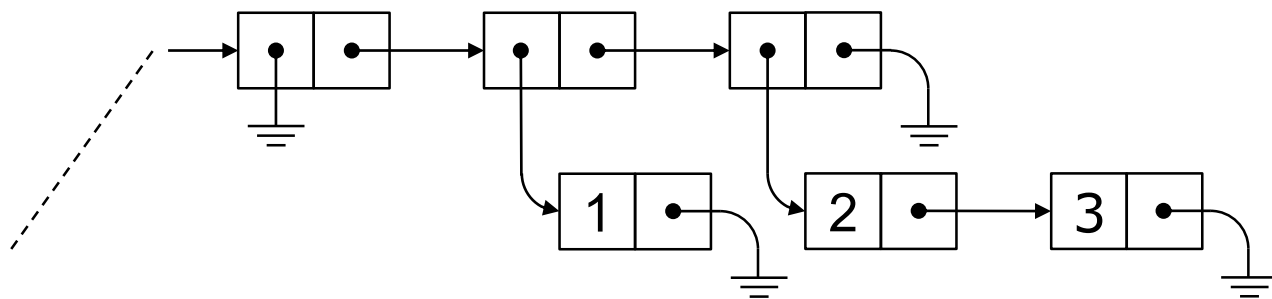
- []

- [[]]

- [[]; [1]; [2; 3]]

- [5; false]

- これらのリストは、それぞれどのような型をもつか？



- ◆ リストは先頭からの**逐次アクセス**のみ、配列は**ランダムアクセス**が可能。しかし...
- ◆ リストは先頭要素の追加削除が容易、配列要素の追加削除は困難
 - cf. Java の ArrayList は両機能を備えるが軽量ではない
- ◆ リストは再帰関数による操作に適する。配列は繰返し構文による操作に適する。
 - **再帰関数とパターンマッチング (後述) によるプログラミングに慣れることが、本講義の重要な目的の一つです。**

理由1: メモリの確保と解放は処理系が自動的にやってくれる
(malloc, free, new 等は一切不要)

- 最近のプログラミング言語が備える **ガーベジコレクション** (garbage collection) の考え方と技法は、関数型言語のルーツである Lisp 処理系の中で60年前から培われてきた

理由2: ポインタの概念をプログラマから安全に隠蔽している

- NullPointerException (何も指していないポインタをたどる) は起こりえない

リストは帰納的 (inductive) に定義される

24

“再帰的” (recursive)
という人もいる

◆ 「リストとは何か」をきちんと定義すると

1. 空のリスト $[]$ は, T 型を要素とするリストである
2. T 型を要素とするリスト l の先頭に T 型の値 a を追加したものは, T 型を要素とするリストである
 - これを $a :: l$ と書く
3. 以上のように構成されるものだけが T 型を要素とするリストである

◆ リストを扱う関数の大多数は, 入力を 1. と 2. に場合分けして, 再帰を用いて定義する.

◆ $a :: l$ の $::$ を list constructor (リスト構成子) という

- $::$ は cons と読む (もともとは Lisp 用語)
- $::$ は右結合的, つまり $2 :: 3 :: 5 :: [] = 2 :: (3 :: (5 :: []))$

【注意】 $::$ でリストとリストをつなげることはできない

- 《NG》 $[1; 2; 3] :: [4; 5]$
 - リストとリストをつなげるには, 後述の **append** または組込みの **@**演算子を使う
- 《NG》 $[1; 2; 3] :: 4$
 - リストの最後尾に要素を 1 個追加するには?

◆ $[e_1; e_2; \dots; e_n]$ は, 実は $e_1 :: e_2 :: \dots :: e_n :: []$ を略記したもの
にすぎない

- 両者はまったく同じものを表す
- 要素を具体的に列挙するときは略記法を使うのが普通.
引数でもらってきたリストの先頭に要素を追加したり,
パターンマッチングで分解したりするときに $::$ を使う
(次頁)

【注意】 “;” のかわりに “,” で区切ると別の型のデータ (値の
組を要素とするリスト) になる

- 例: $[1; 3]$ vs. $[1, 3]$ ($= [(1, 3)]$)

◆ 要素の総和を計算する

```
# let rec sum l s =  
  match l s with  
  | [] -> 0  
  | a::rest -> a + (sum rest) ;;
```

match 式

◆ 以下のように書いてもよい

```
# let rec sum = function l s ->  
  match l s with  
  | [] -> 0  
  | a::rest -> a + (sum rest) ;;
```

先頭要素 a と
残りのリスト rest
に分解できる

◆ 一般形は

```
match <式> with  
| <パターン1> -> <式1>  
|  
| ...  
| <パターンn> -> <式n>
```

◆ 例: # let rec fib = function i ->
match i with
| 0 -> 0
| 1 -> 1
| j -> fib (j-1) + fib (j-2);;

単なる変数は wildcard

上から下に
マッチング
を試みる

パターンマッチングで代用可能な基本関数

- ◆ リストの先頭要素を返す

```
# let hd ls = function  
  h::t -> h;;
```

```
[ # let hd (h::t) = h;;  
  と略すこともできる ]
```

- ◆ リストの第2要素以降を返す

```
# let tl (h::t) = t;;
```

- ◆ リストが空かどうか調べる

```
# let null ls = (ls = []);;
```

- ◆ 例：要素の総和を求める

```
# let rec sum ls =  
  if null ls then 0  
  else hd ls + sum (tl ls);;
```

アドバイス：

hd, tl はライブラリ関数として用意されているが、リストの分解や検査にはできるだけパターンマッチを使おう

リストを扱う関数の例 (2)

◆ 2本のリストをつなげる

```
# let rec append a b =
  match a with
  | [] -> b
  | h::t -> h :: append t b;;
```

@ という中置演算子として
Top-levelで提供されている

```
# append [1; 2; 3] [4; 5];;
- : int list = [1; 2; 3; 4; 5]
# append ["ki wi"; "apple"] ["grape"];;
- : string list = ["ki wi"; "apple"; "grape"]
```

いろいろな型のリストの
連結に使える (多相性)

リストを扱う関数の例 (3)

- ◆ a がリスト $l s$ に含まれているかどうか調べる

```
# let rec member a l s =  
    match l s with  
    | [] -> false  
    | h::t -> (a=h) || member a t;;
```

- ◆ 以下のようにも書ける

```
# let rec member a = function  
    | [] -> false  
    | h::t -> (a=h) || member a t;;
```

リストを扱う関数の例 (4)

- ◆ 整数 m と n をもらい, m 以上 n 以下の整数を並べたリストを作る (練習問題)

- 例: `range 1 9` \rightarrow `[1; 2; 3; 4; 5; 6; 7; 8; 9]`

```
# let rec range m n =
```

```
  if  then []
```

```
  else  ;;
```


リストを扱う関数の例 (5)

33

- ◆ 2本のリストをもらい、いずれかのリストに現れる要素のリストを作る (練習問題)

● 例 : `union [5; 1; 3] [7; 5; 3] → [1; 7; 5; 3]`

```
# let rec union a b =  
  match a with
```

```
| [] -> 
```

```
| h::t -> if member 
```

```
  then 
```

```
  else 
```

```
;;
```

個々のリストは相異なる要素からなるものとする

member を読み済みか確認しよう

累積引数を用いた再帰的定義

◆ リストの和を求めるもう一つの関数

```
# let rec sum_iterative l s n =
  match l with
  | [] -> n
  | a::rest -> sum_iterative rest (a+n);;
# let sum l s = sum_iterative l s 0;;
```

これまでに見た
値の合計を保持
する補助引数
(accumulator)

cf.

```
# let rec sum l s =
  match l with
  | [] -> 0
  | a::rest -> a + sum rest s;;
```

累積引数
(accumulator)
の準備と初期化

◆ 前頁の補助関数

```
# let rec sum_iterative l s n =  
  match l with  
  | [] -> n  
  | a::rest -> sum_iterative rest (a+n);;
```

のような, 「再帰呼出し後の作業が何もない」再帰関数を
末尾再帰関数 (tail-recursive function) と呼ぶ.

- 一般化して, 最後の作業が自分自身または別の関数の呼出しである場合, それを**末尾呼出し (tail call)** と呼ぶ.

- ◆ 末尾呼出しは戻り番地をスタックしなくてよい.
 - 例: **f** が **g** を呼び、**g** が **h** を末尾呼出しするとき、**h** は直接 **f** に戻ればよい
- ◆ したがって、末尾再帰関数は手続き型言語のループのように高い効率で実行できる.
 - このため、末尾再帰関数を「**繰返しの**」(iterative) とよぶこともある.
- ◆ プログラムの可読性は、accumulator を使った末尾再帰関数よりも「素直」な再帰関数の方が一般に高い.

末尾再帰形への変換

```

let rec fact n = fact_loop 1 n
and fact_loop k n =
  if n=0 then k
  else fact_loop (n*k) (n-1);;

```

は、下の手続き型プログラムの末尾再帰的表現

```

int fact(int n) {
  int k = 1;
fact_loop: while (!(n==0)) {
  k = n*k; n = n-1;
}
return k;
}

```

両者の関係を
よく確認しよう