

# 関数・論理型プログラミング実験

## 第4回

江口 慎悟  
酒寄 健  
塚田 武志  
松下祐介

# 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
- 名前/学籍番号/希望アカウント名をメールを  
    tsukada@kb.is.s.u-tokyo.ac.jp  
    までメールしてください。
  - 件名は「FL/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取れるように

# 今日の内容

- 副作用とモナド
- 副作用から値へ
- 計算の繋げ方
- モナドによる副作用の表現

# 副作用とモナド

副作用から値へ

計算の繋げ方

モナドによる副作用の扱い

# OCaml の int型 と副作用

- OCaml の int 型のプログラムには、  
整数値を返す以外の副作用がありえる
  - 入出力
  - 参照の書き換え
  - 例外の生成

```
# let r = ref 0;;
r : int ref = { contexts = 0 }

# let f x = let z = !r in (r := x; z);;
f : int -> int = <fun>

# f 10;;
- : int = 0
```

# 副作用の功罪

- 上手く使うとパフォーマンス / 可読性が向上
- 入出力は必須の機能
- プログラムを理解しやすくする
  - 秋にコンパイラを作る際に実感するかも？

```
let x = f 10 in  
let y = g 20 in  
y + 5
```

一見すると無駄だけど  
消してもよいかは不明  
(= f の中身に依る)

オナニスト いじめよめもしない。  
ごこじどんなんのいじめ) べ起もつてみから まぐな。

# 副作用とモナド

副作用から値へ

計算の繋げ方

モナドによる副作用の扱い

# 代替案：「副作用の情報込み」の値

- 副作用のあるint型の式

⇒ 副作用のない複雑な型の式

副作用で何が起きるかの情報を値に押し込む

例:

- 失敗するかもしれない int 型の式  
⇒ 副作用のない int option 型の式
- string 型の参照を使った int 型の式  
⇒ 副作用のない string -> (int \* string) 型の式
- 非決定性分岐のある int 型の式  
⇒ 副作用のない int list 型の式

# 例 1 : 失敗しうる計算

## ○ テーブルの lookup

```
(* myLookup : 'a -> ('a * 'b) list -> 'b *)
```

```
let rec myLookup key xs =  
  match xs with  
  | []           -> raise Not_found  
  | ((k,v)::rest) ->  
    if key = k then v  
    else myLookup key rest
```

失敗して終了するか

値を返すか

key が一致しない場合は、

key の list に入らない場合である。

# 例 1 : 失敗しうる計算

## ○ テーブルの lookup

```
(* myLookup : 'a -> ('a * 'b) list -> 'b *)
let rec myLookup key xs =
  match xs with
  | []           -> raise Not_found
  | ((k,v)::rest) ->
    if key = k then v
    else myLookup key rest
```

失敗して終了するか

値を返すか

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

値を返す

失敗して終了する

決して失敗しない関数になります。

# 例 1：失敗しうる計算

## ○テーブルの lookup

```
(* myLookup : 'a -> ('a * 'b) list -> 'b option *)  
let rec myLookup key xs =  
  match xs with  
  | []           -> None  
  | ((k,v)::rest) ->  
    if key = k then Some v  
    else myLookup key rest
```

失敗して終了するか

値を返すか

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

値を返す

失敗して終了する

# 例 2 : 例外が発生する計算

- 整数の割り算

```
(* myDiv : int -> int -> int *)
let myDiv x y =
  if y = 0 then raise (Failure "Div by Zero")
  else x / y
```

例外を発行するか  
値を返すか

# 例 2：例外が発生する計算

## ○ 整数の割り算

```
(* myDiv : int -> int -> int *)
let myDiv x y =
  if y = 0 then raise (Failure "Div by Zero")
  else x / y
```

例外を発行するか  
値を返すか

```
type 'a myErr = Ok of 'a | Err of string
```

値を返す

例外を発行する

# 例 2：例外が発生する計算

## ○ 整数の割り算

```
(* myDiv : int -> int -> int myErr *)
```

```
let myDiv x y =  
  if y = 0 then Err "Div by Zero"  
  else Ok (x / y)
```

例外を発行するか

値を返すか

```
type 'a myErr = Ok of 'a | Err of string
```

値を返す

例外を発行する

# 例 3 : 状態を持つ計算

- 参照に値を足す、参照を読む

```
(* addVal : int -> unit *)
(* getVal : unit -> int *)

let r          = ref 0
let addVal x  = (r := !r + x)
let getVal () = !r
```

参照を読んで  
参照に書いて  
値を返す

# 例 3：状態を持つ計算

- 参照に値を足す、参照を読む

```
(* addVal : int -> unit *)
(* getVal : unit -> int *)
let r          = ref 0
let addVal x  = (r := !r + x)
let getVal () = !r
```

参照を読んで  
参照に書いて  
値を返す

```
type 'a int_state = int -> ('a * int)
```

返値

参照の古い値

参照の新しい値

# 例 3 : 状態を持つ計算

- 参照に値を足す、参照を読む

```
(* addVal : int -> unit int_state *)
(* getVal : unit -> int int_state *)
```

この書き込みよこへ  
参照の書き込みが別書き

参照を読んで  
参照に書いて  
値を返す

```
let addVal x = (fun i -> ((), i+x))
let getVal () = (fun i -> (i, i))
```

参照の古い値

返値

参照の新しい値

返値

```
type 'a int_state = int -> ('a * int)
```

参照の古い値

参照の新しい値

# 副作用とモナド

副作用から値へ

計算の繋げ方

モナドによる副作用の扱い

# 計算を繋げる

## 例 1：失敗しうる計算

- lookup の結果を鍵に lookup

```
let myLookup2 k1 t1 t2 =
  match myLookup k1 t1 with
  | None    -> None
  | Some k2 -> myLookup k2 t2
```

# 計算を繋げる

## 例 1：失敗しうる計算

- lookup の結果を鍵に lookup  
した結果を鍵に lookup

```
let myLookup3 k1 t1 t2 t3 =
  match myLookup k1 t1 with
  | None    -> None
  | Some k2 -> (match myLookup k2 t2 with
                  | None    -> None
                  | Some k3 -> myLookup k3 t3)
```

# 計算を繋げる

## 例 1：失敗しうる計算

- `lookup` の結果を鍵に `lookup` した結果を鍵に `lookup`

```
let myLookup3 k1 t1 t2 t3 =
  match myLookup k1 t1 with
  | None      -> None
  | Some k2   -> (match myLookup k2 t2 with
                    | None      -> None
                    | Some k3   -> myLookup k3 t3)
```

- 類似のパターンの繰り返し

# 計算を繋げる

## 例 2：例外が発生する計算

- $x / y / z$  の計算

```
let myDiv2 x y z =  
  match myDiv x y with  
  | Err msg -> Err msg  
  | Ok v      -> myDiv v z
```

# 計算を繋げる

## 例 2：例外が発生する計算

- $x / y / z / u$  の計算

```
let myDiv3 x y z u =
  match myDiv x y with
  | Err msg -> Err msg
  | Ok v    -> (match myDiv v z with
                  | Err msg -> Err msg
                  | Ok w    -> myDiv w u)
```

- 類似パターンの繰り返し

# 計算を繋げる

## 例 2：例外が発生する計算

- $x / y / z / u$  の計算

```
let myDiv3 x y z u =
  match myDiv x y with
  | Err msg -> Err msg
  | Ok v    -> (match myDiv v z with
                  | Err msg -> Err msg
                  | Ok w    -> myDiv w u)
```

- 類似パターンの繰り返し

# 計算を繋げる

## 例 3：状態を持つ計算

- addVal 3; addVal 7 の計算

```
let addAddVal =
  fun init ->
    let (r1, s1) = addVal 3 init in
    let (r2, s2) = addVal 7 s1   in
      (r2, s2)
```

↑新しい状態のs1をx+7...`

# 計算を繋げる

## 例 3：状態を持つ計算

- addVal 3; addVal 7; getVal () の計算

```
let addAddGetVal =  
  fun init ->  
    let (r1, s1) = addVal 3 init in  
    let (r2, s2) = addVal 7 s1 in  
    let (r3, s3) = getVal () s2 in  
      (r3, s3)  
      ↑  
      s2がcurrent...  
      ...
```

# 計算を繋げる

## 例 3：状態を持つ計算

- addVal 3; addVal 7; x = getVal (); addVal x の計算

```
let addAddGetAddVal =
  fun init ->
    let (r1, s1) = addVal 3 init in
    let (r2, s2) = addVal 7 s1 in
    let (x, s3) = getVal () s2 in
    let (r4, s4) = addVal x s3 in
      (r4, s4)
```

- 類似パターンの繰り返し

# 計算を繋げる

## 例 3：状態を持つ計算

- addVal 3; addVal 7; x = getVal (); addVal x の計算

```
let addAddGetAddVal =
  fun init ->
    let (r1, s1) = addVal 3 init in
    let (r2, s2) = addVal 7 s1 in
    let (x, s3) = getVal () s2 in
    let (r4, s4) = addVal x s3 in
      (r4, s4)
```

- 類似パターンの繰り返し

# 計算を繋げる

## 例 3：状態を持つ計算

- addVal 3; addVal 7; x = getVal (); addVal x の計算

```
let addAddGetAddVal =
  fun init ->
    let (r1, s1) = addVal 3 init in
    let (r2, s2) = addVal 7 s1 in
    let (x, s3) = getVal () s2 in
    let (r4, s4) = addVal x s3 in
      (r4, s4)
```

- 類似パターンの繰り返し

# 計算を繋げる

## 例 3：状態を持つ計算

- addVal 3; addVal 7; x = getVal (); addVal x の計算

```
let addAddGetAddVal =
  fun init ->
    let (r1, s1) = addVal 3 init in
    let (r2, s2) = addVal 7 s1 in
    let (x, s3) = getVal () s2 in
    let (r4, s4) = addVal x s3 in
      (r4, s4)
```

- 類似パターンの繰り返し

# 計算を繋げる

## 例 3：状態を持つ計算

- addVal 3; addVal 7; x = getVal (); addVal x の計算

```
let addAddGetAddVal =
  fun init ->
    let (r1, s1) = addVal 3 init in
    let (r2, s2) = addVal 7 s1 in
    let (x, s3) = getVal () s2 in
    let (r4, s4) = addVal x s3 in
      (r4, s4)
```

- 類似パターンの繰り返し

# 計算を繋げる

## 例 3：状態を持つ計算

- addVal 3; addVal 7; x = getVal (); addVal x の計算

```
let addAddGetAddVal =
  fun init ->
    let (r1, s1) = addVal 3 init in
    let (r2, s2) = addVal 7 s1 in
    let (x, s3) = getVal () s2 in
    let (r4, s4) = addVal x s3 in
      (r4, s4)
```

- 類似パターンの繰り返し

# 計算を繋げる

## 例 3：状態を持つ計算

- addVal 3; addVal 7; x = getVal (); addVal x の計算

```
let addAddGetAddVal =
  fun init ->
    let (r1, s1) = addVal 3 init in
    let (r2, s2) = addVal 7 s1 in
    let (x, s3) = getVal () s2 in
    let (r4, s4) = addVal x s3 in
      (r4, s4)
```

※ 初期値を与えると計算結果を得られる

```
# addAddGetAddVal 10;;
- : unit * int = ((), 40)
```

↓  
めんと“くわいひじ。  
一定のパラメータの抽出(け)だし。

# 副作用とモナド

副作用から値へ

計算の繋げ方

モナドによる副作用の扱い

# 一般的な計算の繋げ方

- 副作用のある計算の繋げ方には一定のパターンがあった

- ただし副作用ごとに異なる

この概念を

- 「繋げ方」を一般化・抽象化したら？

⇒ モナド

型とつなげ方を含めてモナドという。

# モナド

- 以下の演算を持つような型構成子 ' $a\ m$

- Bind : 副作用のある計算の繋げ方

```
(>>=) : 'a m -> ('a -> 'b m) -> 'b m  
         わかはず    何かをやつして、    わかれはす.
```

- Return : 副作用のない計算 (=値を返すだけ)

```
return : 'a -> 'a m
```

# Bind ( $>>=$ ) の例 1

- 'a option 型の値が、
  - None なら、 None を返す
  - Some k なら、 k を引数に続きの計算を呼び出す

```
(>>=) : 'a option -> ('a -> 'b option) -> 'b option
```

```
let (>>=) x f = match x with
  | None    -> None
  | Some k -> f k
```

↑  
引数  
↓  
戻り値

```
let myLookup3 k1 t1 t2 t3 =
  myLookup k1 t1 >>= (fun k2 ->
    myLookup k2 t2 >>= (fun k3 ->
      myLookup k3 t3))
```

# Bind (>>=) の例 1

- 'a option 型の値が、
  - None なら、None を返す
  - Some k なら

```
(>>=) : 'a o  
let (>>=) x
```

cf. (心の目)

```
let k2 = myLookup k1 t1 in  
let k3 = myLookup k2 t2 in  
    myLookup k3 t3
```

```
let myLookup3 k1 t1 t2 t3 =  
    myLookup k1 t1 >>= (fun k2 ->  
        myLookup k2 t2 >>= (fun k3 ->  
            myLookup k3 t3))
```

# Bind (>>=) の例 2

- 'a myErr 型の値が、
  - Err msg なら、 Err msg を返す
  - Ok y のときは、 y を引数に続きの計算を呼び出す

```
(>>=) :: 'a myErr -> ('a -> 'b myErr) -> 'b myErr
```

```
let (>>=) x f = match x with
    | Err msg -> Err msg
    | Ok y      -> f y
```

```
let myDiv3 x y z u =
    myDiv x y >>= (fun v ->
        myDiv v z >>= (fun w ->
            myDiv w u))
```

# Bind (>>=) の例 2

- 'a myErr 型の値が、

- Err msg なら cf.
- Ok y のとき cf.

```
(>>=) :: 'a myErr -> 'a myErr -> 'a myErr
let (>>=) x = let v = myDiv x y in
let w = myDiv v z in
myDiv w u
```

```
let myDiv3 x y z u =
  myDiv x y >>= (fun v =>
  myDiv v z >>= (fun w =>
  myDiv w u))
```

# Bind (>>=) の例 3

- 状態の初期値 init を受け取る
- init を初期値として、x を評価。
  - 返値を r、新しい状態を s とする
- 初期値を s として、fx を評価

```
(>>=) :  
'a int_state -> ('a -> 'b int_state) -> 'b int_state  
  
let (>>=) x f =  
  fun init ->  
    let (r, s) = x init in  
      f r s
```

# Bind (>>=) の例 3

- 状態の初期値 init を受け取る
- init を初期値として、x を評価。
  - 返値を r、新しい状態を s とする
- 初期値を s として、fr を評価

(\* プログラム例 \*)

```
let addAddGetAddVal =
  addVal 3 >>= (fun _ ->
  addVal 7 >>= (fun _ ->
  getVal () >>= (fun x ->
    addVal x))))
```

# Bind (>>=) の例 3

- 状態の初期化
- init を初期化する  
• 返値を r、
- 初期値を s

(\* プログラム例 \*)

cf.

```
let _ = addVal 3 in
let _ = addVal 7 in
let x = getVal () in
    addVal x
```

```
let addAddGetAddVal =
  addVal 3 >>= (fun _ ->
  addVal 7 >>= (fun _ ->
  getVal () >>= (fun x ->
    addVal x)))
```

# Return の例

- 失敗しうる計算

```
let return x = Some x
```

- 例外を返しうる計算

```
let return x = Ok x
```

- 状態を持った計算

```
let return x = fun init -> (x, init)
```

# モナド

- 以下の演算を持つような型構成子 ' $a\ m$ '

- Bind : 副作用のある計算の繋げ方

```
(>>=) : 'a m -> ('a -> 'b m) -> 'b m
```

- Return : 副作用のない計算 (=値を返すだけ)

```
return : 'a -> 'a m
```

# その他のモナドの例

↑ たいてい primitive を直接使うのは  
いけうなじみ。

- 非決定的計算

- 例

- ~を満たすものを全て返せ
    - generate-and-test

- リストモナド

- ロギングする計算

- Writer モナド

イタリックなまま  
callcc

- 大域脱出を含む計算

- Continuation モナド

# モナド則

- モナドが満たさなければならない等式
  - `return` は「作用」をしない
  - グルーピングの仕方は、結果に影響しない
    - 順番は結果に影響する

$(\text{return } x) \gg= f = f x$

$m \gg= \text{return} = m$

$(m \gg= f) \gg= g$

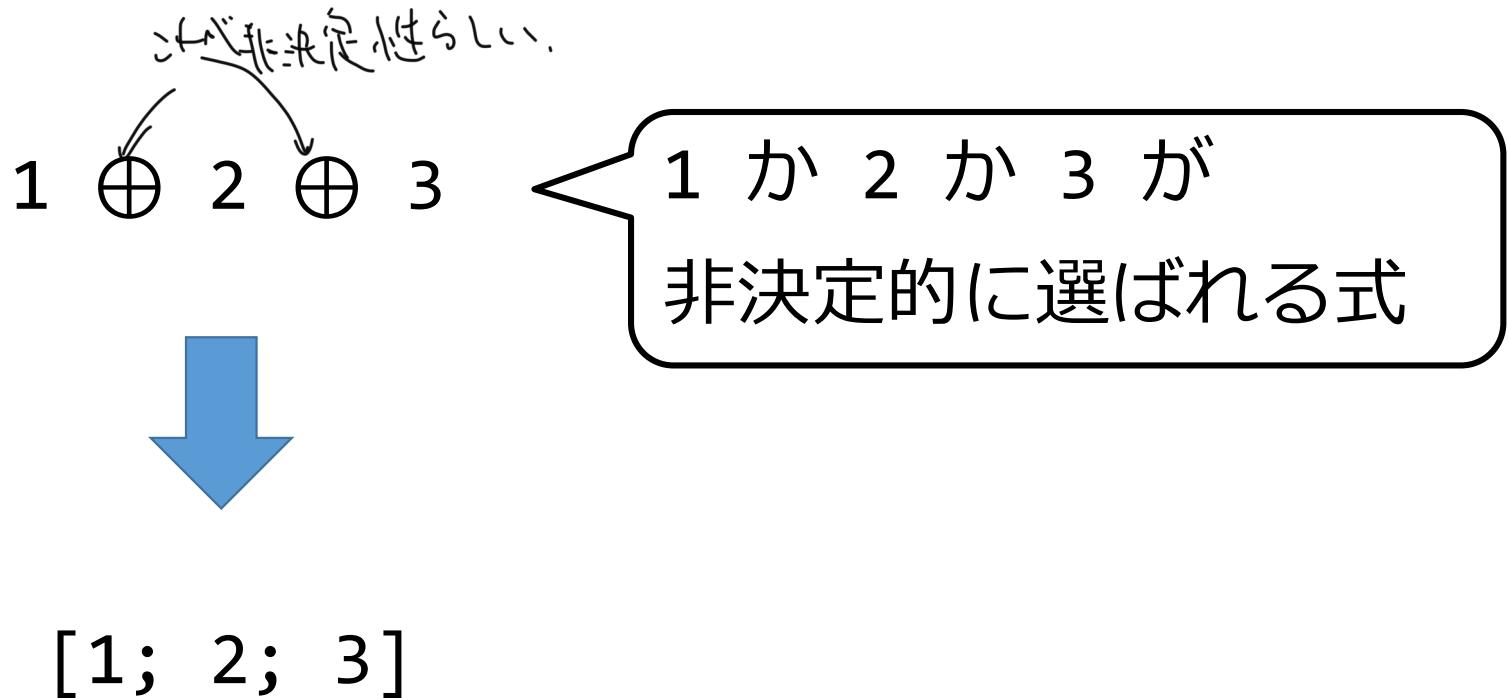
$= m \gg= (\text{fun } x \rightarrow f x \gg= g)$

この3つの  
モリクベ  
じゅくを  
可能といふ。

例：非決定的計算  
(リストモナド)

# 非決定的な計算を表現するアイデア

- 非決定的な計算を取りえる値のリストで表現



# プログラム例

```
# let find =
[1; 2; 3]           >>= (fun x ->
[4; 5; 6]           >>= (fun y ->
(guard (x + y > 7)) >>= (fun _ ->
    return (x, y))));;
- : (int * int) list = [(2, 6); (3, 5); (3, 6)]
```

```
cf. let find =  
    let x = 1 ⊕ 2 ⊕ 3 in  
    let y = 4 ⊕ 5 ⊕ 6 in  
    let _ = guard (x + y > 7) in  
        (x, y) 下の図を使うモードに特有、演算
```

# プログラム例

```
# let find =
[1; 2; 3]           条件に合わない選択肢を排除
[4; 5; 6]
(guard (x + y > 7)) >>= (fun _ ->
    return (x, y))));;
- : (int * int) list = [(2, 6); (3, 5); (3, 6)]
```

cf. let find =  
 let x = 1 ⊕ 2 ⊕ 3 in  
 let y = 4 ⊕ 5 ⊕ 6 in  
 let \_ = guard (x + y > 7) in  
 (x, y)

# プログラム例

```
# let find =
  [1; 2; 3]           >>= (fun x ->
  [4; 5; 6]           >>= (fun y ->
  (guard (x + y > 7)) >>= (fun _ ->
    return (x, y))));;
- : (int * int) list = [(2, 6); (3, 5); (3, 6)]
```

```
cf. let find =
      let x = 1 ⊕ 2 ⊕ 3 in
      let y = 4 ⊕ 5 ⊕ 6 in
      let _ = guard (x + y > 7) in
        (x, y)
```

# 実装

```
type 'a m = 'a list
```

リストの各要素に対し、

fを適用する

可能な値それぞれに f を適用

```
let (">>>=) x f = List.concat (List.map f x)
```

リストのリストをリストに変換

```
let return x = [x]
```

```
let guard b = if b then return () else []
```

# bind の実装の補足説明

```
# let x = [1; 10; 100];;
# let f = fun i -> [i - 1; i + 1];;
```



引数の整数に対して  
非決定的に 1 を足すか引く

```
# List.map f x;;
- : int list list = [[0; 2]; [9; 11]; [99; 101]]

# List.concat (List.map x f);;
- : int list = [0; 2; 9; 11; 99; 101]

# x >>= f;;
- : int list = [0; 2; 9; 11; 99; 101]
```

# 例題

理解の確認をするための課題です

課題提出システム上での提出の必要はありません

例題を解きTAに見せることで出席とします

分からぬことがあったら、積極的に質問しましょう

# 例題

- 3引数ブール関数を与えられたとき  
それを充足する割り当てを全て返す

```
sat : (bool -> bool -> bool -> bool)
      -> (bool * bool * bool) list
```

を非決定性モナドを使って書け

- 素朴な解法は次のようになる

```
let sat f = let x = true ⊕ false in
            let y = true ⊕ false in
            let z = true ⊕ false in
            let () = guard (f x y z) in
              (x, y, z)
```

# レポート課題 4

締切：2019/5/21 13:00(JST)

# 問 1

- キーが見つからないときに例外を返す

eLookup : 'a -> ('a \* 'b) list -> 'b myErr  
を定義せよ

- myDiv と eLookup を使って、  
以下の計算を行うプログラムを書け
  - 第一引数を eLookup でテーブルから引き、xとする
  - 第二引数を eLookup でテーブルから引く、yとする
  - myDiv x y を計算する  
(例外はモナドの機構が処理してくれるはず)

# 問 1

- キーが見つからないときに例外を返す

eLookup : 'a -> ('a \* 'b) list -> 'b myErr

を定義せよ

- myDiv と eLookup を使って、  
以下の計算を行うプログラムを書け

```
(* lookupDiv かぎり　ひざな　Table
   : 'a -> 'a -> ('a * int) list -> int myErr *)
lookupDiv kx ky t =
  (eLookup kx t) >>= (fun x ->
    ....
  )
```

# 問1 (つづき)

```
# let table = [("x", 6); ("y", 0); ("z", 2)];;
val table : (string * int) list = .....
```

```
# lookupDiv "x" "y" table;;
- : int myErr = Err "Div by Zero"
```

```
# lookupDiv "x" "z" table;;
- : int option = Ok 3
```

```
# lookupDiv "x" "b" table;;
- : int option = Err "Not found: b"
```

```
# lookupDiv "a" "z" table;;
- : int option = Err "Not found: a"
```

## 問 2

- 非決定性モナドを使って次の覆面算を解け

ばなな + ばなな = しなもん

(Wikipedia: 覆面算 より)

- 異なる文字に同じ数字を使ってもよい
- 先頭の文字に 0 が割り当てられてもよい
  - こういうことがない解に絞ってもよい
- 同様に次の覆面算を解け

実行時間は 3 秒以内

SEND + MORE = MONEY

- 異なる文字には異なる数字を割り当てること
- 先頭の文字 (= S と M) には 0 を割り当てない

# 問 3

- 文字列を書き込む副作用を表す  
Writer モナドを定義せよ

```
type 'a m = 'a * string
(>>=) : 'a m -> ('a -> 'b m) -> 'b m
return : 'a -> 'a m
writer : string -> unit m
```

- 配布資料を参考にせよ

# 問3 (つづき)

```
# let f x =
  (x+1, "call f(^(string_of_int x)^"), ");
val f : int -> int m = <fun>

# let g x =
  (2*x, "call g(^(string_of_int x)^"), ";
val g : int -> int m = <fun>

# (f 3) >>= (fun a ->
  (g a) >>= (fun b ->
    (f b) >>= (fun c ->
      return c))));;
- : int * string =
  (9, "call f(3), call g(4), call f(8), ")
```

# 問 4

- 整数型関数のメモ化をするモナドを定義せよ

```
type 'a m
(>>=) : 'a m -> ('a -> 'b m) -> 'b m
return : 'a -> 'a m
memo : (int -> int m) -> int -> int m
runMemo : 'a m -> 'a
```

- メモ化：最適化の一種。これまでの呼び出しの引数・返値の組を覚えておいて、新しい呼び出しが過去にあったものと同じなら、再計算せずに、記憶しておいた値を返す処理

# 問4 (つづき)

```
# let rec fib n =
  if n <= 1 then
    return n
  else
    (memo fib (n-2)) >>= (fun r1 ->
      (memo fib (n-1)) >>= (fun r2 ->
        return (r1 + r2)))
val fib : int -> int m
# runMemo (fib 80);;
- : int = 23416728348467685
```

この計算がすぐに終わるように

# 発展 1

- 講義で出てこなかったモナドを探し、  
実装した上で、使用法などを解説せよ
  - 例えば次のサイトは参考になるかもしれない

[https://wiki.haskell.org/Monad#Interesting\\_monads](https://wiki.haskell.org/Monad#Interesting_monads)

# 発展 2

リストをモナド化  
する機能ある。

- 'a list 型がモナドになるように

```
(>>=)  : 'a list -> ('a -> 'b list) -> 'b list
return : 'a -> 'a list
```

を定義せよ

- ただし講義資料のモナドと異なるように
  - ( $>>=$ )だけ異なればよい
- モナド則を満たすように注意すること
- どのような計算を表現しているか