µC: A Simple C Programming Language
Programming Assignment III
µC Compiler for Java Assembly Code Generation

Due Date: 23:59, 6/20, 2019

This assignment involves the code generation part, which is commonly found in modern compilers. Together with the code developed in the first two assignments, in this assignment, your **µC** compiler is expected to generate the Java assembly code (using Jasmin Instructions) with the input **µC** program. The generated code will then be translated to the **Java bytecode** by the **Java assembler, *Jasmin***. Lastly, the **Java Virtual Machine (JVM)** runs the generated **Java bytecode** and outputs the expected execution results.

1. Prerequisite

In Linux-based environments, you can prepare the required development tools with the following commands:

- **Lexical analyzer (Flex) and syntax analyzer (Bison):**

  $ sudo apt-get install flex bison

- **Java Virtual Machine (JVM):**

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

- **Check if JVM is installed properly:**

  You will see a short message about the version of your installed JVM if your JVM is installed properly with the command below.

  $ java -version

2. Workflow Of The Assignment

You are required to build a complete **µC** compiler based on the previous two assignments. Figure 1 illustrates the overall execution flow for your compiler.

The execution steps are described as follows.

- Run your ***compiler***, which is built by *lex* and *yacc*, with the given **µC** code (*.c 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***.
- Run the Java program (*.class file*) with ***Java Virtual Machine (JVM)***; the program should generate the **execution results** required by this assignment.

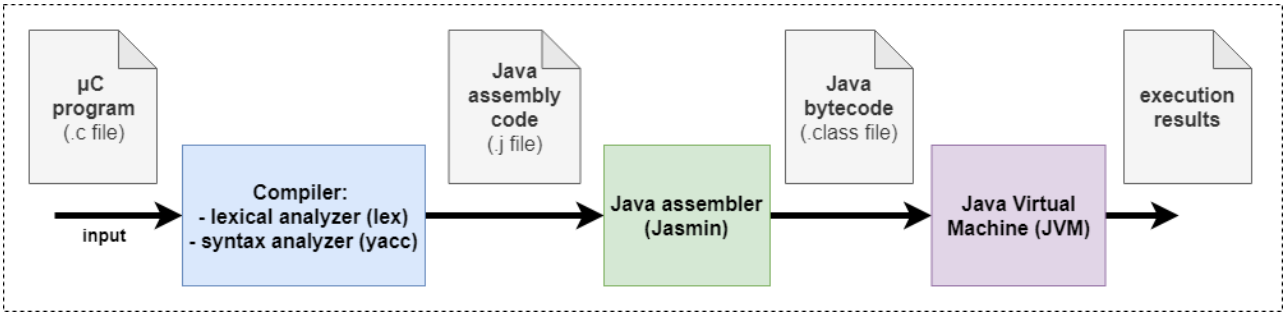

Figure 1. The execution flow for compiling the **µC** program into Java bytecode for JVM

3. Generating Java Assembly Code (Jasmin Instructions)

In this section, we list the Jasmin instructions that you may use in this assignment.

• Initialization

A **μC** program is translated into a Java class. Before the translated Java program can be executed, the **μC** compiler should generate the code for setting up the execution environment for it. An empty **μC** program is listed below.

```
// example_input.c

int main(){

}
```

The corresponding Java assembly code for the example program above.

```
; example_input.j

.class public example_input
.super java/lang/Object
.method public static main([Ljava/lang/String;)V
   .limit stack 15 ; up to 15 items can be pushed to the stack

   return
.end method
```

Once a program is parsed by your compiler, a declaration of the corresponding class name must be generated. Furthermore, the Java method *main*, which is declared public and static, must be generated in the program body.

• Operators for integer and floating-point numbers

The table below lists the **μC** operators and the corresponding assembly code defined in ***Jasmin.***

μC Operator	Jasmin Instr. (int)	Jasmin Instr. (float)
+	iadd	fadd
-	isub	fsub
*****	imul	fmul
/	idiv	fdiv
%	irem	-

- ## Constants

The table below lists the constants defined in **µC** language and the ***Jasmin*** instructions that we use to ***load*** the constants into the Java stack (int, float and string respectively). More about the ***load*** instructions could found in the course slides in "Intermediate Representation."

Constant in µC	Jasmin Instr.
94	ldc 94
8.7	ldc 8.7
"string"	ldc "string"

- ## Arithmetic operations

The following example shows the standard binary arithmetic operation in **µC** and Jasmin code.

µC Code	Jasmin Code
5 + 3	ldc 5
	ldc 3
	iadd

- ## More about variables and constants

While generating Java assembly code, you have to allocate storage for declared variables and constants. Variables can be classified into two types: global and local. We have already defined them clearly in Assignment 2. For example, all variables are said to be local if they are declared inside compound statements within the definition of a function, whereas other variables are global.

Global Variables

The global variables are considered as the fields of classes in Java assembly language. Fields will be declared right after the class name declaration. Each global variable will be declared as a static field by the form.

.field public static <field-name> <type descriptor> [= <value>]

The <type descriptor> is the type of variable <field-name>. The example **µC** program will be converted into the Jasmin code.

µC Code	Jasmin Code
int a; *float b = 3.14;* *bool c;*	*.field public static a I* *.field public static b F = 3.14* *.field public static c Z*

Local Variables

Local variables will not be declared *explicitly* in Java assembly programs. In fact, local variables will be numbered and accessed by the ***load*** / ***store*** instructions. In order to number local variables, symbol tables (implemented by Assignment 2) should be maintained to store their numbers (i.e., identifiers).

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

Jasmin Code

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

Data structure of storage

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

If the constant type is floating-point, you should use the floating-point instructions, *fstore*, *fload*, *fadd* and etc.

Loading Global Variables

If you want to load the global variables, use ***getstatic*** instruction.

```
// example_input.c

int a = 6; //global variable

void main(){
  int b;
  int c = 6;
  int d;
  b = a;
   ...
  d = c;
}
```

The above **µC** program is converted to the following Java assembly code.

.field public static a I = 6; declare a global variable a with integer type.
...
getstatic example_input/a I ; get global variable a in example_input field which is integer type
...
iload 2 ; local variable number of c is 2

- ## Print function

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 print functions invoked by the ***invokevirtual*** instructions, i.e., int, string and float.

µC	Jasmin Code
print(30);	ldc 30 getstatic java/lang/System/out Ljava/io/PrintStream; swap invokevirtual java/io/PrintStream/println(**I**)V
print(3.14);	ldc 3.14 getstatic java/lang/System/out Ljava/io/PrintStream; swap invokevirtual java/io/PrintStream/println(**F**)V
print("Hello");	ldc "Hello" getstatic java/lang/System/out Ljava/io/PrintStream; swap invokevirtual java/io/PrintStream/println(**Ljava/lang/String;**)V

- ## Casting instruction

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

µC Code	Jasmin Code
x = x + y	iload 0 i2f fload 1 fadd f2i istore 0

- ## Jump instruction (if, while statements)

The following example shows how to use jump instructions.

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

µC Code	Jasmin Code

if (x == 10) {
/* do something */
} else {
/* do something */
}
	iload 0
	ldc 10
	isub
	ifne Label_0
	/* do something */
	goto EXIT_0
	Label_0:
	/* do something */
	EXIT_0:

- ## Function definition and function fall

Function Definition

Functions are defined as **static methods** in **Java assembly languages**, as shown below.

.method public static <function name> (<arg1 type> <arg2 type> ...) <return type>

If there are ***n*** formal parameters for a **µC** function, these parameters are considered as the local variables for the Java method and the numbers (or identifiers) of the local variables are numbered from ***0*** to ***n – 1***.

```
int foo (int a, int b){
    ...
   return a + b;
}
```

The above **µC** program will be converted into the following Java assembly code.

```
.method public static add(II)I
.limit stack 15   ; Sets the maximum size of the operand stack required by the method
.limit locals 15 ; Sets the number of local variables required by the method
   iload 0
   iload 1
   iadd
   ireturn
.end method
```

If a **µC** function is declared without a return value, the return type of its corresponding Java method will be **void** and its Jasmin instruction will be ***return***.

Function Call

To invoke a static method, the instruction ***invokestatic*** will be used. The following code

µC Code	Jasmin Code
... = foo (a, 6);	*iload 2 ; if local variable of a is stored in 2* *ldc 6 ; constant 6* *invokestatic test/add(II)I*

- ## Execution Environment Setup

A valid Jasmin program should include the code for the execution environment setup, as described in "Initialization." 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 for setting up the execution environment is listed as below.

```
.class public compiler_hw3
.super java/lang/Object
.method public static main([Ljava/lang/String;)V
.limit stack 50              ; Define your storage size.
.limit locals 50                        ; Define your local space number.

;  ... Your generated Jasmin code for the uC program ...

.end method
```

NOTE:

1. The size for the stack and local variables of every method **will not exceed 50** in our test cases.
2. Please refer to "**References"** for more about the Jasmin instructions and rules. When in doubt, you should always check the references looking for your answers.

4. Assignment Requirement (130 pt)

The scores that you will get depending on how many test cases your compiler can pass. That is, **TAs will check only the execution results** of the Java assembly code produced by your **µC** compiler. If the results are incorrect, you will not get any scores for that test case. The categories of the test cases are listed below.

- **Variable declaration, including local, global and formal parameters. (20pt)**
  - If the variable is declared without given the initial value, your compiler should automatically initialize its value to 0.
  - Your compiler should have the **type casting** support during the generation of the variable declaration code. For example, the floating point number (3.14239) is assigned to the integer variable (a) during the variable declaration, and the content of the variable is the rounded integer number (3) in your symbol table.

```
int a = 3.14239; // -> a = 3;
float b = 3; // -> b = 3.0;
```

- **Arithmetic operations for both integer and float variables. (20pt)**
  - You compiler should handle the following operators.

Precedence	Operators	Category
1	++ --	Postfix
2	* / %	Multiplication
3	+ -	Addition
4	== != < <= > >=	Comparison
5	+= -= *= /= %=	Assignment

  - Your compiler must handle type casting.
  - Only integer variable can be used in the modulo operation (% and %=), as shown below.

```
int a = 3;
int b = 14;
int c;
c = 14 % a;
c = b % a;
c = 14 % 5;
```

- **Print function. (20 pt)**
  - Arithmetic operations will not be tested in your print function. That is, only the following four cases should be implemented in your compiler.

```
int a = 6;
string b = “hello”;
print(a);
print(8);
print(b);
print(“world”);
```

- **If statements (20pt) and while statements. (20pt)**
  - If statements and while statements are defined below. Your compiler should also consider about the nested structure of them.

```
if (expression 1){
   /* executes when expression 1 is true */
}
else if (expression 2){
   /* executes when expression 2 is true */
}
else{
   /* executes when all expression is false*/
}
```

```
while (expression 3){
   /* executes when expression 3 is true */
}
```

- **Functions. (20pt)**
  - Functions should be implemented with **return values** if there is a specific return type (other than *void*) in the function prototype. For example, the return type should be *int*, *float* or *bool*, if one of them is appeared in the return type of the function prototype.
  - The use (i.e., invocation) of a function should always happen after the function definition, which is the case used in all of our test cases. For example, the function *foo* is *defined* before it can be called in the *main* function.

```
int foo (int a, int b)
{
   return a + b;
}

void main()
{
   foo (6, 6);
   return 0;
}
```

  - All function return a value in our test cases, include **void** type function (return;). Main function is also void type because of the limitation of **Jasmin**. for example:

```
void foo1 (int a, int b){
   print(a + b);
   return;
}

int foo2 (int a, int b){
   return a + b;
}

void main(){
   return;
}
```

- **The syntax and semantic errors that should be caught by your compiler are listed as below. Not that some of them are identical to those in Assignment 2. (10pt)**
  - Your error messages should include the type of error and line number.
  - **Syntactic errors'** output and format are **the same with Assignment 2.**
  - **Semantic errors'** output and format are **the same with Assignment 2.**

```
printf("\n|---------------------------------------------|\n");
printf("| Error found in line %d: %s\n", yylineno, code_line);
printf("| %s", error_message);
printf("\n|---------------------------------------------|\n\n");
```

1. **Declaration and definition errors.**
   - Re-declarations and un-declaration variables / functions
2. **Arithmetic errors.**
   - Variables of numbers that divided by zero.
   - Modulo operator (%) with floating point operands.
3. **Function invocation.**
   - A function must be declared before it is invoked (used).
   - The type of the return statement of the generated code must match the return type of the function declaration.
   - The types of the actual parameters must be identical to the formal parameters in the function declaration.
   - The number of actual parameters must be identical to the function declaration.

5. Example Output and Jasmin Assembly code

Example input and ***Jasmin*** code will be found in Moodle, for example (function call) :

Example input:

```
int foo(int a) {
   a += 6;
   return a;
}

void lol (int a) {
   print(a);
   return;
}

void main(){
   int a;
   a = foo(4);
   lol(a);
   return;
}
```

Example generated *Jasmin* assembly code

```
.class public compiler_hw3
.super java/lang/Object
.method public static foo(I)I
.limit stack 50 ; stack size won't over 50 in our test cases
.limit locals 50 ; local size won't over 50 in our test cases
   iload 0
   ldc 4
   iadd
   istore 0
   iload 0
   ireturn
.end method
.method public static lol(I)V
.limit stack 50
.limit locals 50
   iload 0
   getstatic java/lang/System/out Ljava/io/PrintStream;
   swap
   invokevirtual java/io/PrintStream/println(I)V
   return
.end method
.method public static main([Ljava/lang/String;)V
.limit stack 50
.limit locals 50
   ldc 0
   istore 0
   ldc 6
   invokestatic compiler_hw3/foo(I)I
   istore 0
   iload 0
   invokestatic compiler_hw3/lol(I)V
   return
.end method
```

6. Submission

- Please upload your homework to Moodle before the deadline.
  - **Assignment Deadline: 23:59, June 20, 2019 (Thursday).**
  - **Late submission will not be allowed for Assignment 3, since the demonstration of your code is hled on the next day, from 9:00 to 14:00, June 21, 2019 (Friday).**
- **Your uploaded code must be organized as follows.**
  - All your files must be included in a **Compiler_<StudentID>_HW3** file and compress it either with .zip or .rar format (also named it as **Compiler_<StudentID>_HW3**). For example, John has the student id, F6905555, and the file name of his code is **Compiler_F6905555_HW3.rar**.
  - You are welcome to design your code or file structure to meet your expectations. Just make sure that you upload all of them to Moodle and it is able to compile your assignment with an easy ***make*** command.

Compiler_<StudentID>_HW3.zip
↳ *Compiler_<StudentID>_HW3/*
  ↳ *Makefile*
  ↳ *compiler_hw3.l*
  ↳ *compiler_hw3.y*
  ↳ *jasmin.jar*
  ↳ *other files if needed …*

7. Live Demonstration of Your Assignment 3

You are required to demonstrate your Assignment 3 from **9 a.m. to 2 p.m.** on **June 21, 2019** in Room **65704**, CSIE Building in our campus. Demonstration schedule 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. **The scores that you get for your Assignment 3 depend totally on how good your answer is.** By default, the demonstration should be performed on TA's PC. Nevertheless, you can bring your laptop to the demonstration site, so that it can be used to do the demonstration in case something goes wrong.

8. References

- The Java Virtual Machine Specification, 2nd Edition (The Java Language Specification, Java SE 12 Edition)
  https://docs.oracle.com/javase/specs/
- Java bytecode instruction listings (Wikipedia)
  https://en.wikipedia.org/wiki/Java_bytecode_instruction_listings
- Jasmin homepage
  http://jasmin.sourceforge.net/