Assignment 6 & 7

Assignment 6 Due: Sun. Dec 3 at 11:59pm

Assignment 7 Due: Friday Dec 8 at 11:59pm

Implement a CodeGenVisitor class to traverse the decorated AST and generate code. The abstract syntax has been annotated to indicate which part of the language is to be implemented in Assignment 5 and how it maps into JVM elements. The rest of the language will be implemented in Assignment 6.

Complete the CodeGenVisitor class as described below. I have tried to add new instructions in red.

Program ::= name (Declaration | Statement)*

Generate code for a class called name.

statements are added to main method

Declaration ::= Declaration_Image | Declaration_SourceSink | Declaration_Variable

Declaration_Image :: = name (xSize ySize | ε) Source

Add a field to the class with type java.awt.image.BufferedImage. If there is a source, visit the AST node to load the String containing the URL of file name onto the stack. Use the cop5556fa17.ImageSupport.readImage method to read the image. If no index is given pass null for the xSize and ySize (called X and Y elsewhere) parameters, otherwise visit the index to leave the values on top of the stack. These are ints, use java.lang.Integer.valueOf to convert to Integer.

If no source is given use the makeImage method to create an image. If no size is given use values of the predefined constants def_X and def_Y.

Store the image reference in the field.

Declaration_SourceSink ::= Type name Source

Add a field to the class with the given name. If there is a Source, visit it to generate code to leave a String describing the sport on top of the stack and then write it to the field.

Declaration_Variable ::= Type name (Expression | ε)

Add field, name, as static member of class.
If there is an expression, generate code to evaluate it and store the results in the field.
See comment about this below.

Statement ::= Statement_Assign | Statement_In | Statement_Out

Statement_Assign ::= LHS Expression

REQUIRE: LHS.Type == Expression.Type

StatementAssign.isCartesian <= LHS.isCartesian

If the type is integer or boolean visit the expression to generate code to leave its value on top of the stack. Then visit the LHS to generate code to store the top of the stack in the lhs variable.

If the type is Image, generate code to loop over the pixels of the image. For each pixel, visit the expression to generate code to leave its value on top of the stack and visit LHS to generate code to store the value in the image. The range of x is [0, X) and y is [0, Y) where X and Y can be obtained from the image object using methods ImageSupport.getX and imageSupport.getY. Note that x, y, X, and Y are all predefined variables that need to be added to the class. See comments below. (Hint: write a little Java program to see how to implement the loops).

To handle polar coordinates, you still loop over x and y, but calculate r and a using RuntimeFunctions.polar_r and RuntimeFunctions.polar_a.

Statement_In ::= name Source

Generate code to get value from the source and store it in variable name.
For Assignment 5, the only source that needs to be handled is the command line.

Visit source to leave string representation of the value on top of stack
Convert to a value of correct type: If name.type == INTEGER generate code to invoke
Java.lang.Integer.parseInt. If BOOLEAN, invoke java/lang/Boolean.parseBoolean

If Image, you have already generated code to put string with image source on stack. Handle it similarly to declaration with source.

Statement_In.Declaration <= name.Declaration

~~REQUIRE: if (name.Declaration != null) & (name.type == Source.type)~~

Statement_Out ::= name Sink

For INTEGERS and BOOLEANS, the only "sink" is the screen, so generate code to print to the console here. Use java.io.PrintStream .println. This is a virtual method, you can use the static field PrintStream "out" from class java.lang.System as the object.

IF IMAGE, load the image and visit the sink.

Expression ::= Expression_Binary | Expression_BooleanLit | Expression_Conditional | Expression_FunctionApp | Expression_FunctionAppWithExprArg | Expression_FunctionAppWithIndexArg | Expression_Ident | Expression_IntLit | Expression_PixelSelector | Expression_PredefinedName _ Expression_Unary

For each expression kind, generate code to leave the value of the expression on top of the stack.

Expression_Binary ::= Expression_0 op Expression_1

Generate code to evaluate the expression and leave the value on top of the stack.

Visiting the nodes for $Expression_0$ and $Expression_1$ will generate code to leave those values on the stack. Then just generate code to perform the op.

Expression_BooleanLit ::= value

Generate code to leave the value of the literal on top of the stack

Expression_Conditional ::= $Expression_{condition}$ $Expression_{true}$ $Expression_{false}$

Generate code to evaluate the $Expression_{condition}$ and depending on its Value, to leave the value of either $Expression_{true}$ or $Expression_{false}$ on top of the stack. Hint: you will need to use labels, a conditional instruction, and goto.

Expression_FunctionApp ::= Expression_FunctionAppWithExprArg

| Expression_FunctionAppWithIndexArg

Expression_FunctionAppWithExprArg ::= function Expression

Visit the expression to generate code to leave its value on top of the stack. Then invoke the corresponding function in RuntimeFunctions. The functions that belong here are abs and log. (You do not need to implement sin, cos, or atan)

Expression_FunctionAppWithIndexArg ::= function Index

Visit the index to leave two values on top of the stack. Then invoke the corresponding function in RuntimeFunctions. The functions that belong here are cart_x, cart_y, polar_r, and polar_a. These functions convert between the cartesian (x,y) and polar (r,a) (i.e. radius and angle in degrees) representations of the location in the image.

Expression_Ident ::= name

Generate code to get the value of the variable and leave it on top of the stack.

Expression_IntLit ::= value

Generate code to leave constant on stack.

Expression_PixelSelector ::= name Index

Generate code to load the image reference on the stack. Visit the index to generate code to leave Cartesian location of index on the stack. Then invoke ImageSupport.getPixel which generates code to leave the value of the pixel on the stack.

Expression_PredefinedName ::= predefNameKind

Generate code to load value of variable onto the stack. See comments below.

Expression_Unary ::= op Expression

Generate code to evaluate the unary expression and leave its value on top of the stack. Which code is generated will depend on the operator. If the op is OP_PLUS, the value that should be left on the stack is just the value of Expression

Index ::= Expression_0 Expression_1

Visit the expressions to leave the values on top of the stack. If isCartesian, you are done. If not, generate code to convert r and a to x and y using (cart_x and cart_y). Hint: you will need to manipulate the stack a little bit to handle the two values. You may find DUP2, DUP_X2, and POP useful.

LHS ::= name Index

If LHS.Type is INTEGER or BOOLEAN, generate code to

store the value on top of the stack in variable name.

If LHS.Type is IMAGE, a pixel is on top of the stack. Generate code to store it in the image at location (x,y). (Load the image ref, load x and y, invoke ImageSupport.setPixel)

Sink ::= Sink_Ident | Sink_SCREEN

Sink_Ident ::= name

The identifier should contain a reference to a String representing a filename. Generate code to write the image to the file. The image reference should already be on the stack, so load the filename and invoke ImageSupport.write.

Sink_SCREEN ::= SCREEN

Generate code to display the image (whose ref should be on top of the stack already) on the screen. Call ImageFrame.makeFrame. Note that this method returns a reference to the frame which is not needed, so pop it off the stack.

Source ::= Source_CommandLineParam | Source_Ident | Source_StringLiteral

Source_CommandLineParam ::= $\text{Expression}_{\text{paramNum}}$

Generate code to evaluate the expression and use aaload to read the element from the command line array using the expression value as the index. The command line array is the String[] args param passed to main.

Source_Ident ::= name

This identifier refers to a String. Load it onto the stack.

Source_StringLiteral ::= fileOrURL

Load the String onto the stack

Implementing Predefined Variables

You need to implement the predefined (integer) variables x,y,X,Y,r,a,R,A, and Z in this assignment. You may add static fields to the class, like you have done for the declared variable, or you may define them

as local variables in main. The former you already know how to do, the latter is more convenient during code generation. If you make them local variable, you need to assign each one a slot number in the local variable array and call mv.visitLocalVariable before calling mv.visitMaxs. For example, if x is assigned slot 1, then invoke mv.visitLocalVariable("x", "I", null, mainStart, mainEnd, 1); . (Remember that the String[] parameter in main which contains the command line parameters is in slot 0).

x: the locations in the x direction, used as a loop index in assignment statements involving images

y: the location in the y direction, used as a loop index in assignment statements involving images

X: the upper bound on the value of loop index x. It is also the width of the image. Obtain by invoking the ImageSupport.getX method

Y: the upper bound on the value of the loop index y. It is also the height of the image. Obtain by invoking the ImageSupport.getY method.

r: the radius in the polar representation of cartesian location x and y. Obtain from x and y with RuntimeFunctions.polar_r.

a: the angle, in degrees, in the polar representation of cartesian location x and y. Obtain from x and y with RuntimeFunctions.polar_a.

R: the upper bound on r, obtain from polar_r(X,Y)

A: the upper bound on a, obtain from polar_a(0,Y)

The final three are fixed constants. You can handle this by defining and initializing a variable, or just by letting visitExpression_PredefinedName load the constant value.

DEF_X: the default image width. For simplicity, let this be 256.

DEF_Y: the default image height. For simplicity, let this be 256.

Z: the value of a white pixel. This is 0xFFFFFF or 16777215.

Logging info for grading

Since this was confusing in Assignment 5, we will do something a little simpler this time. For grading purposes, you should delete all calls to the CodeGenUtils functions with GRADE parameter. **Then, in visitStatement_Out, call CodeGenUtils.genLogTOS(GRADE, mv, type); as necessary to log the boolean or int value, or the reference to the image, before it is output and consumed.** The new versions of CodeGenUtils and RuntimeLog that will handle images.

Corrections to previous assignment.

Change the type checking of

Statement_In ::= name Source

Statement_In.Declaration <= name.Declaration
~~REQUIRE: if (name.Declaration != null) & (name.type == Source.type)~~

Requirements:

~~For grading purpose, it is essential that you insert calls to CodeGenUtils.genLog(..) and CodeGenUtils.genLogTOS(…) at locations indicated in the comments of the provided class CodeGenVisitor.~~

~~The provided class CodeGenVisitorTest contains several test cases that should be useful to you. The empty program test should pass immediately. The others should pass in a completed assignment. These are not complete, you will need additional test cases.~~

Turn in a jar file containing your source code for CodeGenVisitor.java, CodeGenVisitorTest.java, TypeCheckVisitor.java, Parser.java, Scanner.java, all of the AST classes, TypeUtils.java, RuntimeLog.java, CodeGenUtils.java, ImageSupport.java and any classes that you have added.

Your CodeGenTest will not be graded, but may be looked at in case of academic honesty issues. We will subject your parser to our set of unit tests and your grade will be determined solely by how many tests are passed.

Name your jar file in the following format: *firstname_lastname_ufid_hw6.jar*

Comments and Suggestions

- IMPORTANT: Review the lectures from 11/20 before you begin.
- **Remember that when you submit your assignment, you are attesting that have neither given nor received inappropriate help on the assignment. In this course, all assignments must be your own individual work, including the Scanner, Parser, and TypeChecker and previous versions of CodeGenVisitor after they have been graded.**
- Hint: be careful to keep the throw UnsupportedOperationException in place until a feature is implemented. This will save you a tremendous amount of time if you accidentally invoke a test program containing an unimplemented feature.