CS 536 — Spring 2016

Programming Assignment 5
CSX Code Generator

Due: Friday, May 6, 2016

Not accepted after Midnight, Tuesday, May 10, 2016

Your final assignment is to write methods in the class `CodeGenerating` that walk the AST for a CSX program and generate JVM assembler code. Your main program will call the CSX parser. If the parse is successful, the type checker is called. If the program contains no type errors, the code generator is called.

The CSX source program to be compiled is named on the compiler's command line or entered through a GUI. Error messages are written to standard output, and the JVM code generated is placed in file `name.j` where `name` is the identifier that names the CSX class.

A complete code generator for CSX-lite may be found at the following URL: www.cs.wisc.edu/~fischer/cs536.s16/course/proj5/startup/eclipse/.

The Code Generator

You will generate assembler code for the Java Virtual Machine (JVM). This is the same target machine that Java compilers assume. You will assemble the symbolic JVM instructions your compiler generates using the **Jasmin** assembler. Jasmin documentation is available on its homepage, which is linked to the class homepage (under "Useful Programming Tools"). The JVM instruction set (often called "bytecodes") is also described in the Jasmin documentation. Jasmin produces a standard format "`.class`" file, which can be executed using `java`, just as compiled Java programs are.

You will initiate code generation by creating an instance of class `CodeGenerating` by calling the constructor

```
new CodeGenerating(PrintStream asmFile)
```

The file parameter is the file into which JVM instructions are to be written. You then call the boolean-valued method `startCodeGen(root)` where `root` is the root of the AST built by the parser. This method will begin traversal of the AST, generating JVM code into `asmFile`.

Your code generator is only expected to handle type-correct programs; don't worry about translating type-incorrect programs. If any errors are detected during code generation, `startCodeGen` should return `false`; the contents of `asmFile` need not be valid. If no errors are detected by the code generator, `true` is returned and the contents of `asmFile` should be a valid JVM assembly program that can be assembled using `jasmin`.

Consider the following simple CSX program:

```
class simple{
   void main() {
        int a;
        read(a);
        print("Answer = ", 2*a+1, '\n');
}  }
```

This program might be translated into the following JVM assembler code:

```
.class public simple     ; This is a public class named simple
.super java/lang/Object  ; The superclass is Object

; JVM interpreters start execution at main(String[])
.method public static main([Ljava/lang/String;)V
invokestatic simple/main()V    ; call main()
return                         ; then return
.limit stack 2                 ; Max stack depth needed
.end method                    ; End of body of main(String[])

.method public static main()V  ; Beginning of main()
invokestatic CSXLib/readInt()I ; Call CSXLib.readInt()
istore 0                       ; Store int read into local 0 (a)
ldc "Answer = "                ; Push string literal onto stack
;   Call CSXLib.printString(String)
invokestatic CSXLib/printString(Ljava/lang/String;)V
ldc 2                          ; Push 2 onto stack
iload 0                        ; Push local 0 (a) onto stack
imul                           ; Multiply top two stack values
ldc 1                          ; Push 1 onto stack
iadd                           ; Add top two stack values
invokestatic CSXLib/printInt(I)V ; Call CSXLib.printInt(int)
ldc 10                         ; Push 10 ('\n') onto stack
invokestatic CSXLib/printChar(C)V    ; Call CSXLib.printChar(char)
return                         ; return from main()

.limit stack 25                ; Max stack depth needed(overestimate)
.limit locals 1                ; Number of local variables used
.end method                    ; End of body of main()
```

This program would be written into file `simple.j`, since the name of the CSX class is `simple`. The following command could be used to assemble the program into `simple.class`:

```
jasmin   simple.j
```

`simple.class` would then be executed using the command

```
java   simple
```

Translating AST Nodes

The following table outlines what your code generator is expected to do for each kind of AST node. Details of the translation process will be discussed in class and in handouts. Further information may also be found in the class notes.

Kind of AST Node	Code Generator Action
`classNode`	Generate beginning of class; generate body of `main(String[]);` translate `members`.
`memberDeclsNode`	Translate `fields`, then `methods`.
`fieldDeclsNode`	Translate `thisField`, then `moreFields`.
`methodDeclsNode`	Translate `thisMethod`, then `moreMethods`.
`varDeclNode`	Allocate a field or local variable index for `varName`. If `initValue` is non-null, translate it and generate code to store `initValue` into `varName`.
`constDeclNode`	Allocate a field or local variable index for `constName`; translate `constValue`; generate code to store `constValue` into `constName`.
`arrayDeclNode`	Allocate a field or local variable index for `arrayName`; generate code to allocate an array of type `elementType` whose size is `arraySize`; generate code to store a reference to the array in `arrayName`'s field or local variable.
`methodDeclNode`	Generate the method's prologue; translate `args`; translate `decls`; translate `stmts`; generate the method's epilogue.
`argDeclsNode`	Translate `thisDecl`, then `moreDecls`.
`valArgDeclNode`	Allocate a local variable index to hold the value of a scalar parameter.
`refArrayDeclNode`	Allocate a local variable index to hold a reference to an array parameter.
`stmtsNode`	Translate `thisStmt`, then `moreStmts`.
`asgNode`	If `source` is an array, generate code to clone it and save a reference to the clone in `target`. If `source` is a string literal, generate code to convert it to a character array and save a reference to the array in `target`. If `target` is an indexed array, generate code to push a reference to the array (using `varName`), then translate `target.subscriptVal`. Translate `source`; generate code to store `source`'s value in `target`.

Kind of AST Node	Code Generator Action

`incrementNode`
 	
`decrementNode`
	If `target.subscriptVal` is null generate code to push `target.varName`'s value onto stack. Push the integer 1 and generate an `iadd` or `isub`. Then store stack top into `target.varName`.
	
	Otherwise push the array reference stored at `target.varName`. Translate `target.subscriptVal`. Duplicate top two stack values using `dup2`. Generate an `iaload` or `caload`. Then push integer 1 and generate `iadd` or `isub`. Finally, generate an `iastore` or `castore`

`ifThenNode`	Translate `condition`; generate code to conditionally branch around `thenPart`; translate `thenPart`; generate a jump past `elsePart`; translate `elsePart`.
`whileLoopNode`	Create assembler labels for head-of-loop and loop-exit. If `label` is non-null store head-of-loop and loop-exit in `label`'s symbol table entry. Generate head-of-loop label; translate `condition`; generate a conditional branch to loop-exit label; translate `loopBody`; generate a jump to head-of-loop; generate loop-exit label.
`readNode`	Generate a call to `CSXLib.readInt`() or `CSXLib.readChar`() depending on the type of `targetVar`; generate a store into `targetVar`; translate `moreReads`.
`printNode`	Translate `outputValue`; generate a call to `CSXLib.printString(String)` or `CSXLib.printInt(int)` or `CSXLib.printChar(char)` or `CSXLib.printBool(boolean)` or `CSXLib.printCharArray(char[])`, depending on the type of `outputValue`; translate `morePrints`.
`callNode`	Translate `procArgs`; generate a static call to `procName`.
`returnNode`	If `returnVal` is non-null then translate it and generate an `ireturn`; otherwise generate a `return`.
`breakNode`	Generate a jump to the loop-exit label stored in `label`'s symbol table entry.
`continueNode`	Generate a jump to the head-of-loop label stored in `label`'s symbol table entry.
`blockNode`	Translate `decls`; translate `stmts`;

Kind of AST Node	Code Generator Action
`argsNode`	Translate `argVal`; translate `moreArgs`.
`binaryOpNode`	Translate `leftOperand`; translate `rightOperand`; generate JVM instruction corresponding to `operatorCode`.
`unaryOpCode`	Translate `operand`; generate JVM instruction corresponding to `operatorCode`.
`fctCallNode`	Translate `functionArgs`; generate a static call to `procName`.
`castNode`	If `resutltType` is `bool` and `operand` is an `int` or `char` then if `operand` is non-zero generate code to convert it to 1 (which represents true). If `resutltType` is `char` and `operand` is an `int` then generate code to extract the rightmost 7 bits of `operand`.
`name`	If `subscriptVal` is null generate code to push value at `varName`'s field name or local variable index. Otherwise, generate code to push the array reference stored at `varName`'s field name or local variable index; translate `subscriptVal`; generate an `iaload` or `baload` or `caload` based on `varName`'s element type.
`intLitNode`	Generate code to push `intval` onto the stack.
`charLitNode`	Generate code to push `charval` onto the stack.
`trueNode`	Generate an `iconst_1`.
`falseNode`	Generate an `iconst_0`.
`strLitNode`	Push `strval` onto stack using `ldc` instruction.
`nullNode`	Do nothing.
`intTypeNode`	Do nothing.
`boolTypeNode`	Do nothing.
`charTypeNode`	Do nothing.
`identNode`	Do nothing (name or index of identifier is used by parent nodes based on context).

How to Proceed

Start with simple constructs like read, print, assignment and simple expressions. Implement harder constructs like ifs, loops and methods after the simpler constructs are working. For each construct you implement, you have two things to do. First, you must decide *what* JVM code you want to generate. Try out the code you selected by creating (by hand) simple Jasmin assembler programs. Run them to verify that the code you selected really works.

Once you know the code you selected is viable, modify your code generator to generate that code. Look at the output of your code generator (the `name.j` file) to verify that what is generated *looks* correct. If the output looks correct, run it through Jasmin and `java` to verify that it *is* correct.

Once you've implemented a few simple constructs, you'll see how it all works. You can then add additional features until all of CSX is supported.

If you're in doubt as to what JVM code to generate, here's a useful trick. CSX programs closely correspond to Java classes (with all fields and methods declared static). Create a Java program that's equivalent to a particular CSX program. Compile the Java program using your favourite Java compiler (perhaps `javac`). Then run

```
javap -c file
```

where `file.class` is the class file created by `javac`. This will show you the JVM instructions selected by the Java compiler (in a slightly different format than that used by Jasmin). In most cases these instructions could be generated by your compiler to translate the CSX program in question.

Be careful that the JVM instructions that you generate don't try to access operands that aren't on the stack. Such instructions are illegal and can cause the Java interpreter (`java`) to crash.

What to hand in

As was the case for Project 4, your program should expect the name of a CSX program to be compiled on its command line (if no program name is found, a GUI will prompt you to enter one). If the CSX program is invalid, appropriate error messages should be written to standard output. Otherwise a translation of the CSX program should be placed in `name.j` where `name` is the program's class name. `name.j` should be executable using `jasmin` and then `java`.

Create a folder (directory) and name it using your first and last name (e.g., `CharlesFischer`). Copy into this folder a `README` file, a `build.xml` file and all source files necessary to build an executable version of your program (`.java` source files, a `csx.jlex` file and a `csx.cup` file). Do not hand in any `.class` files. Name the class that contains your `main` method `CSX.java`. Upload this handin folder to the `cs 536 project 5` Dropbox folder in learn@uw. You may compress your handin folder into a single file using zip if you wish. Partners should submit only one solution. The other partner should submit only a README file identifying the partnership

You may test your CSX compiler using the test programs at this URL: www.cs.wisc.edu/~fischer/cs536.s16/course/proj5/tests/. These programs are named `test-00.csx, test-01.csx,`.... Create a file named `CSXtests` that contains the results produced by compiling, assembling and running each of these programs. We'll also run your compiler on a variety of our own test programs.

If you wish to claim extra credit, *clearly* state (in the `README` file) what you've added and include examples of its operation.