

## 付録 A Oolong について

### A.1 Oolong とは

Oolong は JVM ( Java Virtual Machine ) のコードをターゲットとするアセンブリ言語である<sup>1</sup>。JVM は、仮想 CPU ( つまりソフトウェア上でシミュレートされる CPU ) の一種である。JVM の命令セットは本質的な部分は Intel x86 や MIPS などの現実の CPU と似ている。しかし、レジスタベースではなく、スタックベースなので、レジスタ割り付けの必要がなく、現実の CPU を対象とするよりはコード生成が容易である。

このため、本演習では、作成するコンパイラのターゲットとして、Intel x86 などの現実の CPU の機械語ではなく、この Oolong ( すなわち JVM のアセンブリ言語 ) を使用する。

JVM はもともと Java というプログラミング言語のコンパイラのターゲットとして設計された仮想機械である。Java 言語自体は、C 言語によく似た制御構造を持つ “高水準” プログラミング言語であり、Java とアセンブリ言語である Oolong とは、まったくの別物である。( C 言語と Intel x86 のアセンブリ言語が全く異なるのと同じことである。 ) 実際、JVM をターゲットとする Java 言語以外のプログラミング言語のコンパイラも、数多く存在する。

### A.2 Oolong の実行方法

Oolong の処理系 ( アセンブラ ) 自体も JVM 上で実装されているので、Oolong を実行するためには、まず JVM ( JRE という Java の実行環境 ) と oolong.jar というファイルを手に入れる必要がある。

Oolong ソースファイルの拡張子は .j である。 *Filename.j* という名前の Oolong ファイルをアセンブルする時のコマンドは次のようになる。

```
java -jar oolong.jar Filename.j
```

oolong.jar を別のフォルダに置いている場合は、上記の oolong.jar の部分は oolong.jar のフルパスを記述する。このコマンドで、 *Filename.class* というファイルが生成される。このファイルの中身は、仮想機械用のコードなので、直接実行することはできない。実行するには、次のように java コマンドを用いる。

<sup>1</sup>Oolong については、以下の書籍で紹介されている。

Joshua Engel: “Programming for the Java Virtual Machine” ADDISON-WESLEY, ISBN 0-201-30972-6

```
java Filename
```

この時は、最後に拡張子の.class をつけないことに注意する。

### A.3 Oolong のファイル構造

本演習で使用する Oolong ファイルは、すべて次のような雛型に従う。そして、斜字体の部分の部分を必要に応じて書き換える。Oolong の文法では、改行は意味を持っているので、この例のとおり改行を入れる必要がある。

```
.super java/lang/Object  
.method public static main([Ljava/lang/String;)V  
  
    Statements  
  
.end method
```

なお、クラス名はソースファイル名の拡張子(.j)を除いた部分と同じ名前になる。

### A.4 Oolong の命令文 ( *Statements* )

*Statements* は命令文 ( *Statement* ) の並びである。Oolong では、必ず 1 行に 1 つの命令文を書く。本演習では、次のような命令文を使用する。浮動小数点数関係など、ここで紹介していないその他の Oolong ( すなわち JVM ) の命令文については、Sun Microsystems 社の Web ページ ( <http://java.sun.com/docs/books/vmspec/> ) や Jasmin<sup>2</sup> の Web ページ ( <http://cat.nyu.edu/~meyer/jvmref/> ) で知ることができる。

JVM はスタックベースの仮想機械である。つまり、Oolong のほとんどの命令文は ( レジスタではなく ) スタックに置かれているデータをパラメータとして動作する。以下では、スタック操作関係、分岐関係、整数演算関係、変数操作関係、その他にわけて Oolong の命令文を紹介する。以下の説明中で、a はスタックの先頭の要素、b はスタックの 2 番目の要素を表す。「増減」はスタック中の要素数の変化を表す。

<sup>2</sup>Jasmin は Oolong の基になった JVM アセンブラである。Jasmin と Oolong の文法はほぼ同一であるが、Jasmin で必要ないいくつかの宣言を、Oolong では省略することが可能である。

## スタック操作関係

| 命令                      | 増減 | 説明                               |
|-------------------------|----|----------------------------------|
| <code>ldc Int</code>    | 1  | 整数定数をスタックにプッシュする。(load constant) |
| <code>ldc String</code> | 1  | 文字列定数をスタックにプッシュする。               |
| <code>dup</code>        | 1  | スタックの先頭の要素 a を複製する。(duplicate)   |
| <code>pop</code>        | -1 | スタックの先頭の要素 a を取り除く。              |
| <code>swap</code>       | 0  | スタックの先頭の 2 要素 a, b の順番を入れ換える。    |
| <code>nop</code>        | 0  | 何もしない。(no operation)             |

分岐関係 JVM は `goto` などの無条件分岐命令と、条件付きの分岐命令を持っている。JVM のコード中では、オフセットを指定することによりジャンプするが、Oolong のソースコードでは *Label* で分岐先を指定することができる。*Label* 名には、アルファベットからはじまり、空白文字を含まない任意の文字列を使用することができる。

| 命令                           | 増減  | 説明  |
|------------------------------|-----|---|
| <code>Label :</code>         | (0) | <code>goto</code> 文などの分岐命令の分岐先のラベルを設定する。  |
| <code>goto Label</code>      | 0   | <i>Label</i> に無条件に分岐する。   |
| <code>if.icmpeq Label</code> | -2  | a, b をスタックから取り除き (以下同様)<br><code>b==a</code> ならば、 <i>Label</i> に分岐する。<br>(if integer compare equal) |
| <code>if.icmpne Label</code> | -2  | <code>b!=a</code> ならば、 <i>Label</i> に分岐する。<br>(if integer compare not equal)                        |
| <code>if.icmpge Label</code> | -2  | <code>b&gt;=a</code> ならば、 <i>Label</i> に分岐する。<br>(if integer compare greater than or equal)         |
| <code>if.icmpgt Label</code> | -2  | <code>b&gt;a</code> ならば、 <i>Label</i> に分岐する。<br>(if integer compare greater than)                   |
| <code>if.icmple Label</code> | -2  | <code>b&lt;=a</code> ならば、 <i>Label</i> に分岐する。<br>(if integer compare less than or equal)            |
| <code>if.icmplt Label</code> | -2  | <code>b&lt;a</code> ならば、 <i>Label</i> に分岐する。<br>(if integer compare less than)                      |
| <code>ifeq Label</code>      | -1  | <code>a==0</code> ならば、 <i>Label</i> に分岐する。  |
| <code>ifne Label</code>      | -1  | <code>a!=0</code> ならば、 <i>Label</i> に分岐する。  |
| <code>return</code>          | -   | メソッドから値を返さずに <code>return</code> する。<br>注: A.3 節で紹介した雛型でも、メソッドの最後で必ず <code>return</code> する必要がある。   |

**整数演算関係** ここでは整数に関する算術演算の命令のみを紹介する。

| 命令   | 増減 | 説明   |
|------|----|--|
| iadd | -1 | b+a、加算 ( スタックから a と b を取り除き、b+a をスタックに積む。以下同様。 )<br>( <b>integer add</b> ) |
| isub | -1 | b-a、減算 ( <b>integer subtract</b> )   |
| imul | -1 | b*a、乗算 ( <b>integer multiply</b> )   |
| idiv | -1 | b/a、( 整数としての ) 除算 ( <b>integer divide</b> )                                |
| irem | -1 | b%a、( 整数としての ) 剰余 ( <b>integer remain</b> )                                |

**変数操作関係** JVM では、変数は番号で参照され、利用できる番号は 0 ~ 65535 までである。ちなみに、A.3 の難型の場合は、このうち 0 番の変数はメソッドの引数として使用済みである。

| 命令                | 増減 | 説明   |
|-------------------|----|--|
| iload <i>Int</i>  | 1  | <i>Int</i> 番目の変数の値をスタックにプッシュする。            |
| istore <i>Int</i> | -1 | スタックから a をポップし、その値を <i>Int</i> 番目の変数に格納する。 |

**その他** 整数や文字列を画面に出力するために必要な命令を紹介する。スタックの先頭 ( a ) に出力したいデータ、スタックの 2 番目 ( b ) に出力ストリームが入っている状態で、出力のための命令を呼び出す。

| 命令   | 増減 | 説明                           |
|--|----|------------------------------|
| getstatic java/lang/System/out                                 |    | Ljava/io/PrintStream;        |
|  | 1  | 標準出力ストリームをスタックにプッシュする        |
| invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V |    |                              |
|  | -2 | 出力ストリーム b に a ( 文字列 ) を出力する。 |
| invokevirtual java/io/PrintStream/println(I)V                  |    |                              |
|  | -2 | 出力ストリーム b に a ( 整数 ) を出力する。  |
| invokevirtual java/io/PrintStream/println(C)V                  |    |                              |
|  | -2 | 出力ストリーム b に a ( 文字 ) を出力する。  |

## A.5 Oolong のプログラム例

まず、“Hello World!” と出力する Oolong のコードを紹介する。

ファイル名: Hello.j

```
.super java/lang/Object
.method public static main([Ljava/lang/String;)V
  getstatic java/lang/System/out Ljava/io/PrintStream;
  ldc "Hello World!"
```

```

invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
return ; 最後に returnが必要
.end method

```

次は、繰り返しを用いて “Hello World!” を 10 回出力するプログラムである。  
 ファイル名: ManyHello.j

```

.super java/lang/Object
.method public static main([Ljava/lang/String;)V
  ldc 0
  istore 1 ; 変数 1 に 0 を代入する

  loop: ; ここからループ
    iload 1
    ldc 10
    if_icmpge exit ; 変数 1 が 10 以上なら exit へ
    getstatic java/lang/System/out Ljava/io/PrintStream;
    ldc "Hello_World!"
    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
    iload 1
    ldc 1
    iadd
    istore 1 ; 変数 1 に 1 を足す
    goto loop ; loop へジャンプ

  exit:
    return ; 最後に returnが必要
.end method

```

ちなみに、これらは、それぞれ次のような Java のプログラムのコンパイル結果に相当する。この中で System.out.println は C 言語の puts に相当する出力メソッドである。

ファイル名: Hello.java

```

public class Hello {
  public static void main(String[] args) {
    System.out.println("Hello_World");
    return;
  }
}

```

ファイル名: ManyHello.java

```

public class ManyHello {
  public static void main(String[] args) {
    int i = 0
    while(true) {
      if (i>=10) break;
      System.out.println("Hello_World");
      i=i+1;
    }
  }
}

```

```
    return;  
  }  
}
```