

関数・論理型プログラミング実験

第1回

江口 慎悟

酒寄 健

塚田 武志

松下祐介

「関数・論理型」の狙い

- 「実際の計算機」から離れた計算モデル
 - Cf. 手続き型 = 計算機で処理すべき命令列を記述
 - 例：リスト型
 - 実際の計算機 : ポインタを使った実装
 - 関数型の理解 : 要素の有限 (or 無限) 列の集合

理解や性質の証明が容易

担当者

- 江口 慎悟
 - 酒寄 健
 - 塚田 武志
 - 松下祐介
-
- f1-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/~tsukada/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
- アカウントの発行は上のページの「ログイン」
 - 登録フォームは一週間で閉じます

全体の予定

- 関数型プログラミング演習
 - 第1回～第4回: ML(OCaml) 演習
 - 第5回～第9回: インタプリタ作成
- 論理型プログラミング演習
 - 第10回～第12回: Prolog演習
- まとめ
 - 第13回: リバーシプログラム作成

評価について

- 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
- 出席が加点材料になるケースも？

関数型言語演習

- ML言語 (OCaml) を学ぶ (第1回～第4回)
 - 関数型プログラミングの基礎を学ぶ
- インタプリタを作る (第5回～第9回)
 - 関数型言語がどうやって動くのか知る

今日の内容

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

- レポート課題について

OCaml とは

- 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ング
 - 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：コンパイル時に型をチェックする
 - 実行時のオーバーヘッドがない
- 型推論
 - 変数等の型を書かなくてもよい
- 型多相

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>
- Real World Ocaml <https://realworldocaml.org/>

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

インタプリタの使い方

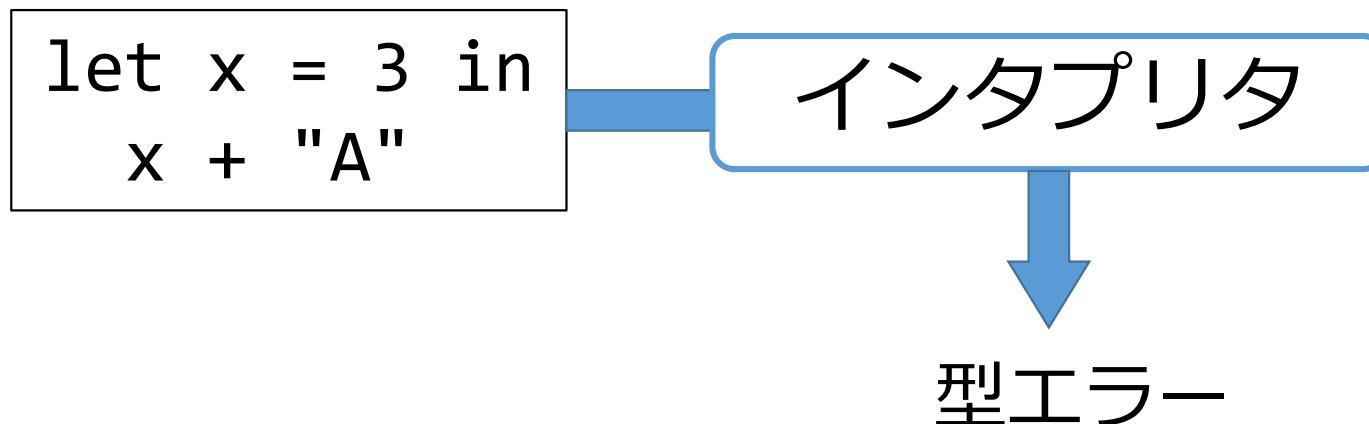
インタプリタとは

- 。「式」を入力として受け取り
その式を「評価」して
その評価の結果の「値」を返すプログラム



インタプリタと型検査

- OCaml インタプリタは式を評価する前に「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障される



インタプリタの準備

- 貸与PC
 - sudo apt-get install ocamli
- Linux
 - apt, yum YaST などパッケージの利用
- Mac
 - MacPorts
- Windows
 - cygwin or 公式ビルド

起動・終了

引数なしで起動すると対話的に動作

```
$ ocaml
```

OCaml version 4.02.0

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

1+2 を評価せよ

```
# #quit;;
```

評価結果は 3 で
その型は int

インタプリタを終了
(Ctrl+Dでも可)

基本的な使い方

- 式;;

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

式 $1+2$ を評価せよ

- 定義;;

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

$1+2$ の評価結果を x とせよ

式 x の値は？

ファイルに書いたプログラムの利用

- 直接式・宣言を書く代わりにテキストファイルの内容を読み込ませることができる

```
# #use "test.ml";;
val x : int = 3
val y : int = 10
# x + y;;
- : int = 13
```

test.ml

```
let x = 1 + 2;;
let y = 10;;
```

基本的な構文

コメント

基本型と演算

変数、関数と再帰

組、リスト

コメント

- (* コメント *)
 - 入れ子にできる

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

基本的な構文

コメント

基本型と演算

変数、関数と再帰

組、リスト

整数定数

3という「式」を評価すると

```
# 3;;          3という「値」になる
- : int = 3

# 123;;      (* 10進 *)
- : int = 123

# 0xdeadBEEF;; (* 16進 *)
- : int = 3735928559

# 0b1110;;    (* 2進 *)
- : int = 14

# 0o775;;     (* 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
```

整数演算

```
# 13 + 4          (* 和 *);;
- : int = 17

# -(13 + 4)      (* 符号反転 *);;
- : int = -17

# (3 + 5) * 8   (* 括弧も使える *);;
- : int = 64
```

浮動小数点数の定数

1.25 という「式」を評価すると

```
# 1.25;;
- : float = 1.25
# 1.25e-2 (* 指数表記 *);;
- : float = 0.0125
```

1.25 という「値」になる

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

- 整数と浮動小数点数は混ぜられない

```
# 13 +. 4.0;;
Characters 0-2:
 13 +. 4.0;;
 ^^
```

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

注意

- 整数と浮動小数点数は混ぜられない

```
# 13 + 4.0;;
Characters 5-8:
 13 + 4.0;;
            ^^^
```

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

注意

- 整数と浮動小数点数は混ぜられない

```
# 13.0 + 4.0;;
Characters 0-4:
 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 13;;
- : float = 13.
```

- int_of_float

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

文字列

- 「"」で囲う
- 連接は「^」

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

真偽値

true という「式」を評価すると

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

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 = true
```

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

条件分岐

- o if e then e_1 else e_2

```
# if true then 2 else 3;;
- : int = 2
# if false then 2 else 3;;
- : int = 3
# if 3 < 2 then 2 else 3;;
- : int = 3
```

基本的な構文

コメント

基本型と演算

変数、関数と再帰

組、リスト

変数の定義（トップレベル）

変数

式

- let x = e
 - トップレベル定義 자체は式ではない

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

注意

- 変数は書き変えられない

```
# let x = 3;;
val x : int = 3
# let y = 4 * x;;
val y : int = 12
# let x = 10;;
val x : int = 10
# y;;
- : int = 12
```

注意

- 変数は書き変えられない

```
# let x = 3;;
val x : int = 3
# let y = 4 * x;;
val y : int = 12
# let x = 10;;
val x : int = 10
# y;;
- : int = 12
```

変数の定義（局所定義）

xはこの式の中でのみ有効

- `let x = e1 in e2`
 - 式 e_1 を評価して結果を x に束縛し、 e_2 を評価

```
# 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

- $\text{let } x = e_1 \text{ in } e_2$ は式
- $\text{let } x = e_1$ は式ではない

```
# let x = (let y = 3);;  
Characters 18-19:  
  let x = (let y = 3);;  
                      ^
```

Error: Syntax Error: operator expected

```
# let x = (let y = 3 in y+4) in 2*x;;  
- : int = 14
```

関数

○ `fun x -> e`

```
# fun x -> x + 1;;
- : int -> int = <fun>
```

2引数関数

```
# fun x -> fun y -> x + y;;
- : int -> int -> int = <fun>
```

```
# fun x y -> x + y;;
- : int -> int -> int = <fun>
```

上の略記

関数の適用

- $e \ e_1 \ e_2 \dots e_n$

```
# (fun x -> x + 1) 2;;
- : int = 3
# (fun x y -> x + y) 5 12;;
- : int = 17
```

関数の束縛

- 関数は「値」 \Rightarrow let で変数に束縛可能

```
# let inc = fun x -> x + 1;;
val inc : int -> int = <fun>
# inc 2;;
- : int = 3
# let add = fun x y -> x + y;;
val add : int -> int -> int = <fun>
# add 5 12;;
- : int = 17
```

略記

```
# let inc = fun x -> x + 1;;
val inc : int -> int = <fun>
# let inc x = x + 1;;
val inc : int -> int = <fun>
```

上の略記

```
# let add = fun x y -> x + y;;
val inc : int -> int -> int = <fun>
# let add x y = x + y;;
val inc : int -> int -> int = <fun>
```

上の略記

部分適用

- 関数に引数の一部だけを与える

```
# let add x y = x + y;;
val add : int -> int -> int = <fun>
# let f = add 40;;
val f : int -> int = <fun>
# f 2;;
- : int = 42
# f 7;;
- : int = 47
```

再帰関数

- let rec x = e

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

再帰関数

- let rec x = e

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

気づきにくい間違い

```
# let rec fact n =
  if n = 1 then 1 else
    n + fact (n - 1);; おっと間違えた
val fact : int -> int = <fun>
# let fact n =
  if n = 1 then 1 else
    n * fact (n - 1);; 直したつもり
val fact : int -> int = <fun>
# fact 10;;
- : int = 450 ??
```

気づきにくい間違い

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

注意

- 関数適用は整数演算よりも強く結合する

```
# let rec fact n =
  if n=0 then 1
    else n * fact n-1;;
val fact : int -> int = <fun>
# fact 10;;
Stack overflow during evaluation
(looping recursion?)
```

注意

- 関数適用は整数演算よりも強く結合する

```
# let rec fact n =
  if n=0 then 1
    else n * fact (n-1);;
val fact : int -> int = <fun>
# fact 10;;
- : int = 3628800
```

括弧が必要

相互再帰

- let rec $x_1 = e_1$ and $x_2 = e_2 \dots$

```
# 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
```

基本的な構文

コメント

基本型と演算

変数、関数と再帰

組、リスト

組 (tuple)

- (e_1, e_2, \dots, e_n)
 - 異なりうる型の要素を**固定個**まとめる

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

組の分解

- let で

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

使わない要素は _ で無視

リスト

- $[e_1; e_2; \dots; e_n]$
 - 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; 5];;
- : int list = [1; 2; 3; 4; 5]
```

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, true);;
- : int * float * bool = (1, 2., true)
```

- 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: ...
```

パターンマッチング

パターンマッチング

- 値の「パターン」で処理を分岐させること
 - `match e with | p1 -> e1`
 | p₂ -> e₂ ...
 - p₁, p₂, … : パターン
- 上から順にマッチするか試す
- マッチしたら対応する式を評価

```
# let rec sum xs =
  match xs with
  | []      -> 0
  | y :: ys -> y + sum ys;;
val sum : int list -> 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
 - 定数と比較（一致すればマッチ）
- 変数パターン
 - 任意の値とマッチ
 - マッチした値は本体中で使用可能

```
# 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マッチすればOK
 - 変数パターンとの組み合わせで要素を取り出せる

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

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 = <fun>
# sum [1; 2; 3; 4];;
- : int = 10
```

ワイルドカード

- 任意の値にマッチ
 - マッチした値を利用しないことの明示

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

ワイルドカード

- 任意の値にマッチ
 - マッチした値を利用しないことの明示

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

関数定義におけるパターンマッチ

- let (rec) や fun の仮引数部分には
パターンが書ける

```
# let fst (x, _) = x
val fst : 'a * 'b -> 'a = <fun>
# fst (1, 2.0);;
- : int = 1
# (fun (x, y) -> (4*x, 4*y)) (2, 3);;
- : int * int = (8, 12)
```

網羅的でないパターンマッチング

- 網羅的でない = マッチしない値がある
- 定義はされるが、実行時例外が出る

```
# let head xs =
  match xs with
  | 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; 3; 4];;
- : int = 1
# head [];;
Exception: Match_failure("//toplevel//", 3, -16).
```



警告



例外の発生

パターンの注意 (1)

- 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

- パターンに条件を追加

```
# 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できない
 - 変数は変数パターンとなる

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

レポート課題について

課題の種類と配点

- 問
 - 全員が取り組むべき課題
 - 配点: 70点/回 × 13回 = 910点

- 発展
 - 問ではものたりない人向け
 - 解くことは必須ではない（解けたら加点）
 - マニュアルや論文を読む必要があることも
 - 配点: 30点/回 × 13回 = 390点

締切・質問

- 出題後 2 週間以内に提出（毎回期限を明示）
- 締切厳守
 - 締切を過ぎた課題の提出については、質問メールアドレスまで連絡してください
- 質問メールは以下のアドレスまで：
fl-enshu@kb.is.s.u-tokyo.ac.jp
- 講義時間中にも質問ができます：
 - その週以外の講義内容についてでも構いません

提出物の形式

Written in Japanese or English

- レポート本文
 - プレーンテキスト（拡張子 .txt）または PDF
- 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
- **ファイル名は以下のようにすること**
 - 出題回-学籍番号下 6 桁.拡張子
 - 空白および非ASCII文字は避ける

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ように
 - (不要と書かれていなければ) 考察
- ソースコード一式
 - 適切なコメント
 - 意図や動作が分かるように
 - (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

- 構文エラー/型エラーがないことを確認しよう
- 「エラーが期待通りの結果」の場合は、動作例や考察できちんと議論すること

提出方法

- サポートページから

<http://www.kb.is.s.u-tokyo.ac.jp/~tsukada/cgi-bin/m/>

- 利用にはアカウントが必要
- アカウントの発行は上記ページ「ログイン」から
- その後「コース登録」を行う
 - この講義は「関数論理型プログラミング実験 2017」

例題

理解の確認をするための課題です

これに解答することで、出席とします

分からぬことがあつたら、積極的に質問しましょう

例題 1

- 半径（浮動小数点数）を受け取り、円の面積を返す関数 circle を書け

circle : float -> float

- 上で定義した circle を使って、
 - 半径 10.0 の円の面積を求めよ
 - 半径 15.0 の円の面積を求めよ

```
# circle 10.0;;
- : float = 314.159
```

例題 2

- 整数関数 f と正の整数 n を受け取り、
$$f(0) + f(1) + \cdots + f(n)$$
を返す関数 sigma を書け
 $\text{sigma} : (\text{int} \rightarrow \text{int}) \rightarrow \text{int} \rightarrow \text{int}$
- sigma を使って $\sum_{k=0}^{10} k^2 + k$ を求めよ

$$\sum_{k=0}^{10} k^2 + k = 440$$

例題 3

- 関数 f と
リスト $[x_1; x_2; \dots; x_n]$ を受け取り、
リスト $[f x_1; f x_2; \dots; f x_n]$ を返す
関数 map を書け
- $\text{map} : ('a \rightarrow 'b) \rightarrow 'a \text{ list} \rightarrow 'b \text{ list}$

```
# map (fun x -> x * x) [1; 2; 3];;
- : int list = [1; 4; 9]
```

レポート課題 1

締切：2019/4/23 13:00(JST)

問 1 (考察不要)

- 以下の関数を書け。
 - 正の整数 n を受け取って、0 から n までの整数の和を求める関数
`sum_to : int -> int`
 - 正整数 n が素数かどうかを判定する関数
`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 (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 と正整数 n を受け取って、
 f を n 回合成した関数を返す `repeat`
 - 例 : `repeat (fun x -> 2*x) 4 3 = 48`

問 3

$$\text{fix } f \ x = f(\text{fix } f) \ x$$

$\text{fix}: f \in x \vdash f$ を返す。

$$\begin{array}{c} \text{fix calc } a \ b \\ \downarrow \end{array}$$

不動点コンピュータの仕組み

- 関数 fix を以下で定義する

```
# let rec fix f x = f (fix f) x;;
val fix : (('a -> 'b) -> 'a -> 'b) -> 'a
```

$\text{fix } f$

この関数以外の再帰関数を使わずに問 1 の関数を実装せよ

$$\begin{aligned} & \text{fix } f \ 3 \\ &= f(\text{fix } f) \ 3 \\ &= 3 + \text{fix } f \ 2 \\ &= 3 + 2 + f(\text{fix } f) \ 1 \\ &= 3 + 2 + 1 + f(\text{fix } f) \ 0 \end{aligned}$$

$$\begin{aligned} & \text{fix } f \ 4 \\ &= f(\text{fix } f) \ 3 \\ &= \text{fix } f \ 3 . \xrightarrow{\text{false}} \end{aligned}$$

$$\begin{array}{c} \xrightarrow{\quad} \xrightarrow{\quad} b' \\ \text{fix } f \ x \\ \xrightarrow{\quad} \xrightarrow{x \rightarrow b'} \\ f \\ \xrightarrow{\quad} \text{fix } f \end{array}$$

問 4

- 以下を満たす `fold_right` を実装せよ

$$\begin{aligned} \text{fold_right } f \ [x_1; x_2; \dots; x_n] \ e \\ = f \ x_1 \ (f \ x_2 \ (\dots \ (f \ x_n \ e) \ \dots)) \quad (n \geq 0) \end{aligned}$$

- 以下を満たす `fold_left` を実装せよ

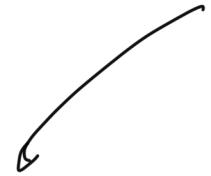
$$\begin{aligned} \text{fold_left } f \ e \ [x_1; \dots; x_{n-1}; x_n] \\ = f \ (f \ (\dots \ (f \ e \ x_1) \ \dots) \ x_{n-1}) \ x_n \quad (n \geq 0) \end{aligned}$$

- 未尾再帰

この「未尾再帰」

問 5

リストからを取り除かない



- 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

- @を使用しないこと
 - 判定関数とリストを受け取り、リストの要素のうち判定条件を満たす要素だけから成るリストを生成する関数 filter

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

$f, e, [x]$ $\mapsto f(f(\dots(f(x_1)\dots)x_{n-1})x_n)$ f_left

問 6 $a \in b$ を append $\rightarrow e$ を a に、 x_i を b の i 番目 \rightarrow 不可 \rightarrow なぜなら b は逆順 \downarrow
 $e \in b, x_i$ を b の逆順要素 \rightarrow $reversed$ のじゅう.

- 問 5 の関数を `fold_left` を用いて実装せよ
- 問 5 の関数を `fold_right` を用いて実装せよ
- 問 5 の実装と問 6 の実装を比較せよ
 - 書きやすさとか
 - 実行速度とか

`fold-right` & `fold-left` は C-like 文法で書かれている.

f_right

f, \vec{x}, e

$\hookrightarrow f(x_1(f(x_2(\dots(f(x_n, e) \dots)))$

元々 a, e が "b"

への要素 \rightarrow $n!$ 通りの並べ方.

TYPE OCaml a #11

問 7

$$n \text{ の要塞の下限 } = \binom{n \text{ の上端}}{\text{下限}} : \binom{n-1 \text{ の上端}}{\text{下限}} \quad \text{or} \quad \binom{n-1 \text{ の上端}}{\text{下限}} @ \binom{n \text{ の}}{n-1 \text{ の上端}}$$

- 。リストを受け取り全ての順列のリストをリストにして返す関数

`perm: 'a list -> 'a list list`

gcc = ジェーシーシー

を実装せよ

$$a \in \text{Int}(U_{\eta, T=K}) \cdot a + \binom{n-1}{\lfloor \frac{n}{m} \rfloor}$$

- 引数のリストには重複がないと仮定してよい

```
# perm [1];;
```

```
- : int list list = [[1]]
```

```
# perm [1;2];;
```

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

発展 1



- リストを受け取り、逆順のリストを返す
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
- reverse 関数を fold_right を用いて書け
 - 入力リストの長さに対し線形時間で動作すること
 - ※ なお、代わりに fold_left を使うなら簡単

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

発展 1 の補足

- リストの連接 $xs @ ys$ は
 xs について線形時間かかることに注意

$fold-right$ + $x e$

$f x_1 (f x_2 (\dots (f x_n e) \dots))$

右から左へ評価する.

x_1 が $x_2 \dots x_n$ に適用される.

$f x_i h = h \circ x_i^*$

右から左へ評価は可能.

$h \circ x_i^*$

$f x_1 (f \underset{\downarrow}{x_2} e^*)$

$f x_1 (e^* \underset{\downarrow}{x_2}^*)$

$e^* x_2^* x_1^* []$

発展 2

$\text{fold_left } f e \vec{x}$
 $= f(f(\dots(f(e, x_1) \dots) x_{n-1})) x_n$

$\underbrace{g x_1(g x_2 e^*)}_{\downarrow} \quad \text{初期値を } g \text{ に置く。}$
 $\underbrace{g x_1(e^* x_2^*)}_{\downarrow} \quad \text{g は引数 } 2. \text{ (かんすう)}$
 $e^* x_2^* x_1^* \quad \text{左から取る。} \quad \text{fex, e* と入れ替わる。}$

○ fold_left を fold_right を用いて書け

- 末尾再帰にならないが気にしないでよい

$e^* x_2^* (fex)$
 $\underbrace{e^* f(fex,) x_2}_{\downarrow}$

○ fold_right を fold_left を用いて書け

$g(g(\dots(g(e, x_1) \dots) \dots) x_{n-1})) x_n$

$\underbrace{g(g(e, x_1)) x_2}_{\downarrow} \quad \text{eを右側へ}$
 $\underbrace{g(g(e, x_1^*)) x_2}_{\downarrow} \quad \rightarrow x_2 e を入れ込む$
 $e^* x_1^* x_2^* \quad \text{e* x_1^* x_2^*} \quad \text{e* x_1^* f x e}$
 $\underbrace{e^* x_1^* f x e}_{\downarrow} \quad \text{f x e}$
 $f x_1(f x_2 e)$

発展 3

n個

○ $\underline{n} = \text{fun } f \ x \ -> \ f \ (\underbrace{f \ (\dots \ (f \ x) \ \dots)}_{n\text{個}}))$

と定義する。以下の条件を満たす関数を書け

- $\text{add } \underline{n} \ \underline{m} = \underline{n+m}$ ← 全成りんすうにやればよい。
- $\text{mul } \underline{n} \ \underline{m} = \underline{n \times m}$ ← fold-left とかで順行うのが。
- $\text{sub } \underline{n} \ \underline{m} = \underline{n-m}$ ← ある。

```
# add (fun f x -> f x)
      (fun f x -> f (f x))
      (fun y -> y + 1)
      0;;
- : int = 3
```

発展 3 の補足

○ $n - m$ の定義

$$\underline{n - m} = \begin{cases} \underline{k}, & n \geq m, k = n - m \\ \underline{0}, & n < m \end{cases}$$

add n m = n + m

(fun f x → λ f(m f x))

mul n m = (fun f x →