プログラミング言語 No. 2-5

2025.5.16 中川博之

[ミニレポート解説] 条件分岐

- TRUE： λxy. x
- FALSE： λxy. y
- if b then x else y： λb. (λ x. (λy. (b x y)))

- if TRUE then A else B： λb. (λ x. (λy. (b x y))) TRUE A B → λ x. (λy. (TRUE x y)) A B

→ λy. (TRUE A y) B → TRUE A B → (λxy. x) A B → (λy. A) B → A

- if FALSE then A else B は？

[ミニレポート解説] 条件分岐

- TRUE： λxy. x
- FALSE： λxy. y
- if b then x else y： λb. (λ x. (λy. (b x y)))

```
val TRUE = fn x => fn y => x;
val FALSE = fn x => fn y => y;
val IFTHEN = fn b => fn x => fn y => (b x y);
```

- if TRUE then A else B： λb. (λ x. (λy. (b x y))) TRUE A B → λ x. (λy. (TRUE x y)) A B

→ λy. (TRUE A y) B → TRUE A B → (λxy. x) A B → (λy. A) B → A

- if FALSE then A else B は？

```
- IFTHEN TRUE 0 1;
val it = 0 : int
- IFTHEN FALSE 0 1;
val it = 1 : int
```

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 -> で結ぶことで表現
  - int -> int: 整数を引数に取り, 結果も整数である関数
  - real list -> real: 実数のリストを引数に取り, 結果が実数である関数
  - int list -> int list * 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);　←　どれでもよい．
  fun f2 x = x * x :real;

カリー化

関数の引数は1つ

- ラムダ項では関数の引数は1つ
  - λ x. M はラムダ項
    λxy.x = λx.(λy.x) = λx.λy.x
- 関数型言語での関数は引数を1つしか取らない
- しかし，2つ以上の引数をもつ関数を書きたいことも少なくない
  - 例: 実数 x の i 乗． ← x と i の2つの引数が必要
    fun multiply (_, 0) = 1.0
    | multiply (x, i) = x * multiply(x, i-1);
    ここでの（）は組を表すものであり，
    関数の引数を（）で囲む必要はない

カリー化 (Curry)

- 複数の引数を取る際に，組ではない別の形でそれを可能とする形式
- カリー化とは，「複数の引数を取る関数」を「最初の引数を引数として，残りの引数を引数として値を返すような関数を返す関数」として定義すること

```
(カリー化形式でない plus)
fun plus (x, y) = x + y;
val plus = fn : int * int → int
```

組を引数として，値を返す

```
(カリー化形式の cplus)
  fun cplus x y = x + y;
  val cplus = fn : int → int → int
  →は右結合なのでこれは int → (int → int)
```

intを引数として，関数を返す

[再掲] 匿名関数

- ラムダ式では関数に名前をつけない: λ 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

カリー化された匿名関数

- 前回のラムダ計算で自然数を表す例で用いているのは，カリー化された匿名関数
  - val zero = fn f => fn x => x;
- これは2つの引数を取る匿名関数を zeroという名前にバインドしたもの

カリー化の利点

- 複数の引数を取る関数を高階関数として実現できる

  例えば，一部の引数だけを具体化した関数を作成することができる

```
- fun cplus x y = x + y;
val cplus = fn : int -> int -> int

- val plus5 = cplus 5;          ← 5加算するという関数を定義
val plus5 = fn : int -> int

- plus5 10;
val it = 15 : int

- plus5 12;
val it = 17 : int
```

高階関数

高階関数 (higher-order-function)

- 関数を引数とする関数
- 関数を結果とする（返す）関数

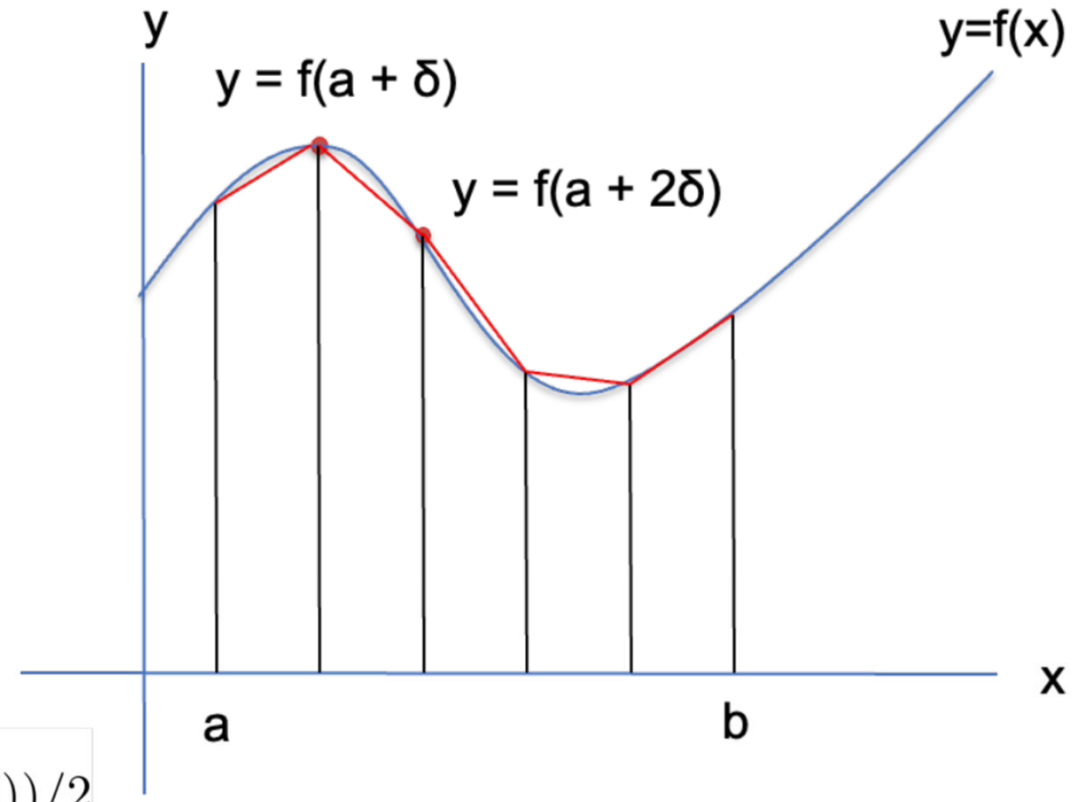
- 関数を引数とする関数の例：積分計算

$$\int_a^b f(x)dx$$

関数を引数とする関数

- 例：台形則による積分計算

$$\int_a^b f(x)dx$$



$$\int_a^b f(x)dx = \sum_{i=1}^{n} \delta(f(a+(i-1)\delta) + f(a+i\delta))/2$$

問題2-5-1) 積分を計算する関数

- 積分を計算する関数trapを以下の式に従って書け

$$\int_a^b f(x)dx = \sum_{i=1}^{n} \delta(f(a+(i-1)\delta) + f(a+i\delta))/2$$

関数trapの引数は
a: 定積分の下限
b: 定積分の上限
n: 領域の分割数
f: 積分する関数 ←関数も引数とする

δはa, b, nから計算可能なので引数にはしない

問題2-5-1) 積分を計算する関数

- 積分を計算する関数trapを以下の式に従って書け

$$\int_a^b f(x)dx = \sum_{i=1}^{n} \delta(f(a+(i-1)\delta) + f(a+i\delta))/2$$

```
fun trap a b n f =
    if n<=0 orelse (b-a) <= 0.0 then 0.0
```

問題2-5-1) 積分を計算する関数

- 積分を計算する関数trapを以下の式に従って書け

$$\int_a^b f(x)dx = \sum_{i=1}^{n} \delta(f(a+(i-1)\delta) + f(a+i\delta))/2$$

```
fun trap a b n f =
    if n<=0 orelse (b-a) <= 0.0 then 0.0
    else
        let
            val delta = (b-a)/ real n
        in
```

問題2-5-1) 積分を計算する関数

- 積分を計算する関数trapを以下の式に従って書け

$$\int_a^b f(x)dx = \sum_{i=1}^{n} \delta(f(a+(i-1)\delta) + f(a+i\delta))/2$$

```
fun trap a b n f =
    if n<=0 orelse (b-a) <= 0.0 then 0.0
    else
        let
            val delta = (b-a)/ real n
        in
            delta * (f a + f (a+delta))/2.0 + trap (a + delta) b (n-1) f
        end;
```

問題2-5-1) 実際に積分を計算してみよう

$$\int x^i dx = \frac{1}{i+1} * x^{i+1} + C$$

より，0.0から1.0までの x * x の積分は 1/3

x * x * x の積分は 1/4

x * x * x * x の積分は 1/5

- 適用例： trap 0.0 1.0 5 (fn (x:real) => x * x);

map 関数・畳み込み関数

関数型言語由来のよく利用される関数

- map 関数: 引数として与えられた関数をリストの各要素に適用する
- 畳み込み関数: リストを一つの値に「畳み込む」関数
  - そのリストを要約した値を返すと考えればよい
    どのような要約をするかは引数の関数で与える
  - 具体的な畳み込み関数: fold, reduce
    - SML の畳み込み関数: foldr, foldl
- filter 関数: 条件に合致した要素を残す

map 関数

map 関数

- map;

  val it = fn : ('a -> 'b) -> 'a list -> 'b list

('a -> 'b) を引数にとって，'a list -> 'b list を結果とする高階関数.

1つ目の引数: 関数，2つ目の引数: リスト，結果: リスト

- 2つ目の引数で与えられたリストの各要素に1つ目の引数で与えられた関数を適用

多相型 'a, 'b はそれぞれ別の型であってもよいことを示す

map 関数の使い方 (問題2-5-2)

- 問題2-5-2) 整数のリストのすべての要素に 1 を加えるmap関数を定義せよ

```
map (fn x => x + 1) [1,2,3];
```

定義例

```
- val listInc = map (fn x => x + 1);
val listInc = fn : int list -> int list
- listInc [1, 2, 3];
val it = [2,3,4] : int list
```

```
- fun add1 x = x + 1;
- map add1 [1,2,3];
```

でもよいが，このためだけに関数add1を定義するのは無駄なので匿名関数を使う方がよい.

map 関数の使い方 (問題2-5-2～問題2-5-4)

- 問題2-5-2) 整数のリストのすべての要素に 1 を加えるmap関数を定義せよ

  map (fn x => x + 1) [1,2,3];

- 問題2-5-3) 実数のリストの中の負の数を 0.0 で置き換え，非負の数はそのままにする map関数を定義せよ

  map (fn x => if x < 0.0 then 0.0 else x) [1.0, ~3.2, 5.4, ~0.3];

- 問題2-5-4) 文字のリストのうち，小文字を対応する大文字に変換する map関数を定義せよ

  map (fn c=> if c>= #"a" andalso c<= #"z" then chr(ord(c)-ord(#"a")+ord(#"A")) else c) [#"a", #"N", #"f", #"r"];

chr は文字コードを文字に変換する組み込み関数.
ord は文字を文字コードに変換する組み込み関数.

map 関数はどのように定義されているか

- 実現例1 (局所環境を使わない例)

```
fun map1 F [] = []
| map1 F (x::xs) = F x::map1 F xs;
```

- 実現例2 (局所環境を使った例)

```
fun map2 F =
  let
    fun M [] = []
    |  M (x::xs) = F x :: M xs
  in
    M
  end;
```

どちらも動作は同じ.
実現例2の方が，関数 (F) を引数に取り，関数 (R) を結果とする関数であることがより明確になっている.
SML/NJではmapは実現例2で定義されている.

問題2-5-5)

- map (map square) [[1,2], [3,4,5]] は何をするか？
  ただし，fun square x = x * x; とする.

問題2-5-5)

- map (map square) [[1,2], [3,4,5]] は何をするか？
  ただし，fun square x = x * x; とする.

- map square: 整数のリストを取りその各要素を二乗する操作.
- map (map square) はその操作をあるリストに対して行う.
- ということはmap (map square)は整数のリストのリストを引数に取り,そのリストの各要素である整数のリストにmap squareを施す.
- つまり, map (map square)は整数のリストのリストを引数に取り各整数を二乗するものである.

問題2-5-5)

- map (map square) [[1,2], [3,4,5]] は何をするか？
  ただし，fun square x = x * x; とする．

  - map;

  val it = fn : ('a -> 'b) -> 'a list -> 'b list

  - square;

  val it = fn : int -> int

  map square は(‘a -> ’b)の部分をsquareのint -> int で確定． すなわち‘a ’b ともにint となり

  - map square;

  val it = fn : int list -> int list

  - map (map square);

  val it = fn : int list list -> int list list

畳み込み関数 foldr, foldl

関数 foldr

- foldr;

val it = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> ‘b

- ('a * 'b -> 'b) 型の関数を引数に取り，'b -> 'a list -> 'b を結果とする高階関数
- 1つ目の引数 (関数): リスト要素にどのような演算を施し集約するかを指定する関数
  - この関数は引数を2つ取る（カリー化形式ではない）
- 2つ目の引数: 初期値（デフォルト値）
- 3つ目の引数: 対象とするリスト

関数 foldr の使い方

- 実数のリストの最大値:

```
foldr (fn (x, y) => if x > y then x else y) 1.1 [1.1, 2.3, ~4.2];
```

一般にLを大小比較のできる値のリストであるとすると，その最大値は

```
foldr (fn (x, y) => if x > y then x else y) (hd L) L;
```

- 整数のリストの要素の合計:

```
foldr (fn (x, y) => x + y) 0 [1,2,3,4,5];
```

- リストの長さ:

```
foldr (fn (x, y) => y + 1) 0 L;
```

- リストを逆順にする:

```
foldr (fn (x, xs) => xs@[x]) [] L;
```

関数 foldr と関数 foldl の定義

- fun foldr F y [] = y

  |　 foldr F y (x::xs) = F(x, foldr F y xs);

  foldrはリストを右から計算

- fun foldl F y [] = y

  |　 foldl F y (x::xs) = foldl F (F (x,y)) xs;

  foldlはリストを左から計算

foldr と foldl の使い分け

- 第一引数である関数 F が可換である場合，foldrと foldlの結果は同じ
  - foldr (fn (x, y) => x + y) 0 [1,2,3,4,5,6];
    (1+(2+(3+(4+(5+(6+0))))))
  - foldl (fn (x, y) => x + y) 0 [1,2,3,4,5,6];
    (6+(5+(4+(3+(2+(1+0))))))
- 第一引数である関数 F が可換でない場合，foldrとfoldlの結果は異なる
  - foldr (fn (x, y) => x - y) 0 [1,2,3,4,5,6];
    (1-(2-(3-(4-(5-(6-0))))))
  - foldl (fn (x, y) => x - y) 0 [1,2,3,4,5,6];
    (6-(5-(4-(3-(2-(1-0))))))