

# プログラム言語論

亀山幸義

筑波大学 情報科学類

No. 8: 命令型言語

## 目次

- 1 命令型言語
- 2 C 言語の型
- 3 命令型言語つづき
- 4 関数呼び出し

## 命令型プログラム言語

Imperative Programming Language

- ▶ プログラム=命令の列=状態変更の列
- ▶ cf. 数学的関数は、「状態」を持たない。(同じ引数に対しては、必ず同じ値を返す.)

典型的な命令型言語:

- ▶ C, C++, Java, Perl, Python, Ruby, ...

## 命令型言語 micro-C

構文 (式 expression):

```
type expr =  
  | CstI of int  
  | Var of string  
  | Prim of string * expr * expr
```

構文 (文 statement):

```
type stmt =  
  | Asgn of string * expr  
  | If of expr * stmt * stmt  
  | Block of stmt list  
  | For of string * expr * expr * stmt  
  | While of expr * stmt  
  | Print of expr
```

C 言語	micro-C 言語 (教科書)
<code>x = y + 3;</code>	<code>Asgn("x",Prim("+",Var("y"),CstI(3)))</code>
<code>if e1 s1 else s2</code>	<code>If(e1,s1,s2)</code>
<code>{s1 s2 s3}</code>	<code>Block([s1;s2;s3])</code>
<code>for (i=e1;i&lt;e2;i++) s1</code>	<code>For(e1,e2,s1)</code>
<code>while (e1) s1</code>	<code>While(e1,s1)</code>
<code>printf("%d",e1);</code>	<code>Print(e1)</code>

```
{ int sum, x;
  sum = 0;
  while (x > 0) {
    sum = sum + x;
    x = x - 1; }
  printf("%d", sum); }

Block([
  Asgn("sum",CstI 0);
  While(Prim(">",Var "x",CstI 0),
    Block([
      Asgn("sum",Prim("+",Var "sum",Var "x"));
      Asgn("x",Prim("-",Var "x",CstI 1));
    ]));
  Print(Var "sum"); ])
```

変数宣言はない。(すべてグローバル変数)

命令型言語の意味; 式と文

micro-C の意味

	関数型言語	命令型言語
式		
文	-	

関数型言語の処理

- ▶ 環境: 変数を、その値 (整数やクロージャなど) に対応付け。値は更新不能。

命令型言語の処理

- ▶ 環境: 変数を、ストア中の場所 (location) に対応付け。値は更新不能。
  - ▶ ストア (store): 場所を値に対応付けるもの。値は更新可能。
- ただし、micro-C は単純なため、環境は持たない (ストアのみ)。

式の評価は、micro-ML と本質的に同じ:

```
let rec eval e store =
  match e with
  | CstI i → i
  | Var x → getSto store x
  | Prim(ope, e1, e2) →
    let i1 = eval e1 store in
    let i2 = eval e2 store in
    begin
      match ope with
      | "*" → i1 * i2
      ...
```

## micro-C の意味

文の評価関数 `exec`:

```
let rec exec stmt store =  
  ...
```

ポイント:

- ▶ 評価器 (`exec`) は、文とストアをもらってストアを返す。
- ▶ 環境とストアの違い: ストアの中の値は変更可能。

例: 文 `Asgn("x", Prim("+", Var("x"), CstI 5))` の実行

- ▶ 実行前のストア: `[x->10, y->20]`
- ▶ 実行後のストア: `[x->15, y->20]`

## ストアの操作

ストア==書き換え可能なメモリ:

- ▶ `emptystore ...` 空のストア (初期値)
- ▶ `getSto s x ...` ストア `s` において、`x` に対応する値を返す
- ▶ `setSto s (x, v)` ストア `s` において `x` に対応する値を `v` に変更

```
let s = emptystore ;;           []  
getSto s x ;;                   []  
==> error  
setSto s (x, 10) ;;              [x→10]  
==> ()  
setSto s (y, 20) ;;              [x→10; y→20]  
==> ()  
getSto s x ;;                    [x→10; y→20]  
==> 10  
setSto s (x, (getSto s x) + 30) ;;  
==> ()                            [x→40; y→20]
```

## micro-C の意味

代入文 (Assignment statement, 割り当て文) の処理:

```
let rec exec stmt store =  
  match stmt with  
  | Asgn(x, e) →  
    setSto store (x, eval e store)  
  ...
```

式 `e` は (文でなく式であるので) `exec` でなく `eval` で評価する。

例: 文 `Asgn("x", Prim("+", Var("x"), CstI(5)))` の実行

- ▶ 実行前のストア: `[x->10, y->20]`
- ▶ 実行後のストア: `[x->15, y->20]`

## micro-C の意味

ブロック文の処理:

```
let rec exec stmt store =  
  match stmt with  
  ...  
  | Block stmts →  
    let rec loop ss sto =  
      match ss with  
      | [] → sto  
      | s1::sr → loop sr (exec s1 sto)  
    in loop stmts store
```

例: `Block([s1; s2])` (文 `s1, s2` を順に実行する文)

```
loop [s1; s2] store1  
-> loop [s2] (exec s1 store1)  
-> loop [] (exec s2 store2)
```

For 文や While 文も同様に処理できる。ここでは、メタ言語 (OCaml) の再帰関数を使って、対象言語 (micro-C) の For 文を処理している。

```
| For(x, estart, estop, stmt) →
  let start = eval estart store in
  let stop = eval estop store in
  let rec loop i sto =
    if i > stop then sto
    else loop (i+1) (exec stmt (setSto sto (x, i)))
  in loop start store
```

命令型言語の構文と意味:

- ▶ 命令型言語のプログラム=状態変更を行なう「命令」の列。
- ▶ 状態: 書き換え可能なストアとして表現。
- ▶ 式の意味: 関数型言語と同様。
- ▶ 文の意味: 状態(ストア)を変更するものとして定義。

命令型言語の意味の記述は特に難しいところはない。(ただし、状態変更と高階関数が組み合わさると、意味を精密に決めるのが格段に困難になる。)

## 目次

① 命令型言語

② C 言語の型

③ 命令型言語つづき

④ 関数呼び出し

## C 言語の型の例

```
void foo (int *p, int *q) {
  while (*p) {
    *(q++) = *(p++);
  }
}
```

- ▶ ポインタ型の型構成子「\*」を便宜上 Ptr と書くと、
- ▶ 関数 foo の型は、(Ptr(int) \* Ptr(int)) -> void

## C言語の型の例

<code>int x</code>	integer
<code>int *x</code>	pointer to an integer
<code>int x[10]</code>	array of 10 integers
<code>int x[10][3]</code>	array of 10 arrays of 3 integers
<code>int *x[10]</code>	array of 10 pointers to integers
<code>int *(x[10])</code>	array of 10 pointers to integers
<code>int (*x)[10]</code>	pointer to an array of integers
<code>int **x</code>	pointer to a pointer to an integer

C言語では、ポインタ型と配列型の構文がやや非直感的であるが、型システムそのものはシンプル。(基本型と型構成子からなる。)

### C言語の型検査

C言語のプログラムが与えられたとき、型付けが成功するかどうかを検査するアルゴリズム(型検査アルゴリズム)が存在する。

## 目次

- 1 命令型言語
- 2 C言語の型
- 3 命令型言語つづき
- 4 関数呼び出し

## C言語の文

以下のもののうち、構文的に正しい式はどれか？

```
int x, y; int a[100]; int *p;
x = x + 10;
x + 10 = x;
a[5] = x + 10;
x = a[5] + 10;
x++ = y + 10;
p = &y + 1;
p = &(a[5]) + 1;
*p++ = y + 10;
```

## lvalue と rvalue

lvalue (left hand side value, 左辺値)

- ▶ ストア中の場所 (location) を表す。

rvalue (right hand side value, 右辺値)

- ▶ 値を表す。
- ▶ x は、lvalue も rvalue も持つ。
- ▶ a[10] は、lvalue も rvalue も持つ。
- ▶ \*x は、lvalue も rvalue も持つ。
- ▶ 8+2 は、rvalue 10 を持つが、lvalue を持たない。
- ▶ &x は、rvalue を持つが、lvalue を持たない。

構文的に正しい式は？

```
int x, y; int a[100]; int *p;
x = x + 10;           OK
x + 10 = x;          Not good
a[5] = x + 10;       OK
x = a[5] + 10;       OK
x++ = y + 10;        Not good
p = &y + 1;          OK
p = &(a[5]) + 1;     OK
*p++ = y + 10;      OK
```

\*p++は \*(p++) であることに注意。

- ① 命令型言語
- ② C 言語の型
- ③ 命令型言語つづき
- ④ 関数呼び出し

## 関数呼び出しでの引数の渡し方

関数型言語での主要な関数呼び出し方法

- ▶ 値呼び call by value
- ▶ 名前呼び call by name
- ▶ 必要呼び call by need

命令型言語での主要な関数呼び出し方法

- ▶ 値呼び call by value
- ▶ 参照呼び call by reference
- ▶ ... call by value return

## 命令型言語における引数の渡し方

値呼び (C, Java):

```
static void swapV (int x, int y) {
    int tmp = x; x = y; y = tmp;
}
main () {
    int u = 10;
    int v = 20;
    swapV(u, v);
    printf("%d %d\n", u, v); ==> 10 20
}
```

- ▶ 実引数の rvalue (10 と 20) を渡す。
- ▶ 仮引数 x, y は、それらを指す。
- ▶ swapV を呼んでも、u と v の値は変わらない。

## 命令型言語における引数の渡し方

Note. このスライドは、swapR の呼び出しの付近のコードに間違いがあり、授業後に修正しました。(教科書 p.119 参照)

参照呼び (C#):

```
static void swapR (ref int x, ref int y) {  
    int tmp = x; x = y; y = tmp;  
}  
...  
swapR(ref u, ref v);  
...
```

- ▶ 実引数の lvalue (場所) を渡す。
- ▶ 仮引数 x, y は、それらを指す。
- ▶ swapV を呼ぶと、u と v の値が入れかわる。

## 命令型言語における引数の渡し方

C 言語で、参照呼びを真似する。

```
static void swapR (int *x, int *y) {  
    int tmp = *x; *x = *y; *y = tmp;  
}  
main () {  
    int u = 10;  
    int v = 20;  
    swapR(&u, &v);  
    printf("%d %d\n", u, v); ==> 20 10  
}
```

C 言語は、あくまで「値呼び」だが、ポインタを使えば、参照呼びと同じことができる。(ML 言語でも ref 型を使えば同様にできる。)

## 命令型言語における引数の渡し方

Call by value return (FORTRAN, Ada):

- ▶ 実引数の rvalue を格納した新しい場所を作り、その lvalue を渡す。
- ▶ 関数呼び出しの終了時に、その lvalue に格納された値を、実引数の lvalue の場所に格納する。
- ▶ 結果として、swapR と同様の結果となる。

## 命令型言語における引数の渡し方：まとめ

命令型言語での主要な関数呼び出し方法

- ▶ 値呼び call by value
- ▶ 参照呼び call by reference
- ▶ ... call by value return

	PASCAL	C	C++	C#	Ada	Java	Fortran
call by value							
call by reference		-				-	-
call by value return	-	-	-	-	○	-	

(Sestoft の教科書から引用)

## Short quiz

以下の問に答えよ。

- ▶ 命令型言語の値呼びと参照呼びについて、それぞれのメリット、デメリットを1つ以上あげなさい。

解答の一例

- ▶ 値呼びでも参照呼びでも同様に記述できるプログラムで、かつ、渡される引数が整数や浮動小数点数など(基本的な値)の場合、値呼びの方が参照呼びより実行性能が良い(ことが多い)。
- ▶ 一方、swapR の例のように、「呼び出された関数内で、呼び出し元の値を変更する」というプログラムは、参照呼びの方が自然に記述できる。C言語では(上記のスライドのように)値呼びで swapR を記述することが可能であるが、その場合、「アドレスを値として渡す」といったプログラミングが必要であり、そのようなプログラミングスタイルは、エラーが起きやすく、またデバッグも困難であることが多い。