CS 541 — Spring 2014

Programming Assignment 5
CSX Code Generator

Your final assignment is to extend the AST node classes to generate JVM assembler code for CSX programs. Your main program calls the CSX parser. If the parse is successful, it calls the type checker. If the program contains no type errors, it calls the code generator.

Your program takes the CSX source program to be compiled the command line, writes error messages to standard output, and places generated JVM code in file `name.j`, where `name` is the identifier that names the CSX class. Skeletons for the code generator may be found in `~raphael/courses/cs541/public/proj5/startup`.

The Code Generator

Your program generates assembler code for the Java Virtual Machine (JVM), which is the same target machine that Java compilers assume. You then 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 "bytecode") 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.

Initiate code generation by calling the member function

```
boolean codeGen(PrintStream asmfile)
```

in the root of your AST (which must be a `ProgramNode`). The parameter is the file into which JVM instructions are to be written. `codeGen` should store the `PrintStream` in an internal variable or field called `aFile`. It then calls the member function `cg`(), which traverses the AST, generating JVM code into `aFile`.

Your code generator need only handle type-correct programs; don't worry about translating type-incorrect programs. If it detects any errors during code generation, `codeGen` should return `false`; the contents of `aFile` need not be valid. If it detects no errors, it returns **`true`**, and the contents of `aFile` 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');
     } // main()
} // class simple
```

This program might translate into the following JVM assembler code:

```
   .class public simple        ; This is a public class named simple
   .super java/lang/Object     ; The super class 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()
   .limit locals 1             ; Number of local variables used
   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)
   .end method                 ; End of body of main()
```

Your generator stores this program in file `simple.j`, since the name of the CSX class is `simple`. The following command assembles the program into `simple.class`:

```
jasmin  simple.j
```

You would then execute `simple.class` 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.

Kind of AST Node	Code Generator Action
`ProgramNode`	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 ref erence to the array (using `varName`), then translate `target.subscriptVal`. Translate `source`; generate code to store `source`'s value in `target`.

`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(bool)` 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`;
`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 the 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 `var Name`'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, and assignment statements and simple expressions. Implement harder constructs like **`if`**, 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 you support all of CSX.

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 `javac`. Then run

```
javap -c file
```

where `file.class` is the class file created by `javac`. `javap` will show you the JVM instructions selected by the Java compiler (in a slightly different format than that used by **Jasmin**). In most cases, your compiler could generate these instructions 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 invalid and can cause the Java interpreter (`java`) to crash.

What to hand in

Test your CSX compiler using all the test programs included in `~raphael/-courses/cs541/public/proj5/tests`. These programs are named `test1.csx, test2.csx,`.... Create a file named `CSXtests` that contains the results produced by compiling, assembling and running each of these programs. You should add tests that cover things that these tests miss.

Your compiler program should take the name of a CSX program to be compiled on its command line. If the CSX program is invalid, your program should write appropriate error messages to standard output. Otherwise, it should place a translation of the CSX program in `name.j` where `name` is the program's class name. `name.j` should be executable using **jasmin** and then **java**. Submit a `README` file, a `Makefile`, your `CSXtests` file and all source files necessary to build an executable version of your program. Do not hand in `.class` files. Name the class that contains your `main` method `csx.java`. The grader will test your CSX compiler by compiling and executing a series of test programs.