Assignment 5: Code Generation

COMP3131/9102: Programming Languages and Compilers

Term 1, 2020

Worth: 28/100 marks

Due: 11:59pm Tuesday 28 April 2020

Revision Log

Nothing at this stage.

End Of Revision Log

Congratulations for having gone this far!
Let us conquer our last assignment!

1. Specification

This is the last assignment for the project! You are now on the way to build a compiler for VC, a fairly substantial language for a year 3 course!

You are to write a code generator for VC. This last assignment involves developing a visitor class (named `Emitter`) that implements the set of visitor methods in the interface `VC.ASTs.Visitor.java`. Your code generator will be a *visitor object* that traverses the decorated AST representing the source program, emitting Jasmin assembly instructions incrementally. Both the order of traversal and the instructions to be emitted are specified by code templates. Code templates for almost all language constructs are discussed in Lecture 10.

As before, if no lexical, syntactic or semantic error is found, your compiler should announce success by printing:

```
Compilation was successful.
```

and write the Jasmin assembly code in a file. Otherwise, the following message should be printed:

```
Compilation was unsuccessful.
```

If the input file name is test.vc, then the output file must be test.j.

There are four points to be noted below:

- In this assignment, we make one simplification about the VC language. In VC, int[] can be passed as an argument to a formal parameter expecting float[]. In this assignment, you can assume that this is NOT allowed. As a consequence, a formal parameter of type T[] requires the type of the corresponding argument to be exactly T[].
- In an assignment statement of the form LHS=RHS, the LHS must be translated before the RHS. This is the same as in Java. Please refer to the lecture slides on Code Generation for the code template used and an illustrating example.
- Following C, all global variables are initialised with constant expressions only. This is not checked in Assignment 4. This restriction will be satisfied by all test cases used for marking Assignment 5. You don't need to do anything special for this restriction.
- For an array declaration with an initialising expression as follows:

  ```
  T a[] = { blah1, blah2, ..., blahn };
  ```

  where the size of the array is not specified, you can assume that the AST constructed after our checker, which is used for marking your Assignment 5, has been invoked is exactly same as that for the same expression except than the array size is now explicit:

  ```
  T a[n] = { blah1, blah2, ..., blahn };
  ```

  The following example illustrates this:

  ```
  Program:                        int i[] = { 1, 2 };
  AST built by Parser:            AST-parser.gif
  AST decorated by Checker:       AST-checker.gif
  ```

2. Jasmin Assembly Language

Jasmin Home Page

Sun has not defined an assembler format for Java byte code. Jasmin is an assembly language for Java byte code. Jasmin is an assembler that reads as input a Jasmin assembly program and produces as output a Java class file ready to be run on a JVM. To access Jasmin installed in the 3131/9102 class accounts, run `3131` or `9102` to set up your environment. Then the command

```
jasmin Test.j
```

will assemble the file `Test.j` and place the output class file `Test.class` in the current directory.

If you have a PC yourself, you can download Jasmin from its home page. Alternatively, you can download the source code from the cs3131 account:

```
cd ~cs3131/VC/CodeGen
cp jasmin-2_04.zip your-directory
```

All the classes are pre-compiled. You can run the jasmin assembler once you have included `~INSTALL-DIR/jasmin/classes` in your CLASSPATH. The command to run the assembler on `Test.j` is:

```
java jasmin.Main Test.j
```

Jasmin Ops,Limc Instruction Reference,Manual

This manual describes all Jasmin instructions. You can also find more information at Jasmin Instructions. Only those listed in the file `VC.CodeGen.JVM.java` will be used. For every instruction used, you need to understand the effect of its execution on the operand stack. As an example, an `iadd` instruction adds the two values on the top of the operand stack and replaces them with the result of the operation. As another example, `i2f` converts the integer value on the top of the stack to the float result. Such a basic understanding is necessary for you to calculate the maximum depth of the operand stack used for a function.

Java Class File Disassembler

You can use the Java class file disassembler, javap, to translate a Java class file into assembly code. The command

```
javap -c Test > Test.j
```

will disassemble the class file and produce `Test.j`.

There are some other disassemblers:

- jasper
- BCEL

3. Installing the Built-In Functions

All the built-in functions of the VC language are implemented as static methods in the class `System.java`.

You can install this class as follows:

1. Create the subpackage `lang` under the package `VC`.
2. Copy `~cs3131/VC/lang/System.java` into your `VC.lang`.
3. Compile this Java file into a class file.

4. Interpreting VC Programs as Java Programs

The following assumptions are made about every VC program:

1. The name of the class is the same as the file name with the "." and the suffix "vc" removed.
2. All global variables are assumed to be class variables with the package access.
3. All functions (except the main function) are assumed to be instance methods with the package access.
4. The main function is interpreted to be the following Java main method:

   ```
   public static void main(String argv[]) {
    Classname vc$
    The original variable declarations
    vc$ = new Classname();
    The original statements
   }
   ```

   where Classname is the name of the class defined in Assumption 1. `vc$` is declared implicitly to be an object reference to the class and initialised at the end of the original variable declarations, if any. Since every non-main function is an instance method, every call f(...) in the main method is assumed implicitly to be `vc$.f(...)`. It is further assumed that the main method cannot be called recursively either directly or indirectly.

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

5. Attributes Useful in ASTs

Two additional attributes in the AST are used during code generation.

- There is an instance variable called `index` in the abstract class `VC.ASTs.Decl`. This variable, inherited by the concrete classes `VC.ASTs.LocalVarDecl` and `VC.ASTs.FormalPara`, is used to store the local variable index allocated to a variable or a formal parameter.
- There is an instance variable called `parent` in the abstract class `VC.ASTs.AST`. Therefore, `parent` is inherited by all the other AST classes. During the construction of the AST, as a node is linked to its child nodes, the child nodes are also linked to its parent at the the same time. In our implementation, this parent link is used in two occasions:
  - In a CompoundStmt node that represents the body of a function, its parent link can be used to access the formal parameters of the function so that the PL node of the function can be traversed.
  - In an AssignExpr node, its parent link can be used to determine if the top stack value needs to be duplicated by emitting a `dup` instruction, as discussed in Lecture 10.

6. Writing Your Code Generator

1. Set your current directory to VC.
2. Copy `~cs3131/VC/CodeGen/CodeGen.zip` into VC.
3. To extract all the files once you have retrieved the zip archive, type:

   ```
   unzip CodeGen.zip
   ```

The files in this zip file are listed below:

```
VC Package:

  vc.java                main compiler module

VC.CodeGen Package:

  Emitter.java:          Code generator skeleton
  Frame.java:            Information useful about translating a function
  JVM.java:              Simple JVM definition
  Instruction.java:      Instruction definition

  Test Files:            gcd.vc, max.vc

  Solution Files:        gcd.j, max.j
```

You need to at least change the two files: `JVM.java` and `Emitter.java`.

Modifying JVM.java

Presently, `JVM.java` defines all necessary instructions for dealing with scalar variables. You need to add to this file all instructions required for translating array-related constructs.

Modifying Emitter.java

`Emitter.java` does not compile. But we have already provided the implementation for the following:

- The generation of field declarations and the class initialiser <clinit> for all global **scalar** variables in `visitProgram`. But you are required to modify this method to deal with all array-related declarations and initialisations.
- The generation of the non-arg constructor initialiser <init>.
- Various visit methods.

Presently, Emitter.java consists of about 700 lines of code. You need to implement all missing visitor methods and complete those as indicated in the file. You will write approximately 300 lines of Java code.

The code templates for all constructs are discussed in Lecture 10. Those that are not have been implemented for you in `Emitter.java`.

All auxiliary methods provided in `Emitter.java` are self-explanatory. However, you are not obliged to use any of them if you feel constrained. You can develop your code generator from scratch if you so wish.

Only two test files are provided. The solution files are provided to give an indication about the Jasmin files produced. You are not required to generate exactly the same code.

The two directives `.limit locals` and `.limit stacks` must be contained in every Jasmin method. The total number of locals is simply the value returned from the call `frame.getNewIndex()`.

However, you must calculate yourself the maximum depth of the operand stack used for every function. As discussed in Lecture 10 and further demonstrated in the partially implemented `Emitter.java`, one way to find out the stack size is to simulate the execution of the byte code generated. You can delay counting the maximum stack size by using a constant number:

```
.limit stack 50
```

The generation of the `.var` directive is required and can be done in `visitLocalVarDecl` and `visitFormalPara` by using the information from the LocalVarDecl or FormalDec node and the labels from the two stacks `frame.scopedStart` and `frame.scopedEnd`.

The generation of the `.line` directive is optional.

If you are uncertain what Jasmin instructions to generate for a construct, you can compile a similar Java program and use the disassembler to convert the class file back into a Jasmin assembly file.

7. Parser

If you want to use our parser in case yours does not work properly, copy ~cs3131/VC/Parser/Parser-Sol.zip to the directory containing package VC and type:

```
unzip Parser-Sol.zip
```

This installs the class Parser.class under package VC.Parser. It is not necessary to understand how this parser works. Your type checker will only work on the AST constructed for the program by the parser.

8. Type Checker

You will be required to generate Jasmin code only for both syntactically and semantically correct programs. So this means that how well your type checker detects semantic errors is irrelevant. However, your type checker is assumed to have produced an annotated AST correctly. There are two kinds of annotations, as explained in Type Coercions in Assignment 4 Spec and illustrated by an example there:

- All operators have been replaced with non-overloaded integer or floating-point operators (such as `i+` and `f+`).
- The type conversion `i2f` operator has been added wherever it is necessary to change a number from integer to floating-point representation at run time.

If you want to use our checker in case yours does not work properly, copy ~cs3131/VC/Checker/Checker-Sol.zip to the directory containing package VC and type:

```
unzip Checker-Sol.zip
```

This installs Checker.class, SymbolTable.class and IdEntry.class under package VC.Checker and StdEnvironment.class under VC. It is not necessary to understand how our checker works.

However, our Checker solution will be released on 17 April, two days after the due date for Assignment 4 has passed.

9. Debugging Your Code Generator

The compiler options are still the same as in Assignment 4:

```
[jxue@daniel Checker]$ java VC.vc
====== The VC compiler ======

[# vc #]: no input file

Usages: java VC.vc [-options] filename

where options include:
        -d [1234]               display the AST (without SourcePosition)
                                1:  the AST from the parser (without SourcePosition)
                                2:  the AST from the parser (with SourcePosition)
                                3:  the AST from the checker (without SourcePosition)
                                4:  the AST from the checker (with SourcePosition)
        -t [file]               print the (non-annotated) AST into
                                (or filename + "t" if  is unspecified)
        -u [file]               unparse the (non-annotated) AST into
                                (or filename + "u" if  is unspecified)
```

If the source file being compiled is `test.vc`, the output Jasmin file will be `test.j`.

When designing your own test files, you need to be aware of the following differences between VC and Java:

Variable Initialisations

In Java, all variables must be initialised before used. Otherwise, the JVM will throw an "accessing value from uninitialised register ..." error. In VC, as in C, the same requirement is unspecified for local variables. But global variables are initialised according to the VC spec except that all global variables must be initialised only with constant expressions – **this is not checked in Assignment 4**. Therefore, you should use only test files for which all variables have been initialised.

return

In Java, the compiler makes sure that every method returning a value contains an appropriate return in every possible execution path. In VC, this context-sensitive restriction is specified in the language definition but not enforced by the type checker. If a return is missing, the JVM will throw a "falling off the end of the code" error. Therefore, you should use only test files in which return statements have been provided in all required places.

You will probably find that the easiest way to debug your code generator is to produce assembly code and look at it, rather than attempting to run it.

10. Marking Criteria

Your code generator will be assessed by running your compiler on some test cases and comparing the output with the correct output. Therefore, correct but highly inefficient code will not be penalised.

The test files used for marking your compiler will all be correct programs free of any kind of errors. In each test file, all variables are appropriately initialised and all functions contain return statements wherever they are required.

As before, there are no subjective marks.

11. Submitting Your Code Generator

```
give cs3131 emitter your-Java-files
```

You will definitely need to submit `Emitter.java` and `JVM.java`.

Submit `Frame.java`, `JVM.java` and `Instruction.java` if you have modified them.

You can ignore all the four files provided and do everything from scratch. But you are only allowed to submit the files with these names. No additional files will be accepted.

If your code generator need some "personal" information you have decorated in the AST, you may also submit your `checker.java`.

12. Late Penalties

This assignment is worth 28 marks (out of 100). You are strongly advised to start early and do not wait until the last minute. You will lose 5 marks for each day the assignment is late.

Extensions will not be granted unless you have legitimate reasons and have let the LIC know ASAP, preferably one week before its due date.

13. Plagiarism

As you should be aware, UNSW has a commitment to detecting plagiarism in assignments. In this particular course, we run a special program that detects similarity between assignment submissions of different students, and then manually inspect those with high similarity to guarantee that the suspected plagiarism is apparent.

If you receive a written letter relating to suspected plagiarism, please contact the LIC with the specified deadline. While those students can collect their assignments, their marks will only be finalised after we have reviewed the explanation regarding their suspected plagiarism.

This year, CSE will adopt a uniform set of penalties for the programming assignments in all CSE courses. There will be a range of penalties, ranging from "0 marks for the assessment item", "negative marks for the value of the assessment item" to "failure of course with 0FL."

Here is a statement of UNSW on plagiarism:

> **Plagiarism is the presentation of the thoughts or work of another as one's own.***
>
> Examples include:
>
> - direct duplication of the thoughts or work of another, including by copying material, ideas or concepts from a book, article, report or other written document (whether published or unpublished), composition, artwork, design, drawing, circuitry, computer program or software, web site, Internet, other electronic resource, or another person's assignment without appropriate acknowledgement;
> - paraphrasing another person's work with very minor changes keeping the meaning, form and/or progression of ideas of the original;
> - piecing together sections of the work of others into a new whole;
> - presenting an assessment item as independent work when it has been produced in whole or part in collusion with other people, for example, another student or a tutor; and,
> - claiming credit for a proportion a work contributed to a group assessment item that is greater than that actually contributed.†
>
> Submitting an assessment item that has already been submitted for academic credit elsewhere may also be considered plagiarism. Knowingly permitting your work to be copied by another student may also be considered to be plagiarism. An assessment item produced in a non-text form, or involving live presentation, may similarly contain plagiarised material.
>
> The inclusion of the thoughts or work of another with attribution appropriate to the academic discipline does not amount to plagiarism.
>
> Students are reminded of their Rights and Responsibilities in respect of plagiarism, as set out in the University Undergraduate and Postgraduate Handbooks, and are encouraged to seek advice from academic staff whenever necessary to ensure they avoid plagiarism in all its forms.
>
> The Learning Centre website is the central University online resource for staff and student information on plagiarism and academic honesty. It can be located at:
>
> www.lc.unsw.edu.au/plagiarism
>
> The Learning Centre also provides substantial educational written materials, workshops, and tutorials to aid students, for example, in:
>
> - correct referencing practices;
> - paraphrasing, summarising, essay writing, and time management;
> - appropriate use of, and attribution for, a range of materials including text, images, formulae and concepts.
>
> Individual assistance is available on request from The Learning Centre.
> Students are also reminded that careful time management is an important part of study and one of the identified causes of plagiarism is poor time management. Students should allow sufficient time for research, drafting, and the proper referencing of sources in preparing all assessment items.
>
> * Based on that proposed to the University of Newcastle by the St James Ethics Centre. Used with kind permission from the University of Newcastle
> † Adapted with kind permission from the University of Melbourne.

Have fun!

Jingling Xue
Last updated 04/06/2020 18:10:59