μC: A Simple C Programming Language

Programming Assignment III

μC Compiler for Java Assembly Code Generation

Due Date: 12:00, 6/23, 2017

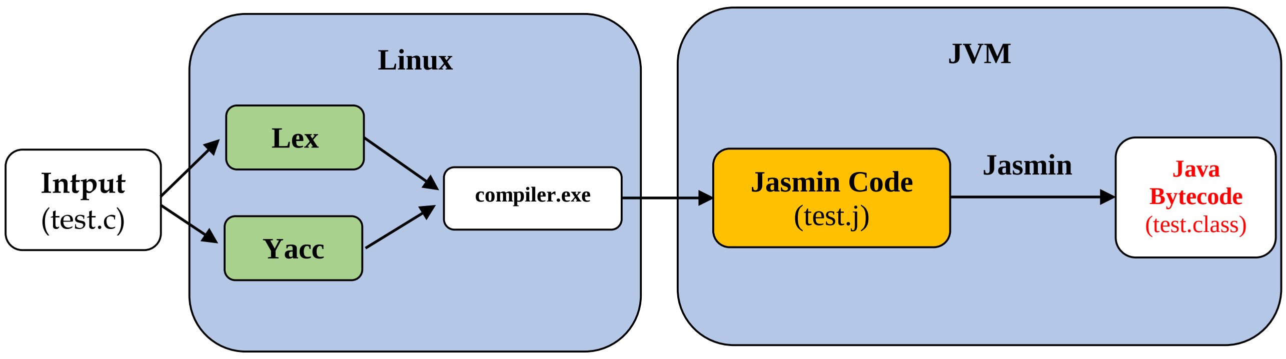
This assignment is to generate code (in Java assembly language) for the **μ**C language. The generated code will then be translated to the Java bytecode by the Java assembler, Jasmin. The generated Java bytecode should be run by the Java Virtual Machine (JVM) successfully.

1. Java Assembly Code Generation

In this assignment, you have to build a **μ**C compiler. The figure below illustrates the overall execution flow for your built compiler.

In particular, you need to do the following:

- Build your **μ**C ***compiler*** by injecting the Java assembly code into your **lex** and/or **yacc** code developed in the previous assignments.
- Run the ***compiler*** with the given **μ**C code (e.g., test.c 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 results to be displayed are divided into two parts:
      a) the parsing procedure for the given **μ**C code (similar to the output of the previous assignment), and
      b) the JVM execution result of your generated Java bytecode.
    - ■ The example output messages during the parsing procedure and JVM execution are displayed in Section 3 What Should Your Program Do?



2. Java Assembly Language (Jasmin Instructions)

- **Operators**

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

µC Operator	**Jasmin Instruction**
+	iadd
−	isub
*	imul
/	idiv

- **Constants**

  The table below lists the constants defined in µ**C** 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 µC	**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 **µ**C and Jasmin code.

µC Code	**Jasmin Code**
	ldc 5
5+3	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. In addition, it 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. Furthermore, 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	Content
0	9
1	13
2	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** and **string**.

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)

```
30
Hello
```

- **Execution Environment Setup**

  A valid Jasmin program should include the code 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 uC program ...   */

.end method
```

- **Misc.**

  You may refer to here for the official documentation of Jasmin.

3. What Should Your Program Do?

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

Basic features (100pt)

Compile the given **μ**C code and generate the Jasmin code 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 the μC code with arithmetic operations for integers. (60pt)
- Handle the **μ**C code with variable declarations using local variables. (30pt)
- Detect syntax error (e.g., undeclared and re-declared variables) and display the error during compilation process (dump with Yacc code) and during JVM execution. Error messages may include the type of error and line number. (10pt)

Advanced features (35pt)

The following features are the same with the features in the second programming assignment, except that you should extend the previous assignment to generate the corresponding Jasmin code. Of course, the generated Jasmin code after converting to the class file should be executed successfully by JVM.

- Handle arithmetic operations for integers and doubles. The compiler should consider brackets, print function, precedence, and data type. (20pt)
- Design the grammar for the C-style **while loop**. (15pt)

Example:

Input 1 :

```
int a;
int b = 5;
a = b * 20;
print(a);
b = a / (5+5);
print(b);
print(“Hello”);
```

Input 2 :

```
int a;
int b = 5;
a = b * 20;
print(a);
a = c * 15;
print(b / 0);
```

Output 1 (Dump by yacc) :

```
Create symbol table
Insert symbol: a
Insert symbol: b
Mul
ASSIGN
Print : 100
Add
Div
ASSIGN
Print : 10
Print : "Hello"

Total lines : 7
The symbol table :
ID       Type    Data
a        int     100
b        int     10

Generated: Assignment_3.j
```

Output 1 (Dump by JVM) :

```
100
10
Hello

Compile Success!
```

Output 2 (Dump by yacc) :

```
Create symbol table
Insert symbol: a
Insert symbol: b
Mul
ASSIGN
Print : 100
<ERROR> cannot find the variable c -
----- on 5 line
Mul
ASSIGN
<ERROR> The divisor can not be 0 ---
--- on 6 line
Print : 0

Total lines : 6

The symbol table :
ID       Type    Data
a        int     0
b        int     5

Generated: Assignment_3.j
```

Output 2 (Dump by JVM) :

```
100
0

Compile Failure!
Exist 2 errors
```

4. Template

Lex

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include "y.tab.h"       /* header file generated by bison */
extern int yylineno;
%}

%%
[0-9]+"."[0-9]+    {
                        //printf("%s is a float number \n", yytext);
                        sscanf(yytext,"%lf",&yylval);
                        return NUMBER;
                   }
[0-9]+             {
                        //printf("%s is a number \n", yytext);
                        sscanf(yytext,"%lf",&yylval);
                        return NUMBER;
                   }
[\n]               { yylineno++; return '\n'; }
[\t]    ;

.                  { return yytext[0]; }

%%
int yywrap()
{
    return 1;
}
```

Yacc

```
%{
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
extern int yylineno;
extern int yylex();
void yyerror(char *);
FILE *file;
%}
%union {
       int intVal;
}
%token NUMBER
%type <intVal> factor group term expression NUMBER
```

```
%%
lines
    :
    | lines expression '\n'    {
         printf(" = %d\n", $2);
         fprintf(file, "getstatic java/lang/System/out Ljava/io/PrintStream;\n"
                       "swap;swap the top two items on the stack \n"
                       "invokevirtual java/io/PrintStream/println(I)V\n" );
                             }
    ;
expression
    : term     { $$ = $1; }
    | expression '+' term {printf("Add  \n"); $$ = $1 + $3; fprintf(file,"iadd \n");}
    | expression '-' term {printf("Sub  \n"); $$ = $1 - $3; fprintf(file,"isub \n");}
    ;
term
    : factor   { $$ = $1; }
    | term '*' factor  {printf("Mul  \n"); $$ = $1 * $3; fprintf(file,"imul \n");}
    | term '/' factor  {printf("Div  \n"); $$ = $1 / $3; fprintf(file,"idiv \n");}
    ;
factor
    : NUMBER   { $$ = $1; fprintf(file,"ldc %d\n" , $1);}
    | group    { $$ = $1; fprintf(file,"ldc %d\n" , $1);}
    ;
group
    : '(' expression ')' { $$ = $2; }
    ;
%%

int main(int argc, char** argv)
{
       file = fopen("Computer.j","w");
       fprintf(file,    ".class public main\n"
                        ".super java/lang/Object\n"
                        ".method public static main([Ljava/lang/String;)V\n"
                        "       .limit stack %d\n"
                        "       .limit locals %d\n",10,10);
       yyparse();
       fprintf(file,    "return\n"
                        ".end method\n");
       fclose(file);
       printf("Generated: %s\n","Computer.j");
       return 0;
}

void yyerror(char *s) {
    printf("%s on %d line \n", s , yylineno);
}
```

Java Assembly Code (j file, generated by your compiler (lex/yacc)):

```
.class public main
.super java/lang/Object
.method public static main([Ljava/lang/String;)V
.limit stack 10
.limit locals 10

ldc 45
ldc 52
iadd
getstatic java/lang/System/out Ljava/io/PrintStream;
swap                ;swap the top two items on the stack
invokevirtual java/io/PrintStream/println(I)V

ldc 65
ldc 92
iadd
getstatic java/lang/System/out Ljava/io/PrintStream;
swap                ;swap the top two items on the stack
invokevirtual java/io/PrintStream/println(I)V

ldc 9
ldc 52
imul
getstatic java/lang/System/out Ljava/io/PrintStream;
swap                ;swap the top two items on the stack
invokevirtual java/io/PrintStream/println(I)V

return

.end method
```