PROJECT ASSIGNMENT 4
Code Generation

Due Date: 11:59PM, January 17, 2017
(No late submission is accepted!)

Your assignment is to generate code (in Java assembly language) for the C− (C minus) language using a syntax-directed translation or a translation scheme. The generated code will then be translated to Java bytecode by a Java assembler.

1 Assignment

Your assignment will be divided into the following parts:

- parsing declarations for constants and variables

- code generation for expressions and statements

- code generation for conditional statements and while/for loops

- code generation for procedure calls

1.1 Language Restrictions

In order to keep the code generation assignment simple that we can implement most of the features of the language, only a subset set of C− language will be considered in this assignment:

- No declaration or use of arrays.

- No string variables, i.e. no assignments to string variables. Only string constant and string literals are provided for uses in PRINT statements.

1.2 What to Submit

You should submit the following items:

- your compiler

- Makefile (**Please make sure it works well. TAs will rebuild your parser with this makefile. No further grading will be performed if the** *make* **process fails.**)

1.3 Java Assembler

Jasmin (`http://jasmin.sourceforge.net/`) is an assembler for the Java Virtual Machine. It takes ASCII descriptions of Java classes, written in a simple assembler-like syntax using the Java Virtual Machine instruction set. It converts them into binary Java class files, suitable for loading by a Java runtime system. It is available for non-commercial purpose. The program can be downloaded from SourceForge (`http://sourceforge.net/project/showfiles.php?group_id=100746`).

2 Generating Java Assembly Code

This section describes the four major pieces (see Section 1) of the translation of C− programs into Java assembly code. This document presents methods for code generation in each piece and gives sample C− along with the Java assembly code. Please note the label numbers in the examples are arbitrarily assigned. In addition, an extra piece will be added to generate code for initialization.

2.1 Initialization

A C− program is translated into a Java class. An empty C− program

```
// test.c
void main() {
}
```

will be translated into the following Java assembly code

```
; test.j
.class public test
.super java/lang/Object

.method public static main([Ljava/lang/String;)V
    .limit stack 15  ; up to 15 items can be pushed
    return
.end method
```

Consequently, once the program name is parsed, a declaration of the corresponding class name must be generated. Furthermore, a method `main` that is declared public and static must be generated for the compound statement in the program body.

2.2 Declarations for Variables and Constants

Before generating Java assembly commands for C− statements, you have to allocate storage for declared variables and store values of constants.

2.2.1 Allocating Storage for Variables

Variables can be classed into two types: global and local. All variables that are declared inside compound statements are local, while other variables are global.

Global Variables

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

```
.field public static <field-name> <type descriptor>
```

where `<type descriptor>` is the type of variable `<field-name>`. For example,

```
int a, b;
bool c;
```

will be translated into the following declaration statements in Java assembly

```
    .field public static a I
    .field public static b I
    .field public static c Z
```

Note: see `http://java.sun.com/docs/books/jvms/second_edition/html/ClassFile.doc.html#1169` for detailed specification of descriptors in JVM.

Local Variables

Local variables in C− will be translated into local variables of methods in Java assembly. Unlike fields (i.e. global variables of C−), local variables will not be declared explicitly in Java assembly programs. Instead, local variables will be numbered and instructions to reference local variables take an integer operand indicating which variable to use. In order to number local variables, symbol tables should maintain a counter to specify "the next available number". For example, consider the following program fragment:

```
{
  int i, j;

  {
    int k;
  }

  {
    int i, j, k;
  }
}
```

the symbol table information will be

```
entering block, next number 1
  i = 1, next number 2
  j = 2, next number 3
entering block, next number 3
  k = 3, next number 4
leaving block, symbol table entries:
  <"k", variable, integer, 3>
entering block, next number 4
  i = 4, next number 5
  j = 5, next number 6
  k = 6, next number 7
leaving block, symbol table entries:
  <"i", variable, integer, 4>
  <"j", variable, integer, 5>
  <"k", variable, integer, 6>
leaving block, symbol table entries:
  <"i", variable, integer, 1>
  <"j", variable, integer, 2>
```

2.2.2 Store Constants in Symbol Table

Constant variables in C− will not be transformed into fields or local variables in Java assembly. The values of constant variables will be stored in symbol tables.

2.3 Expressions and Statements

2.3.1 Expressions

An expression can be either an variable, a constant variable, an arithmetic expression, or a boolean expression.

Variables

Since string variables are not considered in this project and furthermore Java Virtual Machine does not have instructions for boolean, a variable will be loaded to the operand stack by *iload* (for integer type) or *fload* (for float type) or *dload* (for double type) instruction if it is local or *getstatic* if it is global. Consider the following program fragment

```
int a;
float b;
bool c;

int main() {
    int d;
    float e;
    bool f;
    ... = a ...;
    ... = b ...;
    ... = c ...;
    ... = d ...;
    ... = e ...;
    ... = f ...;
}
```

The translated program will contain the following Java assembly instructions

```
getstatic test/a I
getstatic test/b F
getstatic test/c Z
...
iload 1  ; local variable number of d is 1
fload 2  ; local variable number of e is 2
iload 3  ; local variable number of f is 3
```

Constants

The instructions to load a constant onto the operand stack in Java Virtual Machine are *iconst value*, *sipush value*, and *ldc value*. If the constant is a string, a boolean, or an integer/float/double number, we use *ldc string* for a string constant or *ldc float_point_constant* for a float constant. Consider the following program fragment

```
const int a = 10;
const bool b = true;
const string s = "string";
const float r = 1.23;
```

```
int main() {
    ... = a ...;
    ... = b ...;
   print s;
    ... = 5 ...;
    ... = r ...;
}
```

The translated program will contain the following Java assembly instructions

```
sipush 10
...
iconst_1
...
ldc "string"
...
sipush 5
...
ldc 1.23
```

Arithmetic and Boolean Expressions

Once the compiler performs a reduction to an arithmetic expression or a boolean expression, the operands of the operation will be already on the operand stack. Therefore, only the operator will be generated. The following table lists the mapping between operators in C− and corresponding instructions in Java assembly languages.

C− Operator	Boolean	Integer	Float
+	–	iadd	fadd
–	–	isub	fsub
*	–	imul	fmul
/	–	idiv	fdiv
%	–	irem	–
– (neg)	–	ineg	fneg
&&	iand	–	–
||	ior	–	–
!	ixor	–	–

Boolean expressions with relational operators and integer operands will be modeled by a subtraction instruction followed by a conditional jump. For instance, consider $a < b$.

```
    isub  ; a and b are at stack top
    iflt L1
    iconst_0  ; false = 0
    goto L2
L1:
    iconst_1  ; true = 1
L2:
```

Boolean expressions with relational operators and floating-point operands will be modeled by a floating-point comparison instruction (fcmpl) followed by a conditional jump. For instance, consider $a < b$.

```
    fcmpl  ; a and b are at stack top
    iflt L1
    iconst_0  ; false = 0
    goto L2
L1:
    iconst_1  ; true = 1
L2:
```

The following table summarizes the conditional jumps for each relational operator:

$<$	**iflt**
$<=$	**ifle**
$!=$	**ifne**
$>=$	**ifge**
$>$	**ifgt**
$==$	**ifeq**

2.3.2 Statements

Assignments $id = expression$;
The right-hand side, i.e. $expression$**, will be on the operand stack when this production is reduced. As a result, the code to generate is to store the value at stack top in** id**. If** id **is a local variable, then the instruction to store the result is**

```
    istore 2  ; local variable number is 2
```

On the other hand, if id **is global, then the instruciton will be**

```
    putstatic test/id <type descriptor>
```

where `<type descriptor>` is the type of id**.**

PRINT Statements $print\ expression$;
The PRINT statements in C$-$ are modeled by invoking the $print$ **method in** $java.io.PrintStream$ **class using the following format**

```
    getstatic java/lang/System/out Ljava/io/PrintStream;
    ...  ; compute expression
    invokevirtual java/io/PrintStream/print(Ljava/lang/String;)V
```

if the type of $expression$ **is a string. Types** int**,** $float$**, or** $boolean$ **will replace** $string$ **if the type of** $expression$ **is integer, float, or boolean.**

For instance, consider the type of $expression$ **is integer:**

```
    getstatic java/lang/System/out Ljava/io/PrintStream;
    ...  ; compute expression
    invokevirtual java/io/PrintStream/print(I)V
```

READ Statements $read\ variable\_reference$;
The READ statements in C$-$ are modeled by invoking the $nextInt$**,** $nextFloat$**,** $nextDouble$**, or** $nextBoolean$ **methods in** $java.util.Scanner$ **class using the following format**

```
; test.j
; hidden static field used to invoke java.util.Scanner.nextInt()...
.field public static _sc Ljava/util/Scanner;


; in the beginning of main block, create an instance of java.util.Scanner
new java/util/Scanner
dup
getstatic java/lang/System/in Ljava/io/InputStream;
invokespecial java/util/Scanner/<init>(Ljava/io/InputStream;)V
putstatic test/_sc Ljava/util/Scanner;
...
; invoke java.util.Scanner.nextInt()
getstatic test/_sc Ljava/util/Scanner;
invokevirtual java/util/Scanner/nextInt()I
```

For example, consider the following C− program:

```
int a;
bool b;
float c;
double r ;
...
read a;
read b;
read c;
read r;
```

The following code will be generated:

```
getstatic test/_sc Ljava/util/Scanner;
invokevirtual java/util/Scanner/nextInt()I
istore 1   ; local variable number of a is 1

getstatic test/_sc Ljava/util/Scanner;
invokevirtual java/util/Scanner/nextBoolean()Z
istore 2   ; local variable number of b is 2

getstatic test/_sc Ljava/util/Scanner;
invokevirtual java/util/Scanner/nextFloat()F
fstore 3   ; local variable number of c is 3

getstatic test/_sc Ljava/util/Scanner;
invokevirtual java/util/Scanner/nextDouble()D
fstore 4   ; local variable number of r is 4
```

2.4   If Statements and While/For Loops

It is fairly simple to generate code for IF and WHILE statements. Consider this if-then-else statement:

```
if( d == 1 ) {
```

```
        i = 5;
    }

    else {
        i = 10;
    }
```

The following code will be generated:

```
    iload 2 ; local variable number of d is 2
    ldc 1
    isub
    ifeq Ltrue_0
    iconst_0
    goto Lfalse_0
 Ltrue_0:
    iconst_1
 Lfalse_0:
    ifeq Lelse_1
    ldc 5
    istore 1 ; local variable number of i is 1
    goto Lexit_1
 Lelse_1:
    ldc 10
    istore 1 ; local variable number of i is 1
 Lexit_1:
```

For each **WHILE** loop, a label is inserted before the boolean expression, then a test and a conditional jump will be performed after the boolean expression. Consider the following **WHILE** loop:

```
    i = 1;

    while ( i < 10 ) {
        ...
        i = i + 1;
    }
```

and its correspondent using loop representation:

```
    for( i = 1 ; i < 10 ; i = i + 1 ) {
    ...
    }
```

The following instructions will be generated:

```
    ldc 1  ; constant 1
    istore 1  ; local variable number of i is 1
 Lbegin_0:
    iload 1
    sipush 10
```

```
    isub
    iflt Ltrue_1
    iconst_0
    goto Lfalse_1
  Ltrue_1:
    iconst_1
  Lfalse_1:
    ifeq Lexit_0
    iload 1
    sipush 1
    iadd
    istore 1
    goto Lbegin_0
  Lexit_0:
```

2.5   Procedure Declaration and Invocation

Procedures and functions in C− will be modeled by static methods in Java assembly languages.

2.5.1   Procedure Declaration

if n arguments are passed to a static Java method, they are received in the local variables numbered 0 through $n-1$. The arguments are received in the order they are passed. For example,

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

compiles to

```
.method public static add(II)I
.limit stack 2 ; Sets the maximum size of the operand stack required by the method
.limit locals 2 ; Sets the number of local variables required by the method
    iload 0
    iload 1
    iadd
    ireturn
.end method
```

If a procedure is declared without a return value, the type of its corresponding method will be void and the return instruction will be return.

2.5.2   Procedure Invocation

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

```
= add(a, 10) ...;
```

will be compiled into

```
iload 1  ; local variable number of a is 1
ldc 10   ; constant 10
invokestatic test/add(II)I
```

where the first and second I are the types of formal parameters and the last I is return type of the function.

3   Implementation Notes

The minimal requirement of this assignment is to implement *print*, *read*, and *assignment* statements with all basic data types (integer, float, double, boolean). Test patterns will be generated based on, but not limit to, these features. Notice that TAs will check only the execution result of the Java assembly code produced by your C− language compiler. If the result is not correct, you won't get credits for that pattern.

3.1   Local Variable Numbers

Local variables in a procedure or a function are numbered starting from 0, while local variables in the entry function are numbered starting from 1.

3.2   Java Virtual Machine and Java Disassembler

Once a C− program is compiled, the resulted Java assembly code can then be transformed into Java bytecode using the Java assembler *Jasmin*. The output of *Jasmin* will be a class file of the generated bytecode, which can be executed on the Java Virtual Machine. For example, type the following commands to generated test.class and to run the bytecode

```
% java -jar jasmin.jar test.j
% java test
```

Bytecode programs can be viewed using the Java Disassembler and the command format is

```
% javap -c test
```

There is also a tool, called *D-Java*, which disassembles Java bytecode into Java assembly code in *Jasmin* format. See https://www.vmth.ucdavis.edu/incoming/D-Java/djava.html for more information.

4   Project Demonstration

Project demonstration is requied for all course takers. Demonstration schedule will be later announced.

5   References

- **The Java Virtual Machine Specification, 2nd Edition**
  http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html

- **Java bytecode instruction listings (Wiki)**
  http://en.wikipedia.org/wiki/Java_bytecode_instruction_listings

- **Jasmin**
  http://jasmin.sourceforge.net/

- **D-Java**
  https://www.vmth.ucdavis.edu/incoming/D-Java/djava.html
  (This Website is down. See a backup from http://www.cs.nctu.edu.tw/~ypyou/courses/
  Compiler-f16/projects/D-Java/D-Java.htm.)

6 How to Submit the Assignment?

Create a directory, named "YourID" and store all files of the assignment under the direc-
tory. Zip the directory as a single archive, name the archive as "YourID.zip". Upload the
zipped file to the e-Campus (E3) system.

7 Example

Source C− program:

```
// test.c
int a, b;
bool d;
int foo( int a ) {
    int i, result;
    result = 0;
    i = 1;
    while( i <= a ) {
        result = result + i;
        i = i + 1;
    }
    return result;
}
int main() {
    int c ;
    read a;
    c = foo( a );
    print c;
    print "\n";
    if( c >= 100 ) {
        print "c >= 100 \n";
    }
    else {
        print "c < 100 \n";
    }
}
```

Generated Java assembly program:

```
; Line #1: // test1.c
; test1.j
.class public test1
.super java/lang/Object
```

```
.field public static _sc Ljava/util/Scanner;
; Line #2: int a, b;
.field public static a I
.field public static b I
; Line #3: bool d;
.field public static d Z
; Line #4: int foo( int a ) {
; Line #5:     int i, result;
.method public static foo(I)I
.limit stack 100    ;Sets the maximum size of the operand stack required by the method
.limit locals 100
; Line #6:     result = 0;
    ldc 0
    istore 2
; Line #7:    i = 1;
    ldc 1
    istore 1
; Line #8:    while( i <= a ) {
Lbegin_0:
    iload 1
    iload 0
    isub
    ifle Ltrue_1
    iconst_0
    goto Lfalse_1
Ltrue_1:
    iconst_1
Lfalse_1:
    ifeq Lexit_0
; Line #9:        result = result + i;
    iload 2
    iload 1
    iadd
    istore 2
; Line #10:       i = i + 1;
    iload 1
    ldc 1
    iadd
    istore 1
    goto Lbegin_0
; Line #11:    }
Lexit_0:
; Line #12:    return result;
    iload 2
    ireturn
; Line #13: }
.end method
```

```
; Line #14: int main() {
; Line #15:     int c ;
.method public static main([Ljava/lang/String;)V
.limit stack 100 ; up to 100 items can be pushed
.limit locals 100
    new java/util/Scanner
    dup
    getstatic java/lang/System/in Ljava/io/InputStream;
    invokespecial java/util/Scanner/<init>(Ljava/io/InputStream;)V
    putstatic test1/_sc Ljava/util/Scanner;
; Line #16:     read a;
    getstatic test1/_sc Ljava/util/Scanner;
    invokevirtual java/util/Scanner/nextInt()I
    putstatic test1/a I
; Line #17:     c = foo( a );
    getstatic test1/a I
    invokestatic test1/foo(I)I
    istore 1
; Line #18:     print c;
    getstatic java/lang/System/out Ljava/io/PrintStream;
    iload 1
    invokevirtual java/io/PrintStream/print(I)V ; this is print
; Line #19:     print "\n";
    getstatic java/lang/System/out Ljava/io/PrintStream;
    ldc "\n"
    invokevirtual java/io/PrintStream/print(Ljava/lang/String;)V ; this is print
; Line #20:     if( c >= 100 ) {
    iload 1
    ldc 100
    isub
    ifge Ltrue_2
    iconst_0
    goto Lfalse_2
Ltrue_2:
    iconst_1
Lfalse_2:
    ifeq Lelse_3
; Line #21:         print "c >= 100 \n";
    getstatic java/lang/System/out Ljava/io/PrintStream;
    ldc "c >= 100 \n"
    invokevirtual java/io/PrintStream/print(Ljava/lang/String;)V ; this is print
; Line #22:     }
    goto Lexit_3
; Line #23:     else {
Lelse_3:
; Line #24:         print "c < 100 \n";
    getstatic java/lang/System/out Ljava/io/PrintStream;
    ldc "c < 100 \n"
```

```
        invokevirtual java/io/PrintStream/print(Ljava/lang/String;)V ; this is print
; Line #25:     }
Lexit_3:
; Line #26: }
    return
.end method
```

Execution step for test1.j:

```
% java -jar jasmin.jar test1.j       // generate test1.class
% java test1 < input

file "input":
100


result:
5050
c >= 100
```