µGo: A Simple Go Programming Language

Compiler 2022 Programming Assignment III
µGO Compiler for Java Assembly Code Generation
Due Date: June 9, 2022 at 23:59

Online Demonstration: June 10, 2022 from 12:00 to 18:30

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.

- Environmental Setup
  - Recommended OS: Ubuntu 18.04
  - Install dependencies: `$ sudo apt install flex bison`
  - Java Virtual Machine (JVM): `$ sudo apt install default-jre`
  - Java Assembler (Jasmin) is included in the Compiler hw3 file.
  - Judgmental tool: `$ pip3 install local-judge`

1. Java Assembly Code Generation

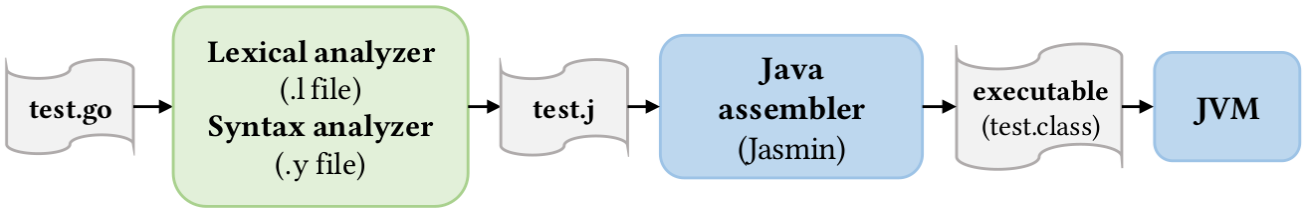

Figure 1. The execution flow for compiling the µGO program into Java bytecode for JVM

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.

2. Java Assembly Language (Jasmin Instructions)

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

2.1 Literals (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`
`"Hello world"`	`ldc "Hello world"`
`true` / `false`	`iconst_1` / `iconst_0` (or `idc 1` / `ldc 0`)

2.2 Operations

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

2.2.1 Unary Operators

µGO Operator	Jasmin Instruction (**`int32`**)	Jasmin Instruction (**`float32`**)
`+`	- (ignore or a blank)	- (ignore or a blank)
`-`	`ineg`	`fneg`

2.2.2 Binary Operators

µGO Operator	Jasmin Instruction (**`int32`**)	Jasmin Instruction (**`float32`**)
`+`	`iadd`	`fadd`
`-`	`isub`	`fsub`
`*`	`imul`	`fmul`
`/`	`idiv`	`fdiv`
`%`	`irem`	-

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

- µGO Code:

```
-5 + 3 * 2
```

- Jasmin Code (for reference only):

```
ldc 5
ineg
ldc 3
ldc 2
imul
iadd
```

2.2.3 Boolean Operators

µGO Operator	Jasmin Instruction
`&&`	`iand`
`||`	`ior`
`!`	`ixor` (`true xor b` equals to `not b`)

- µGO Code:

```
// Precedence: ! > && > ||
true || false && !false
```

- Jasmin Code (for reference only):

```
iconst_1    ; true (1)
iconst_0    ; false (2)
iconst_1    ; load true for "not" operator
iconst_0    ; false (3)
ixor        ; get "not" result (4) from (3)
iand        ; get "and" result (5) from (2),(4)
ior         ; get "or" result from (1),(5)
```

2.2.4 Comparison operators

You need to use subtraction and jump instruction to complete comparison operations. For int32, you can use `isub` . For float32, there is an instruction `fcmpl` is used to compare two floating-

point numbers. Note that the result should be bool type, i.e., 0 or 1. Jump instruction will be mentioned at section 2.6.

- µGO Code:

```
1 > 2
2.0 < 3.1
```

- Jasmin Code (for reference only):

```
    ldc 1
    ldc 2
    isub
    ifgt L_cmp_0
    iconst_0
    goto L_cmp_1
L_cmp_0:
    iconst_1
L_cmp_1:

    ldc 2.000000
    ldc 3.100000
    fcmpl
    iflt L_cmp_2
    iconst_0
    goto L_cmp_3
L_cmp_2:
    iconst_1
L_cmp_3:
```

2.3 Store/Load Variables

Relative operators: `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `++`, `--`.

2.3.1 Primitive Type

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 as below.

- µGO Code:

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

- Jasmin Code (for reference only):

```
ldc 9
istore 0         ; store 9 to x

ldc 4
iload 0          ; load x
iadd             ; add 4 and x
istore 1         ; store the result to y

ldc "Hello"
astore 2         ; store a string to z
```

2.4 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., int32 (`I`), float32 (`F`), and string (`Ljava/lang/String;`). Note also that you need to treat bool type as string when encountering print statement, and the corresponding code segments are shown as below.

- µGO Code:

```
println(30)
print("Hello")
print(true)
```

- Jasmin Code (for reference only):

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

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

```
    iconst_1 ; true
    ifne L_cmp_0
    ldc "false" ; we should load "false" and
                ; "true" as string literal for printing
    goto L_cmp_1
L_cmp_0:
    ldc "true"
L_cmp_1:
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/print(Ljava/lang/String;)V
```

2.5 Type Conversions (Type Casting)

The following example shows the usage of the casting instructions, `i2f` and `f2i`, where x is int32 local variable 0, y is float32 local variable 1.

- µGO Code:

```
x = x + int32(y)
```

- Jasmin Code (for reference only):

```
iload 0     ; x
fload 1     ; y
f2i         ; convert y to int32
iadd        ; add them
istore 0    ; store to x
```

2.6 Jump Instruction

The following example shows how to use jump instructions (both conditional and non-conditional branches). Jump instruction is used in if statement and for statement.

Jasmin Instruction	Description
`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 statement, x is an int32 variable):

```
if x == 10 {
    /* do something */
} else {
    /* do the other thing */
}
```

- Jasmin Code (for reference only):

```
    iload 0         ; load x
    ldc 10          ; load integer 10
    isub
    ifeq L_cmp_0    ; jump to L_cmp_0 if x == 0;
                    ; if not, execute next line
    iconst_0        ; false (if x != 0)
    goto L_cmp_1    ; skip loading true to the stack
                    ; by jumping to L_cmp_1
L_cmp_0:            ; if x == 0 jump to here
    iconst_1        ; true
L_cmp_1:
    ifeq L_if_false
                    ; -> do something
    goto L_if_exit
L_if_false:
                    ; -> do the other thing
L_if_exit:
```

- µGO Code (for statement, x is an float32 variable):

```
var x float32 = 10.0
for x > 0.0 {
  x--
  println(x)
}
```

- Jasmin Code (for reference only):

```
  ldc 10.000000
  fstore 0          ; store 10.0 to x
  L_for_begin:
  fload 0           ; load x for comparison
  ldc 0.000000
  fcmpl             ; compare float32 numbers
  ifgt L_cmp_0
  iconst_0
  goto L_cmp_1
L_cmp_0:
  iconst_1
L_cmp_1:
  ifeq L_for_exit   ; exit when the condition is false
  fload 0           ;---+
  ldc 1.0           ;   +--- (x--)
  fsub              ;   |
  fstore 0          ;---+
  fload 0           ; load x for println
  getstatic java/lang/System/out Ljava/io/PrintStream;
  swap
  invokevirtual java/io/PrintStream/println(F)V
  goto L_for_begin  ; goto loop begin
L_for_exit:
```

- µGO Code (switch statement, x is an int32 variable):

```
switch x {
    case 0: {
        /* do something A */
    }
    case 1: {
        /* do something B */
    }
    default: {
        /* do something C */
    }
}
```

- Jasmin Code (for reference only):

```
  iload 0   ; x
  goto L_switch_begin_0
L_case_0:
                        ; -> do something A
  goto L_switch_end_0   ; exit switch statement
L_case_1:
                        ; -> do something B
  goto L_switch_end_0   ; exit switch statement
L_case_2:
                        ; -> do something C
  goto L_switch_end_0   ; exit switch statement

L_switch_begin_0:
lookupswitch            ; a table with keys and labels
  0: L_case_0
  1: L_case_1
  default: L_case_2
L_switch_end_0:
  return
```

2.7 Method Invocation

There are several forms of method-calling instructions in the JVM. In this homework, methods are called using the `invokestatic` instruction. The usage can be shown by following example. A function `foo` has signature `(II)I` that you have implemented in hw2, and this information is used during the code generation. The `invokestatic Main/foo(II)I` is used to invoke the method `foo` after two actual argumants (`3` , `4`) are loaded to the stack, and then the result of the function output will be pushed to the stack.

- µGO Code:

```
package main
func foo(x int32, y int32) int32 {
    return x + y
}
func main() {
    var z int32 = foo(3, 4)
    println(z)
    return
}
```

- Jasmin Code (for reference only):

```
.method public static foo(II)I  ; Define foo function
.limit stack 20
.limit locals 20
  iload 0 ; load the first argument
  iload 1 ; load the second argument
  iadd
  ireturn
.end method

.method public static main([Ljava/lang/String;)V
.limit stack 100
.limit locals 100
  ldc 3     ; push argument to the stack
  ldc 4     ; push argument to the stack
  invokestatic Main/foo(II)I ; invoke `foo` method in `Main` class
  istore 2  ; store the result to z
  iload 2   ; load z for println
  getstatic java/lang/System/out Ljava/io/PrintStream;
  swap
  invokevirtual java/io/PrintStream/println(I)V
  return
.end method
```

2.8 Setup Code

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 translated Jasmin instructions (as shown in the previous paragraphs). The example code is listed as below.

- Filename: `hw3.j` (generated by your compiler)

```
.source hw3.j
.class public Main
.super java/lang/Object

; ... Your generated Jasmin code for the input µGO program ...

.method public static main([Ljava/lang/String;)V  ; main function
.limit stack 100  ; Define your storage size.
.limit locals 100 ; Define your local space number.

    ; ... Your generated Jasmin code for the input µGO program ...

    return
.end method
```

2.9 Workflow Of The Assignment

You are required to build a µGO compiler based on the previous two assignments. The execution steps are described as follows.

- Build your compiler by `make` command and you will get an executable named `mycompiler`.
- Run your compiler using the command `$ ./mycompiler < input.go`, which is built by lex and yacc, with the given µGO code (`.go` file) to generate the corresponding Java assembly code (`.j` file).
- The Java assembly code can be converted into the Java Bytecode (`.class` file) through the Java assembler, Jasmin, i.e., use `$ java -jar jasmin.jar hw3.j` to generate `Main.class`.
- Run the Java program (`.class` file) with Java Virtual Machine (JVM); the program should generate the execution results required by this assignment, i.e., use `$ java Main` to run the executable.

3. What Should Your Compiler Do?

In Assignment 3, the flex/bison file only need to print out the error messages, we score your assignment depending on the JVM execution result, i.e., the output of the command: `$ java Main`.

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

There are 11 test cases (each test case is 10pt and the total score is 110pt) in the Compiler hw3 file, and you can check the correctness by local-judge (type `judge` command in your terminal) as hw1 and hw2.

4. Submission

- Hand in your homework with Moodle.
- Only allow `.zip` format for compression.
- The directory organization should be (change all `StudentID` to your student ID number):

```
Compiler_StudentID_HW3.zip/
└── Compiler_StudentID_HW3/
    ├── compiler_hw3.l
    ├── compiler_hw3.y
    ├── compiler_hw_common.h
    ├── jasmin.jar
    └── Makefile
```

!!! Incorrect format will lose 10pt. !!!

5. Online Demonstration of Your Assignment 3

Demonstration will be held in virtual. The form and schedule of demonstration will be announced on Moodle later. During the demonstration, you will be asked to demonstrate your assignment downloaded from Moodle and you need to answer the questions about the logics of your codes in 5 ~ 10 minutes. The scores that you get for your Assignment 3 depend totally on how good your answers are. By default, the demonstration should be performed on TA's PC.

6. References

- Jasmin instructions: http://jasmin.sourceforge.net/instructions.html
- Java bytecode instruction listings: https://en.wikipedia.org/wiki/Java_bytecode_instruction_listings
- Java Language and Virtual Machine Specifications: https://docs.oracle.com/javase/specs/
- The Go (not µGo) Playground: https://go.dev/play/