Assignment 5: Code Generation

CSI4104 Compiler Design, Fall 2019
Document Version 1.5

Due date

- Please **do not submit** Assignment 5 before Friday, Dec 20, 2019! The grace-period of Assignment 4 ends on the previous day, and any submission before the end of the grace-period will overwrite your Assignment 4 submission!
- The due-date for Assignment 5 is Wednesday, Dec 25, 2019, **23:00.**
- Please note: **no extension nor grace period** can be granted for Assignment 5, because the university requires us to finish marking by Dec 27.

Introduction

Congratulations, you have made it to the last assignment of the project! After this assignment, you'll have accomplished a full compiler for a C-like programming language, which is a substantial achievement for a one-semester compiler course!

We will finish our compiler by adding a code generator. This code generator will take our decorated ASTs from Assignment 4 as input. We will generate code for the Java Virtual Machine (JVM). Specifically, we will generate assembly instructions that will be assembled to Java class files by the Jasmin bytecode assembler. This class files can then be executed on the JVM.
Our code generator will again be a Visitor of our AST class hierarchy. A skeleton of this visitor is provided (file Emitter.java). This visitor will traverse the AST and output (emit) Jasmin assembly instructions. The order of AST traversals and the instructions to be emitted are specified by the code templates in the lecture slides on Java bytecode generation.
To keep the workload manageable, the Visitor provided to you already covers the complete MiniC language, and you will have to fill in your code at specific places. Code generation for arrays and for floating point variables will be left out. The assignment walks you through the complete code generation process, to ensure maximum efficiency.

Specification

As before, if no lexical, syntactic or semantic error is found, the compiler should announce "`Compilation was successful.`", otherwise it should print "`Compilation was unsuccessful.`".
If the MiniC compiler is given input file `hello.mc`, then the output file will be `hello.j`. File `hello.j` will contain the Jasmin assembly code that has been generated

for file `hello.mc`. The Jasmin assembler will assemble `hello.j` into a classfile named `hello.class`. The bytecode in `hello.class` can then be run on the JVM.

Example (assuming you're in the top-level directory of your source-tree):

```
java -jar build/libs/MiniC-CodeGen.jar hello.mc
                           // compiles hello.mc to hello.j
jasmin hello.j             // assembles hello.j to classfile hello.class
java -cp build/classes/java/main:. Hello
// Executes the bytecode on the JVM. Note that the Java classpath  (cp)
// must be set to include the path to the MiniC lang package (see
// Section "MiniC built-in functions" below).
```

The Jasmin assembler is installed on our server. A jar-file is provided with the Assignment 5 archive. To use the jar-file, you need to provide the path to the file on the command-line. Assuming you're in the top-level directory of the Assignment 5 source-tree:

```
java -jar resources/jasmin_2.4/jasmin.jar hello.j
```

Alternatively, you can download Jasmin from http://jasmin.sourceforge.net/ if you want a full installation on your own computer.

We use a disassembler called *classfileanalyzer* (or *cfa*, for short) to disassemble Java class files into Jasmin assembly code. This is useful if you want to find out how the Java compiler translates a particular piece of Java code into bytecode. It is then easy to figure out how a similar MiniC statement should be translated (but note that most code templates are already given to you in the lecture slides on bytecode generation). The classfileanalyzer is installed on our server. You can download it from http://classfileanalyzer.javaseiten.de/ if you want to install it on your own computer.

Example: assume you are interested in how to generate code for MiniC procedure calls. As discussed in the lecture, MiniC procedures must be compiled to Java instance methods. We can now use a Java program that calls an instance method in order to figure out how our MiniC assembly code for procedure calls should look like. You could use the following piece of code to see what the Java compiler does:

```
// File small.java:
import lang.System; // the MiniC Standard Environment
public class small {

   public void foo() {
      System.putString("hello there!\n");
   }

   public static void main(String[] argv) {
      small s = new small();
      s.foo();
      return;
   }
}

javac -cp build/classes/java/main small.java  // compile to classfile
cfa small.class > small.j // disassemble small.class to small.j
```

Note: the above Java class uses the method System.putString() from the MiniC standard environment. To compile, the javac compiler needs the System.class file. You thus must provide the path to the class-files via the `-cp` command-line argument. In the above example, it is assumed that the small.java source file is in the toplevel directory of your source tree.

You are invited to have a look at the System.java class to see how the MiniC standard environment functions are implemented for/with Java.

Notes on the JVM

Bytecode that runs on the JVM must fulfill the requirements of the JVM specification. Most JVM implementations use a bytecode verifier before they execute bytecode. This bytecode verifier checks that bytecode conforms to the JVM specification. If bytecode violates this specification, the JVM will generate an exception and exit.
The JVM requires that bytecode methods end with a `return` statement. However, it is not required with MiniC to end a function with a `return` statement. Moreover, in Java, the main function is of return-type `void`, whereas the MiniC main function is of return type `int`. The quickest way to get around this issue is as follows.

(1.) Do not use a `return` statement in the MiniC main function. Our code generator in `Emitter.java` will insert a `return` statement at the end of main for you.
(2.) For all other MiniC functions, you should add proper `return` statements in the MiniC code.

The code that we will use for marking adheres to the above rules. If your own test cases violate Rule (2) above, the JVM will complain with "falling off the end of the code".

Proper MiniC example:

```
  void foo() { return; } // "return" is required at end of function
  int main(){}           // "return" will be inserted by code generator
```

The JVM requires that all variables are initialized before they are used. For global MiniC variables, a default initialization has already been implemented for you in the class initializer. For local variables, you should make sure in your MiniC code that variables are initialized before they are used. Note: the code generator skeleton provided to you does not implement variable initializations yet. Therefore you cannot use initializers like

```
int a = 0;
```

Instead, you have to use

```
int a;
a = 0;
```

The JVM requires to specify the number of operand stack cells that a method will use. This amount must be specified using the "`.limit stack <nr of cells>`" directive. We use an upper bound of 150 cells (this has already been implemented for you).

The JVM requires to specify the number of slots required for the local variable array of a method. This number must be provided using the "`.limit locals <nr of local variables>`". This number includes the "this pointer" of instance methods, the number of parameters of a method, and the number of local variables of the method. This has already been implemented for you.

The MiniC Built-In Functions

All built-in functions of MiniC have been implemented as static methods in the class `System.java`. Class `System.java` belongs to package `lang` (see the MiniC package description below).

Compiling MiniC Programs to Java Bytecode

The following assumptions have been made for every MiniC program:

(1.) A MiniC program is compiled as if it was a Java class. The name of that class is the filename of the MiniC program, with the suffix ".mc" removed.

(2.) All global MiniC variables become class variables (aka "static variables") in the bytecode.

(3.) All MiniC functions except the main function are compiled to *instance* methods (see lecture slides on the JVM regarding *instance* methods and *class* methods).

(4.) The MiniC main function is compiled like the following Java main method would be compiled by the javac compiler:

```
public static void main (String argv[]) {
  Classname mc;
  The original local variable declarations of the MiniC main function
  mc = new Classname();
  The original statements of the MiniC main function
  return;
}
```

Therein "Classname" is the name of the class as defined in Assumption 1. "mc" is an object reference to the class and initialized at the end of the variable declarations. Because every non-main MiniC function is an instance method in bytecode, every function call foo() is assumed to be mc.foo(). It is assumed that `main` cannot be called recursively.

The translation of those changes to the MiniC main function has been implemented for you in `Emitter.java`.

Useful AST Attributes

A new instance variable `int index` has been added to `MiniC.AstGen.Decl.java`. This variable is inherited by variable declarations (`MiniC.AstGen.VarDecl.java`) and by formal parameter declarations (`MiniC.AstGen.FormalParamDecl.java`). We use this instance variable to store the slot number (also called index number) of a local variable in the local variable array of a method.

When we process a variable declaration in the AST, we have to decide whether it is a local variable. If it is a local variable, we have to allocate a new variable slot from the current method frame (see description of the method frame below) and store the number of the slot in instance variable `index`. Later on when we encounter applied occurrences of this variable, we use the `index` value with JVM load and store instructions.

In order to distinguish global MiniC variables from local MiniC variables, another instance variable `boolean global` has been added to `MiniC.AstGen.Decl.java.` The value of this variable is already set for you. `MiniC.AstGen.Decl.java` provides you with method `isGlobal()` to determine if a variable is global or local.

The Method Frame

We generate a new method frame for every method that we visit. The method frame keeps track of the used local variable slots. It maintains a counter that keeps track of the number of slots already occupied. Whenever we allocate a new variable slot in the local variable array, this counter is incremented and the current number is returned (to be stored in `index`).

The second purpose of the method frame is to keep track of label numbers. Labels are symbolic addresses used in the bytecode to mark the targets of jump instructions (goto, ifeq, etc.).

```
;; Example bytecode using labels:
ifeq Label2   ;; conditional jump to Label2
goto Label3   ;; unconditional jump to Label3
Label2:       ;; definition of Label2
    i2f
Label 3:      ;; definition of Label3
    ....
```

Label definitions must be unique within a single method. The method frame maintains a counter for labels. Whenever we need to allocate a label, `frame.getNewLabel()` will return a unique label number.

For further details on the method frame please refer to package `MiniC.CodeGen.Frame.java` and the use of the method frame in the skeleton code generator in package `MiniC.CodeGen.Emitter.java.`

Writing Your Code Generator

We provide you with an updated version of the skeleton compiler that you should extend to create your MiniC code generator. You will find the new skeleton on the server in the archive file /opt/ccugrad/Assignment5/CSI4100_Assignment_5.tgz. Please set up your programming environment as specified in Assignment 1.

In the following, you find a description of the skeleton-files that have been added or changed compared to Assignment 4. All directories are relative to the top-level directory `CSI4100_Assignment_5`:

Package MiniC (directory MiniC)

`MiniC.java`: the new version of the compiler driver. If no errors occur during analyses, the compiler driver calls the code generator in Emitter.java.

Package Scanner (directory MiniC/Scanner)

`Scanner.java`: empty scanner implementation which you should replace by your own scanner from Assignment 1. In case your scanner from Assignment 1 does not work properly, you can use scanner classfiles provided with this assignment instead (see Section ``Provided Classfiles'' below).

Package Parser (directory MiniC/Parser)

`Parser.java:` empty parser implementation which you should replace by your own parser from Assignment 3. In case your parser from Assignment 3 does not work properly, you can use scanner and parser classfiles provided with this assignment instead (see Section "Provided Classfiles" below).

Package SemanticAnalysis (directory MiniC/SemanticAnalysis)

`SemanticAnalysis.java`: empty semantic analyzer implementation which you should replace by your own semantic analyzer from Assignment 4.

Package CodeGen (directory MiniC/CodeGen)

`Emitter.java`: this is the skeleton of a visitor to perform code generation on MiniC ASTs. You will add your code at specific places in this file. Those places are already marked by "`//TBD`".
`JVM.java, Frame.java`: the files implementing JVM instructions and the method frames.
Subdirectory tst/base/testcases: contains several MiniC testcases.
Subdirectory tst/base/solutions: contains Jasmin assembly code that our code generator produces for the testcases.

MiniC/scripts/code_test.sh

This is a simple bash-script which you might find useful to test your code generator. Note that this script might not be able to handle all oddities of code generators under development. Use at your own risk! If you experience problems, you might have to revert to manual testing. Note especially that this script must be run in the MiniC directory for the hard-coded paths to work!

resources/jasmin_2.4/

`jasmin.jar`: A binary distribution of the Jasmin bytecode assembler discussed in the lecture. See the above instructions in Section ``Specification" on how to use Jasmin.
`license-jasmin.txt`: jasmine license file.

Note that you do not have to produce exactly the same Jasmin assembly code. In fact, there are infinitely many correct ways to translate a program to Jasmin assembly code. During marking, we will only compare the output (generated by `put*` instructions) of programs compiled with your compiler against the output from our compiler.
The output for the testcases is provided on the website, like with Assignment 3 and 4, at http://elc.yonsei.ac.kr/courses/csi4104/csi4104_assignment_5.html . In addition, you can find it with the Jasmin solutions in directory CodeGen/tst/base/solutions.

Your Task:

File Emitter`.java` contains a visitor to perform code generation of MiniC ASTs. This analyzer is not complete yet---it has to be completed by you. At specific places in this code you find instructions on what you should do to complete this code generator. Those places have been marked with “`//TBD`”. Note that `Emitter.java` contains many helper functions at the beginning of the file. Those helper functions will be useful for code generation. A short comment describing the purpose of each function is provided.
You do not have to generate code for arrays or for floating point numbers, except for the i2f integer-to-floating point conversion with AST-node UnaryExpr.

`Emitter.java` already compiles. The simplest testcase, `HelloWorld.java`, already works out of the box. All functions from the MiniC StdEnvironment already work. This means that you can output integers, floats and Boolean values, which is useful for debugging of your code generator. The only function from the MiniC StdEnvironment that has not been implemented is `getString()`. We will not use this function.

The **code templates** from the lecture slides on Java bytecode generation outline the code that you should generate for each MiniC construct. If you are uncertain what Jasmin instructions to generate for a MiniC construct, you can compile a similar Java program and use the disassembler (classfileanalyzer) to convert the class file to a Jasmin assembly file.

The easiest way to debug your code generator is probably to compile small MiniC examples that contain only those constructs that you are currently implementing in your code generator. Looking at the generated code usually reveals problems quickly. Once the code looks sufficiently mature, you can assemble it to bytecode and run it on the JVM.

Provided Classfiles

In case your scanner or parser do not work properly, you can use classfiles provided in the ``resources'' directory-tree of the CSI4100_Assignment_5.tgz archive. The following gradle tasks are provided:

1) `jarNoScanner`: uses the classfiles in resources/scanner_only to provide a working scanner implementation. (Parser and semantic analyzer must be provided by the course participant.)
2) `jarNoScannerNoParser`: uses the classfiles in resources/scanner_and_parser to provide a working scanner and parser implementation. (The semantic analyzer must be provided by the course participant.)

Building, Running and Testing Your Code Generator

Like with the previous assignments, we use the Gradle build automation system to compile our MiniC compiler. The gradle build tasks are the same as with Assignment 4.

The MiniC compiler now accepts a set of command-line switches:

- `-ast1` to display the AST from the parser on-screen.
- `-ast2` to display the AST from semantic analysis on-screen.
- `-envast` to display the AST of the MiniC standard environment on-screen.
- `-t <file>` prints the AST from the parser in file <file>, using the TreePrinter.
- `-u <file>` unparses the AST from the parser into file <file>.

You can use the `-ast2` option to compare the AST with the bytecode that your compiler produced. You can compare the output of your compiler with the output provided on the course web site (http://elc.yonsei.ac.kr/course.htm).

Marking

Your code generator will be assessed by running your compiler on test cases and comparing the output with the correct output. Efficiency of the generated code will not be evaluated. The test files used for marking will be correct MiniC programs that are free of any kinds of errors. In each test case, all variables are initialized and all functions contain proper return statements.

Deliverables and Submission

Please do **not** submit Assignment 5 prematurely! See Section “Due date” at the beginning of this document for the time-period submission is allowed for Assignment 5.

You should extend the code generator in file `Emitter.java`. Note that for this assignment it is not necessary to modify any other file. Submission will therefore be restricted to this single file. If you need to submit additional files, you must let me know in advance. *You are not allowed to modify other files, e.g., the AST class hierarchy, Token.java aso, otherwise automated testing of your compiler might no longer work.*

To submit your file Emitter.java, please log in on our server, create a submission directory, e.g., ~/myassignment5, and copy your modified Emitter.java to this directory.

After copying relevant files to your submission directory, you must cd to the submission directory and type 'submit'. This will collect your file(s) and copy it to our repository.

```
[you@elc1 ~] cd ~/myassignment5
[you@elc1 myassignment5] submit
```

Please note: There is no submission unless you issue the 'submit' command! Failing to issue the 'submit' command will result in 0 points. The timestamp of your submission will be recorded by the 'submit' command. Please note also that if you re-submit, then a previous submission will be overwritten and a new timestamp will be recorded.

If you have questions, don't hesitate to contact us. ☺