

# プログラミングAIII

2025年度講義資料 (4)

新潟大学 工学部工学科 知能情報システムプログラム

青戸等人

# 目次

- ① 関数の再帰的な定義
- ② 局所変数, 変数のスコープ
- ③ 関数を返す関数

# 目次

- 1 関数の再帰的な定義
- 2 局所変数, 変数のスコープ
- 3 関数を返す関数

## 再帰的な定義 (教科書2.3節)

関数 $f$ の定義で $f$ 自身を用いる場合, つまり, 関数 $f$ を(今定義しようとしている) $f$ 自身を使って定義しているような定義を, **再帰的な定義**であるという.

例. 自然数 $n$ の階乗 $n \times (n - 1) \times \dots \times 1$ を求める関数 $fact$

$$fact(x) = \begin{cases} 1 & (x = 0 \text{ の場合}) \\ x * fact(x - 1) & (x > 0 \text{ の場合}) \end{cases}$$

再帰的に定義された関数の計算結果を求めるためには, 定義を繰り返し展開する必要がある.

$$fact(3) = 3 * fact(2) = 3 * 2 * fact(1) = 3 * 2 * 1 * fact(0) = 3 * 2 * 1 * 1 = 6$$

## SML#における再帰的定義の利用

SML#では, (そのまま素直に)関数を再帰的に定義してよい.

```
# fun fact n = if n = 0 then 1
                else n * fact (n - 1);
val fact = fn : int -> int
# fact 3;
val it = 6 : int
```

関数factの定義の右辺に, 関数factが使われていることに注意しよう.

コードが最適化されることもあるが, 前ページのように, 素直に関数定義が展開されて, 計算が行われる.

## 再帰的定義における注意

再帰的な定義においては、関数が返す値を求めるために、定義を繰り返し展開する必要があった。

場合によっては、この展開が終わらない場合もある。階乗の例では、引数に負の数を与えると、定義の展開が止まらない。

$$\text{fact}(-1) = (-1) * \text{fact}(-2) = (-1) * (-2) * \text{fact}(-3) = \dots$$

数学的には、 $-1$ の階乗は「定義されない」ということで済むが、プログラムにおいては、計算が止まらず、無限ループに入ってしまう。

```
# fact ~1;  
Segmentation fault (コアダンプ)  
aoto$
```

## 関数定義における再帰の利用

実は, 計算可能性の理論的な成果の結果, “再帰” は計算可能な関数を特徴づけるツールの1つであることがわかっている. 実際, 再帰的な定義を利用すると, 関数プログラムは容易に書けることが非常に多い.

例.  $n$ 個の元をもつ集合から $k$ 個の元を選び出す選び方の種類 ( ${}_n C_k$  や  $\binom{n}{k}$  と書くことが多い) ( $1 \leq k \leq n$ ) を計算する関数  $C(n, k)$  を考える. ここで,

$$\begin{aligned} n \text{個の元をもつ集合から} k \text{個の元を選び出す選び方} \\ &= (n-1) \text{個の元をもつ集合から} k \text{個の元を選び出す選び方} \\ &\quad + (n-1) \text{個の元をもつ集合から} (k-1) \text{個の元を選び出す選び方} \end{aligned}$$

という性質に着目する.

(例の続き)素直に再帰的に定義すると,

```
# fun C (n,k) = C (n-1,k) + C (n-1,k-1)
```

となりますが, このままでは定義の展開が止まらない。  
例として, 集合{0, 1, 2}から2個の数を選ぶ方法を考えると

$$C(3, 2) = C(2, 2) + C(2, 1) = 1 + 2 = 3$$

となる. ここで, 「 $n$ 個から $n$ 個選ぶ方法は1通り」, 「 $n$ 個から1個選ぶ方法は $n$ 通り」, というのが最後に使われていることがわかる.

```
# fun C (n,k) = if n = k then 1
                else if k = 1 then n
                else C (n-1,k) + C (n-1,k-1);
val C = fn : int * int -> int
# C (6,3);
val it = 20 : int
```

## 再帰的な関数定義のコツ

再帰を使って関数を定義するときは、以下のような順番で考えるとよい：

Step 1 少し簡単な場合がどんな場合か考える。

Step 2 少し簡単な場合からどのように計算できるか、考える。

Step 3 最後に、最も簡単になる場合にはどうなるか、考える。

問題. 非負整数 $n$ の2進数表現を文字列で返す関数`toBinary`を定義せよ。

```
# toBinary 254;  
val it = "11111110" : string
```

10進数 表現	2進数 表現
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
...	...

(Step 1) `toBinary n` は, `toBinary (n div 2)` を使うと作れそう.

(Step 2)  $w = \text{toBinary } (n \text{ div } 2)$  とするとき, `toBinary n` は:

- $n$  が偶数のとき:  $w$  の末尾に "0" をつけばよい.
- $n$  が奇数のとき:  $w$  の末尾に "1" をつけばよい.

(Step 3)  $n = 0, 1$  のときは, `toBinary n` は,  $n$  になる.

と考えると, 以下のような関数定義ができるだろう:

```
fun toBinary n = if n < 2 then Int.toString n
                 else if n mod 2 = 0 then toBinary (n div 2) ^ "0"
                 else toBinary (n div 2) ^ "1";
```

## 関数の同時定義(教2.8節)と相互再帰(教2.5節)

2つ(以上)の関数を, 相互に再帰することで, 関数が見通しよく実現できる場合がある. これには, `and` キーワードを使って, 2つ(以上)の関数を同時に定義する.

例えば, 以下は, ジグザクに何ステップか座標を移動する関数を定義する.

```
fun up ((x,y),n) = if n = 0 then (x,y)
                  else right ((x,y+1),n-1)
and right ((x,y),n) = if n = 0 then (x,y)
                     else up ((x+1,y),n-1);
```

この例では, 関数 `up` は関数 `right` を呼びだし, 関数 `right` は関数 `up` を呼びだしている. このため, どちらも単独では定義できない. `up` も `right` も再帰的に定義されていることになるが, このような再帰の仕方を **相互再帰** とよぶ.

## 実習課題 (1)

以下の関数を再帰を使って定義せよ。

- ① 0以上の整数 $n$ を受けとって,  $n + (n - 1) + \dots + 0$ を返す関数sum

```
# sum;  
val sum = fn : int -> int  
# sum 5;  
val it = 15 : int
```

- ② 0以上の整数の対 $(n, k)$ を受けとって,  $n^k$ を返す関数power

```
# power;  
val power = fn : int * int -> int  
# power (2,5);  
val it = 32 : int
```

- ③ 正整数 $a, b$ の最大公約数 ( $a$ と $b$ の共通の約数のうち、最大の数)を  $\text{gcd}(a, b)$ と書く.

ユークリッドの互除法は,

「正整数 $a, b$  ( $b < a$ )について,  $b$ が $a$ の約数でないときに,  $a$ を $b$ で割ったときの余りを $r$  ( $r > 0$ )とおくと,  $\text{gcd}(a, b) = \text{gcd}(b, r)$ となる」

ことを利用して, 最大公約数を求める方法である.  
ユークリッドの互除法を用いて, 2つの正整数の最大公約数を求める関数 $\text{gcd}$  を定義せよ.

```
# gcd;  
val gcd = fn : int * int -> int  
# gcd (321,843);  
val it = 3 : int
```

# 目次

- ① 関数の再帰的な定義
- ② 局所変数, 変数のスコープ
- ③ 関数を返す関数

## 局所変数の使用(教2.4節)

- インタープリタで関数や変数を定義すると, その定義以降で, その識別子は見えるようになる.
- 関数定義が複雑になってくるに従って, 補助関数を定義ながら, 目的の関数を作ることが多くなってくる. そのような補助関数は, メインに定義したい関数の補助として定義するためであれば, 他のところでは参照したくない.
- つまり, 大きなプログラムになると, 名前の衝突を避けるために, 名前空間(関数名や変数名がどこから見れるか)に制限を設けることが必須.
- これを小規模な単位で実現する手段として, let式が用意されている.

## let式による局所定義

### let 式

```
1 let
2   変数または関数定義
3   .....
4   変数または関数定義
5 in body式
6 end;
```

- 式全体として, body式の評価結果を返す.
- letとinの間にある定義は body式の中でのみ参照され, let式全体の外では見えない.

## 局所定義の有効範囲

以下のインタプリタでの入力では, 変数 $x$ と関数 $f$ を, `let`文のなかで局所的に定義している. `body`の中では $f$ と $x$ が参照されているが, 定義の外側では $f$ も $x$ も見えない.

```
# let val x = 2
>     fun f y = y + 3
> in f x
> end;
val it = 5 : int
# f;
(interactive):15.0-15.0(201) Error: (name evaluation "190")
unbound variable: f
# x;
(interactive):16.0-16.0(0) Error: (name evaluation "190")
unbound variable: x
```

## let式が置ける場所

let式自体は式であるから、式が書けるところにはどこにでもlet式を入れることができる。

```
# (let val y = 1 in y + y end) + 3;  
val it = 5 : int  
# if true then let val y = 1 in y + y end else 5;  
val it = 2 : int  
# let val y = 1 in let val z = y + y in z + z end end;  
val it = 4 : int  
#
```

## let式の利用 (1)

let式を使うことで、同じ処理や同じ値の式を、一箇所にまとめることができる。

```
fun f str = (substring (str, 0, (size str) div 2)
            = substring (str, (size str) div 2,
                        (size str) div 2));
```

```
fun f str = let val half = (size str) div 2
            val left = substring (str, 0, half)
            val right = substring (str, half, half)
            in left = right
            end;
```

処理単位をまとめることで、プログラムの可読性が高まったり、再利用をしやすくなる。

## let式の利用 (2)

let式を使うことで, 2重計算を避けることができる.

```
let fun f x = ...とても時間のかかる計算...  
in (f 0) + (f 0)  
end;
```

```
let fun f x = ...とても時間のかかる計算...  
    val y = f 0  
in y + y  
end;
```

前者では(f 0)が2回計算され時間がかかるが, 後者では(f 0)の計算は1回で済む.

## let式の利用 (3)

let式は途中の計算結果を確認するのもにも有用である。以下のプログラムは、階乗を計算する

```
fun fact x =  
  if x = 0 then 1  
  else let val ih = fact (x - 1)  
        val _ = print ((Int.toString x) ^ " * "  
                        ^ (Int.toString ih) ^ "\n")  
      in x * ih  
  end;
```

「fact 5;」の実行結果を調べ、なぜそのような表示が得られるか考えよ。

なお、「val \_ = expr」は、exprを評価するが、変数に評価結果を束縛するかわりに、結果を捨てる。

## 変数のスコープ(教2.8節) (1)

定義の有効範囲を**スコープ**とよぶ.

```
$ smlsharp
SML# unknown for x86_64-pc-linux-gnu with LLVM 13.0.1
# val x = 1;
val x = 1 : int
# ...
```

例えば, インタープリタのトップ環境で定義された変数定義の  
スコープは, 「変数の定義が行われた後から, インタープリタ  
が終了するまで」, である.

## 変数のスコープ(教2.8節) (2)

ただし, 正確には, 同じ識別子を用いた変数定義が行われると, 以前の定義は見えなくなる.

```
# val x = 1;
val x = 1 : int
# val x = 2;
val x = 2 : int
# x + 1; (* x = 1 は見えない *)
val it = 3 : int
```

しかし, 「見えない」 = 「なくなった」ではない. let式による局所定義を考えると, それがよくわかる. (⇒ 次ページ)

## 局所定義のスコープ

```
# val x = 1; (* x=1 *)
val x = 1 : int
# let val x = 2
>   val y = x + 3 (* x=2 *)
> in (x,y)         (* x=2 *)
> end;
val it = (2, 5) : int * int
# let val y = x + 3 (* x=1 *)
>   val x = 2
> in (x,y)         (* x=2 *)
> end;
val it = (2, 4) : int * int
# x;
val it = 1 : int
```

- let式の局所定義は、上から順番に評価され、束縛が追加される。
- 局所定義のスコープは、その定義がされた後から end まで。
- 局所定義であっても、識別子が重なるときは、以前にあった束縛は見えなくなる。
- 局所定義のスコープの外では、局所定義で行われた束縛は見えない。

## 実習課題 (2)

let式を利用して, 以下の関数を定義せよ.

- ① 与えられた文字列が与えられたとき, 中央で区切った2つの文字列の対を返す関数 `splitString` を定義せよ. ただし, 長さが奇数のときは前の文字列を1文字短かくせよ.

```
# splitString "hello";  
val it = ("he", "llo") : string * string
```

- ② 整数 $n$ について, その倍数で100を越えない最大の数を $f(n)$ とおくとき, 2つの整数 $x, y$ を受けとって,  $f(x) + f(y)$ を計算する関数 `addMultipleLe100`.

```
# addMultipleLe100 (8,9);  
val it = 195 : int
```

# 目次

- ① 関数の再帰的な定義
- ② 局所変数, 変数のスコープ
- ③ 関数を返す関数

## 関数を返す関数(教2.6節) (1)

前回見たように, いくつかの2引数ライブラリ関数は,  
f (arg1, arg2)のように引数を与えるのではなく,  
f arg1 arg2のように引数を与える仕様になっている.

```
# String.isPrefix;  
val it = fn : string -> string -> bool  
# String.isPrefix "abc" "abcdef";  
val it = true : bool  
# Char.contains;  
val it = fn : string -> char -> bool  
# Char.contains "alpha" #'o';  
val it = false : bool
```

ここでは, このような関数に見ていこう.

## 関数を返す関数(教2.6節) (2)

このような関数は `f arg1 arg2` とよびだすこともできるが、  
`(f arg1) arg2` のようによびだすこともできる。

```
# (String.isPrefix "abc") "abcdef";  
val it = true : bool  
# (Char.contains "alpha") #"o";  
val it = false : bool
```

`(f arg1) arg2` という式は、`f arg1` が `arg2` に適用した形になっていることに注意する。実は以下のように、`f arg1` も式としてはまったく正しく、式全体で関数となっている。

```
# String.isPrefix "abc";  
val it = fn : string -> bool  
# Char.contains "alpha";  
val it = fn : char -> bool
```

## 関数を返す関数(教2.6節) (3)

このような形の関数を自分で定義するには、使うときと同様、**関数定義において、スペースで区切って仮引数を与える。**

```
# fun f x y = x + y
val f = fn : ['a::{int,...}]. 'a -> 'a -> 'a]
# f 2; (* 「yを受けとって、2+yを返す関数」が得られる *)
val it = fn : int -> int
# it 3;
val it = 5 : int
```

(f 2) 3は f 2 3のように括弧を省略できる。2つのスペースは“関数適用”を表わしていることに注意すると、これは、「**関数適用が左結合**」ということを意味している。

## 関数を返す関数(教2.6節) (4)

```
# fun f x y = x + y  
val f = fn : ['a::{int,...}]. 'a -> 'a -> 'a]
```

ここで,  $f$ の型  $'a \rightarrow 'a \rightarrow 'a$  に注目してみよう.

- $'a \rightarrow 'a \rightarrow 'a$  の括弧付けは,  $'a \rightarrow ('a \rightarrow 'a)$  のようになっている.

この括弧の省略の仕方は, 論理式の $\rightarrow$ (ならば)と同じ.

- つまり,  $f$ は, 「型 $'a$ の要素を受けとり, 型 $'a \rightarrow 'a$ の関数を返す」ような関数となっている.

## 部分適用

以下の関数fとgは、結局、計算したいことは同じだが、異なる型をもち、異なる関数適用の形になっている：

```
# fun f x y = x + y;  
val f = fn : ['a::{int,...}]. 'a -> 'a -> 'a]  
# fun g (x,y) = x + y;  
val g = fn : ['a::{int,...}]. 'a * 'a -> 'a]
```

前者の書き方では、 $(f \text{ expr})$  という最初の引数だけを与えた関数を使い回すことが可能。このように一部の引数だけを適用することを、**部分適用**とよぶ。

## 実習課題 (3)

指定された型をもつ関数を定義せよ。

- 1 正整数 $k$ と整数の対 $(n, m)$ を受けとって,  $n$ と $m$ が $k$ を法として等しいかを返す関数 `equalModulo`

```
# equalModulo;
val it = fn : ['a#eq::{int,...}. 'a -> 'a * 'a -> bool]
(* もしくは fn : int -> int * int -> bool など *)
# equalModulo 3 (2,5);
val it = true : bool
```

- 2 文字 $x$ , 正整数 $n, m$  ( $n \leq m$ とする)と文字列 $s$ を受けとって,  $s$ の $n$ 文字目から $m$ 文字目までを $x$ に変更した文字列を返す関数 `maskWith`

```
# maskWith;
val it = fn : char -> int * int -> string -> string
# maskWith #"x" (3,5) "alphabet";
val it = "alxxxbet" : string
```