付録 A　Oolongについて

A.1　Oolong とは

Oolong は JVM（Java 仮想機械）のコードをターゲットとするアセンブラである[1]。JVM は、Java というポピュラーな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（仮想 CPU）である。このため、JVM は、Windows, Linux, Solaris など多くのプラットフォームで利用可能である。しかし、JVM の命令セットは本質的な部分は Pentium や MIPS などの現実の CPU と似ている。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、Pentium などの現実の CPU ではなく、この Oolong（すなわち JVM のアセンブラ）を使用する。

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、まったく異なる別物である。（C 言語と Pentium のアセンブリ言語が全く異なるのと同じことである。）本演習では Java 自体の知識は全く必要としない。（Java は 3 年生の「計算機ネットワーク I」という授業で取り扱う。）

A.2　Oolong の実行方法

Oolong 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 JDK（Java の開発キット）と oolong.jar というファイルを入手する必要がある。

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 -classpath oolong.jar COM.sootNsmoke.oolong.Oolong 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。（`java` は JVM を起動するためのコマンドである。）

```
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。

---

[1] Oolong については、以下の書籍で紹介されている。

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

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、この例のとおりに改行を入れる必要がある。

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

Filename は生成されるクラスファイルの名前である。なお、慣習上、*Filename* 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する。

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[2]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。（この点で JVM は Pentium など現実の CPU よりも扱いが楽である。）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。

スタック操作関係

命令	増減	説明
ldc *Int*	1	整数定数をスタックにプッシュする。
ldc *String*	1	文字列定数をスタックにプッシュする。
dup	1	スタックの先頭の要素 a を複製する。
pop	-1	スタックの先頭の要素 a を取り除く。
swap	0	スタックの先頭の 2 要素 a, b の順番を入れ換える。
nop	0	何もしない。

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

[2]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 で必要ないくつかの宣言を、Oolong では省略することが可能である。

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
Label：	*(0)*	goto 文などの分岐命令の分岐先のラベルを設定する。
goto *Label*	0	*Label* に無条件に分岐する。
if_icmpeq *Label*	-2	a==b ならば、 *Label* に分岐する。
if_icmpne *Label*	-2	a!=b ならば、 *Label* に分岐する。
if_icmpge *Label*	-2	a>=b ならば、 *Label* に分岐する。
if_icmpgt *Label*	-2	a>b ならば、 *Label* に分岐する。
if_icmple *Label*	-2	a<=b ならば、 *Label* に分岐する。
if_icmplt *Label*	-2	a<b ならば、 *Label* に分岐する。
ifeq *Label*	-1	a==0 ならば、 *Label* に分岐する。
ifne *Label*	-1	a!=0 ならば、 *Label* に分岐する。
return	–	メソッドから値を返さずに return する。 **注:** A.3 の雛型でも、メソッドの最後で必ず return 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	a+b、加算
isub	-1	a-b、減算
imul	-1	a*b、乗算
idiv	-1	a/b、（整数としての）除算
irem	-1	a%b、（整数としての）除算の剰余

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 まである。ちなみに、A.3 の雛型の場合は、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 *Int*	1	*Int* 番目の変数の値をスタックにプッシュする。
istore *Int*	-1	スタックからポップし、その値を *Int* 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。スタックの先頭に出力したいデータ、スタックの 2 番目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	文字列を出力する。
`invokevirtual java/io/PrintStream/println(I)V`		
	-2	整数を出力する。
`invokevirtual java/io/PrintStream/println(C)V`		
	-2	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

```
.class public Hello
.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

```
.class public ManyHello
.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 `printf` に相当する（ただし、最後に改行文字を出力する）出

力関数である。 Java の制御構造の構文は C 言語とほとんど同じなので、説明なしでもだいたいのところは理解できるだろう。

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;
  }
}
```

Java のコンパイラのコマンドは javac である。上記のプログラムは、それぞれ次のようなコマンドでコンパイルできる。

```
javac Hello.java

javac ManyHello.java
```

Hello.class, ManyHello.class というファイル名で JVM 用のコードが生成される。