

## プログラム言語論

亀山幸義

筑波大学 情報科学類

命令型言語

## 命令型プログラム言語

### Imperative Programming Language

プログラム = 「命令」の列

命令 = 「状態」の変更, 入出力 etc.

cf. (純粋な, あるいは, 数学的な) 関数 = 「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: 変数の値の変更.

### 命令型言語の例

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

注意: 「命令型言語かどうか」のボーダーは必ずしも明確ではない.

Lisp/Scheme も見方によっては命令型.

(set! x (+ x 1)) という関数は, 数学的な関数ではなく, 状態を変更する命令.

OCaml では, 参照型 (ref 型) を使えば, 命令型言語と同様なプログラムが書ける.

## 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
```

## micro-C と C の対応

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

```
{ 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

式の評価:

```
let rec eval e (store : naivestore) : int =
  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-ML と (本質的に) 同じ。

	関数型言語	命令型言語
式と文: 式	○	○
文	—	○

関数型言語の処理

環境: 変数 (変数名) を、その値 (整数やクロージャなど) に対応付けるもの。値は更新不能。

命令型言語の処理

環境: 変数 (変数名) を、ストア中の場所 (location) に対応付けるもの。値は更新不能。

ストア (store): 場所 (location) を値に対応付けるもの。値は更新可能。

micro-C は C より非常に単純なため、環境なしで (ストアのみで) 処理をしている。(ストアは、変数を、値に対応付けるが、更新可能。)

文の評価:

```
let rec exec stmt (store : naivestore) : naivestore =
  match stmt with
  | Asgn(x, e) ->
    setSto store (x, eval e 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。  
ストアの中の値は更新可能。

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

実行前のストアが `[x->10, y->20]` のとき、  
実行後のストアが `[x->15, y->20]` となる。

```

| 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])

```

      loop [s1; s2] store1
-> loop [s2] (exec s1 store1)
-> loop [s2] store2
-> loop [] (exec s2 store2)
-> loop [] store3
-> store3

```

exec 関数が完成したら、、、

```

let run stmt =
  let _ = exec stmt emptystore in
  ()

```

空のストアからはじめて実行する。(返すものは何もないので、() を返している。つまり、実行「結果」はなく、結果を print して終わり。)  
!!chap7.ml の演習問題をやってください。(締切は少し余裕をもたせて 6/17 とします。)

## C の型の例

C 言語でも、型は結構複雑:

```

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

```

void, int などの基本型

\* という型構成子 (便宜上 Ptr(.) と書くことにする)

引数の int \*p は、「p が Ptr(int) 型である」ことを意味する。

p++ の型は Ptr(int) で、\*(p++) の型は int である。

関数 foo の型は、(Ptr(int) \* Ptr(int)) -> void

C 言語は、コンパイル時に型検査を行なう。

たとえば q++ = \*p++; はエラー

## C の型の例

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

「型システム」が複雑なのではなく、ポインタ型 (と配列型) の構文がやや非直感的。

命令型言語

プログラム=「状態」の変更をおこなう「命令」の列

式と文

文の意味: 「状態」(ストア)の変更として定義

事実: 状態変更と高階関数が組み合わせると、意味をきちんと決めるのが格段に難しくなる。

lvalue と rvalue

命令型言語の引数の渡し方

C 言語の文

構文的に正しい式は？

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

lvalue to rvalue –C 言語の場合

lvalue (left hand side value, 左辺値):

`x = e;` の左辺の `x`

`x++` の中の `x`

`&x` の中の `x`

これらは、ストア中の場所 (location) を表す。

rvalue (right hand side value, 右辺値):

`x + 10` の中の `x` など

これは、ストア中に蓄えられた値を表す。

例:

`8+2` は、rvalue 10 を持つが、lvalue を持たない。

`x = 8+2;` は OK だが、`8+ 2 = x;` は NG。

`x` は、lvalue も rvalue も持つ。

`x = x + 10;` は OK。

`a[10]` は、lvalue も rvalue も持つ。

構文的に正しい式は？

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

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

## 命令型言語における引数の渡し方

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, ML):

```
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
}
```

swapV(10,20) の呼び出し:

実引数 (actual parameter) の rvalue (10 と 20) を渡す。

仮引数 (formal parameter) x, y は、それらを指す。

x と y の値を変更しても、呼び出し元の値は変わらない。

## 命令型言語における引数の渡し方

参照呼び (C#):

```
static void swapR (ref int x, ref int y) {  
    int tmp = x; x = y; y = tmp;  
}  
... swapR(10,20) ...
```

swapR(10,20) の呼び出し:

- 実引数 (actual parameter) の lvalue (場所) を渡す。
- 仮引数 (formal parameter) x, y は、それらを指す。
- x と y の値を変更すると、呼び出し元の値が変わる。

## 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。

## 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

(Sestoft のテキストより引用)

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