# 関数・論理型プログラミング実験
# ML演習第1回

松田 一孝

TA: 武田広太郎 寺尾拓

# 担当者

- 松田 一孝，武田広太郎，寺尾拓
  - fl-enshu@kb.is.s.u-tokyo.ac.jp

# 講義のサポートページ

- http://www-kb.is.s.u-tokyo.ac.jp/~kztk/cgi-bin/m/
  - 講義資料等が用意される
  - レポートの提出先
  - 利用にはアカウントが必要
    - 自分の名前と学籍番号を書いたメールを kztk@is.s.u-tokyo.ac.jp までメールすること
      - 件名は「FP/LP実験アカウント申請」
      - アカウント名/パスワードを返信
        - PCからのメールを受けとれるように

# 本実験の予定（全15回）

- 4/8
- 4/15
- 4/22
- 5/01
- 5/13
- 5/20
- 5/27
- 6/3
- 6/10
- 6/17
- 6/24
- 7/1
- 7/8
- 7/15
- 7/22

**ML(OCaml)演習:**
関数プログラミングと関数型言語の基礎
関数型言語のインタプリタの作成

**Haskell演習:**
OCamlとは別の抽象化・プログラミング技法
より先進の関数プログラミング技法

**Prolog演習+α:**
論理プログラミングと
論理型言語の基礎

**最終課題**

# 評価方法

- レポートが主
  - 基本的には平均点=成績
  - 全課題について提出すること
    - 3回以上未提出 = 不可
- 出席も多少考慮するかも？
  - 素点が59点や79点だったときに活躍？
  - 不可になりそうな時も活躍？

# ML演習の内訳

- 第1回～第3回
  - ML言語（OCaml）を学ぼう
    - 関数プログラミングの基礎
- 第4回～第7回
  - MLインタプリタを作ろう
    - 関数型言語がどうやって動くのかを知る

# 参考資料

- OCaml公式 http://caml.inria.fr/
  - 処理系のダウンロード
  - マニュアルなど

- Developing Applications with Objective Caml
  - Online Pre-release: http://caml.inria.fr/pub/docs/oreilly-book/

# 日本語の参考資料

- 「プログラミングの基礎」
  - 浅井 健一 著


- 「プログラミング in OCaml」
  - 五十嵐 淳 著

# 今日の内容

- OCamlとは？
- インタプリタの使い方
- 基本的な構文
- パターンマッチ

- レポート課題

# OCaml?

- MLの方言の一種
  - 強力な型システム
  - 柔軟なデータ型定義とパターンマッチ
  - 強力なモジュールシステム

  - さかんな開発

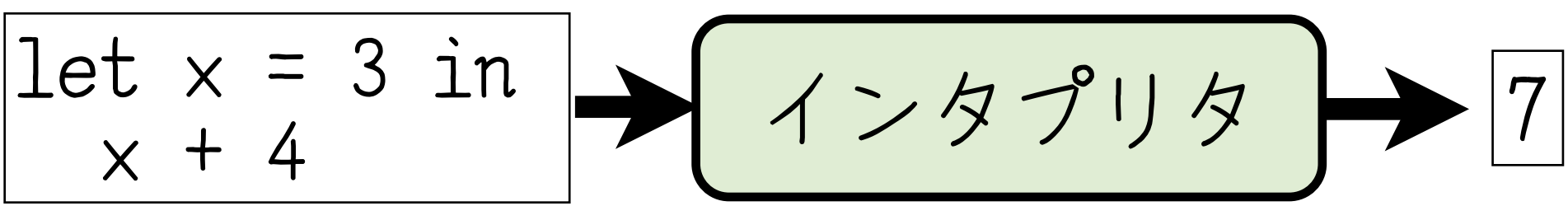
# OCamlの型システム

- 強い静的型付け
  - 強い：型整合を強制
    - メモリエラーなどが絶対に生じない
  - 静的：コンパイル時に型検査
    - 実行時オーバヘッドなし
- 型推論
  - 変数等の型を書かなくてよい
- 型多相 (Parametric Polymorphism)

# ocamlインタプリタの使い方

# インタプリタ

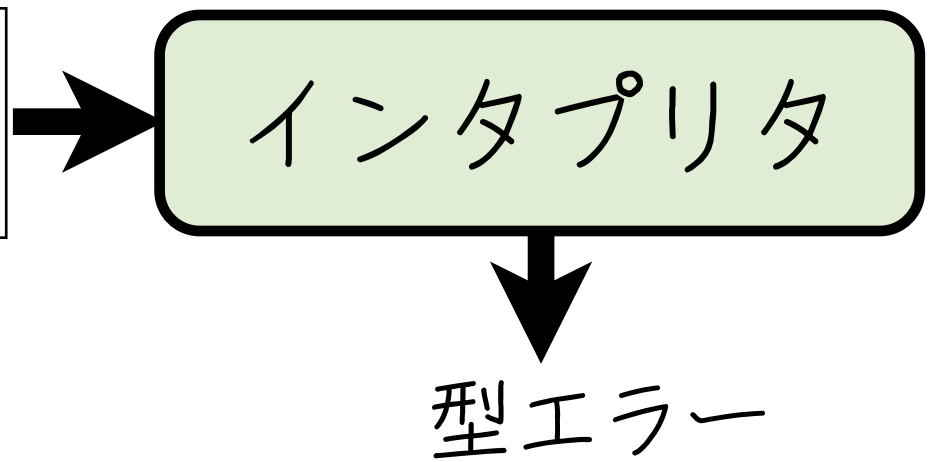
- 「式」を入力として受けとり
  その式を「評価」して
  その評価結果の「値」を返すプログラム



# インタプリタと型検査

- OCamlのインタプリタは式の評価前に「型検査」を行う
  - 型検査にパスした式の評価は失敗しないことが保証
    - Well-typed programs cannot go wrong.

```
let x = 3 in
  x + "A"
```



# ocamlインタプリタの準備

- 貸与PC
  - sudo apt-get install ocaml
    - sudo apt-get install rlwrapも推奨
- Linux
  - apt, yum, YaST等パッケージの利用
- Mac
  - MacPorts
- Windows
  - cygwin or 公式ビルド

# 起動・終了

引数なし起動するとで対話的に動作

```
$ ocaml
        Objective Caml version 4.00.1

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

1+2を評価せよ

その結果は3という値で型はint

インタプリタを終了
(Ctrl+Dでも可)

# 基本的な使い方

- 式;;

式1+2を評価せよ

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

結果は値3

- 定義;;

式1+2の結果をxとせよ

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

式xの値は？

# 基本的な式

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

注意：ここで紹介するものがすべてではない
詳細はマニュアル参照

# コメント

- (* コメント *)
  - ネストできる

```
# (* this is a comment *) 1 + 2;;
- : int = 3
# 1 (* a (* nested *) comment *) + 2;;
- : int = 3
```

# 整数定数

1という「式」を評価すると

1という「値」になる

```
# 1;;
- : int = 1
# 12              (* 10進 *);;
- : int = 12
# 0xdeadBEEF      (* 16進 *);;
- : int = 3735928559
# 0b11101111      (* 2進 *);;
- : int = 239
# 0o755           (* 8進 *);;
- : int = 493
```

# 整数演算

```
# 13 + 4   (* 和 *);;
- : int = 17
# 13 - 4   (* 差 *);;
- : int = 9
# 13 * 4   (* 積 *);;
- : int = 52
# 13 / 4   (* 商 *);;
- : int = 3
# 13 mod 4 (* 剰余 *);;
- : int = 1
# - (13 + 4) (* 符号反転 *);;
- : int = -17
```

# 浮動小数点数の定数

1.25という「式」を評価すると

1.25という「値」になる

```
# 1.25;;
- : float = 1.25
# 1.25e-2 (* 指数表記 *);;
- : float = 0.0124
```

# 浮動小数点数の演算

- 整数とは異なる演算子を使用

```
# 13.0 +.  4.0   (* 和 *);;
- : float = 17.
# 13.0 -.  4.0   (* 差 *);;
- : float = 9.
# 13.0 *.  4.0   (* 積 *);;
- : float = 52.
# 13.0 /.  4.0   (* 商 *);;
- : float = 3.25
# 13.0 **  4.0   (* 累乗 *);;
- : float = 28561.
```

# 注意

- 整数と小数は混ぜられない

```
# 13 +.  4.0;;
Error: This expression has type int but
an expression was expected of type float
# 13.0 + 4.0;;
Error: This expression has type float
but an expression was expected of type
int
```

# 整数と小数の変換

- float_of_int

```
# float_of_int 3;;
- : float = 3.
```

- int_of_float

```
# int_of_float 3.3;;
- : int = 3
# int_of_float (-3.3);;
- : int = -3
```

# ブール値

```
# true;;
- : bool = true
# false;;
- : bool = false
```

trueという「式」を評価すると

trueという「値」になる

# 比較演算，ブール演算

```
# 13.0 = 4.0;;
- : bool = false
# 13 < 4;;
- : bool = false
```

```
# true && false (* 論理積 *);;
- : bool = false
# true || false (* 論理和 *);;
- : bool = true
# not false      (* 否定 *);;
- : bool = true
```

# 変数の定義（トップレベル）

- let 変数 = 式
  - トップレベル定義自体は式ではない

```
# let x = 3;;
val x : int = 3
# x;;
- : int = 3;;
# 4 * x;;
- : int = 12;;
# let y = 4 * x;;
val y : int = 12
# y;;
- : int = 12;;
```

# 変数（局所定義）

- let x = $e_1$ in $e_2$
  - 式$e_1$の評価結果をxに束縛して$e_2$を評価

xはこの式の中でのみ有効

```
# let x = 3 in x + x;;
- : int = 6
# x;;
Error: Unbound value x
# let x = 3 in let x = 5 in x;;
- : int = 5;;
```

# 注意

- let x = $e_1$ in $e_2$は式
- let x = eは式ではない

```
# let x = let y = 3;;
Error: Syntax Error
```

# 関数

- fun x -> e

```
# fun x -> x + 1;;
- : int -> int = <fun>
# fun x -> fun y -> x + y (* 2引数関数 *);;
- : int -> int -> int = <fun>
# fun x y -> x + y          (* 上の略記 *);;
- : int -> int -> int = <fun>
```

# 関数の適用

- $e\ e_1\ \ldots\ e_n$

```
# (fun x -> x + 1) 2;;
- : int = 3
# (fun x y -> x + y) 1 2;;
- : int = 3;;
```

# 関数とlet

- 関数は「値」⇒letで束縛可

```
# let inc = fun x -> x + 1;;
val inc : int -> int = <fun>
# inc 1;;
- : int = 2
# let inc x = x + 1    (* 略記法 *);;
val inc : int -> int = <fun>
# let add = fun x y -> x + y;;
val add : int -> int -> int = <fun>
# let add x y = x + y (* 略記法 *);;
val add : int -> int -> int = <fun>
# add 1 2;;
- : int = 3
```

# 関数の部分適用

```
# let add x y = x + y;;
val add : int -> int -> int = <fun>
# add 1 2;;
- : int = 3
# let inc = add 1;;
val inc : int -> int
# inc 2;;
- : int = 3
# inc 3;;
- : int = 4
# (add 1) 2;;
- : int = 3
```

部分適用：n引数関数にn未満個の引数を適用

# 条件分岐

- if e then e1 else e2

```
# if true then 2 else 3;;
- : int = 2
# if false then 2 else 3;;
- : int = 3
# let abs x = if x<0 then -x else x;;
val abs : int -> int = <fun>
# abs 10;;
- : int = 10
# abs (-10);;
- : int = 10
```

# 再帰関数

- let rec x = e

let rec fact n = …と略記可

```
# let rec fact = fun n ->
    if n = 1 then
        1
    else
        n * fact (n-1);;
val fact : int -> int = <fun>
# fact 10;;
- : int = 3628800
```

# 相互再帰

- let rec $x_1$ = $e_1$ and $x_2$ = $e_2$ ...

```
# 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 10;;
- : bool = true
# odd 10;;
- : bool = false
```

# 文字列

- 「″」で囲う
- 連接は「^」

```
# ″Hello World″;;
- : string = ″Hello World″
# ″Hello″ ^ ″ ″ ^ ″World″ ;;
- : string = ″Hello World″
```

# 組（tuple）

- $e_1, e_2, ..., e_n$
  - 異なりうる型の式を固定個まとめる

```
# (1, 2.0);;
- : int * float = (1, 2.)
# (1, 2.0, false);;
- : int * float * bool = (1, 2., false)
```

# 組の分解

- letで

```
# let t = (1, 2.0, false);;
val t : int * float * bool = (1, 2., false)
# let (x,y,z) = t;;
val x : int = 1
val y : float = 2.
val z : bool = false
# let (_,y,_) = t in y < 0.0;;
- : bool = false
```

使わない要素は_で無視

# リスト

- $[e_1;e_2;...;e_n]$
  - 同じ型の要素を可変個まとめる

「'a」は次回説明

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

# リストの演算

- コンス $e_1$ :: $e_2$

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


- 連接 $e_1$ @ $e_2$

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

# リストと組の違い

- 組：異なる型の固定個の要素

```
# (1, 2.0);;
- : int * float = (1, 2.)
# (1, 2.0, 3);;
- : int * float * int= (1, 2., 3)
```

- リスト：同じ型の可変個の要素

```
# [1;2];;
- : int list = [1; 2]
# [1;2;3];;
- : int list = [1; 2; 3]
# [1;2.0];;
Error: …
```

# パターンマッチ

# パターンマッチ

- 値が「パターン」に照合するか？
  - 値1 と パターン1 は照合
  - 値1 と パターン0 は不照合
  - 値1 と パターンx は
    x=1 とすることで照合

  - 場合分けに利用

# match式

- match e with | $p_1$ -> $e_1$
  | $p_2$ -> $e_2$ …
  - * $p_1, p_2, ...$：パターン
  - ◆ 上から順に照合するか試す
  - ◆ 照合したら対応する式を評価

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

# パターンの種類

- 定数パターン
- 変数パターン
- 組パターン
- リストパターン
- ワイルドカード
- …

# 定数&変数パターン

- 定数パターン
  - 値とパターンが同じとき照合
- 変数パターン
  - 任意の値と照合，束縛を生じる
  - 変数は本体で使用可

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

# 組パターン

- 組の要素それぞれが照合すればOK
  - 変数パターンと組合し要素を取り出し

```
# let scale m p =
      match p with
        | (x,y) -> (m*x,m*y);;
val scale : int -> int*int -> int*int = <fun>
# scale 2 (1,2);;
- : int * int = (2, 4)
```

# リストパターン

- []と，$p_1$ :: $p_2$

```
# let rec sum xs =
    match xs with
          | []    -> 0
          | y::ys -> y + sum ys;;
val sum : int list -> int
# sum [1;2;3;4];;
- : int = 10
```

# ワイルドカード

- 任意の値にマッチ
  - 照合結果を利用しないことを明示

```
# let null xs =
    match xs with
          | []    -> true
          | _::_  -> false;;
val null : 'a list -> bool
# null [1;2;3;4];;
- : bool = false
```

# 関数定義とパターンマッチ

- let (rec)やfunの仮引数部分にはパターンが書ける

```
# let fst3 (a, _, _) = a;;
val fst3 : 'a * 'b * 'c -> 'a = <fun>
# fst3 (1, 2.0, false);;
- : int = 1
# (fun n (x, y) -> (n*x, n*y)) 2 (1, 2);;
- : int * int = (2, 4)
```

# 網羅的でないパターン

- 網羅的でない = 照合しない値がある

```
# let head (x::_) = x;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val head : 'a list -> 'a = <fun>
# head [1;2];;
- : int = 1
# head [];;
Exception: Match_failure ("", 3, -7).
```

警告

例外の発生

# パターンの注意

- 同じ変数は一つのパターンで一回だけ

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

# ガード

- パターンに照合の条件を付加

```
# let is_diag p =
    match p with
      | (x,y) when x=y -> true
      | _              -> false;;
val is_diag : 'a * 'a -> bool
# is_diag (1,1);;
- : bool = true
# is_diag (1,2);;
- : bool = false
```

# パターンの注意2

- 定義された変数を
  定数パターンとして使うことは不可
  - 変数は変数パターンとなる

```
# let x = 0;;
_ : int = 0
# match (1,2) with
    | (x,_) -> true
    | _     -> false;;
Warning 11: this match case is unused.
- : bool = true
```

# その他の使い方

# ファイルの読み込み

- #use "FILENAME.ml"

```
(* test.ml *)
1 + 2;;
3.0 +. 4.0;;
```

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

# 補足

```
(* test.ml *)
let pi = 3.14
let area_circ r =  3.14 *. r *. r ;;
area_circ 4.0
let x = 5.0;;
area_circ x;;
area_circ (2.*.x)
```

「定義」の前に;;は不要

同様

最後にもいらない

```
# #use "test.ml";;
val pi : float = 3.14
val area_circ : float -> float = <fun>
- : float = 50.24
val x : float = 5.
- : float = 78.5
- : float = 314.
```

# ファイルの読み込み2

- ocaml FILENAME.ml
  - 評価し，評価結果の値を捨てる

```
(* test.ml *)
let _ = print_string "Hello";;
1 + 2;;
print_string " "
let _ = print_string "World.\n";;
```

```
$ ocaml test.ml
Hello World.
```

# rlwrap, ledit

- ocaml対話的インタプリタの使い勝手を改善
  - rlwrap ocamlやledit ocamlで起動
    - 「左」をおせばカーソルが左に
    - 「上」でヒストリをさかのぼれる
  - なお，ocamlで起動すると…

```
$ ocaml
…
# ^[[D^[[C
…
# ^[[A
```

カーソルキー「左」や「右」で変な文字が

「上」でも同様

# レポート課題について

# 課題の種類

- 問
  - 普通の課題
- 発展
  - 問じゃものたりない人向け
    - 解くことは必須ではない
      - 解けなくても減点しない
      - 解けたら加点
    - 一問の点数配分は 問 ＞ 発展
    - 難易度は 問 ＜ 発展
    - マニュアルや論文を読む必要があることも

# 提出物の形式

- レポート本文
  - プレーンテキスト（拡張子 .txt）
    - 必要ならばPDFも可
- ソースコード一式
  - 単一ならそのまま
  - 複数の時はtar.gz/tar.bz2かzipで
- ファイル名は以下を推奨

出題回-学籍番号下6桁.拡張子

  - 少なくとも空白や非ASCII文字は避けよ

# 提出物に含めるもの

- レポート本文
  - 名前，学生証番号
  - 動作例
    - プログラムが「正しく」動作することを示すのに適切な例を考えよ
  - （不要と書かれてなければ）考察
- ソースコード一式
  - 適切なコメント
    - 動作や意図がわかるようにする
  - 実行する必要があれば，実行の仕方
  - ビルドする必要があれば，ビルドの仕方

# 注意

- 構文・型エラーが出ないことを確認
  - 構文エラーや型エラーのあるものはMLのプログラムではない
  - ただし，「エラーが出ること」が「動作例」として適切ならばそのようなコードをレポートに含めよ

# 提出方法

- サポートページから
  - http://www-kb.is.s.u-tokyo.ac.jp/~kztk/cgi-bin/m/
  - 利用にはアカウントが必要
    - アカウントを用意するので，
      自分の名前と学籍番号を書いたメールを
      kztk@is.s.u-tokyo.ac.jp
      までメールすること
    - IDと仮パスワードが送られてくる
    - ギリギリにメールして
      発行が間に合わなくても知りません

# レポート課題1

締切：2週間後の火曜13:00

# 注意

- 今回の課題を解くのに，
  標準ライブラリの関数は
  基本的には使ってはならない
  - 四則演算や比較演算はOK
  - 問8や発展1でList.fold_rightや
    List.fold_leftを使うのはOK

# 問1（考察不要）

- 以下の関数を書け
  - 非負整数 n を受け取って0からnまで（n含む）の整数の和を求める
    sum_to : int -> int
  - 正整数nが素数かどうか判定する
    is_prime : int -> bool
  - ユークリッドの互除法に基づき最大公約数を計算する
    gcd : int -> int -> int

# 問2

- n番目のフィボナッチ数を計算する関数fibを書け
  - ただし，以下をそれぞれ書くこと
    - 再帰の回数が n の指数
    - 再帰の回数が n に対し線形

注意：このような問では本当にそうなっている
ことをレポートで説明するように

  - オーバフローは気にしなくてOK
  - 浮動小数点数を用いないこと

# 問3（考察不要）

- 以下の関数を書け
  - 関数 f を受け取って，
    fを二回合成した関数を返す関数 twice
    * 例：twice (fun x -> 2*x) 3 = 12

  - 関数 f と整数 n を受け取って，
    fをn回合成した関数を返す関数 repeat
    * 例：repeat (fun x -> 2*x) 4 3 = 48

# 問4

- 関数fixを以下で定義する

```
let rec fix f x = f (fix f) x
```

この関数以外の再帰関数を使わずに問1と問2の関数を実装せよ

- for等も使用を禁止
- ライブラリ関数もダメ

# 問5

- 以下を満たすfold_rightを実装せよ
  - fold_right f [$x_1$;$x_2$;…;$x_n$] e
    = f $x_1$ (f $x_2$ (…(f $x_n$ e)…))
- 以下を満たすfold_leftを実装せよ
  - fold_left f e [$x_1$;…;$x_{n-1}$;$x_n$]
    = f (f (… (f e $x_1$) …) $x_{n-1}$) $x_n$
  - 末尾再帰

# 問6

- 以下の再帰関数を書け
  - @と同じ動作をする関数append
    append: 'a list -> 'a list -> 'a list
    * 当然@を使用してはならない
  - リストの末尾を返す関数
    last: 'a list -> 'a
  - 関数fとリストxsを受けとり，fをxsの各要素に適用したリストを返す関数
    map: ('a -> 'b) -> 'a list -> 'b list

# 補足

- 使用例

```
# last [1;2];;
- : int = 2
# map (fun x -> x+1) [1;2;3];;
- : int list = [2; 3; 4]
```

# 問7

- リストを受け取り，
  そのリストを逆順にしたリストを返す
  関数 reverse を書け
  - 入力リストの長さに対し，
    線形時間で動作すること

# 問8

- 問6の関数をfold_leftおよびfold_rightを用いて実装し，それぞれを比較せよ
  - 書きやすさとか
  - 実行速度とか

  - 他の再帰関数を使ってはならない
    - ただし@を利用してもよい
      - 利用しなくても書ける（発展1参照）

# 問9

- リストを受け取りの全ての順列をリストにして返す関数
  perm : 'a list -> 'a list list
  を実装せよ
  - 重複は考慮しない

```
# perm [1]
- : int list list = [[1]]
# perm [1;2]
- : int list list = [[1; 2]; [2; 1]]
```

# 発展1

- 問7の関数 reverse を fold_rightを用いて実装せよ
  - ただし，reverse xs がxsの長さに対し線形で動作するように
  - 他の再帰関数を使ってはならない
    - @は再帰関数であることに注意

# 発展2

- fold_leftを，
  fold_rightを用いて実装せよ
  - let fold_left f e xs =
      ... fold_right (...) xs ...
    という形
    - …の部分に再帰関数（fold_rightも）や
      let recを用いてはならない
    - 末尾再帰にならないがここでは気にしない
- 上と同様の条件で，fold_rightを，
  fold_leftを用いて実装せよ