README for the bytecode code-generator project

Wuu Yang
National Chiao-Tung University, Taiwan, Republic of China
July 28, 2022

For the 4th project, you need to generate and run Java bytecode code. We recommend Java bytecode for this project. Java bytecode is much easier than other assembly languages. Jasmin (http://jasmin.sourceforge.net/) is the assembler for Java bytecode.

> (Quote from the Jasmin page)Jasmin 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.
>
> Jasmin was originally created as a companion to the book "Java Virtual Machine", written by Jon Meyer and Troy Downing and published by O'Reilly Associates. Since then, it has become the de-facto standard assembly format for Java. It is used in dozens of compiler classes throughout the world, and has been ported and cloned multiple times. For better or worse, Jasmin remains the oldest and the original Java assembler.

In order to use Jasmin, your compiler will generate Java bytecode (which is a text file) similar to test_array.j, fibonacci_recursive.j and qsort.j in the SAMPLE-CODE directory. The above three bytecode files are generated from the corresponding minipascal programs in the same directory.

Please study the example in SAMPLE-test-for-JASMIN-2.p (minipascal source code) and SAMPLE-test-for-JASMIN-2=This-contains-bugs.j (the corresponding Java bytecode). It includes many basic features, such as class,

functions, local variables, 1-dimensional and 2-dimensional arrays, function parameters, return values of functions, if-statements, while-loops, writeln, simple arithmetic expressions, assignments.

Java bytecode executes on Java Virtual machine. The "Jasmin Instructions.pdf" file explains the bytecode instructions. The "Jasmin User Guide20120117.pdf" file is a simple user guide. The "jasmin.jar" file is used to run the bytecode:

`java -jar jasmin.jar foo.j` (see below).

To run your project, use the following process:

1. `mycompiler.exe test00.p` (or `mycompiler.exe < test00.p`)

mycompiler.exe is your compiler. test00.p is a mini-pascal program. This command should generate a bytecode file named `foo.j`.

2. `java -jar jasmin.jar foo.j`

Next use jasmin to translate bytecode into a Java class file for execution. The above command would generate the foo.class file.

3. `java foo`

Finally, use Java Virtual Machine to execute foo.class.

Code generation for Jasmin

Java (and hence Java bytecode) is an object-oriented langauage. Minipascal is a procedural language. We do not all the features of bytecode in our code generator. In particular, features related to object orientation are not used.

To compile a minipascal program into bytecode, we create a single class, whose name (`foo` in the following example) is the same as the name of the minipascal program.

The first two lines of the bytecode file are usually

```
.class public foo
.super java/lang/Object
```

All global variables in a mini-pascal program are treated as static fields of the class.

```
.field public static a I
.field public static b I
.field public static d [I
.field public static e [F
.field public static g F
.field public static h F
.field public static x F
.field public static k [[F
.field public static l [[I
```

The global variables are accessed through

```
getstatic java/lang/System/out Ljava/io/PrintStream;
putstatic foo/c I
```

All functions are compiled into static methods in the bytecode. An example of a function is shown below.

```
.method public static addition(FF)F
.limit locals 100
.limit stack 100
ldc 0.0
fstore 2
fload 0
fload 1
fadd
fstore 2
fload 2
```

```
freturn
.end method
```

Because functions are compiled into static methods, function invocations are compiled into the `invokestatic` instructions. Your compiler will not use `invokevirtual` unless you have a different design.

```
invokestatic foo/addition(FF)F
```

The standard initializer method `<init>` of a class looks like

```
.method public <init>()V
aload_0
invokenonvirtual java/lang/Object/<init>()V
return
.end method
```

The parameters and local variables are treated as local variables. Operand stack size:

```
.limit stack 100
```

The number of local variables:

```
.limit locals 100
```

Jumps and labels:

```
if_icmple L8
ldc 111
```

```
putstatic foo/a I
goto L9
L8:
```

Input can be implemented with the Scanner class. For example, `read s` (where `*s` is a float value) can be implemented as follows:

```
getstatic allStatement/_sc Ljava/util/Scanner;
invokevirtual java/util/Scanner/nextFloat()F
fstore 0   // store in s
```

Output can be implemented with (for `printInt(u)`).

```
getstatic java/lang/System/out Ljava/io/PrintStream;
getstatic foo/u I
invokestatic java/lang/String/valueOf(I) Ljava/lang/String;
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
```

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

More details of the compilation is discussed in the file "minipascal-to-bytecode-guideline.txt".

Codegen test cases

In the CodegenTestCases directory are many test cases for your compiler. You may try your compiler on them.

Handin

For the 4th project, you need to turn in your lex files, yacc files, semantic routine files, code generator routines, test cases and the corresponding assembly files generated by your compiler, all related files, and the executable code. You need to prepare for Makefile. The TA will run your compiler with

```
make run
```

You also need to write a readme.txt file for the project, telling the TA how to build and run your compiler and additional details. In the readme.txt file, you also need to list all the features that your compiler includes.

Put all of the above files in a single zip file which will be named "DDDDDDD-codegen.zip", where DDDDDDD is your student id. Upload the zip to the e3new platform.

Deadline of the 4th project is January 20, 2023, Thursday, 23:55 pm. Please turn in your compiler project even if it only partially works. You will receive partial credits in that case.