µGO: A Simple C Programming Language

Programming Assignment III

µGO Compiler for Java Assembly Code Generation

Due Date: 23:59, 6/28, 2018

This assignment is to generate Java assembly code (for Java Virtual Machines) of the given **µGO** program. The generated code will then be translated to the Java bytecode by the Java assembler, **Jasmin**. The generated Java bytecode should be run by the Java Virtual Machine (JVM) successfully.

1. Prerequisite

In Linux environment, you could prepare the development tools with following commands:

- **Lexical Analyzer** (Flex) and **Syntax Analyzer** (Bison)

```
$sudo apt-get install flex bison
```

- **Java Assembler** (Jasmin) is attached to the **Compiler hw3** file.
- **Java Virtual Machine** (JVM)

```
$sudo add-apt-repository ppa:webupd8team/java
$sudo apt-get update
$sudo apt-get install default-jre
$sudo apt-get install oracle-java8-installer
```

2. Java Assembly Code Generation

In this assignment, you have to build a µGO compiler. Figure 1 shows the big picture of this assignment and the descriptions for the execution steps are as follows.

- Build your µGO compiler by injecting the Java assembly code into your **flex**/**bison** code developed in the previous assignments.
- Run the compiler with the given µGO program (e.g., **test.go** file) to generate the corresponding Java assembly code (e.g., **test.j** file).
- Run the Java assembler, ***Jasmin***, to convert the Java assembly code into the Java bytecode (e.g., test.class file).
- Run the generated Java bytecode (e.g., **test.class** file) with JVM and display the results.
    - The example output messages during the parsing procedure and JVM execution are displayed in **Section 4 What Should Your Compiler Do?**

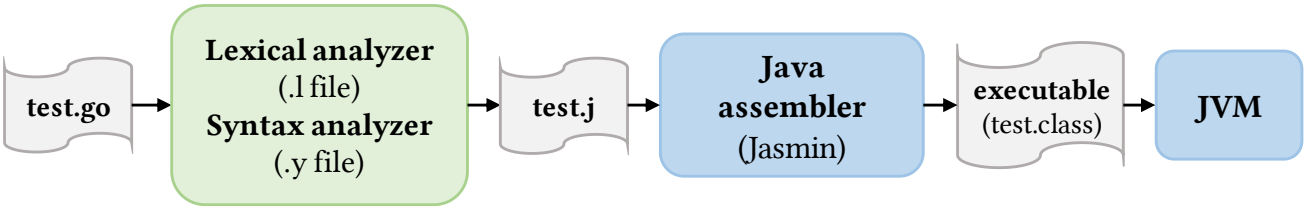

Figure 1. The execution flow for compiling the µGO program into Java bytecode for JVM

3. Java Assembly Language (Jasmin Instructions)

In this section, we list the Jasmin instructions that you may use in developing your compiler.

- **Operators**

The table below lists the µGO operators and the corresponding assembly code defined in Jasmin (i.e., Jasmin Instruction).

µGO Operator	Jasmin Instruction
+	iadd / fadd
-	isub / fsub
*	imul / fmul
/	idiv / fdiv
%	irem

- **Constants**

The table below lists the constants defined in µGO language. Also, the Jasmin instructions that we use to ***load*** the constants into the Java stack are given. More about the ***load*** instructions could be found in the course slides, Intermediate Representation.

Constant in µGO	Jasmin Instruction
94	ldc 94
8.7	ldc 8.7
“string”	ldc “string”

- **Arithmetic Operations**

The following example shows the standard binary arithmetic operation in µGO and the corresponding Jasmin instructions.

µGO Code	Jasmin Code
5+3	ldc 5 ldc 3 iadd

- **Store/Load Variables**

  The following example shows how to load the constant at the top of the stack and store the value to the local variable (`x = 9`). In addition, it then loads a constant to the Java stack, loads the content of the local variable, and adds the two values before the results are stored to the local variable (`y = 4 + x`). Furthermore, the example code exhibits how to store a string to the local variable (`z = "Hello"`). The contents of local variables after the execution of the Jasmin code are shown in the right.

µGO Code

```
x = 9
y = 4 + x
z = "Hello"
```

Jasmin Code

```
ldc 9
istore 0
ldc 4
iload 0
iadd
istore 1
ldc "Hello"
istore 2
```

Data structure of storage

Local Variable	Content
0 (x)	9
1 (y)	13
2 (z)	Hello

- **Print**

  The following example shows how to print out the constants with the Jasmin code. Note that there is a little bit different for the actual parameters of the `println` functions invoked by the **invokevirtual** instructions, i.e., **int (I)**, **float32 (F)**, and **string (Ljava/lang/String;)**.

µGO Code

```
println(30)
println("Hello")
```

Jasmin Code

```
ldc 30
getstatic java/lang/System/out Ljava/io/PrintStream;
swap
invokevirtual java/io/PrintStream/println(I)V
ldc "Hello"
getstatic java/lang/System/out Ljava/io/PrintStream;
swap
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
```

Screen (Output)

- **Casting Instruction**

  The following example shows the usage of the casting instructions, `i2f` and `f2i`, where `x` is int local variable 0, `y` is float32 local variable 1.

µGO Code	Jasmin Code

`x = x + y`	`iload 0`
	`i2f`
	`fload 1`
	`fadd`
	`f2i`
	`istore 0`

- **Jump Instruction**

  The following example shows how to use jump instructions (both conditional and non-conditional branches). **(x** is int variable**)**

Jasmin Instruction	Meaning
`goto <label>`	direct jump
`ifeq <label>`	jump if zero
`ifne <label>`	jump if nonzero
`iflt <label>`	jump if less than zero
`ifle <label>`	jump if less than or equal to zero
`ifgt <label>`	jump if greater than zero
`ifge <label>`	jump if greater than or equal to zero

µGO Code

```
if (x == 10) {
    /* do something */
} else {
    /* do something */
}
```

Jasmin Code

```
    iload 0
    ldc 10
    isub
    ifne Label_0
    /* do something */
    goto EXIT_0
Label_0:
    /* do something */
EXIT_0:
```

- **Execution Environment Setup**

  A valid Jasmin program should include the code segments for the execution environment setup. **Your compiler should be able to generate the setup code, together with the core Jasmin code** (as shown in the previous paragraphs). The example code is listed as below.

```
.class public main
.super java/lang/Object
.method public static main([Ljava/lang/String;)V
.limit stack 10      /* Define your storage size. */
.limit locals 3      /* Define your local space number. */

/*  … Your generated Jasmin code for the uGO program …  */

.end method
```

  Hint: You may refer to [1] [2] for the official documentations of Jasmin.

4. What Should Your Compiler Do?

Your compiler is expected to offer the basic features. To get bonus points, your compiler should be able to provide the advanced features.

In Assignment 3, the flex/bison file only need to print out the error messages, we score your assignment depending on your .j file and the JVM execution result.

Basic features (100pt)

Compile the given μGO program and generate the Jasmin program in a **.j file**. In particular, the points you get depend on what your compiler can do. Of course, the generated Jasmin code (e.g., arithmetic and variable store/load instructions in the .j file) after converting to the class file should be executed successfully by JVM.

- Handle **variable declarations** using local variables. (20pt)
  Note: When the variable declares without given the initial value, your compiler should automatically initialize its value to 0.
  Note: You do not have to worry about the casting in variable declaration.
- Handle **arithmetic operations** for integers and float32. (30pt)
  Note: You should handle the following operators: `+ - * / % ++ -- += -= *= /= %=` (**"++" and "--"** are **postfix expressions.**)
  Note: When the float32 variable/constant involves with the mod (%) operation, your compiler should take it as an illegal action.
- Handle the **print and println function**. (10pt)
  Note: Do not worry about the following situations: `println(x++)` and `print(x--)`.
- Handle the **if...else if...else statement**. (40pt)
  Note: The basic feature do not handle the scoping.

When ERROR occur during the parsing phase, we expect your compiler to print out **ALL** error messages, as Assignment 2 did, and **DO NOT** generate the Java assembly code (.j file).

Advanced features (30pt)

If you decide to challenge the advanced features in Assignment 3, please attach the README for explaining WHAT and HOW advanced function(s) you have implemented.

- Handle the **for statement**. (10pt)
- Handle the **scoping** for JVM. (10pt)
- Handle **user defined function**. (10pt)

Example

Input:

```
var x int
var a int = 2
var y float32 = 1.3

// pure int operation
x = a + 5 * 2

println(x)
println(y * 1.3)
println("Hello")

if (x > a) {
    x++
} else {
    x--
}

println(x)
```

JVM output:

```
12
1.69
Hello
13
```