CS 541 — Fall 2021

Programming Assignment 5
CSX_go Code Generator

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

Your program takes the file name of the CSX_go source program to be compiled on 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_go package. 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 machine that Java compilers target. 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 `.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 should be a `ProgramNode`). The parameter is the file into which JVM instructions are to be written. `codeGen()` 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 the output file need not be valid. If it detects no errors, it returns **`true`**, and the contents of the output file should be a valid JVM assembly program that *Jasmin* can assemble.

Consider the following simple CSX_go program:

```
package simple
  func main() {
      var a int;
      read a;
      print "Answer = ", 2*a+1, '\n';
  } // main()
// package 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_go package 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
```

Extra credit

Generate correct code for variables declared within the block bodies of **`if`** and **`for`** statements.

Generate correct code for expressions used to initialize global variables.

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 `variables`; translate `functions`.
`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.
`FuncDeclNode`	Generate the function's prologue; translate `args`; translate `decls`; translate `stmts`; generate the method's epilogue.
`ArgDeclsNode`	Translate all the formal declarations.
`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 all the statements.
`AsgNode`	If `source` is an array, generate code to clone it and save a reference to the clone in `target`. (This strategy implements shallow-copy semantics as opposed to pointer-copy semantics.) 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`.
`ForNode`	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 `CSX_goLib.readInt()` or `CSX_goLib.read Char()` depending on the type of `targetVar`; generate a store into `targetVar`; translate `moreReads`.
`PrintNode`	Translate `outputValue`; generate a call to `CSX_goLib.printString(String)` or `CSX_go Lib.printInt(int)` or `CSX_goLib.printChar(char)` or `CSX_goLib.printBool(bool)` or `CSX_goLib.print CharArray(char[])`, depending on the type of `out putValue`; translate `moreDisplays`.
`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`.

`FuncCallNode`	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`.
`NameNode`	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`**, assignment statements and simple expressions. Implement harder constructs like **`if`**, **`for`**, and functions after the simpler constructs are working. For each construct you implement, decide what JVM code you want to generate. Try out the code you select 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_go.

If you're in doubt as to what JVM code to generate, here's a useful trick. CSX_go programs closely correspond to Java classes (with all fields and methods declared

static). Create a Java program that's equivalent to a particular CSX_go program. Compile the Java program using *javac*. Then run

```
javap -c -p 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 from that used by **Jasmin**). In most cases, your compiler could generate these instructions to translate the CSX_go program in question.

Don't let the JVM instructions that you generate 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_go compiler using all the test programs included in `~raphael/-courses/cs541/public/proj5/tests`. These programs are named `test1.csx_go, test2.csx_go,`.... 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_go program to be compiled on its command line. If the CSX_go program is invalid, your program should write appropriate error messages to standard output. Otherwise, it should place a translation of the CSX_go 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 `P5.java`. The grader will test your CSX_go compiler by compiling and executing a series of test programs.