

# プログラミング言語 No.2-4

2025.5.9 中川博之

# 最終レポート

- 解くべき問題を作成し, その問題を解く関数をsml上で定義せよ
- レポートに含めるもの
  - 扱う問題
  - プログラム(関数) とその解説
  - 講義で学んだどのような技術を用いたか
  - 工夫した点, アピールポイント
  - SMLや講義に関する所感
- オリジナリティが高いほど, 難易度が高いほど加点する
- 提出締切: 6月4日(水) 23:59

## [再掲] 問題 2-3-5)

「ピッグ・ラテン(Pig Latin)」へ翻訳する簡単な規則は、母音で始まる単語には“yay”を付加し、1つ以上の子音で始まる単語にはその子音を後ろに回してから“ay”を追加する。例えば、“able”は“ableyay”になり、“stripe”は“ipestray”となる。文字列をピッグ・ラテンに変換する関数を書け。

- 母音の無い単語は考慮しなくて良い

各関数をMLで定義せよ（ミニレポート）

## [再掲] 問題 2-3-5) を行うために

- 文字のリストを取り，最初の要素が母音であったら true を，そうでなければ false を返す関数 vowel
  - SML/NJ では文字は #`"a"` のように表す
- 以下の関数も役に立つかもしれない
  - - `explode("abc");`  
val it = [`#"a"`,`#"b"`,`#"c"`] : char list
  - - `implode([#"a",#"b",#"c"]);`  
val it = `"abc"` : string

```
fun vowel(#"a"::_) = true
  | vowel(#"e"::_) = true
  | vowel(#"i"::_) = true
  | vowel(#"o"::_) = true
  | vowel(#"u"::_) = true
  | vowel(#"A"::_) = true
  | vowel(#"E"::_) = true
  | vowel(#"I"::_) = true
  | vowel(#"O"::_) = true
  | vowel(#"U"::_) = true
  | vowel(_) = false;
```

## 解答例：問題 2-3-5)

子音用



```
fun piglatin2(x::xs) = if vowel(x::xs) then x::xs@["a", "y"]  
                        else piglatin2(xs@[x]);
```

```
fun piglatin [] = []  
  | piglatin(x::xs) = if vowel(x::xs) then x::xs@["y", "a", "y"]  
                      else piglatin2(xs@[x]);
```

```
fun PigLatin(L) = implode(piglatin(explode(L)));
```

「ピッグ・ラテン(Pig Latin)」へ翻訳する簡単な規則は、母音で始まる単語には“yay”を付加し、1つ以上の子音で始まる単語にはその子音を後ろに回してから“ay”を追加する。例えば、“able”は“ableyay”になり、“stripe”は“ipestray”となる。文字列をピッグ・ラテンに変換する関数を書け。母音の無い単語は考慮しなくて良い

- 局所環境を使っても、もちろん良い
- 組み込み関数 concat を用いても良い

## 解答例：問題 2-3-6)

べき集合を計算する関数 powerset を, let を使い尾部(tail) のべき集合の計算を一度に済ませるように修正せよ.

(既出) 局所環境を使わない場合の解答例：

```
fun powerset(nil) = [nil]
  | powerset(x::xs) = powerset(xs)@inserteach(x,powerset(xs));
```

局所環境を使った解答例：

```
fun PowerSet(nil) = [nil]
  | PowerSet(x::xs) =
    let
      val s = PowerSet(xs)
    in
      s@inserteach(x,s)
    end;
```

inserteach は問題 2-3-1) で作成したもの

PowerSet の評価結果をsにバインドすることで、PowerSet の再帰を一度で済ませることがポイント

## 解答例：問題 2-3-7)

整数のリストを取り，その偶数番目の数の和と奇数番目の数の和の対 (pair) を返す関数 `paiofSums` を書け．ただし，補助関数を使ってはいけない．

```
fun paiofSums(nil) = (0, 0)
| paiofSums([a]) = (0, a)
| paiofSums(a::b::cs) =
  let
    val (M,N) = paiofSums(cs)
  in
    (b+M, a+N)
  end;
```

問題文に「偶数番目と奇数番目のペア」と書いてあるので偶数番目の方を先にすべき

## 解答例：問題 2-3-8)

リストLとMを連結する関数cat(L,M)を書け。  
ただし、@ 演算子を使ってはいけない。::を使うこと。  
この関数はリストLの長さに比例した時間で動作し、  
リストMの長さには依存しないこと。

```
fun cat([],M) = M  
|   cat(x::xs,M) = x::cat(xs,M);
```



# ラムダ計算

# ラムダ計算 (ラムダ算法 Lambda Calculus)

- 1930年代にAlonzo Church と Stephen Cole Kleeneによって提唱されたもの
- 計算の実行を関数への引数の評価(evaluation)と適用(application)としてモデル化・抽象化した計算体系
- チューリングマシンと同等の計算能力
- 関数型言語の基礎
  - 型付きのラムダ計算がMLの基礎だが、本講義ではラムダ計算の一般的なモデルである型無しのラムダ計算を紹介
  - 型付きは形無しに制約を与えて限定したもの

# ラムダ記法 (Lambda notation)

- 関数表記において仮引数となる変数を明示した記法
- 関数と関数の計算結果をきちんと区別する

例：

$$f(x) = ax^2 + bx + c$$

これは  $f$  という関数を表しているのか、それとも関数  $f$  に値  $x$  を入れたときの計算結果を示しているのか → 文脈依存

# ラムダ記法 (Lambda notation)

- 関数表記において仮引数となる変数を明示した記法
- 関数と関数の計算結果をきちんと区別する

例：

$$f(x) = ax^2 + bx + c$$

ラムダ記法

関数  $f$

$$f = \lambda x. ax^2 + bx + c$$

値  $f(x)$

$$(\lambda x. ax^2 + bx + c)x = ax^2 + bx + c$$

これは  $f$  という関数を表しているのか、それとも関数  $f$  に値  $x$  を入れたときの計算結果を示しているのか → 文脈依存

それを区別しようとするのが  
ラムダ記法



# ラムダ項 (Lambda term) ※ラムダ式ともいう

## 定義 1 ラムダ項 ( $\lambda$ 項) の定義

1. 変数はラムダ項
2.  $M$  がラムダ項で  $x$  が変数のとき,  
 $(\lambda x. M)$  はラムダ項
3.  $M$  と  $N$  がラムダ項のとき,  
 $(M N)$  はラムダ項



# ラムダ項 (Lambda term) ※ラムダ式ともいう

## 定義 1 ラムダ項 ( $\lambda$ 項) の定義

1. 変数はラムダ項
2.  $M$  がラムダ項で  $x$  が変数のとき,  $(\lambda x. M)$  はラムダ項  ラムダ抽象 (Lambda abstraction)
3.  $M$  と  $N$  がラムダ項のとき,  $(M N)$  はラムダ項  適用 (Application)

# ラムダ項 (Lambda term) ※ラムダ式ともいう

## 定義 1 ラムダ項 ( $\lambda$ 項) の定義

1. 変数はラムダ項
2.  $M$  がラムダ項で  $x$  が変数のとき,  $(\lambda x. M)$  はラムダ項  ラムダ抽象 (Lambda abstraction)
3.  $M$  と  $N$  がラムダ項のとき,  $(M N)$  はラムダ項  適用 (Application)

例:  $\lambda f. (\lambda x. (f (f x)))$

$f$  を  $x$  に2回適用した値を返すラムダ項  
(高階関数)

どんな二つの $\lambda$ 式 $M$ と $N$ をとってきても $M$ を $N$ に (関数) 適用することができる  
⇒ 高階関数が自然に表現できる

# 適用の順序・変数の有効範囲

- 適用の順序：左結合

$$M\ N\ N' = (M\ N)\ N'$$

- 変数の有効範囲：できるだけ広く取る

$$\lambda x. xy = \lambda x. (xy)$$

- 意味が明確なときは $\lambda$ は省略可能

$$\lambda xy. x = \lambda x. (\lambda y. x) = \lambda x. \lambda y. x$$



## 変数の束縛 (bound variable) ・ $\alpha$ 変換

$$\lambda x.(\lambda x.x) \equiv \lambda y.(\lambda x.x) \\ \neq \lambda x.(\lambda y.x)$$

$$\lambda x.(\lambda y.x) \equiv \lambda xy.x \equiv \lambda uy.u \equiv \lambda uv.u$$

$\lambda$ 式において変数を置き換えても同じもの =  $\alpha$ 同値

$$\lambda \underline{x}.(\lambda \underline{y}.\underline{x} + \underline{y}) + y$$

束縛変数                      自由変数

$\alpha$ 変換：束縛変数の名前を変更すること

# β簡約(β-変換 Beta reduction)

λ記法で表された関数の（ある値に対する）関数の値を求める操作．関数の適用．

定義： β簡約  $\xrightarrow{\beta}$  は

$$(1) \quad (\lambda x.M)N \xrightarrow{\beta} M[x := N]$$

$$(2) \quad M \xrightarrow{\beta} N \quad \text{ならば} \quad \lambda x.M \xrightarrow{\beta} \lambda x.N$$

$$MP \xrightarrow{\beta} NP$$

$$PM \xrightarrow{\beta} PN$$

$$\text{例3 : } (\lambda x.x + 1)3 \xrightarrow{\beta} 4$$

$$(\lambda f.f * 2)(\lambda x.x + 3) \xrightarrow{\beta} \lambda x.(x + 3) * 2$$

# 自然数の表現

- $\lambda$ 記法で自然数はどのように表現できるのか

1. 基本となる数をまず(勝手に)作る
2. Successor関数を用いて数を定義する

これで完成

- このように定義された自然数をチャーチ数(Church number)と呼ぶ

# 自然数の表現(ゼロ)

1. 基本となる数ゼロ(に相当するもの)を以下のように定義する.

$\lambda f x. x$  ( $= \lambda f. \lambda x. x$ )

これは $f$ と $x$ の2引数をとる関数.

(あるいは $f$ を引数としてとり,  $\lambda x. x$ を結果とする関数)

$\lambda x. x$ は恒等関数. つまり引数をそのまま結果とする関数.

要は, これが自然数の基本となる数だということを人が決める.

これは $f$ に関係なく二つ目の引数である $x$ をそのまま返す.

← ゼロに相当

2. Successor 関数を用いて数を定義する

# 自然数の表現(Successor)

1. 基本となる数ゼロ(に相当するものを)を以下のように定義する.

$\lambda f x. x$  ( $= \lambda f. \lambda x. x$ )

2. Successor 関数を用いて数を定義する.

- Successor関数とは「与えられた引数の次を返す関数」.
- 「次」とは抽象的な概念.
- ゼロの定義で  $f$  で表したものが Successor 関数
- 今の時点ではその具体的な実装は考えない(つまり抽象的な概念を表す関数).

# 自然数の表現(0, 1, 2, ...)

1. 基本となる数ゼロ(に相当するものを)を以下のように定義する.

$\lambda f x. x$  ( $=\lambda f. \lambda x. x$ )

2. Successor 関数を用いて数を定義する.

- ゼロの定義で  $f$  で表したものが Successor 関数
- 1 の表現:  $\lambda f x. f x$   
つまりゼロの次の数
- 2 の表現:  $\lambda f x. f (f x)$   
つまりゼロの次の次の数
- 3 の表現  
...

# 自然数の表現 — MLで確かめてみよう

- Successor 関数を具体的に与えてMLの処理系で動かしてみる
- ここで使っているのは, 匿名関数

まず, ゼロ, 1, 2, Successor  
関数を右のように定義する.

```
ゼロ  $\lambda f x. x$   
    val zero = fn f => fn x => x;  
1  $\lambda f x. f x$   
   val one = fn f => fn x => f x;  
2  $\lambda f x. f (f x)$   
   val two = fn f => fn x => f (f x);  
Successor 関数  
   val succ = fn n => n+1;
```

# 匿名関数

- ラムダ式では関数に名前をつけない:  $\lambda x. x + 1$
- これに対応するSMLでの関数の定義が匿名関数  
fn <パラメータ> => <式>  
例: fn x => x + 1;
  - fun の代わりに fn  
= の代わりに =>  
関数名はない
- ラムダ式やその場でしか使わない関数は匿名関数として定義するとよい



# 自然数の表現 — MLで確かめてみよう (実行)

- Successor 関数を具体的に与えてMLの処理系で動かしてみる
- ここで使っているのは, 匿名関数

関数zero

```
- zero 3 0;  
val it = 0 : int  
- zero succ 0;  
val it = 0 : int  
- zero succ 1;  
val it = 1 : int
```

一つ目の引数に関わらず二つ目の引数の値を値とする.

関数one

```
- one succ (zero succ 0);  
val it = 1 : int
```

二つ目の引数+1 を結果とする.

関数two

```
- two succ (zero succ 0);  
val it = 2 : int
```

二つ目の引数+2 を結果とする.

# 再帰の表現

ラムダ記法で再帰はどのように表現できるのか.

- ラムダ記法では関数に名前がない.
- 名前がないと普通の方法では自分自身を呼び出すことができず, 関数の再帰構造が書けない

## 不動点コンビネータを使う

- コンビネータとは自由変数を含まないラムダ式のこと.
- 不動点コンビネータはいくつも発見されているが, ここでは Curry の不動点演算子 (または Y コンビネータ) と呼ばれているものを紹介する.

# 再帰のモデル 不動点コンビネータ

不動点とは、関数を適用しても値が変わらない点（値）のこと

ラムダ式  $M$  に対して

$$\begin{aligned} M' &\equiv M'' M'' \\ M'' &\equiv \lambda x. M(xx) \end{aligned}$$

とすると

$$\begin{aligned} M' &\equiv (\lambda x. M(xx)) M'' \\ &\xrightarrow{\beta} M(M'' M'') \\ &= M M' \end{aligned}$$

すなわち  $M'$  は  $M$  の不動点

# 再帰のモデル 不動点コンビネータ

不動点とは、関数を適用しても値が変わらない点（値）のこと

ラムダ式  $M$  に対して

$$\begin{aligned} M' &\equiv M'' M'' \\ M'' &\equiv \lambda x. M(xx) \end{aligned}$$

とすると

$$\begin{aligned} M' &\equiv (\lambda x. M(xx)) M'' \\ &\xrightarrow{\beta} M(M'' M'') \\ &= M M' \end{aligned}$$

すなわち  $M'$  は  $M$  の不動点

$Y$  を以下のように定義すると  $YM$  は

$$\begin{aligned} Y &\equiv \lambda y. (\lambda x. y(xx)) (\lambda x. y(xx)) \\ YM &\equiv (\lambda y. (\lambda x. y(xx)) (\lambda x. y(xx))) M \\ &\xrightarrow{\beta} (\lambda x. M(xx)) (\lambda x. M(xx)) \\ &= M'' M'' \\ &= M' \end{aligned}$$

であるから

$$YM \underset{\beta}{=} M' \underset{\beta}{=} M M' \underset{\beta}{=} M(YM)$$

この  $Y$  を Curry の不動点演算子または **Yコンビネータ** と呼ぶ。

# 不動点コンビネータ

- 不動点コンビネータは他にもいくつか発見されている
- 不動点コンビネータを使ってラムダ計算で名前のない関数について再帰を書くことができる, ということが数学的に示されている

ラムダ計算で再帰ができる = ループを実現可能  
別途ラムダ項で条件分岐も記述可能

→ プログラムが書ける  
関数型言語の理論的基礎になっている  
関数型言語で書いたプログラムの検証が数学的に可能

# 条件分岐

- TRUE :  $\lambda xy. x$
- FALSE :  $\lambda xy. y$
- if b then x else y :  $\lambda b. (\lambda x. (\lambda y. (b \ x \ y)))$
- if TRUE then A else B :  $\lambda b. (\lambda x. (\lambda y. (b \ x \ y))) \text{ TRUE } A \ B \rightarrow \lambda x. (\lambda y. (\text{TRUE } x \ y)) \ A \ B$   
 $\rightarrow \lambda y. (\text{TRUE } A \ y) \ B \rightarrow \text{TRUE } A \ B \rightarrow (\lambda xy. x) \ A \ B \rightarrow (\lambda y. A) \ B \rightarrow A$
- if FALSE then A else B は？

型 (type)

# MLの型 (type)

- 基本型

int, real, char, string, bool

- リスト (list)

int list    整数のリスト

real list list    実数のリストのリスト

- 組

int \* real    整数と実数の組

int list \* int list \* int    整数のリストと整数のリストと整数の組



# MLの関数の型

- 関数の型: 引数と結果の型を  $\rightarrow$  で結ぶことで表現
  - $\text{int} \rightarrow \text{int}$ : 整数を引数に取り, 結果も整数である関数
  - $\text{real list} \rightarrow \text{real}$ : 実数のリストを引数に取り, 結果が実数である関数
  - $\text{int list} \rightarrow \text{int list} * \text{int list}$ : 整数のリストを引数に取り, 整数のリストと整数のリストの組を結果とする関数

# 多相型 (polymorphism)

次の関数の型は？

```
fun reverse [] = []  
| reverse (L) =  
  let  
    fun rev1 ([], L) = L  
    | rev1 (x::xs, L) = rev1 (xs, x::L)  
  in  
    rev1 (L, [])  
  end;
```

この関数は連結演算子@を使わないでリストを逆順にする関数.

# 多相型 (polymorphism)

次の関数の型は？

```
fun reverse [] = []  
| reverse (L) =  
  let  
    fun rev1 ([], L) = L  
    | rev1 (x::xs, L) = rev1 (xs, x::L)  
  in  
    rev1 (L, [])  
  end;
```

処理系で実行すればわかるが、  
この関数の型は  
'a list -> 'a list

**'a はどの型ともマッチする多相型**

つまりこの関数reverseはどんなリスト  
でも逆順にする。  
(ただし[]を入れると「何のリストか推  
論できない」という警告が出る)

# 等値型

- 等値演算子 (=) が使える型が等値型
- 多相型だが等値型でなければならない場合SMLでは "a 等と表記
  - 例: 問題2-2-6 a) 関数member  
fun member (\_, nil) = false  
| member (x, y::ys) = if x = y then true else member (x, ys);  
  
stdIn:99.28 Warning: calling polyEqual  
val member = fn : 'a \* 'a list -> bool
- Warning: calling polyEqual : 等値型でないといけないという警告

# SMLで等値型であるものとそうでないもの

## 等値型のもの

- 基本型のうち  
int, bool, char, string
- 等値型の基本型からなる組やリスト

## 等値型でないもの

- 基本型のうち  
real
- 関数
- real 型や関数が含まれる組やリスト

注：real 型は大小比較はできる

# 明示的な型の指定

- 型は基本的にシステムが推論. しかし型を指定したいときもある.

例: 実数 $x$ の二乗を計算したい.

```
fun f1 (x) = x * x;
```

とすると, `val f1 = fn: int -> int` と整数 $x$ の二乗とsmlの処理系に判断される.

\* が整数でも実数でも乗算を表しているため, \*だけではどちらかわからない.  
その場合はsmlは整数と判断するというルール.

- 実数にしたいときは明示的に型を書き込む.

```
fun f2 (x:real) = x * x;
```

注) これは「式」なので $x$ の型指定は最初になくてもいい

```
fun f2 x = (x:real) * x;
```

```
fun f2 x = x * (x:real);
```

```
fun f2 x = x * x :real;
```



どれでもよい.