プログラミング言語論

命令型プログラミング言語

水野嘉明

目次

2

命令型プログラミング言語

- 命令型言語は、
  - 最も一般的なパラダイム
  - ノイマン型コンピュータが、その計算モデルである
  - 命令（つまり代入文）の繰り返しにより、変数の値（「状態」という）を動的に変化させ、計算を行う

3

命令型プログラミング言語

- 命令型言語の基本要素
  - 基本的機械操作は、代入文
  - 代入文の実行を制御する制御構造
  - 操作の対象であるデータ型
  - 手続きを呼出すための仕組み

4

1. 代入

左辺値／右辺値について

5

1. 代入

- 下記の代入文について考える
  <変数> = <式>; （例： x = y + 1; ）
  - 変数とは、
    値を記憶するための「場所」
    （通常は、メモリ上に取られる）
  - 右辺の式が 評価 (evaluate) されて
    その「値」が変数に格納される

6

1. 代入

- 代入文 `x = y;`
  - 左辺の x と右辺の y は、意味が異なる
  - 左辺は、値を記憶する場所を指示している
    左辺値 (l-value) という
  - 右辺は、代入する値を示す
    右辺値 (r-value) という

7

1. 代入

- 変数や式には、左辺値と右辺値がある
  - 通常、「変数の値」と言う時には右辺値を指す
- ある種の式や定数には、右辺値のみがあり、左辺値はない
  - 例) ~~x + y = 5;~~
    ~~5 = x;~~

8

1. 代入

- 一般の代入文

  $\langle$式$\rangle_1$ = $\langle$式$\rangle_2$;

  の効果は、$\langle$式$\rangle_2$ の右辺値を求め、$\langle$式$\rangle_1$ の左辺値の指す場所に置くことである

  例: a[i+3] = (x+y)/2;

9

1. 代入

- 演習5.1
  - Javaにて、次の様に宣言されている

```
int      i = 1, j = 2;
int[ ]   a = new int[10];
```

次の式は、左辺値を持つか

(1) (i)　(2) i++　(3) i+j
(4) a　(5) a[i++] (6) a[i+j]

10

2. 制御構造

逐次／選択／反復
＝ 構造化プログラミング

11

2. 制御構造

- 構造化プログラミング
  - 1967 ダイクストラ (E. W. Dijkstra) 等が提唱
  - *制御構造* による見通しの良い処理と、*モジュール化* による問題分割を基本とする

12

2. 制御構造

- 構造化定理
  1つの入り口と1つの出口を持つようなプログラムは、「*逐次*・*選択*・*反復*」の3つの基本的な制御構造によって記述できる

13

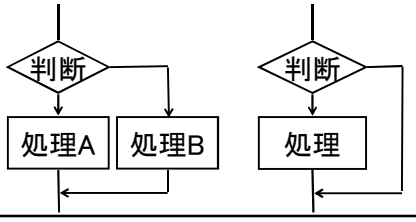
2. 制御構造

- 逐次 (sequence)
  - プログラムに記述された順に、処理を行う

14

2. 制御構造

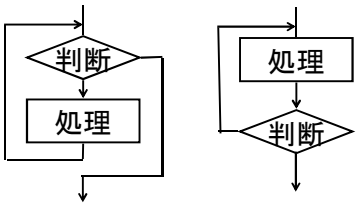
- 選択 (selection)
  - 条件に従い、実行する処理を選択する



15

2. 制御構造

- 反復 (iteration)
  - 一定の条件が満たされている間処理を繰り返す



16

2. 制御構造

- CやJavaにおける制御構造の例
  - 逐次
    - 通常の文の記述

```
discr = b * b - 4.* a * c;
x1 = (-b+sqrt(discr))/(2.*a);
x2 = c/(a * x1);
```

（2次方程式の解法の一部）

17

2. 制御構造

- 選択
  - if文, if-else文, switch-case文

```
if (discr >= 0.0) {
  x1 = (-b+sqrt(discr))/(2.*a);
  x2 = c/(a * x1);
}else
  ErrorMessage("No root¥n");
```

18

2. 制御構造

- 反復
  - for文, while文, do-while文

```
do {
   old_x = x;
   x -= f(x) / df(x);
}while (fabs(x - old_x) > eps);
```

（ニュートン法の一部）

19

2. 制御構造

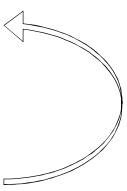
- 構造化されている (well-structured) プログラムとは
  - 前記の制御構造のみで制御されている
    - goto文を用いると、構造化されていないプログラムになりやすい
      - ⇒ goto文有害説
  - モジュール化されている

20

2. 制御構造

- 構造化されてないプログラムの例

```
   for (i:=0; i < n; i++) {
         :
Label:
         :
   }
         :
   i = 3;
   goto Label;
```



21

3. データ型

3.1 データ構造
3.2 データ型について

22

3.1 データ構造

- データ構造
  - 処理対象であるデータの表現形式を与えるもの

  - プログラミング言語の提供する「データ型」を組合わせて実現する

23

3.1 データ構造

アルゴリズム＋データ構造＝プログラム
（N. Wirth）

プログラムとは、

- データ構造によって表現されたデータを、
- アルゴリズムによって示された処理手順に従って 処理するもの

24

3.1 データ構造

- したがって、
  問題に即した適切なデータ構造
  を選択すること が重要

⇩

- ◆分かりやすいプログラム
- ◆効率の良いプログラム

25

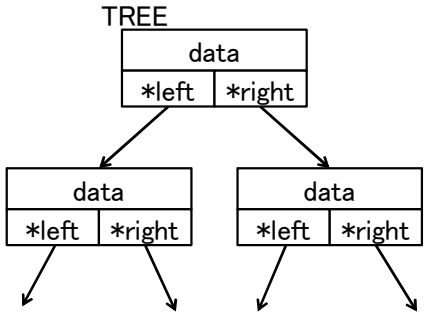
3.1 データ構造

- データ構造の例
  - 木構造 （C言語の構造体）

```
struct  TREE {
  struct  NODE  data:   // この節のデータ
  struct  TREE   *left;   // 左部分木へのポインタ
  struct  TREE   *right; // 右部分木へのポインタ
}
```

26

3.1 データ構造



27

3.2 データ型について

- *基本データ型*
  - 整数、実数など
  - 自然で基本的な型であり、単一の値を持つもの
- *構造型*
  - 基本データ型を構成要素とする、複雑なデータ構造
  - 配列型、レコード型など

28

3.2 データ型について

- *型宣言*
  - 変数にどのようなデータ型のデータを格納するかの宣言

（例）C言語の型宣言

```
int      a, b;
double   x;
char     str[32];
```

29

3.2 データ型について

- *型付けされた言語*（静的型付け）
  - 変数を使用する前に、型宣言を必要とするプログラミング言語
  - ほとんどの命令型言語は、型付けされた言語
    - Perl等のスクリプト言語には、型付けされていないものも多い（動的型付け）

30

3.2 データ型について

- 「プログラミング言語の基礎」の「3.4 型システム」、「3.5 データ型の実際」の章を、もう一度復習して下さい

31

復習

3.2 データ型について

① すべての式は型を持つ
② 型システム
③ 型検査
④ 静的型付け/動的型付け
⑤ 強い型付け/弱い型付け
⑥ 基本型/構造型

32

4. 手続き

4.1 手続きの宣言
4.2 名前の有効範囲
4.3 変数の存続期間
4.4 引数結合方法
4.5 手続きとスタック

33

4. 手続き

- *手続き*（procedure）とは
  - 実行すべき一連の計算ステップを持つ、プログラム単位
  - プロシージャ、サブルーチン、メソッド、関数、副プログラム などと呼ばれる

34

4. 手続き

- 手続きの必要性
  - システムの大規模化、複雑化
  - 要求機能を、より簡素化された複数の機能に分解し、各々を手続きとして実装する。
    これらを組み合わせて全体を実現
    ＝ *モジュール化*

35

4. 手続き

- 処理に必要なデータは、引数として受け渡す
  - 大域変数を使用することもある

（注）大域変数：プログラムのどこからでも参照・更新できる変数

36

4. 手続き

- 処理結果は、
  - 関数値として返す
  - 引数として出力する
  - 大域変数を書き換える（副作用）
  - その他の動作を行う

37

4.1 手続きの宣言

- 手続きの宣言（定義）は、次の4つの部分からなる
  1. 宣言される手続きの名前
  2. 手続きの仮引数
  3. 手続き本体　（宣言と文の並び）
  4. 結果の型　　（オプション）

38

4.1 手続きの宣言

- 例: Cの関数手続き succ の宣言

```
int succ(int i)
{ return (i+1) % size; }
```

  - この関数手続きは、1ヶの仮引数 iを持ち、その本体は次の文である

```
{ return (i+1) % size; }
```

39

4.2 名前の有効範囲

- 変数名などの名前には、*有効範囲*(scope)がある
  - 通常は、静的なプログラムテキストにより定まる
    ＝ 静的有効範囲規則
    (static scope rule)
  - LISPやオブジェクト指向言語では動的有効範囲である

40

4.2 名前の有効範囲

- 名前の宣言
  - 名前の束縛(binding)とも呼ばれる
  - それにより名前が使用できるようになる
- 仮引数と手続きの中で宣言された名前は、その手続き内だけで有効
  - *局所的*(local)である

41

4.2 名前の有効範囲

- 例

x,y,tempの有効範囲

max,minの有効範囲

```
program sample (input, output);
  var  max, min;
  procedure swap(var x,y:integer);
    var temp:integer;
    begin
      temp:=x; x:=y; y:=temp
    end;
  begin
    read(min, max);
    if min>max then swap(min,max);
  end.
```

42

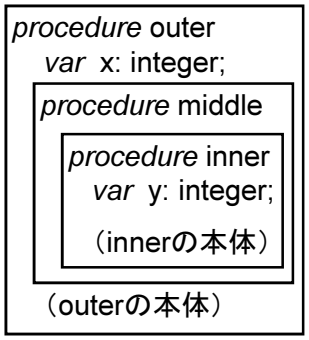
4.2 名前の有効範囲

- 手続きがネストしている場合、外部から内部の名前は見えない
- 内部から外部の名前は、見える（使用できる）

43

4.2 名前の有効範囲

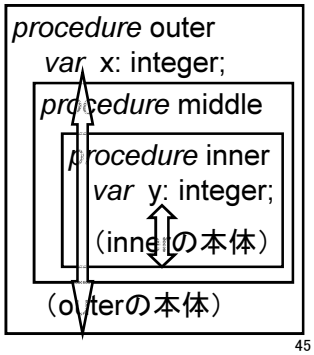
例：手続きouterの中で手続きmiddleが、middleの中でinnerが定義されている場合



44

4.2 名前の有効範囲

- outerで宣言された変数 x は middle/innerでも参照できる
- innerの変数 y は、innerでのみ参照可



45

4.2 名前の有効範囲

- outer の本体からは、
  - middle ← 呼出し可
  - inner ← 呼出し不可

innerは middleの中で定義されている
⇒ middleの外は、innerという名前の 有効範囲外

46

4.2 名前の有効範囲

- 注1：最も外側で定義され、プログラムのどこからでも参照・更新できる変数を、大域変数という
- 注2：C言語では、関数定義のネストはできない（関数内で関数を定義することができない）

47

4.3 変数の存続期間

- 変数の*存続期間*（extent/lifetime）とは
  - 変数に対してメモリ領域を割り当てている期間
    （変数に対して値を格納したり、その値を参照できる期間）

48

4.3 変数の存続期間

- 変数は、少なくとも有効範囲内のコードを実行中は存続している
- メモリの割り当て方式により、変数の存続期間は異なる
  - 動的割り当て
  - 静的割り当て

49

4.3 変数の存続期間

- *動的割り当て*
  - プログラムの実行中、その変数の有効範囲に入った時、割り当てる
    - 例: 手続き開始時
      ブロック開始時
  - 有効範囲を出ると、存続を終了（割り当てたメモリが開放される）

50

4.3 変数の存続期間

- *静的割り当て*
  - コンパイル時に、割り当てメモリが定まっている
  - プログラム（プロセス）開始時から終了時まで存続
  - 値の初期化は、最初に1回だけ（存続し続けているので、途中では初期化による書き換えはない）

51

4.3 変数の存続期間

- C言語の例
  - 関数内で定義された局所変数は
    - auto変数: （動的割り当て）
      その関数を実行中の間だけ存続
    - static変数: （静的）
      プログラムの最初から最後まで
  - 大域変数（関数外で定義;静的）
    プログラムの最初から最後まで

52

4.3 変数の存続期間

- 演習5.2
  - メインプログラムを実行した結果を述べよ。ここで 、staticは静的割当てを、autoは動的割当てを表す。

メイン プログラム

```
auto  int  x, y;
x = f(2) + f(2);
y = g(2) + g(2);
```

53

4.3 変数の存続期間

関数 f(int u)

```
auto int v = 1;
v = v + u;
return  v;
```

関数 g(int u)

```
static int v = 1;
v = v + u;
return  v;
```

（応用情報技術者試験 平23秋 午前 問22 を改）

54

4.4 引数結合方法

- *引数結合*（argument binding）
  - 手続きを呼び出す（駆動する）時
    手続き定義の引数並び（仮引数）
    ⇧ 値を引き渡す
    実際の値（実引数）

55

4.4 引数結合方法

- C言語での引数結合の例
  宣言

```
int func(int x, double y) {・・・}
```

  呼び出し

```
func( n, (x + 3.5));
```

56

4.4 引数結合方法

- 引数結合方法
  - 値呼出し　（call-by-value）
  - 参照呼出し（call-by-reference）
  - その他、
    名前呼出し　（call-by-name）
    入出力呼出し（call-by-value-result）
    などがある（別紙資料参照）

57

4.4 引数結合方法

- *値呼出し*（call-by-value）
  - ➢右辺値を渡す
  - ➢つまり、実引数で与えられた式を評価し、仮引数にその結果を代入する
  - ➢手続き内で仮引数の値を書き換えても、呼び出し側には影響しない

58

4.4 引数結合方法

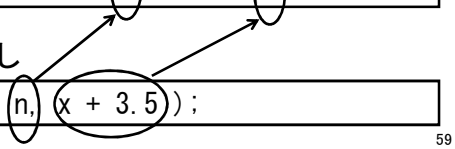
- ➢C言語は、値呼出しの機能のみ
- ➢例1

宣言

```
int func(int x, double y) {・・・}
```

呼出し

```
func( n, x + 3.5));
```



59

4.4 引数結合方法

- ➢例2 （意味のないプログラム）

```
void swap( int x, int y ){
  int   temp;
  temp = x;  x = y;  y = temp;
}
```

swap(a, b) で呼び出した場合、 x=a; y=b; の後 temp=x; x=y; y=temp; が実行される
実引数a、bは変わらない

60

4.4 引数結合方法

- *参照呼出し* (call-by-reference)
  - 左辺値を渡す
  - つまり、実引数のアドレス(左辺値)を仮引数に割り当てることにより、実引数と仮引数の格納場所が同じになる

61

4.4 引数結合方法

参照呼出しの指定

- 例 (Pascalによる swap)

```
procedure swap
          (var x, y:integer);
  var  temp:integer;
  begin
    temp:=x; x:=y; y:=temp
  end;
```

62

4.4 引数結合方法

- 参照呼出しの場合は、実引数は左辺値をとることが出来なければならない
- 前頁の例では、
  swap (a, b); は OK
  swap (5, a+b); は不可

どちらも、左辺値がない

63

4.4 引数結合方法

- 演習5.3
  - 次の手続きについて

```
procedure swap(x, y);
  integer x, y;
  begin
    integer temp;
    temp:=x; x:=y; y:=temp
  end;
```

64

4.4 引数結合方法

(1) 値呼出し

(2) 参照呼出し

の各々の方法で以下のように呼び出した場合の、実行結果を求めよ

```
i:=2; a[2]:=3; a[3]:=4;
swap(i,a[i]);
```

65

4.5 手続きとスタック

- *スタック*(stack)とは、
  - 最も基本的なデータ構造
  - スタックポインタ(SP)と呼ばれるアクセスポートを読み出しと書き込みで共用する線形(一次元)メモリ
  - *LIFO* (Last In First Out：後入れ先出し)方式でアクセスする

66

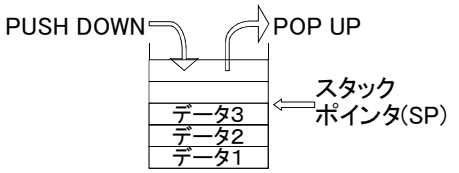
4.5 手続きとスタック

- スタックの操作
  - *PUSH* （PUSH DOWN）
    SPを一つ進め、SPの指すところにデータを格納する
  - *POP*（POP UP）
    データをSPの指すところから取り出し、SPを一つ戻す

67

4.5 手続きとスタック

- スタックポインタの指している個所が、データの先頭



68

4.5 手続きとスタック

- 大抵のコンピュータは、専用のスタックポインタレジスタを持っている
  - このスタックポインタにより、メインメモリの一部をスタックとして利用している
  - スタック領域は、OSが管理し、プロセスごとに用意される

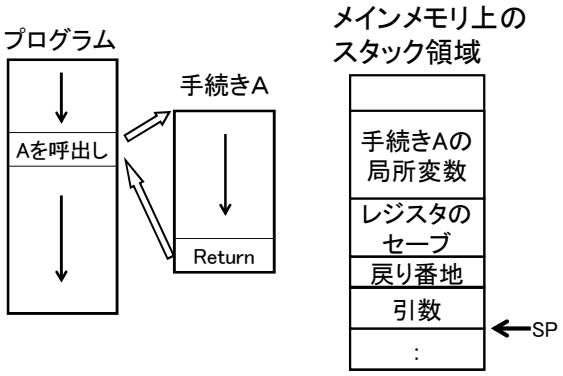
69

4.5 手続きとスタック

- システムに用意されているスタックは次のような用途に使われる
  - 戻り番地の記憶
  - 手続きへの引数受け渡し
  - 変数用領域
  - 作業用領域
    - レジスタ類の退避 など

70

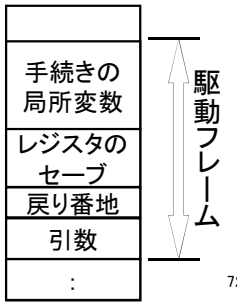
4.5 手続きとスタック



71

4.5 手続きとスタック

- *駆動フレーム*
  （activation frame）
  - 手続きの呼出し毎に、必要な情報を格納するためのメモリ領域
  - 通常、スタックを利用



72

4.5 手続きとスタック

- 再帰呼出しでは、各呼出し毎に局所変数が上書きされることなく、保存されなければならない
  - 呼出し毎に、スタック上にフレームを重ねる
  - 再帰呼出しから戻る際には、スタックフレームを順に「ほぐして」戻る

73

4.5 手続きとスタック

- 例: nの階乗を再帰的に計算する関数 factにより、3の階乗を計算

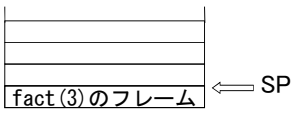
```
fun fact(n) =
  if n≦1 then 1 else n*fact(n-1);
```

```
fact (3) ;          // 3の階乗
```

74

4.5 手続きとスタック

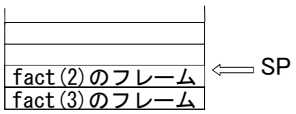
- まず、fact(3)がコールされる



75

4.5 手続きとスタック

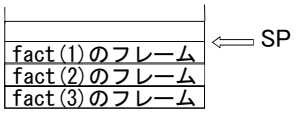
- fact(3) の中で、“3*fact(2)”を計算するため、fact(2) をコール



76

4.5 手続きとスタック

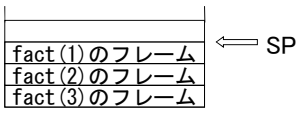
- fact(2) の中で、“2*fact(1)”を計算するため、fact(1) をコール



77

4.5 手続きとスタック

- fact(1) が、値「1」をかえす
- fact(2) が、値「2」をかえす
- fact(3) が、値「6」をかえす



78

お疲れ様でした