Assignment 5

Due April 3 at 11:59pm

In this assignment, we will implement code generation for part of the language. The rest will be added in Assignment 6. The abstract syntax shown below has been modified to indicate which part of the language we need to implement now and how they map into JVM elements.

Program ::= Name List<ParamDec> Block

class Name implements Runnable{

variables declared in List<ParamDec> are instance variables of the class

public Name(String args){

initialize instance variables with values from args.

}

public static void main(String[] args){

Name instance = new Name(args);

instance.run();

}

public void run(){

declarations and statements from block

}

}

ParamDec ::= type ident

instance variable in class, initialized with values from arg array

Block ::= List<Dec> List<Statement>

Decs are local variables in current scope of run method

Statements are executed in run method

Must label beginning and end of scope, and keep track of local variables, their slot in the local variable array, and their range of visibility.

Dec ::= type ident

Assign a slot in the local variable array to this variable and save it in the new slot attribute in the Dec class.

Statement ::= ~~SleepStatement~~ | WhileStatement | IfStatement | ~~Chain~~ | AssignmentStatement

~~SleepStatement ::= Expression~~

AssignmentStatement ::= IdentLValue Expression

store value of Expression into location indicated by IdentLValue

IMPORTANT:

insert the following statement into your code for an Assignment Statement after value of expression is put on top of stack and before it is written into the IdentLValue

```
CodeGenUtils.genPrintTOS(GRADE, mv,assignStatement.getE().getType());
```

~~Chain ::= ChainElem | BinaryChain~~

~~ChainElem ::= IdentChain | FilterOpChain | FrameOpChain | ImageOpChain~~

~~IdentChain ::= ident~~

~~FilterOpChain ::= filterOp Tuple~~

~~FrameOpChain ::= frameOp Tuple~~

~~ImageOpChain ::= imageOp Tuple~~

~~BinaryChain ::= Chain (arrow | bararrow) ChainElem~~

WhileStatement ::= Expression Block

goto GUARD
BODY Block
GUARD Expression
IFNE BODY

IfStatement ::= Expression Block
Expression
IFEQ AFTER
Block
AFTER …

Expression ::= IdentExpression | IntLitExpression | BooleanLitExpression
| ~~ConstantExpression~~ | BinaryExpression
always generate code to leave value of expression on top of stack.

IdentExpression ::= ident
load value of variable (this could be a field or a local var)

IdentLValue ::= ident
store value on top of stack to this variable (which could be a field or local var)

IntLitExpression ::= intLit
load constant

BooleanLitExpression ::= booleanLiteral
load constant

~~ConstantExpression ::= screenWidth | screenHeight~~

BinaryExpression ::= Expression op Expression
Visit children to generate code to leave values of arguments on stack
perform operation, leaving result on top of the stack. Expressions should
be evaluated from left to write consistent with the structure of the AST.

~~Tuple ::= List<Expression>~~

op ::= relOp | weakOp | strongOp

type ::= integer | ~~image | frame | file~~ | boolean | ~~url~~

The provided version of CodeGenVisitor.java also include a partially complete version of visitProgram. It is sufficient to generate code for an empty program. A test case in CodeGenVisitorTest with an empty program has been provided to illustrate how to put the pieces together, get useful output, etc. Use it to ensure that you have asm set up properly and as a starting point for your own tests. You will need asm and asm.util.

Because this fragment of our language does not have output, two routines, genPrint and genPrintTOS have been included with the provided version of CodeGenVisitor.java. These routines generate code to print a String and print the value on top of the stack (without consuming it), respectively. The CodeGenVisitor has a variable DEVEL and a variable GRADE that are set via parameters to the constructor. Both routines take a Boolean parameter that indicates whether these routines generate code or just to nothing. By adding these in appropriate places, you can easily generate code instrumented to show you what it is doing, and turn it off later.

In order to grade this assignment, we will be comparing the sequence of values written in assignment statements. (See assignment statement above.) Please follow directions exactly. This should be the ONLY call to genPrint of genPrintTOS that is passed the value of GRADE. You may insert calls with DEVEL anywhere you want. We will test with DEVEL = false and GRADE = true. The only output from the execution of your generated code should be these print statements.

It is difficult to write unit tests for these programs, nevertheless, Junit provides a convenient way to manage tests, even if they cannot easily automatically check the output. You should also be able to run your generated classfiles as a Java application program with the arguments on the command line.

Provided code:
CodeGenVisitor.java starter code for code generation. visitProgram sets up the class and generates the main method and constructor, and the framework of the run method. You will need to make some changes, but it should compile and deal with the empty program as is. Implement all the methods marked “Implement this”. Those marked `assert false : "not yet implemented"; will be done in assignment 6.`
CodeGenUtils.java some routines helpful during code generation
cop5556sp17.AST.Type.java A few useful modifications from the previous version.
Turn in: CodeGenVisitor.java, CodeGenVisitorTest.java, Parser.java, Scanner.java, SymbolTable.java, TypeCheckVisitor.java, all the classes in cop5555617.AST.
Troubleshooting Hints:
If there are exceptions in the call to visitMaxs, you can temporarily replace

```
cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);
```

at the beginning of visitProgram with

```
cw = new ClassWriter(0);
```

This will not yield a runnable class file, but it will allow you see the code that is being generated.
The JVM lecture also included some troubleshooting tips.