

数理情報演習 第9回

プログラミング言語と
その処理系

様々なプログラミング言語

いくつ知っていますか？

様々なプログラミング言語

ALGOL, AWK, BASIC, BCPL, C, C++, C#,
COBOL, D, Delphi, Eiffel, Forth, Fortran,
Haskell, Java, JavaScript, Lisp, ML,
O'caml, Pascal, Perl, PHP, PL/I,
PostScript, Prolog, Python, R, Ruby, S,
Scheme, sed, sh, smalltalk, SQL, Tcl,
WhiteSpace, などしこ,

... もちろん他にも沢山ある

参考: Computer Language History (<http://www.levenez.com/lang/>)

年代順に見てみる

- ~ 1969年
Fortran (1955), ALGOL (1958), Lisp (1959), COBOL (1960),
BASIC (1964), PL/I (1964), BCPL (1967), Forth (1968), sh(1969),
Smalltalk (1969)
- ~ 1980年
Prolog (1970), Pascal (1970), C (1971), ML (1973),
Scheme (1975), AWK (1979),
- ~ 1990年
PostScript (1982), C++ (1983), S (1984), Eiffel (1986), Perl (1987),
SQL (1987), Haskell (1987), Tcl (1988)
- それ以降
Python (1991), Ruby (1993), Java (1995), JavaScript (1995),
PHP (1995), O'caml (1995), Delphi (1995), R (1997), D (1999),
C# (2000), WhiteSpace (2003), などしこ (2005)

参考: Computer Language History (<http://www.levenez.com/lang/>)

用途による分類

- 汎用言語
 - Fortran, BASIC, Pascal, C, C++, Java, Ruby
- 特定用途 (Domain Specific Languages)
 - COBOL (事務処理)
 - PostScript (画像処理)
 - SQL (データベース処理)
 - AWK, sed (テキスト処理)

Script言語

- 目的を早く達成することを目的
- 通常はコンパイルなどを必要としない

- Ruby, Perl, Python (比較的汎用)
- PHP, JavaScript (Webpage用)
- AWK, sed (テキスト処理用)

プログラムの記述による分類

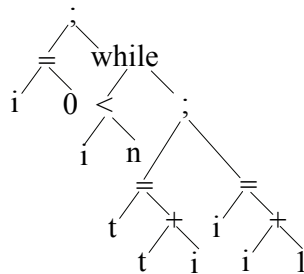
- 手続き型言語
 - 多くの言語はこれ
 - Fortran, BASIC, Pascal, C, C++, Java, Ruby
- 関数型言語
 - 関数を組み合わせて記述
 - Haskell, ML
- 論理型言語
 - 論理を記述すれば答えが出る
 - Prolog

実行形式による分類

- 直接実行
 - Lisp, BASIC, sh, Scheme
- 構文木にして実行
 - Awk, Ruby, Perl, PHP
- 機械語にして実行
 - Fortran, C, C++ など多数
- 中間言語にして実行
 - Prolog, Java

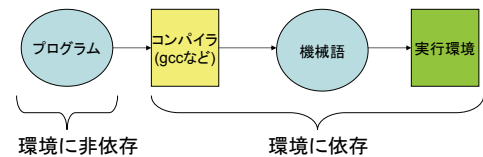
構文木

```
i = 0
while (i < n)
  t = t + i
  i = i + 1
end
```



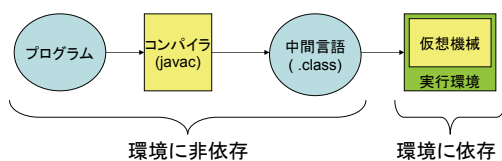
機械語による実行 (例: C言語)

- 機械語: プロセッサが解釈する数字列
- プログラムは環境非依存 (が目的だった)



中間言語による実行 (例: Java)

- 中間言語: 仮想機械 (Virtual Machine) が解釈する数字列
- 中間言語も環境非依存



Javaの利点

- **Write Once, Run Anywhere**
 - 環境 (プロセッサ, OS) が違って動く
 - PC, PDA (Palmなど), 携帯電話
- C++よりもプログラミングが楽
 - ポインタ (多くの人がつまづく) を隠蔽
 - ガベージコレクション (GC)
- 充実したライブラリ
 - GUI, スレッド, ネットワーク, 他いろいろ

本日の課題

- バイトコード (中間言語) を見てみる
 - プロセッサの気分を味わってみる
- Javaのメモリ管理をちょっと知る
 - スタック (Stack)
 - ヒープ (Heap)

バイトコードを見るには

- バイトコード: 命令が1byteにおさまるよう定義されている
 - 1byte = 8bit
 - 0 から 255
- バイトコードを見やすくする
 - javac XX.java
 - javap -c XX
 - みやすい形で表示

関数 sum の例

- ```
int sum(int n) {
 int total = 0;
 for (int i = 0; i < n; i++) {
 total += i;
 }
 return total;
}
```
- javac
- ```

0 iconst_0
1 istore_2
2 iconst_0
3 istore_3
4 goto 14
7 iload_2
8 iload_3
9 iadd
10 istore_2
11 iinc 3 1
14 iload_3
15 iload_1
16 if_icmplt 7
19 iload_2
20 ireturn
    
```

バイトコードの読み方

- 左側の数字: 命令の位置
 - 連続でないのは、直前の命令が1バイトでないため
 - goto 14 → goto (1バイト) 14 (2バイト)
 - 命令は空白を含まない文字列
 - iconst_0 で1命令
 - 変数は出現順に番号がつく
- ```

0 iconst_0
1 istore_2
2 iconst_0
3 istore_3
4 goto 14
7 iload_2
8 iload_3
9 iadd
10 istore_2
11 iinc 3 1
14 iload_3
15 iload_1
16 if_icmplt 7
19 iload_2
20 ireturn

```

## 変数は出現順に番号がつく

- 1から始まる (0は特殊用途)
  - ```
int sum( int n ) {
    int total = 0;
    for ( int i = 0; i < n; i++ ) {
        total += i;
    }
    return total;
}
```
-
- ```

1: n
2: total
3: i

```

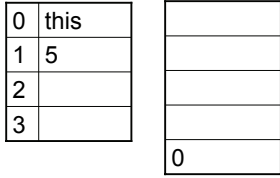
## 仮想機械の動作

- 基本はスタックマシン
- |   |            |
|---|------------|
| 0 | this       |
| 1 | 5<br>(nの値) |
| 2 |            |
| 3 |            |
- 変数テーブル
- |  |
|--|
|  |
|  |
|  |
|  |
|  |
|  |
- スタック
- ```

0 iconst_0
1 istore_2 // total
2 iconst_0
3 istore_3 // i
4 goto 14
7 iload_2 // total
8 iload_3 // i
9 iadd
10 istore_2 // total
11 iinc 3 1 // i
14 iload_3 // i
15 iload_1 // n
16 if_icmplt 7
19 iload_2 // total
20 ireturn
    
```

仮想機械の動作

- 基本はスタックマシン

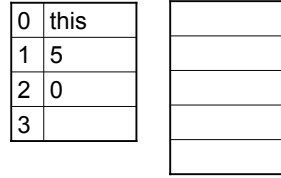


変数テーブル スタック

```
0 iconst_0 // total
1 istore_2 // total
2 iconst_0 // i
3 istore_3 // i
4 goto 14 // total
7 iload_2 // total
8 iload_3 // i
9 iadd // total
10 istore_2 // total
11 iinc 3 1 // i
14 iload_3 // i
15 iload_1 // n
16 if_icmplt 7 // total
19 iload_2 // total
20 return
```

仮想機械の動作

- 基本はスタックマシン

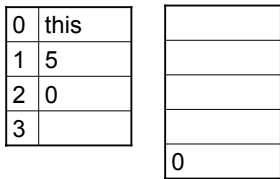


変数テーブル スタック

```
0 iconst_0 // total
1 istore_2 // total
2 iconst_0 // i
3 istore_3 // i
4 goto 14 // total
7 iload_2 // total
8 iload_3 // i
9 iadd // total
10 istore_2 // total
11 iinc 3 1 // i
14 iload_3 // i
15 iload_1 // n
16 if_icmplt 7 // total
19 iload_2 // total
20 return
```

仮想機械の動作

- 基本はスタックマシン

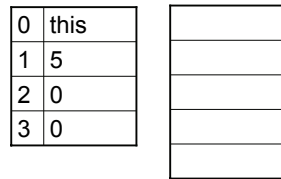


変数テーブル スタック

```
0 iconst_0 // total
1 istore_2 // total
2 iconst_0 // i
3 istore_3 // i
4 goto 14 // total
7 iload_2 // total
8 iload_3 // i
9 iadd // total
10 istore_2 // total
11 iinc 3 1 // i
14 iload_3 // i
15 iload_1 // n
16 if_icmplt 7 // total
19 iload_2 // total
20 return
```

仮想機械の動作

- 基本はスタックマシン

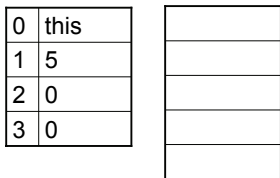


変数テーブル スタック

```
0 iconst_0 // total
1 istore_2 // total
2 iconst_0 // i
3 istore_3 // i
4 goto 14 // total
7 iload_2 // total
8 iload_3 // i
9 iadd // total
10 istore_2 // total
11 iinc 3 1 // i
14 iload_3 // i
15 iload_1 // n
16 if_icmplt 7 // total
19 iload_2 // total
20 return
```

仮想機械の動作

- 基本はスタックマシン

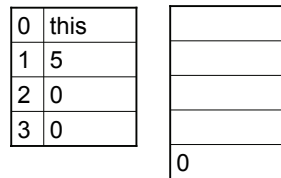


変数テーブル スタック

```
0 iconst_0 // total
1 istore_2 // total
2 iconst_0 // i
3 istore_3 // i
4 goto 14 // total
7 iload_2 // total
8 iload_3 // i
9 iadd // total
10 istore_2 // total
11 iinc 3 1 // i
14 iload_3 // i
15 iload_1 // n
16 if_icmplt 7 // total
19 iload_2 // total
20 return
```

仮想機械の動作

- 基本はスタックマシン



変数テーブル スタック

```
0 iconst_0 // total
1 istore_2 // total
2 iconst_0 // i
3 istore_3 // i
4 goto 14 // total
7 iload_2 // total
8 iload_3 // i
9 iadd // total
10 istore_2 // total
11 iinc 3 1 // i
14 iload_3 // i
15 iload_1 // n
16 if_icmplt 7 // total
19 iload_2 // total
20 return
```

仮想機械の動作

- 基本はスタックマシン

0	this
1	5
2	0
3	0

変数テーブル

5
0

スタック

```

0 iconst_0 // total
1 istore_2 // total
2 iconst_0 // i
3 istore_3 // i
4 goto 14
7 iload_2 // total
8 iload_3 // i
9 iadd // total
10 istore_2 // total
11 iinc 3 1 // i
14 iload_3 // i
15 iload_1 // n
16 if_icmplt 7 // total
19 iload_2 // total
20 return
    
```

⇒ 変数1をpush

仮想機械の動作

- 基本はスタックマシン

0	this
1	5
2	0
3	0

変数テーブル

5 (a)
0 (b)

スタック

```

0 iconst_0 // total
1 istore_2 // total
2 iconst_0 // i
3 istore_3 // i
4 goto 14
7 iload_2 // total
8 iload_3 // i
9 iadd // total
10 istore_2 // total
11 iinc 3 1 // i
14 iload_3 // i
15 iload_1 // n
16 if_icmplt 7 // total
19 iload_2 // total
20 return
    
```

⇒ 2つpopする (a) > (b)なら移動

仮想機械の動作

- 基本はスタックマシン

0	this
1	5
2	0
3	0

変数テーブル

0

スタック

```

0 iconst_0 // total
1 istore_2 // total
2 iconst_0 // i
3 istore_3 // i
4 goto 14
7 iload_2 // total
8 iload_3 // i
9 iadd // total
10 istore_2 // total
11 iinc 3 1 // i
14 iload_3 // i
15 iload_1 // n
16 if_icmplt 7 // total
19 iload_2 // total
20 return
    
```

⇒ 変数2をpush

仮想機械の動作

- 基本はスタックマシン

0	this
1	5
2	0
3	0

変数テーブル

0
0

スタック

```

0 iconst_0 // total
1 istore_2 // total
2 iconst_0 // i
3 istore_3 // i
4 goto 14
7 iload_2 // total
8 iload_3 // i
9 iadd // total
10 istore_2 // total
11 iinc 3 1 // i
14 iload_3 // i
15 iload_1 // n
16 if_icmplt 7 // total
19 iload_2 // total
20 return
    
```

⇒ 変数3をpush

仮想機械の動作

- 基本はスタックマシン

0	this
1	5
2	0
3	0

変数テーブル

0

スタック

```

0 iconst_0 // total
1 istore_2 // total
2 iconst_0 // i
3 istore_3 // i
4 goto 14
7 iload_2 // total
8 iload_3 // i
9 iadd // total
10 istore_2 // total
11 iinc 3 1 // i
14 iload_3 // i
15 iload_1 // n
16 if_icmplt 7 // total
19 iload_2 // total
20 return
    
```

⇒ 2つpopして和をpush

仮想機械の動作

- 基本はスタックマシン

0	this
1	5
2	0
3	0

変数テーブル

スタック

```

0 iconst_0 // total
1 istore_2 // total
2 iconst_0 // i
3 istore_3 // i
4 goto 14
7 iload_2 // total
8 iload_3 // i
9 iadd // total
10 istore_2 // total
11 iinc 3 1 // i
14 iload_3 // i
15 iload_1 // n
16 if_icmplt 7 // total
19 iload_2 // total
20 return
    
```

⇒ 変数2にpop

仮想機械の動作

- 基本はスタックマシン

0	this
1	5
2	0
3	1

変数テーブル



スタック

⇒
変数3に1
足す

```

0 iconst_0 // total
1 istore_2 // i
2 iconst_0 // i
3 istore_3 // total
4 goto 14 // i
7 iload_2 // total
8 iload_3 // i
9 iadd // total
10 istore_2 // total
11 iinc 3 1 // i
14 iload_3 // i
15 iload_1 // n
16 if_icmplt 7 // total
19 iload_2 // total
20 return
    
```

以下同様に動く

Javaのメモリ管理

- スタックとヒープの2種類

– スタック: スタック構造でうまく扱えるもの

- 関数内の変数 (int, doubleなど)
- 関数の呼び出し
- オブジェクトへの参照

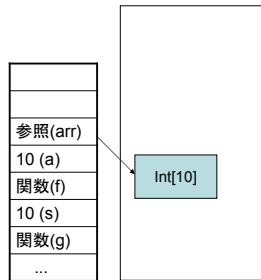
– ヒープ: スタックで扱わないもの

- オブジェクトの実体 (newするもの全て)

スタックとヒープ

```

int f( int a ) {
    int[] arr = new int[a];
}
int g( ) {
    int s = 10;
    f(s);
}
    
```



スタック

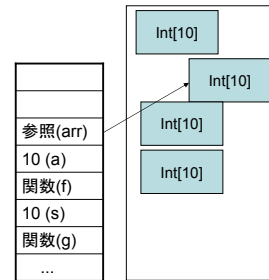
ヒープ

ヒープ

- 関数fをたくさん呼ぶ

```

int f( int a ) {
    int[] arr = new int[a];
}
int g( ) {
    int s = 10;
    f(s); f(s); f(s); f(s);
}
    
```



スタック

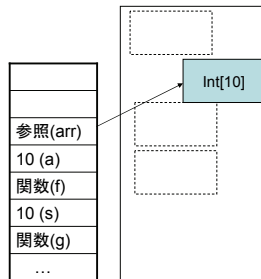
ヒープ

ごみ集め (Garbage Collection)

- ヒープの中の使っていない領域を回収

- Javaの場合, 基本的に自動で動く

- nullを代入すると次回, 回収してくれる
 - 例: arr = null;



スタック

ヒープ