プログラミング言語 2
関数型プログラミング (OCaml)
Programming Languages 2
Functional Programming (OCaml)

田浦

目次

Contents

関数型言語とは

関数型言語とは

- . . . というゴタクから入るよりも，まずは使って見るのがその良さを理解するには一番だろう

関数型言語とは

- . . . というゴタクから入るよりも，まずは使って見るのがその良さを理解するには一番だろう
- そこで関数型言語の中でも人気の高い OCaml を実習する
- OCaml について最低限の情報，自習するためのポインタを与え，ドリルを行う

What are Functional Languages?

What are Functional Languages?

- … can be best answered by trying one

What are Functional Languages?

- ... can be best answered by trying one
- learn OCaml, one of the most popular functional languages
- exercise OCaml with a minimum information and a pointer to teach yourself

Contents

早見表 — トップレベル定義

- 変数定義

```
# let x = 1 + 2 ;;
val x : int = 3
```

- 関数定義

```
# let f x = x + 3 ;;
val f : int -> int = <fun>
```

- 再帰関数定義

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

注:

- 実は ;; は文法の一部ではない
- ocaml インタプリタにプログラムを放り込む区切り
- Jupyter 環境 (SHIFT + ENTER で入力) では不要
- ファイルにプログラムを書く際も不要

Cheatsheet — toplevel definition

- variable definition

```
# let x = 1 + 2 ;;
val x : int = 3
```

- function definition

```
# let f x = x + 3 ;;
val f : int -> int = <fun>
```

- recursive function definition

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

Remarks:

- `;;` is not part of the syntax
- it is a delimiter that marks the end of an input
- it is not necessary in Jupyter (SHIFT + ENTER marks it)
- it is not necessary either when you write programs in a file

早見表 — 式について少し

- 関数定義や適用時の引数はカッコもカンマもなし (`f x y ...`)
- if 式 then 式 else 式
- let 変数 = 式 in 式

```
let f x0 y0 x1 y0 =
  let dx = x0 - x1 in  (* 局所定義 *)
  let dy = y0 - y1 in
  dx * dx + dy * dy ;;
```

Cheatsheet — basic expressions

- function applications (`f x y ...`)
  - note: *no* parens nor commas
- `if` E `then` E `else` E
- `let` *var* `=` E `in` E
  - do not be confused with toplevel definition
- ex.

```
let f x0 y0 x1 y0 =
  let dx = x0 - x1 in  (* local definition *)
  let dy = y0 - y1 in
  dx * dx + dy * dy ;;
```

早見表 — タプル (組)

- タプルリテラル

```
# let trip = 3,4.5,"hello";;
val trip : int * float * string = (3, 4.5, "hello")
```

- 複数の値を組み合わせたひとつの値を作る簡便な手段
- その型は要素の型の組み合わせ ($type * type * \ldots$)

Cheatsheet — Tuple

- tuple literals

```
# let trip = 3,4.5,"hello";;
val trip : int * float * string = (3, 4.5, "hello")
```

- it is a convenient means to combine multiple values into one
- the type of a tuple expression is the combination of component types (*type* $*$ *type* $*$ $\ldots$)

早見表 — リスト

- 空リスト:

```
[]
```

- リテラル:

```
[ 10; 20; 30; 40 ]}
```

- :: は要素 (x) とリスト (l) を取り, l の先頭に x が追加されたリストを作る

```
10 :: [ 20; 30; 40 ]
```

```
10 :: 20 :: 30 :: 40 :: [] ;;
```

- @ で 2 つのリストの連結

```
[ 10; 20 ] @ [ 30; 40 ]
```

Cheatsheet — List

- empty list:

```
[]
```

- list literals:

```
[ 10; 20; 30; 40 ]
```

- `::` takes an element (x) and a list (l) and returns a list that adds x in front of l

```
10 :: [ 20; 30; 40 ]
```

```
10 :: 20 :: 30 :: 40 :: [] ;;
```

- `@` concatenates two lists

```
[ 10; 20 ] @ [ 30; 40 ]
```

早見表 — match

- リストに対するマッチの基本形

```
let rec list_length l =
  match l with
     [] -> 0                   (* 空の場合 *)
   | a :: r -> 1 + list_length r  (* 空じゃない場合 *) ;;
```

- 実はもっと柔軟

```
match 式 with
   [] -> 式  (* 空 *)
 | [a] -> 式 (* 一要素 *)
 | a :: b :: r -> 式 (* 二要素以上 *)
```

など

- タプルにも使える

```
match 式 with
   (x,y,z) -> 式
```

Cheatsheet — match

- the basic form for lists

```
let rec list_length l =
  match l with
     [] -> 0                    (* empty case *)
    | a :: r -> 1 + list_length r  (* non-empty case *) ;;
```

- it is in fact more flexible; e.g.,

```
match E with
   [] -> E  (* empty *)
 | [a] -> E (* singleton *)
 | a :: b :: r -> E (* two or more elements *)
```

- works for tuples too

```
match E with
   (x,y,z) -> E
```

早見表 — 式一覧

```
式 ::= 変数
    | リテラル     (* 整数, 浮動小数点数, '字', "文字列", true, false, () *)
    | 式 式 ...                          (* 関数適用 *)
    | [ 式; 式; ... ]                    (* リスト *)
    | 式, 式, ...                        (* タプル *)
    | if 式 then 式 else 式
    | let [rec] 定義 and 定義 and ... in 式
    | match 式 with
            パターン -> 式
          | パターン -> 式
          | ...
    | ( 式 )

定義 ::= 変数 変数 ... = 式
```

Cheatsheet — Summary of Expressions

```
E ::= identifier
    | literals (* integers, floats, 'c', "string", true, false, () *)
    | E E ...                            (* function application *)
    | [ E; E; ... ]                      (* list *)
    | E, E, ...                          (* tuple *)
    | if E then E else E
    | let [rec] Def and Def and ... in E
    | match E with
          Pattern -> E
        | Pattern -> E
        | ...
    | ( E )

Def ::= identifier identifier ... = E
```

早見表 — ライブラリの利用

- ライブラリ関数は，モジュール名. 関数 で参照. モジュール名は，http://caml.inria.fr/pub/docs/manual-ocaml-4.01/index.html Part IV を参照.

```
Random.int 30
```

- 一部のライブラリは ocaml 起動時，コンパイル時にライブラリのファイルを指定する必要あり.

```
$ ocaml unix.cma
        Objective Caml version 3.12.1

# Unix.gettimeofday ();;
- : float = 1396802697.034235
```

- Jupyter 環境では，最初から組み込まれている以外のものを使うのは難しい (多分)

Cheatsheet — Using Libraries

- use "module.function" to reference a function in a module; see Part IV of http://caml.inria.fr/pub/docs/manual-ocaml-4.01/index.html to find available modules

```
Random.int 30
```

- when using ocaml inerpreter, some libraries require module file name(s) in the command line

```
$ ocaml unix.cma
        Objective Caml version 3.12.1

# Unix.gettimeofday ();;
- : float = 1396802697.034235
```

- when using Jupyter, it seems difficult to use non builtin functions

早見表 — データ型 (1)

- 要素をとる型

```
type rgb_color = Rgb of (int * int * int) ;;
```

  - → Rgb (1,2,3) という式が rgb_color という型を持つようになる
  - ここで定義された Rgb を (rgb_color 型の)「構築子 (constructor)」と呼ぶ
  - 関数のようなものだが関数ではない ("Rgb" 単独ではエラー)

- 複数の構築子を持つ型 (バリアント):

```
type rgb_color = Rgb of (int * int * int)
               | Rgba of (int * int * int * int)
               | Black | White | Blue ;;
```

Cheatsheet — data types (1)

- a type can be built from an existing type

```
type rgb_color = Rgb of (int * int * int) ;;
```

  - → an expression `Rgb (1,2,3)` will have a type `rgb_color`
  - `Rgb` is called a constructor (of `rgb_color` type)
    - a constructor name must begin with a capital letter
    - a variable/type name mustn't
  - a constructor looks like a function but it is not (a standalone "`Rgb`" is not a valid expression)

- a type can have multiple constructors (variant):

```
type rgb_color = Rgb of (int * int * int)
               | Rgba of (int * int * int * int)
               | Black | White | Blue ;;
```

早見表 — データ型 (2)

- 型は型パラメータをとれる:

```
type 'a cell = Cell of 'a ;;
```

  - 型パラメータは引用符 (') で始まる
- 型は再帰的でも良い

```
type hoge = ... | ... of hoge ...
```

- すべてを組み合わせ, 以下で 2 分木が定義できる

```
type 'a bintree = Empty | Node of ('a bintree * 'a bintree);;
```

- 一般には,

```
type (型)変数名 (型)変数名 ... 型名 =
  構築子名 ( of T )?
| 構築子名 ( of T )?
|  ...
```

  - T : 型式 (type expression)

Cheatsheet — data types (2)

- a type can take type parameters:

```
type 'a cell = Cell of 'a ;;
```

  - a type parameter must begin with a quote (')
- a type can be recursive

```
type hoge = ... | ... of hoge ...
```

- all combined, a binary tree type can be defined as follows:

```
type 'a bintree = Empty | Node of ('a bintree * 'a bintree);;
```

- general syntax

```
type identifier identifier ... identifier =
  constructor ( of T )?
| constructor ( of T )?
|  ...
```

  - T : *type expression*

早見表 — match

- match は type で定義された型にも使える

```
let rec tree_sz t =
  match t with
    Empty -> 0
  | Node (l,r) -> 1 + tree_sz l + tree_sz r ;;
```

Cheatsheet — match

- match can be used for user-defined types. e.g.,

```
let rec tree_sz t =
  match t with
    Empty -> 0
  | Node (l,r) -> 1 + tree_sz l + tree_sz r ;;
```

Contents

関数型言語

- ML 族
  - Standard ML of New Jersey (SML)
  - SML# (東北大). ML を「普通の言語に」
    - 他の言語の相互運用，データベース，スレッド，etc.
  - Caml Light, Objective Caml (OCaml)
  - F# (Microsoft. OCaml をもとに設計)
- Lisp, Scheme (動的型)
- Haskell, Miranda (「純粋」関数型．非正則)

Functional Languages

- ML family
  - Standard ML of New Jersey (SML)
  - SML# (Tohoku University). making ML "ordinary languages"
    - interoperability with other languages, databases, multithreading, etc.
  - Caml Light, Objective Caml (OCaml)
  - F# (Microsoft; based on OCaml)
- Lisp, Scheme (dynamically typed)
- Haskell, Miranda ("purely" functional. non-strict evaluation)

関数型言語，特にMLの特徴・ポイント

「関数型」なるもの:

- 型宣言なしで再構築
- 静的な型検査
- パラメトリックな多相型
- 第一級の関数
- データ構造の簡潔な定義，生成，パターンマッチ

$\Rightarrow$ 関数の「宣言的」な ($\approx$ 定義そのものに近い) 記述．それによる「間違いにくく，正しさを理解・証明しやすい」プログラム

Salient features of functional languages and esp. ML family

“Functional” things:

- type reconstruction without type declarations
- static type checking
- parametric polymorphism
- first-class functions
- concise definition, creation and pattern matching of data structures

$\Rightarrow$ “declarative” ($\approx$ similar to math) description of functions; programs less prone to errors and easier to reason/prove the correctness of

Contents

宣言的 vs. 手続き的

例: 以下の数列の第 a_n を計算する関数を書け

$$\begin{cases} a_0 = 1, \\ a_n = a_{n-1} + n \quad (n = 1, 2, \cdots) \end{cases}$$

宣言的: OCaml:

```
let rec a n =
  if n = 0 then 1
  else a (n - 1) + n
```

もちろん C でも:

```
int a(int n) {
  if (n == 0) return 1;
  else return a(n - 1) + n;
}
```

手続き的:

```
int a(int n) {
  int t = 1;
  for (int i = 0; i < n; i++) {
    t = t + (i + 1);
  }
  return t;
}
```

Declarative vs. procedural/imperative

Ex: write a function that computes a_n of the following series

$$\begin{cases} a_0 = 1, \\ a_n = a_{n-1} + n \quad (n = 1, 2, \cdots) \end{cases}$$

declarative: OCaml:

```
let rec a n =
  if n = 0 then 1
  else a (n - 1) + n
```

you can do it in C as well, to be sure:

```
int a(int n) {
  if (n == 0) return 1;
  else return a(n - 1) + n;
}
```

procedural:

```
int a(int n) {
  int t = 1;
  for (int i = 0; i < n; i++) {
    t = t + (i + 1);
  }
  return t;
}
```

Contents

型宣言なしで再構築 (Type Reconstruction)

- 入力を含め，プログラマが型を書かなくて良い

```
# let f x = x + 2;;
val f : int -> int = <fun>
```

- a `->` b は a を受け取り b を返す関数の型
- (大雑把な) 推論過程:
  1. `x + 2` から，(+は int をとるので)，`x` は int
  2. その場合 `x + 2` も整数だから，`f` は `int -> int`
- cf. C の場合

```
int f(int x) { return x + 2; }
```

Type reconstruction without type declaration

- the programmer does not have to declare types of variables, including those of input parameters

```
# let f x = x + 2;;
val f : int -> int = <fun>
```

- a `->` b represents a type of functions taking a and returning b
- a (rough) sketch of how it is inferred:
  1. look at `x + 2`; as `+` takes ints, `x` should be an int
  2. then `x + 2` should be an int, so `f` is `int -> int`
- cf. in C:

```
int f(int x) { return x + 2; }
```

用語: 型の再構築・型推論

- 型の再構築を「型推論 (type inference)」と呼ぶこともある
- ただし「型推論」はどんな言語でも行われている，式の型の導出を意味することもある

```
float f(float * a, int i) { return a[i]; }
```

  - `a` が `float*`, `i` が `int` ⇒ `a[i]` は `float`
- 言葉はさておき，ML の型推論 (型の再構築) は,「関数の入力の型」が型宣言なしで推論されるところが特徴

Terminology: type reconstruction/inference

- the above type reconstruction is commonly called *"type inference"*
- technically, however, it sometimes means a more ordinary process that derives types of compound expressions from component types

```
float f(float * a, int i) { return a[i]; }
```

  - `a` is `float*`, `i` is `int` $\Rightarrow$ `a[i]` is `float`

- regardless of terminologies, the salient feature of type inference (reconstruction) in ML is that it infers function's input types without type declaration

Contents

静的なエラー検査

- 型エラー，未定義の変数の利用を始めとした様々な間違いは「実行前に」検査されている (そこは C 言語と同じ)

```
# let f x = print_int x ; x + true ;;
Characters 28-32:
  let f x = print_int x ; x + true ;;
                              ^^^^
Error: This expression has type bool but an expression was expected of type
         int
```

- `f` が呼ばれてから, `x + true` を実行する瞬間にエラーになるのではない (cf. Python)

Static type checking

- various errors, such as type errors and use of undefined variables, are checked *prior to execution* (like C languages)

```
# let f x = x + true ;;
Characters 14-18:
  let f x = x + true ;;
                ^^^^
Error: This expression has type bool but an expression was expected of type
         int
```

- note that *the error is raised for the definition of* `f`*; not for the execution of* `x + true` *(cf. Python)*

どのような型があるか(1)

いくつかの「基本型」

- int (整数), float (浮動小数点数), bool (真偽), 文字列 (string), 文字 (char), unit

```
# 3;;
- : int = 3
# 3.8 ;;
- : float = 3.8
# true;;
- : bool = true
# "hello";;
- : string = "hello"
# 'h';;
- : char = 'h'
# () ;;
- : unit = ()
```

types (1)

a few primitive types

- **int** (integers), **float** (floating point numbers), **bool** (boolean), **string**, **char** (characters), **unit**

```
# 3;;
- : int = 3
# 3.8 ;;
- : float = 3.8
# true;;
- : bool = true
# "hello";;
- : string = "hello"
# 'h';;
- : char = 'h'
# () ;;
- : unit = ()
```

どのような型があるか (2)

関数型

- ```
  # let f x = x + 1;;   (* 関数の定義 *)
  val f : int -> int = <fun>
  # fun x -> x + 1 ;;   (* 無名関数 *)
  - : int -> int = <fun>
  ```
- 注: +, -, *, /は整数のみ受け取る.
- 浮動小数点数は. をつける (+., -., *., /.)

```
# fun x -> x +. 2.3;;
- : float -> float = <fun>
# fun x -> x + 2.3;;
Characters 13-16:
  fun x -> x + 2.3;;
               ^^^
Error: This expression has type float but an expression was expected of
         type
         int
```
```

types (2)

function types

- ```
  # let f x = x + 1;;   (* function definition *)
  val f : int -> int = <fun>
  # fun x -> x + 1 ;;   (* anonymous function *)
  - : int -> int = <fun>
  ```
- remark: `+`, `-`, `*`, `/` takes only integers
- operators on floating point numbers have `.` (`+.`, `-.`, `*.`, `/.`)

```
# fun x -> x +. 2.3;;
- : float -> float = <fun>
# fun x -> x + 2.3;;
Characters 13-16:
  fun x -> x + 2.3;;
               ^^^
Error: This expression has type float but an expression was expected of
         type
         int
```
```

どのような型があるか(3)

- * (タプル; 複数の値の組)

```
# 3,4.5,"hello" ;;
- : int * float * string = (3, 4.5, "hello")
```

- list (リスト)

```
# [ 1; 2; 3 ];;
- : int list = [1; 2; 3]
# [ "this"; "is"; "a"; "pen" ];;
- : string list = ["this"; "is"; "a"; "pen"]
```

注: リストの要素は ; で区切る. 以下はやりがちな間違い

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

types (3)

- * (tuple; combination of multiple values)

```
# 3,4.5,"hello" ;;
- : int * float * string = (3, 4.5, "hello")
```

- list (list)

```
# [ 1; 2; 3 ];;
- : int list = [1; 2; 3]
# [ "this"; "is"; "a"; "pen" ];;
- : string list = ["this"; "is"; "a"; "pen"]
```

Remark: delimit elements by ';' the following is a common mistake (still type correct).

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

型エラー (型付けできない式) の例

- 関数と引数の型の不一致

```
let f x = x + 1
in f "hello";;
```

- then と else で型が違う if 文

```
if 1 < 2 then 10 else "hello"
```

- 違う型の値が混ざったリスト

```
[ 10 ; "hello" ]
```

Expressions that are not typed (type errors)

- a function application whose function and argument has incompatible types

```
let f x = x + 1
in f "hello";;
```

- an `if` experssion whose `then` and `else` has incompatible types

```
if 1 < 2 then 10 else "hello"
```

- a list expression that mixes incompatible types

```
[ 10 ; "hello" ]
```

Contents

多相関数

- 「色々な値に対して適用できる関数 (多相的な関数; polymorphic function)」を表現する型がある
- しかも関数の定義から，関数が多相型を持つことを推論する

```
# let f x = x;;
val f : 'a -> 'a = <fun>
# let rec len lst = match lst with [] -> 0 | h::r -> 1 + len r ;;
val len : 'a list -> int = <fun>
```

- 'a は「型パラメータ」と呼ぶ
- 直感的には「すべての値」ということ

Polymorphic functions

- there is a type that represents functions that "apppy to various (any) types" (polymorphic functions)
- such types are also automatically reconstructed from function definitions

```
# let f x = x;;
val f : 'a -> 'a = <fun>
# let rec len lst = match lst with [] -> 0 | h::r -> 1 + len r ;;
val len : 'a list -> int = <fun>
```

- `'a` is called a *"type parameter"*
- intuitively, it represents a type of "any value"

よく使うリスト関係の多相関数

- `map` $f\ l$: l の各要素に f を適用したリスト

```
# List.map ;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

- `filter` $p\ l$: l 中で p を満たす要素だけのリスト

```
# List.filter ;;
- : ('a -> bool) -> 'a list -> 'a list = <fun>
```

Frequently used polymorphic functions on lists

- `map` f l : apply f to each element of l returns the list of them

```
# List.map ;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

- `filter` p l : returns the list of elements in l satisfying p

```
# List.filter ;;
- : ('a -> bool) -> 'a list -> 'a list = <fun>
```

- see `List` module for more functions

Contents

第一級の関数

- 関数型言語では,「関数」と「その他の値」を区別せず, 同様に扱う
  - 関数を他の関数に渡せる
  - 関数を関数の返り値にできる
  - 関数をデータ中に入れられる
- もっともこれだけでは C の関数ポインタでも同じ. 本質的な違いは,
  - どこでも関数が定義できる (生成できる)

First class functions

- functional languages generally treat functions and other values similarly without distinctions; namely, a function:
  - can be passed to another function,
  - can be returned from a function,
  - can be put in a data structure, etc.
- it is actually the case in C. the essential difference is
  - you can define (create) a function *anywhere*

どこでも関数が作れる

- ＝トップレベルだけでなく，関数の中でまた関数を定義できる

```
let f x =
  let g y = ... (* 関数の中でまた関数を定義 *)
  in ... ;;
```

- 例

```
# let make_adder x =  (* x 足す関数を返す *)
    let g y = y + x
    in g ;;
val make_adder : int -> int -> int = <fun>
# let a3 = make_adder 3 ;; (* 3を足す関数 *)
val a3 : int -> int = <fun>
# let a4 = make_adder 4 ;; (* 4を足す関数 *)
val a4 : int -> int = <fun>
# a3 10 ;;
- : int = 13
# a4 10 ;;
- : int = 14
```

Functions can be defined anywhere

- = you can define a function not only in the toplevel but also within a function

```
let f x =
  let g y = ... (* define a function in another function *)
  in ... ;;
```

- ex.

```
# let make_adder x =  (* create a function that adds x *)
    let g y = y + x
    in g ;;
val make_adder : int -> int -> int = <fun>
# let a3 = make_adder 3 ;; (* a function that adds 3 *)
val a3 : int -> int = <fun>
# let a4 = make_adder 4 ;; (* a function that adds 4 *)
val a4 : int -> int = <fun>
# a3 10 ;;
- : int = 13
# a4 10 ;;
- : int = 14
```

どこでも関数が作れる

- あえて C 言語風に書けば以下のようなことができるということ

```
int (*)(int) make_adder(int x) {
  int g(int y) { return y + x; } /* 関数の中で関数定義 */
  return g;
}
```

- どこでも関数定義が「書ける」だけなら文法をそうすればいいだけのこと
- 真に大事なのは，中の関数 (g) が，外側の変数 (自由変数; x) を参照でき，しかも,「自分が作られた時の値」をいつまでも覚えているということ
- 関数 = 文面 + 関数定義時の自由変数の値 (closure)

Functions can be defined anywhere

- in a hypothetical C-like syntax, it is like you can write the following

```
int (*)(int) make_adder(int x) {
  int g(int y) { return y + x; } /* a function definition in a function */
  return g;
}
```

- just allowing you to “write” this function is merely a matter of extending the syntax
- what really matters is that the inner function (`g`) can *reference a variable outside of it* (*free variable*; `x`) and it remembers its value when it is defined
- a function = program text + values of free variables at the time of its definition (*closure*)

注: 関数の引数は「常に」ひとつ

- ```
  let f x y = E
  ```

  は,

  ```
  let f = (fun x -> (fun y -> E))
  ```

  の略記に過ぎない (「カリー化された関数」). E の中で `x` が参照できることの重要性に注意.

- 実際, `f x` のように一つだけ引数を与えても良い (部分適用)

  ```
  # let f x y = x + y;;
  val f : int -> int -> int = <fun>
  # f 3 ;;
  - : int -> int = <fun>
  ```
  ```

Note: a function's arity is *always* one

- what looks like a function taking two-parameters (`x` and `y`):

```
let f x y = E
```

  actually takes a parameter (`x`) and returns a function that takes a parametr (`y`), as if:

```
let f = (fun x -> (fun y -> E))
```

- note that this is legitimate only when E can reference `x` (free variable)
- as a matter of fact `f x` is a valid application (partial application)

```
# let f x y = x + y;;
val f : int -> int -> int = <fun>
# f 3 ;;
- : int -> int = <fun>
```

- the concept is called *currying*

演算子も実は関数

- 組み込み演算 ($a+b$, $a<b$ など) も，実は関数適用
- 演算記号 (+, <など) をカッコでくくると，演算子の「関数」を取り出せる

```
# (+);;
- : int -> int -> int = <fun>
# (=);;
- : 'a -> 'a -> bool = <fun>
# (<);;
- : 'a -> 'a -> bool = <fun>
# (<) 3 5 ;;  (* 3 < 5 と同じ意味 *)
- : bool = true
```

- それらももちろん部分適用できる関数．例えば，(<) 3 は，$3 < y$ を判定する「関数」

```
# List.filter ((<) 3) [ 1; 2; 3; 4; 5 ];;
- : int list = [4; 5]
```

Operators are in fact functions

- builtin operations ($a+b$, $a<b$ etc.) are in fact function applications
- put an operator symbol (`+`, `<`, etc.) inside parens and you get the function corresponding to the operator

```
# (+);;
- : int -> int -> int = <fun>
# (=);;
- : 'a -> 'a -> bool = <fun>
# (<);;
- : 'a -> 'a -> bool = <fun>
# (<) 3 5 ;;  (* means 3 < 5 *)
- : bool = true
```

- they are functions so are partially applicable. e.g., `(<) 3` is a function that judges if $3 < y$

```
# List.filter ((<) 3) [ 1; 2; 3; 4; 5 ];;
- : int list = [4; 5]
```

Contents

データ構造の簡潔な定義 (バリアント)

- C の struct 相当

```
# type student = Student of (int * string * float);;
type student = Student of (int * string * float)
# Student(31489678, "Masakazu Mimura", 169.8);;
- : student = Student (31489678, "Masakazu Mimura", 169.8)
```

- C の union 相当 (型は一つ．その作り方が複数)

```
# type staff = Student of (int * string * float)
            | Teacher of (string * float) ;;
```

- C の enum 相当 (上記の特別な場合)

```
# type color = Blue | Red | Orange | Black | ... ;;
```

A concise definition of data structure (variant)

A single `type` construct serves what are distinct constructs in C

- like `struct` in C

```
# type student = Student of (int * string * float);;
type student = Student of (int * string * float)
# Student(31489678, "Masakazu Mimura", 169.8);;
- : student = Student (31489678, "Masakazu Mimura", 169.8)
```

- like `union` in C (multiple constituent types for a single type)

```
# type staff = Student of (int * string * float)
           | Teacher of (string * float) ;;
```

- like `enum` in C (special case of the above)

```
# type color = Blue | Red | Orange | Black | ... ;;
```

安全なパターンマッチ

- Union のような取り違えの起き得ない文法 (場合分けとフィールドの取り出しを「セットで」行う文法)

```
# let height x = match x with
    Student(id, name, h) -> h
  | Teacher(name, h) -> h;;
val height : staff -> float = <fun>
```

- パターンの書き忘れも警告してくれる

```
# let height x = match x with Student(id, name, h) -> h ;;
Characters 15-53:
  let height x = match x with Student(id, name, h) -> h ;;
                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Warning 8:  this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Teacher _
val height : staff -> float = <fun>
```

Safe pattern matches

- a syntax that safely separates cases and extracts component fields (you'll never access wrong fields)

```
# let height x = match x with
    Student(id, name, h) -> h
  | Teacher(name, h) -> h;;
val height : staff -> float = <fun>
```

- a warning against missing cases

```
# let height x = match x with Student(id, name, h) -> h ;;
Characters 15-53:
  let height x = match x with Student(id, name, h) -> h ;;
                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Warning 8:  this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Teacher _
val height : staff -> float = <fun>
```

match の略記

- ```
  let f x = match x with Foo(y) -> ...
  ```

  で，match の選択肢がひとつしかない (|がない) 場合，上記を

  ```
  let f (Foo(y)) = ...
  ```

  と書ける

- タプルのパターンマッチに使うととりわけ自然

  ```
  let f (x,y) = x + y ;;
  ```

  実は以下の略記

  ```
  let f t = match t with (x,y) -> x + y ;;
  ```
  ```

Abbreviations of match expressions

- if there is only a single case for a match expression, like:

```
let f x = match x with Foo(y) -> ...
```

you can write the above as follows

```
let f (Foo(y)) = ...
```

- particularly convenient/natural for pattern-matching tuples.

```
let f (x,y) = x + y ;;
```

is an abbreviation of:

```
let f t = match t with (x,y) -> x + y ;;
```

型定義も多相的にできる

- 自分で明示的に型パラメータ ('a) を書く

```
# type 'a tree = Empty
               | Node of ('a * 'a tree * 'a tree) ;;
type 'a tree = Empty | Node of ('a * 'a tree * 'a tree)
```

- 組み込みのリストは，以下のような型だと理解すれば良い (以下は実際には文法エラー)

```
# type 'a list = [] | ::  of ('a * 'a list) ;;
# 3 :: 4 :: 5 :: [] ;;
```

- 他によく使う型 option は以下のようなもの

```
# type 'a option = None | Some of 'a ;;
```

```
# None ;;
- : 'a option = None
# Some 5 ;;
- : int option = Some 5
```

Types can be made polymorphic

- specify type parameters (`'a`)

```
# type 'a tree = Empty
                | Node of ('a * 'a tree * 'a tree) ;;
type 'a tree = Empty | Node of ('a * 'a tree * 'a tree)
```

- the builtin list type can be understood as if it were defined follows (note: it is not syntactically valid)

```
# type 'a list = [] | ::  of ('a * 'a list) ;;
# 3 :: 4 :: 5 :: [] ;;
```

- a frequently used type `option` is defined as:

```
# type 'a option = None | Some of 'a ;;
```

```
# None ;;
- : 'a option = None
# Some 5 ;;
- : int option = Some 5
```

クイックソート

以上をまとめて，クイックソートがこれだけで書けることは嬉しいと思うのだがどうでしょうか?

```
let rec qs l =
  match l with
     [] -> []
    | h::r ->
       let smaller = List.filter ((>) h)  r in
       let larger  = List.filter ((<=) h) r in
       (qs smaller) @ h :: (qs larger) ;;
```

- 正しいことが理解・証明しやすい (帰納法)
- 変数の間違いや型エラーをはじいてくれるので，型検査を通れば結構な確率であっさりと動く
- どんなリストにでも動作する
- ただし実は問題あり（スタックオーバーフロー．また後日）

Quicksort

Isn't it nice to be able to write a quicksort as concisely as this?

```
let rec qs l =
  match l with
     [] -> []
    | h::r ->
       let smaller = List.filter ((>) h)  r in
       let larger  = List.filter ((<=) h) r in
       (qs smaller) @ (h :: (qs larger)) ;;
```

- easy to reason about its correctness (induction)
- typos and type errors are checked at compile time, so once type-checked, it tends to work in the first run
- it can work on any list type
- disclosure: this version has an issue with large lists (stack overflow; I'll get to it later)

Contents

関数型言語の理想

- OCaml (関数型言語) の利点 (理想) は，再帰呼出しを使って物事を，簡潔に ($\approx$ 定義そのものみたいに) 書けること
- 例: リストの長さ

```
let rec len lst =
match lst with
  []    -> 0               (* 空リストの長さは 0 *)
| h::r -> 1 + (len r)  (* [h;...] の長さは ...の長さ+1 *)
```

- 例: $[a, b)$ に含まれる整数の和

```
let rec sum a b =
if a >= b then
   0                      (* 空区間の和は 0 *)
else
   a + (sum (a+1) b)   (* [a,b)の和は a + [a+1,b)の和 *)
```

Functional languages: the ideal and the reality

- The advantage ("ideal") of OCaml (functional languages) is to be able to write things concisely ($\approx$ as if you are merely writing definitions), which mainly stems from *recursions*
- ex: length of a list

```
let rec len lst =
match lst with
  []   -> 0              (* length of an empty list is 0 *)
| h::r -> 1 + (len r)  (* length of [h;...] is (length of ...) + 1 *)
```

- ex: sum of integers in $[a, b)$

```
let rec sum a b =
if a >= b then
   0                     (* sum of empty interval: 0 *)
else
   a + (sum (a+1) b)   (* sum of [a,b) is a + (sum of [a+1,b)) *)
```

関数型言語であまり使わないもの

関数型言語の「理想」においては，あまり使わなくていいはずのもの

- ループ
- 変数の更新
- データ構造の更新

Not often used in functional languages

Things you should not have to use often in the “ideal” functional programming

- loops
- (destructive) updates of variables
- (destructive) updates of data structures

ループ

- for， while 文はある

```
for i = a to b do
  Printf.printf "hello %d\n" i done ;;
```

- しかし再帰を使えばあまり「ループ」したいと思う必要はなかった
  - sum， range， qs， bs_tree， ...
- なお一般には， for と同じことが以下のようにしてできる

```
for i = a to b do
  E
done
```

=

```
let rec loop i =
  if i <= b then
    E ; loop (i + 1)
in loop a
```

loops

- OCaml does have `for` and `while` expressions, to be sure

```
for i = a to b do
  Printf.printf "hello %d\n" i done ;;
```

- but remember, you already used recursions where you might have used loops in other languages. did you look for loops for them?
  - `sum`, `range`, `qs`, `bs_tree`, ...
- in general, `for` loops in OCaml can be done with recursions as follows:

```
for i = a to b do
  E
done
```

=

```
let rec loop i =
  if i <= b then
    E ; loop (i + 1)
in loop a
```

変数の更新

- `let x = ...` で定義した変数はあとから「更新」できない (immutable).
- 例えば以下は NG

```
let s = 0 in
for i = a to b do
  s に s + i を代入
done ;;
```

- for 文の有用性がそれほどない理由でもある
- 更新できる (mutable な) 変数もあります (後日)
- が，ループ同様,「再帰を使えばいらない」ことが多い

Updating variables

- variables defined by `let x = ...` *cannot* be updated later (they are *immutable*).
- in particular, you cannot do the following to define `sum`

```
let sum* a b =
  let s = 0 in
  for i = a to b do
    update s with s + i
  done ; s
```

- one reason why `for` expression is not that useful
- there are updatable (mutable) variables, to be sure (later)
- you don't want to use it often once you master recursions

Immutable な変数とつきあう

- $[a, b)$ の和を求める「ループ」

```
let s = 0 in
for i = a to b do
  s に s + i を代入
done ;;
```

- ...を再帰に直したもの

```
let rec loop i s =
  if i <= b then
     s
  else
     loop (i + 1) (s + i)
in loop a 0 ;;
```

Working with immutable variables

- a "loop" that sums up integers in $[a, b)$

```
let s = 0 in
for i = a to b do
  update s with s + i
done ;;
```

- ... transformed into a recursion:

```
let rec loop i s =
  if i <= b then
     s
  else
     loop (i + 1) (s + i)
in loop a 0 ;;
```

Immutable な変数とつきあう

- 変数を更新しながら進行するループは多くの場合，ループ内で変更される関数を引数にとるような，再帰関数に形式的に書き換えられる

```
x = x0; y = y0;
while (C) {
  E;
  x = A;
  y = B;
}
return R
```

$\approx$

```
let rec loop x y =
  if not C then R else
  (E ; loop A B)
in loop x0 y0 ;;
```

- これを引き出しに入れておくのは良いが，そもそも最初から「再帰」で考えれば，そんな「技法」が必要だとすら感じない場合が多い
  - 「a から b までの和」$\Rightarrow a+$ 「$(a+1)$ から b までの和」

Working with immutable variables

- a loop that updates variables can often be transformed into a recursive function that takes these variables as parameters

```
x = x0; y = y0;
while (C) {
  E;
  x = A;
  y = B;
}
return R
```

$\approx$

```
let rec loop x y =
  if not C then R else
  (E ; loop A B)
in loop x0 y0 ;;
```

- you may remember this as a formula, but more important is you do not have to master such a technique, if you think with recursions in the first place. e.g.,
  - sum of $[a, b) \Rightarrow a +$ sum of $[a + 1, b)$

データ構造の更新

- これまで紹介したデータ構造 (リスト, バリアント) も immutable = 破壊的更新 (destructive update); その場の更新 (in-place update) 不可能
- 例えばリストに要素を「追加」することもできない
- 以下のような感じで，`range a b` 関数を書くのは無理

```
let l = [] in
  for i = a to b do
     add i to l
  done ; l
```

- cf. C++の vector

```
vector<int> l;
for (i = a; i < b; i++) {
  l.push_back(i);
}
```

Updating data structures

- data structures we have seen so far (list, tuple, variants) are immutable too = "destructive" or "in-place" updates are not allowed
- you cannot "add" an element to a list; you instead create another list that has an additional element (with `::`)
- you cannot write `range a b` along:

```
let range* a b =
  let l = [] in
    for i = a to b do
       add i to l
    done ; l
```

- cf. with C++ vector:

```
vector<int> l;
for (i = a; i < b; i++) {
  l.push_back(i);
}
```

破壊的更新なしでのプログラミング

- すでにあるデータを変更するではなく，得たいデータを「作る」という考え方が基本
  - a から b までのリスト $=$ a :: (($a+1$) から b までのリスト)
  - 空リストを作り，それに要素を「追加している」と考える必要がない
- データを少しずつ「変更」(例: 木構造へデータを挿入) したい場合，データを文字通り変更する代わりに「変更後のデータを新たに作る」のが基本 (関数的更新; functional update)

Working with immutable data structures

- think how to "express" the data you want with recursions, not how to grow it from empty
  - list of $[a, b) = a :: (\text{list of } [a+1, b))$
  - you do not think: "create an empty list first; then add $[a, b)$ to it"
- when you slightly "modify" an existing data structure (e.g., insert an element to a tree), you create a data structure after the operation *(functional update)*

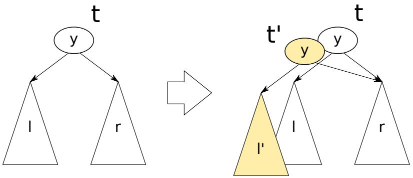
関数的更新: 2 分木の例

- データ構造

```
type 'a bs_tree =
  Empty | Node of ('a * 'a bs_tree * 'a bs_tree) ;;
```

- 挿入する関数は「挿入後の木を作って返す」関数

```
let rec bs_tree_insert x t =
  match t with
    Empty -> Node (x, Empty, Empty)
  | Node (y, l, r) ->
    if x < y then
      Node (y, bs_tree_insert x l, r)
    else
      Node (y, l, bs_tree_insert x r);;
```



- 以下の bs_tree_insert x t 計算後も t はあくまで元の木

```
let t' = bs_tree_insert x t
```

- 一部分 (上記の青下線部分) は再利用されており，毎回全てが作り直されるわけではないことに注意

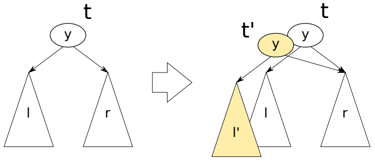
Functional updates: a binary search tree

- data definition

```
type 'a bs_tree =
  Empty | Node of ('a * 'a bs_tree * 'a bs_tree) ;;
```

- a function that “inserts” an element creates another tree

```
let rec bs_tree_insert x t =
  match t with
    Empty -> Node (x, Empty, Empty)
  | Node (y, l, r) ->
    if x < y then
      Node (y, bs_tree_insert x l, r)
    else
      Node (y, l, bs_tree_insert x r)
```



- the variable `t` below still refers to the original tree after `bs_tree_insert x t`

```
let t' = bs_tree_insert x t
```

- note that a part of data structure (<u>the underlined with blue</u> of the above) is reused; the whole data structure is not rebuilt every time

Immutableのどこがいい? (1)

- 何もかもがimmutable ⇒ 同じスコープで同じ式は「同じもの」(参照透明性; referential transparency)
- 人間が紙の上で計算をする時もそのようにしているはず
- コンピュータを習いたての頃，`x = x + 1` が意味不明だったという人には，嬉しい性質のはず
- `x = x + 1` の意味をあえて「等式」で理解すると,「次の瞬間の `x`」と「今の `x`」`+ 1` が等しいということ
- 参照透明性があれば「さっきの `x`」「今の `x`」「明日の `x`」などという区別は不要

What are *good* about immutability? (1)

- everything is immutable ⇒ the same variable (the same expression in the same scope) always refers to the *"same thing (value)" (referential transparency)*
- you do math calculations on papers this way
- if you were confused by a statement like: `x = x + 1` when you first learned a procedural language, you will like it
- to understand the meaning of `x = x + 1` as an "equation", it is: "`x` at the next moment" equals to "`x` at the current moment" `+ 1`
- with referential transparency, there are no such things as "`x` as of yesterday", "`x` as of today" and "`x` as of tomorrow"

Immutableのどこがいい? (2)

参照透明性は，ある種の最適化を簡単にする．

- たとえば共通部分式は「無条件」で最適化できる．

```
f x + f x = 2 * (f x)
```

  - cf. C では，(f x) に「副作用 (データの更新や I/O)」が含まれていないかを気にする必要がある

- 自動的なメモ化 (入力 → 出力の結果を覚えておく)

```
let rec fib n =
  if n < 2 then
    1
  else
    fib (n - 1) + fib (n - 2)
```

- 数学的な式変形の多くが，プログラム上でも正しくなる

What are *good* about immutability? (2)

it makes certain optimizatins straightforward

- e.g., redundant expressions can be "unconditionally" optimized away

```
f x + f x = 2 * (f x)
```

  - cf. in C, you need to know `(f x)` does not have a "side effect (data structure updates or I/O)"

- automatic memoization (record and reuse "input $\rightarrow$ output") relationships)

```
let rec fib n =
  if n < 2 then
    1
  else
    fib (n - 1) + fib (n - 2)
```

- $\approx$ mathematical transformations become valid for programs too

Immutable のどこがいい? (3)

データの更新がないと，並列化可能性 (= 並列実行によって計算結果が変わらないこと) の判定が自明になる

- 以下の (f x) と (g x) は並列に実行可能，(f x) と (h y) はそうではない，etc.

```
let y = f x in
let z = g x in
let w = h y in
  y + z + w
```

- C 言語などでこの判定が困難な理由：(f x) の中で，(g x) に影響のある計算が行われないことを判定するのが困難

What are *good* about immutability? (3)

immutability makes judging *parallelizability* (= parallel execution has an equivalent result to the sequential execution) straightforward

- e.g., `(f x)` and `(g x)` below can be executed in parallel; `(f x)` and `(h y)` cannot (`(h y)` requires the value of `(f x)`), etc.

```
let y = f x in
let z = g x in
let w = h y in
  y + z + w
```

- in languages supporting mutable data structures (e.g., C), it is necessary (and difficult) to judge if execution of `(f x)` never affects `(g x)` (e.g., `(f x)` never updates data structure used by `(g x)`)

Immutableのどこが悪い? (1)

- 不自然なこともある．問題によってはまさしく「同じ変数や式でもその値が変わること (内部状態の変化)」が表現したいことである (世の中は常に変化している)
  - 現在時刻
  - 銀行残高
  - 身長，体重
  - etc.
- 「同じ式は同じ値になる」を守るには，日々刻々変化するものはことごとく,「異なる式」にならなくてはならない

```
let t0,s1 = gettimeofday s0 in
let t1,s2 = gettimeofday s1 in
  ...
```

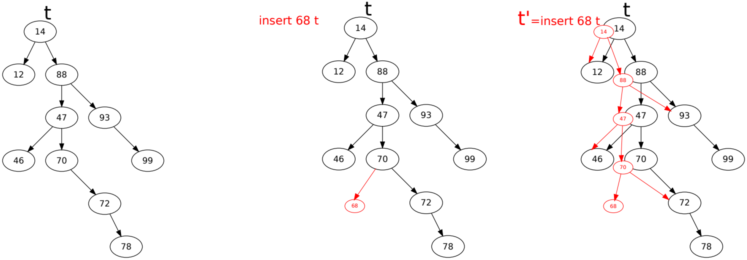
What are *bad* about immutability? (1)

- sometimes it is just unnatural/awkward; some problems exactly want to have the same variable/expression have varying values over time (the world is changing)
  - current time
  - the balance in bank accounts
  - height, weight, etc. of a human
  - etc.
- to maintain referential transparency, different values must be expressed by different expressions

```
let t0,s1 = gettimeofday s0 in
let t1,s2 = gettimeofday s1 in
  ...
```

Immutable のどこが悪い? (2)

- 効率が悪い:「データを少し書き換える」だけの操作が，結構な量のメモリ割り当てになりがち

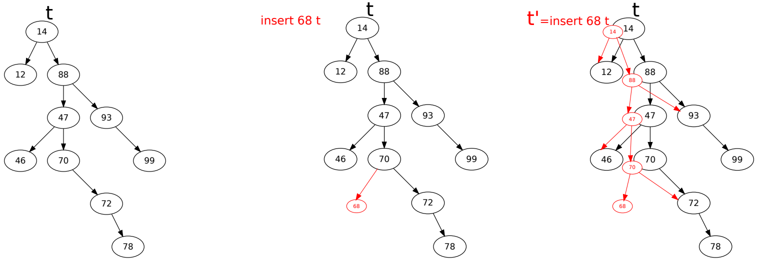


元の木 ⇒ 破壊的更新 関数的更新

- → メモリの割り当て・解放を激しく行う．その性能に強く依存する

What are *bad* about immutability? (2)

- it is inefficient; a "slight modification to data" tends to allocate a fair amount of memory



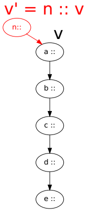
the original tree ⇒ in-place update functional update

- → performs intensive memory allocation/deallocation and relies on their performance

関数的更新：様々なケース

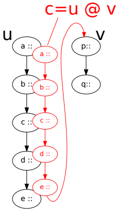
リストの先頭に追加

(`let v’ = n::v`)



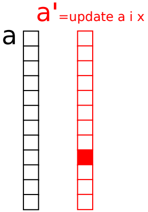
リストの連結

(`let c = u @ v`)



配列の一要素更新 (どうしてもというなら...)

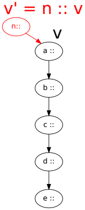
```
let a’ = update a i x
```



Functional updates: vaious cases

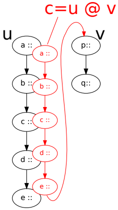
add an element to the head

(`let v' = n::v`)



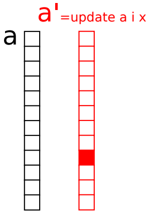
concatenate two lists

(`let c = u @ v`)



update an element of an array (if you insist ...)

`let a' = update a i x`



関数的更新：様々なケースのコスト

- 各セルは,「読み出しコスト/書き込み (+メモリ割り当て) コスト」
- d： 木 (t) の深さ
- n： 配列の大きさ

		破壊的	関数的
木へ挿入	`insert x t`	$O(d)/O(1)$	$O(d)/O(d)$
リストの先頭に追加	`n::v`	$O(1)/O(1)$	$O(1)/O(1)$
リストの連結	`u@v`	$O(\|u\|)/O(1)$	$O(\|u\|)/O(\|u\|)$
配列の要素更新	[i]	$O(1)/O(1)$	$O(n)/O(n)$

Functional updates: cost

- each cell represents the cost of read / write (+ allocation)
- d : depth of a tree (t)
- n : size of an array

		in-place	functional
insert to tree	`insert x t`	$O(d)/O(1)$	$O(d)/O(d)$
add to the head of list	`n::v`	$O(1)/O(1)$	$O(1)/O(1)$
concat two lists	`u@v`	$O(\|u\|)/O(1)$	$O(\|u\|)/O(\|u\|)$
update an array element	`[`i`]`	$O(1)/O(1)$	$O(n)/O(n)$

Immutableのどこが悪い? (3)

- 並列性の判定をしやすいからといって，役に立つ並列性を抽出しやすいとは限らない
- 例：木構造への並列挿入

```
let t1 = insert x0 t0 in
let t2 = insert x1 t1 in
let t3 = insert x2 t2 in
  ...
```

  - 単純には全部逐次にやるしかない ($t0 \to t1 \to t2 \to \cdots$ という依存関係)
  - 直接更新する場合，木の異なる部分を同時に更新して良い

What are *bad* about immutability? (3)

- being able to judge parallelizability easily does not mean being able to parallelize easily
- e.g., insert many elements to tree

```
let t1 = insert x0 t0 in
let t2 = insert x1 t1 in
let t3 = insert x2 t2 in
  ...
```

  - dependencies: $t0 \to t1 \to t2 \to \cdots$ precludes parallelizing these insertions
  - insertions to different parts of a tree could actually be done in parallel

というわけで...

Imperative

はじめました

- ごく一部の「純粋」関数型言語 (Haskell, Miranda など) を除くと，状態の更新をともなう操作を許している
  - array：配列．更新可能．
  - ref：≈ 一要素の配列．後から変更できる変数の代わり
  - record：フィールドに「更新可能 (mutable)」という属性を指定可能
- 「参照の透明性」よりも，型推論，静的型検査，パターンマッチなどの，実利を目指す

After all . . .

Imperative

はじめました

- functional languages except few "purely functional" languages (Haskell, Miranda, etc.) allow updating states
  - arrays are mutable
  - `ref` : ≈ a singleton array. can be used as a mutable variable
  - record : you can label a field with `mutable` attribute
- abandon "referential transparency" but retain practical benefits from declarative programming, type inferance, static type checking, pattern matching, etc.

もうひとつの冷める現実

- 大きな区間の和を求めてみると...

```
# sum 0 (1000 * 100);;
- : int = 4999950000
# sum 0 (1000 * 1000);;
Stack overflow during evaluation (looping recursion?).
```

- 理由はなんとなく想像できるでしょう

```
let rec sum a b =
if a >= b then
   0
else
   a + (sum (a+1) b)
```

sum	0 1000000

もうひとつの冷める現実

- 大きな区間の和を求めてみると...

```
# sum 0 (1000 * 100);;
- : int = 4999950000
# sum 0 (1000 * 1000);;
Stack overflow during evaluation (looping recursion?).
```

- 理由はなんとなく想像できるでしょう

```
let rec sum a b =
if a >= b then
   0
else
   a + (sum (a+1) b)
```

sum	1 1000000
sum	0 1000000

もうひとつの冷める現実

- 大きな区間の和を求めてみると...

```
# sum 0 (1000 * 100);;
- : int = 4999950000
# sum 0 (1000 * 1000);;
Stack overflow during evaluation (looping recursion?).
```

- 理由はなんとなく想像できるでしょう

```
let rec sum a b =
if a >= b then
   0
else
   a + (sum (a+1) b)
```

sum	2 1000000
sum	1 1000000
sum	0 1000000

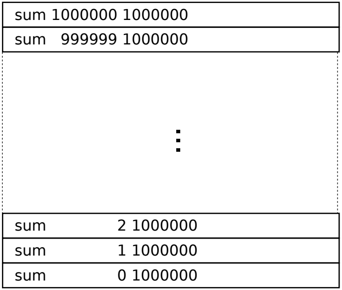
もうひとつの冷める現実

- 大きな区間の和を求めてみると...

```
# sum 0 (1000 * 100);;
- : int = 4999950000
# sum 0 (1000 * 1000);;
Stack overflow during evaluation (looping recursion?).
```

- 理由はなんとなく想像できるでしょう

```
let rec sum a b =
if a >= b then
   0
else
   a + (sum (a+1) b)
```



One more “inconvenient” truth

- let’s find the sum of a very large interval …

```
# sum 0 (1000 * 100);;
- : int = 4999950000
# sum 0 (1000 * 1000);;
Stack overflow during evaluation (looping recursion?).
```

- you can imagine why it happens …

```
let rec sum a b =
if a >= b then
   0
else
   a + (sum (a+1) b)
```

sum	0 1000000

One more “inconvenient” truth

- let’s find the sum of a very large interval . . .

```
# sum 0 (1000 * 100);;
- : int = 4999950000
# sum 0 (1000 * 1000);;
Stack overflow during evaluation (looping recursion?).
```

- you can imagine why it happens . . .

```
let rec sum a b =
if a >= b then
   0
else
   a + (sum (a+1) b)
```

sum	1 1000000
sum	0 1000000

One more "inconvenient" truth

- let's find the sum of a very large interval ...

```
# sum 0 (1000 * 100);;
- : int = 4999950000
# sum 0 (1000 * 1000);;
Stack overflow during evaluation (looping recursion?).
```

- you can imagine why it happens ...

```
let rec sum a b =
if a >= b then
   0
else
   a + (sum (a+1) b)
```

sum	2 1000000
sum	1 1000000
sum	0 1000000

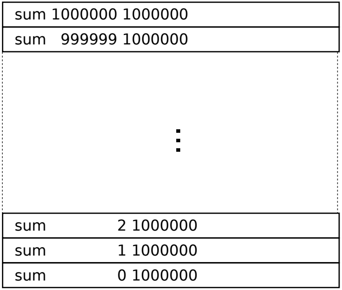
One more "inconvenient" truth

- let's find the sum of a very large interval …

```
# sum 0 (1000 * 100);;
- : int = 4999950000
# sum 0 (1000 * 1000);;
Stack overflow during evaluation (looping recursion?).
```

- you can imagine why it happens …

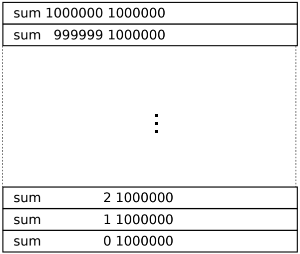
```
let rec sum a b =
if a >= b then
   0
else
   a + (sum (a+1) b)
```



スタックオーバーフロー

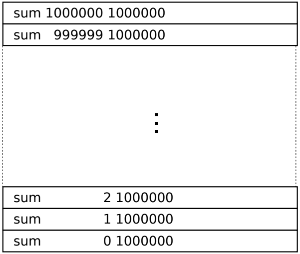
- 関数呼び出しをしようとすると，その関数実行の (変数の値とかを保持する) ための領域が必要になる
- その領域は通常「スタック」と呼ばれる固定サイズの領域から割り当てられている
- → 深い再帰はスタックオーバーフロー
- しかし，何でもかんでも再帰で書く関数型言語にとっては致命的
- C で `sum(a, b)` を再帰で書く人は (あまり) いない

```
int sum(int a, int b) {
  int s = 0;
  for (i = a; i < b; i++) s += i;
  return s; }
```



Stack overflow

- a functionl call requires a space (for holding values of variables or expressions)
- the space is obtained from a region called "stack", which is normally cannot grow very large (even if memory is otherwise abundant)
- → deep recursions overflow the stack
- critical for functional languages, which encourage recursions
- less critical for C, in which programmers are taught to write something like `sum(a, b)` with loops anyways




```
int sum(int a, int b) {
  int s = 0;
  for (i = a; i < b; i++)
    s += i;
  return s; }
```

スタックオーバーフローに関する色々な言語・実装のスタンス

- スタックオーバーフローは本来,「不可避」というような問題ではない
- 多くの言語の実装で，関数呼び出しのためのメモリを特別な固定長の領域 (スタック) から取るために生ずる,「実装上の」問題
  - ほとんどの言語: 深い再帰をやる君がいけない．ループを使え
  - 一部の関数言語実装 (例: SML/NJ): 関数呼び出しのメモリをヒープからとり，メモリが溢れない限りいくらでも深い呼び出しができる.「スタック」オーバーフローと無縁
  - OCaml: 関数型だが，スタックオーバーフローする．といってループは不便．さてどうする?

Positions of various languages/implementations against stack overflows

- the problem is not inevitable
- it is a ramification of an implementation (memory management) strategy, which obtains memory for function calls from a contiguous space (of a fixed size) dedicated for stack
  - most languages: say "avoid deep recursions; use loops instead"
  - a few functional languages (e.g., SML/NJ): abandon stacks altogether; get memory for function calls from heap. there is no such thing as stack overflow. "overflow" occurs only when you run out of the heap ($\approx$ whole memory)
  - OCaml: it is a functional language, yet prone to stack overflow (somewhat irresponsible implementation).
  - what should OCaml programmers do?

末尾呼び出しという「特別な」呼び出し

- OCaml でのプログラム上の工夫として,「末尾呼び出し」という特別な関数呼び出しがある
- 試しに sum 関数の `a +` をなくしてみる

```
let rec sum a b =
if a >= b then
   0
else
   a + (sum (a+1) b)
```

→

```
let rec sum’ a b =
if a >= b then
   0
else
   sum’ (a+1) b
```

- するとどうでしょう

```
# sum’ 0 1000000;;
- : int = 0
```

- もちろんこれは正しくないが，ともかくスタックオーバーフローはしない

Tail calls

- some function calls are called *"tail calls"* and do not require extra space
- for example, let's see what happens if we remove `a +` in the `sum` function

```
let rec sum a b =
if a >= b then
   0
else
   a + (sum (a+1) b)
```

→

```
let rec sum' a b =
if a >= b then
   0
else
   sum' (a+1) b
```

- then . . .

```
# sum' 0 1000000;;
- : int = 0
```

- it's not a correct program, of course, but at least does not cause a stack overflow

末尾呼び出し (tail call)

- f が g を呼び出す時 g の結果が「そのまま」f の結果になるような位置にある関数呼び出しを,「末尾呼び出し」という
- 以下の青は末尾呼び出し，赤は違う

```
let f x =
  if ... then
     g (x - 1)
  else if ... then
     1 + g (x - 2)
  else
     let y = g (x - 3) in
       g (y - 4)
```

- 特に，末尾呼び出しであるような再帰呼出しを,「末尾再帰呼び出し (tail recursive call, tail recursion)」という

```
let rec sum a b =
if a >= b then
   0
else
   a + (sum (a+1) b)
```

```
let rec sum' a b =
if a >= b then
   0
else
   sum' (a+1) b
```

Tail calls

- when f calls g at such a position that the result of g becomes the result of f, it is called a "tail call"
- below, function calls colored blue are tail calls; red are not

```
let f x =
  if ... then
     g (x - 1)
  else if ... then
     1 + g (x - 2)
  else
     let y = g (x - 3) in
       g (y - 4)
```

- in particular, a tail call that is also a recursive call is called "a tail recursive call" or simply "a tail recursion"

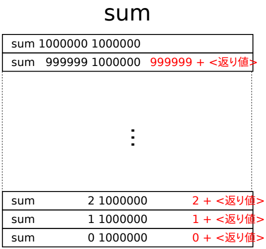
```
let rec sum a b =
if a >= b then
   0
else
   a + (sum (a+1) b)
```

```
let rec sum' a b =
if a >= b then
   0
else
   sum' (a+1) b
```

sum がオーバーフローする「深い」理由

- 関数呼出し時，実際にメモリに記録しておく必要があるのは「その呼び出しの返り値を受け取ったあと何をするか」という指示
- sum の場合,「$a+$ 返り値」という「計算」
- 「呼び出しの返り値」をそのまま返すだけ (つまり末尾呼び出し) だったら，メモリはいらない

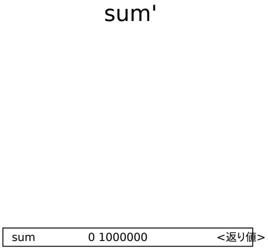
```
let rec sum' a b =
if a >= b then
  0
else
  sum' (a+1) b
```



sum がオーバーフローする「深い」理由

- 関数呼出し時，実際にメモリに記録しておく必要があるのは「その呼び出しの返り値を受け取ったあと何をするか」という指示
- sum の場合,「$a+$ 返り値」という「計算」
- 「呼び出しの返り値」をそのまま返すだけ (つまり末尾呼び出し) だったら，メモリはいらない

```
let rec sum' a b =
if a >= b then
   0
else
   sum' (a+1) b
```

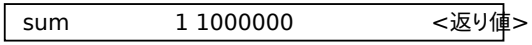


sum がオーバーフローする「深い」理由

- 関数呼出し時，実際にメモリに記録しておく必要があるのは「その呼び出しの返り値を受け取ったあと何をするか」という指示
- sum の場合,「$a+$ 返り値」という「計算」
- 「呼び出しの返り値」をそのまま返すだけ (つまり末尾呼び出し) だったら，メモリはいらない

```
let rec sum' a b =
if a >= b then
   0
else
   sum' (a+1) b
```

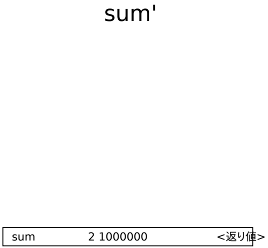
sum'



sum がオーバーフローする「深い」理由

- 関数呼出し時，実際にメモリに記録しておく必要があるのは「その呼び出しの返り値を受け取ったあと何をするか」という指示
- sum の場合,「$a+$ 返り値」という「計算」
- 「呼び出しの返り値」をそのまま返すだけ (つまり末尾呼び出し) だったら，メモリはいらない

```
let rec sum’ a b =
if a >= b then
   0
else
   sum’ (a+1) b
```



sum がオーバーフローする「深い」理由

- 関数呼出し時，実際にメモリに記録しておく必要があるのは「その呼び出しの返り値を受け取ったあと何をするか」という指示
- sum の場合,「$a+$ 返り値」という「計算」
- 「呼び出しの返り値」をそのまま返すだけ (つまり末尾呼び出し) だったら，メモリはいらない

```
let rec sum' a b =
if a >= b then
   0
else
   sum' (a+1) b
```

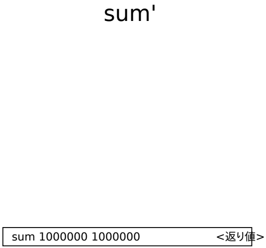
sum'

sum 999999 1000000 <返り値>

sum がオーバーフローする「深い」理由

- 関数呼出し時，実際にメモリに記録しておく必要があるのは「その呼び出しの返り値を受け取ったあと何をするか」という指示
- sum の場合，「$a+$ 返り値」という「計算」
- 「呼び出しの返り値」をそのまま返すだけ (つまり末尾呼び出し) だったら，メモリはいらない

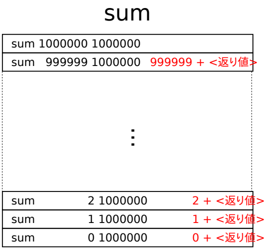
```
let rec sum' a b =
if a >= b then
   0
else
   sum' (a+1) b
```



A “deeper” reason that deep recursions of `sum` overflow

- when you call a function, what truly needs to be recorded in memory is information about “what to do with the return value”
- in the case of `sum`, it is “$a+$ (the return value)”
- if the return value of the call becomes the return value of the caller (i.e., it is a tail call), you don’t need memory

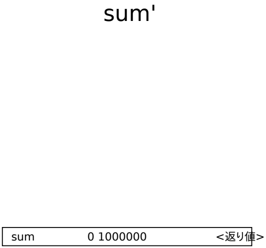
```
let rec sum' a b =
if a >= b then
   0
else
   sum' (a+1) b
```



A “deeper” reason that deep recursions of `sum` overflow

- when you call a function, what truly needs to be recorded in memory is information about “what to do with the return value”
- in the case of `sum`, it is “$a+$ (the return value)”
- if the return value of the call becomes the return value of the caller (i.e., it is a tail call), you don’t need memory

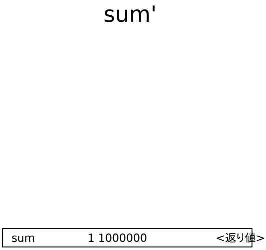
```
let rec sum’ a b =
if a >= b then
   0
else
   sum’ (a+1) b
```



A "deeper" reason that deep recursions of **sum** overflow

- when you call a function, what truly needs to be recorded in memory is information about "what to do with the return value"
- in the case of **sum**, it is "$a+$ (the return value)"
- if the return value of the call becomes the return value of the caller (i.e., it is a tail call), you don't need memory

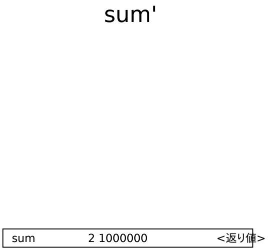
```
let rec sum' a b =
if a >= b then
   0
else
   sum' (a+1) b
```



A "deeper" reason that deep recursions of **sum** overflow

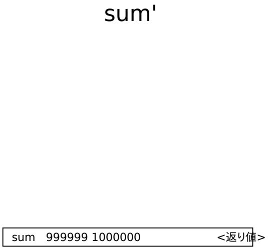
- when you call a function, what truly needs to be recorded in memory is information about "what to do with the return value"
- in the case of **sum**, it is "$a+$ (the return value)"
- if the return value of the call becomes the return value of the caller (i.e., it is a tail call), you don't need memory

```
let rec sum' a b =
if a >= b then
   0
else
   sum' (a+1) b
```



A “deeper” reason that deep recursions of **sum** overflow

- when you call a function, what truly needs to be recorded in memory is information about “what to do with the return value”
- in the case of **sum**, it is “$a+$ (the return value)”
- if the return value of the call becomes the return value of the caller (i.e., it is a tail call), you don’t need memory

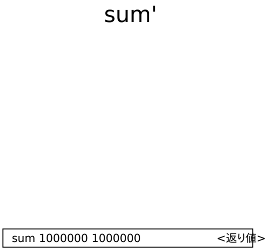



```
let rec sum' a b =
if a >= b then
  0
else
  sum' (a+1) b
```

A “deeper” reason that deep recursions of **sum** overflow

- when you call a function, what truly needs to be recorded in memory is information about “what to do with the return value”
- in the case of `sum`, it is “$a+$ (the return value)”
- if the return value of the call becomes the return value of the caller (i.e., it is a tail call), you don’t need memory

```
let rec sum’ a b =
if a >= b then
   0
else
   sum’ (a+1) b
```



スタックオーバーフローを避けるには

- 深い再帰は「末尾再帰で」
- sum a b の場合:

```
let sum a b =
  (* sum_tail a b s = s + [a,b) の和 *)
  let rec sum_tail a b s =
    if a >= b then
      s
    else
      sum_tail a b (s + a)
  in sum_tail a b 0
```

- 万能変換公式は (実はあるのだが) 汚くなるのであまり期待しない方が良い．例で慣れた方が良い
- 役立つ経験則: 余分な引数 (上記の s) を受け取る関数を作るとうまく行くことが多い

To avoid stack overflows . . .

- deep recursions must be tail recursions
- for `sum a b` . . .

```
let sum a b =
  (* sum_tail a b s = s + [a,b) の和 *)
  let rec sum_tail a b s =
    if a >= b then
      s
    else
      sum_tail a b (s + a)
  in sum_tail a b 0
```

- there is a formula to get rid of all non-tail calls, but you will live without it (learn from examples)
- a good heuristic: consider adding extra parameter (`s` above) to the function

末尾再帰への書き換え練習 — リストを作る (1)

- random_ints

```
let rec random_ints a n =
  if n <= 0 then
    []
  else
    (Random.int a) :: (random_ints (n - 1))
```

→

```
let random_ints a n =
  (* random_ints_tail a n l = n 要素の乱数 @ l  *)
  let rec random_ints_tail a n l =
    if n <= 0 then
      l
    else
      random_ints_tail a (n - 1) ((Random.int a)::l)
  in random_ints_tail a n []
```

- 注: 結果の並び方が逆になる (が，乱数だから気にしていない)

Practicing tail recursions — make a list (1)

- `random_ints` (random integers)

```
let rec random_ints a n =
  if n <= 0 then
    []
  else
    (Random.int a) :: (random_ints (n - 1))
```

→

```
let random_ints a n =
  (* random_ints_tail a n l = n 要素の乱数 @ l  *)
  let rec random_ints_tail a n l =
    if n <= 0 then
      l
    else
      random_ints_tail a (n - 1) ((Random.int a)::l)
  in random_ints_tail a n []
```

- Remark: the latter stores the result in the opposite order (but we ignore that as they are random numbers anyway)

末尾再帰への書き換え練習 — リストを作る (2)

- range

```
let rec range a b =
  if a >= b then
    []
  else
    a :: (range (a + 1) b)
```

→

```
let range a b =
  (*  range_tail a b l = [a,b) のリスト @ l  *)
  let rec range_tail a b l =
    if a >= b then
      l
    else
      range_tail a (b - 1) ((b-1)::l)
  in range_tail a b []
```

- 今度は結果の順番を意識して，a を増やすのではなく，b を減らす再帰に切り替えた

Practicing tail recursions — make a list (2)

- range

```
let rec range a b =
  if a >= b then
    []
  else
    a :: (range (a + 1) b)
```

→

```
let range a b =
  (*  range_tail a b l = [a,b) のリスト @ l  *)
  let rec range_tail a b l =
    if a >= b then
      l
    else
      range_tail a (b - 1) ((b-1)::l)
  in range_tail a b []
```

- the order matters this time, so the tail recursive version decrements `b` rather than incrementing `a`

末尾再帰への書き換え練習 — リストからリストを作る

- map

```
let rec map f lst =
  match lst with
    [] -> []
  | x::xs -> (f x) :: (map f xs)
```

- これをそのまま末尾呼び出しにするのは難しい．こんなことができればいいのだが . . .

```
let rec map_tail f lst l =
  match lst with
    [] -> []
  | all_but_last @ [last]
    -> map_tail f all_but_last ((f last)::l)
```

- 入力は「前から」しか見れない．だが結果は「後ろから」計算したい. . .
- 頻出パターン: 一旦逆順に計算して最後にひっくり返す

Practicing tail recursions — a list from a list

- `map`

```
let rec map f lst =
  match lst with
    [] -> []
  | x::xs -> (f x) :: (map f xs)
```

- it is difficult to make it tail recursive directly. we wish we could do something like this ...

```
let rec map_tail f lst l =
  match lst with
    [] -> []
  | all_but_last @ [last]
    -> map_tail f all_but_last ((f last)::l)
```

- we can only process the input from "the head to the end", but the output should be computed from "the end to the head" ...
- frequent pattern: *get the list in the opposite order and reverse it*

末尾再帰への書き換え練習 — リストからリストを作る

- 一旦逆順に計算して最後にひっくり返す

```
let map f lst =
  (* map_rev f x l = [ f x_{n-1}; ··· ; f x_0 ] @ l *)
  let rec map_rev f lst l =
    match lst with
      [] -> []
    | x::xs -> map_rev f xs ((f x)::l)
  in rev (map_rev f lst [])
```

- rev は末尾再帰だけで書ける? → 書けます

```
let rev lst =
  (* rev_tail x l = [ x_{n-1}; ··· ; x_0 ] @ l *)
  let rec rev_tail lst l =
    match lst with
       [] -> l
     | x::xs -> rev_tail xs (x::l)
  in rev_tail lst []
```

Practicing tail recursions — a list from a list

- compute the list in the opposite order and reverse it

```
let map f lst =
  (* map_rev f x l = [ f x_{n-1}; ··· ; f x_0 ] @ l  *)
  let rec map_rev f lst l =
    match lst with
      [] -> []
    | x::xs -> map_rev f xs ((f x)::l)
  in rev (map_rev f lst [])
```

- can we write **rev** with a tail recursion? → yes

```
let rev lst =
  (* rev_tail x l = [ x_{n-1}; ··· ; x_0 ] @ l *)
  let rec rev_tail lst l =
    match lst with
       [] -> l
     | x::xs -> rev_tail xs (x::l)
  in rev_tail lst []
```

関数型なもののまとめ

- それが自然である場合は，再帰関数・functional update で，アルゴリズムを「宣言的に」記述
- クロージャ(自由変数を含んだ関数を作れる)
- map, filter など多相的な関数で，アルゴリズムを簡潔に記述
- 静的型検査，型推論，多相型

実は「関数型」と呼ばれない言語にも取り入れられている．例えば C++

- ラムダ式
- Standard Template Library
- template

Summary : functional programming

- write algorithms declaratively where it is natural to do so, with recursions and functional updates
- closure (functions with free variables)
- write algorithms concisely with generic functions such as map and filter
- static type checking, type reconstructions and polymorphism

they are incorporated into non functional languages. e.g., C++

- lambdas
- Standard Template Library
- template