

## プログラミングAIII

2024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

## 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。

## 目次

- 1 局所変数、変数のスコープ
- 2 関数を返す関数

## 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
```

## 目次

- 1 局所変数、変数のスコープ
- 2 関数を返す関数

## 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
#
```

## 局所変数の使用(教2.4節)

- インタ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いく。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えるか)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。

## 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。

## let式の利用 (3)

let式は途中の計算結果を確認するのも有用である。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。

## 実習課題 (1)

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
```

## 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、インタプリタが終了するまで」、である。

## 目次

- ① 局所変数, 変数のスコープ
- ② 関数を返す関数

## 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)

## 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、 $f(\text{arg1}, \text{arg2})$ のように引数を与えるのではなく、 $f \text{ arg1 arg2}$ 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 -> 'a -> 'a` に注目してみよう。

- `'a -> 'a -> 'a` の括弧付けは、`'a -> ('a -> '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 -> '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 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 (2)

指定された型をもつ関数を定義せよ。

- 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
```
- 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 = "alxxxxbet" : string
```