Code Generation for Assignment 5

New comments and grammar elements for this assignment are in green. Characters that appear in the generated code are in red.

Pixels are conceptually a 3-tuple with the red, green, and blue color components. Internally, they are stored in a packed form where the three values are packed into a single integer. Because each color component must fit in 8 bits, the values are in [0, 255]. When the color components are provided in the form of an ExpandedPixelExpr, invoke PixelOps.pack to pack them into an int. To extract individual color components from an int, use methods red, green, and blue in PixelOps. To update individual colors in a pixel, use PixelOps methods setRed, setGreen, and setBlue.

The image type in PLC Language will be represented using a java.awt.image.BufferedImage. An image object is instantiated when the declaration is elaborated. An image always has a size—either it obtains the size from a Dimension in the NameDef in its declaration, or its declaration has an initializer, and the size is determined from the initializer. If it does not have either a Dimension or an initializer, throw a CodeGenException. Once instantiated, the size of the image does not change.

Make sure to add import statements to your generated code for all of the edu.ufl.cise.cop4020fa23.runtime classes that are used in your generated Java code.

AST Node	Code to add to StringBuilder (shown in red) _X_ means that visiting _X_ should result in code being added to represent X.

Program::= Type IDENT NameDef* Block	public class _IDENT_ {
	public static _Type_ apply(
	NameDef*
	) _Block
	}
	
	Note: parameters from _NameDef*_ are separated by commas

Block ::= BlockElem*	{ _BlockElem*_ }
BlockElem ::= Declaration | Statement	Note: Declarations and Statements are terminated with ;
NameDef ::= Type Dimension? IDENT	_Type_ _name_

	Where _name_ is the Java name of the IDENT. (The dimension will be visited in the parent Declaration)
Type ::= image | pixel | int | string | void | boolean	BufferedImage | int | int | String | void | Boolean
Declaration::= NameDef	If NameDef.type != IMAGE _NameDef_ If NameDef.type is image, instantiate a BufferedImage object using ImageOps.makeImage.The size of the image comes from the Dimension object in the NameDef. If there is no Dimension object, this is an error: throw a CodeGenException. (Note: you may choose a different way to divide the code generation between visitDeclaration and visitNameDef) final BufferedImage NameDef.javaName = ImageOps.makeImage(_Dimension_)
Declaration::= NameDef Expr	If NameDef.type != IMAGE _NameDef_ = _Expr_ If NameDef.type is an image, there are several options for the type of the Expr. If Expr.type is string, then the value should be the URL of an image which is used to initialize the declared variable. If the NameDef has a size, then the image is resized to the given size. Otherwise, it takes the size of the loaded image. Use edu.ufl.cise.cop4020fa23.runtime.FileURLIO readImage (with or without length and width parameters as appropriate) If Expr.type is an image and the NameDef does not have a Dimension, then the image being declared gets its size from the image on the right side. Use ImageOps.cloneImage.

	If Expr.Type is an image and the NameDef does have a Dimension, then the image being declared is initialized to a resized version of the image in the Expr. Use ImageOps.copyAndResize.
Expr::= ConditionalExpr	
| BinaryExpr	
| unaryOp Expr	
| PostFixExpr	
| StringLitExpr	_StringLitExpr_.getText
| NumLitExpr	_NumLitExpr_.getText
| IdentExpr	_IdentExpr_.getNameDef().getJavaName()

ConstExpr	If ConsExpr.name = Z then 255
	else get hex String literal representing the RGB representation of the corresponding java.awt.Color.
	
	Example:
	Let the PLC Lang constant be BLUE.
	This corresponds to the java Color constant java.awt.Color.BLUE.
	
	Get the packed pixel version of the color with getRGB()
	
	Convert to a String with Integer.toHexString
	
	Prepend “0x” to make it a Java hex literal.
	
	Putting it all together, you get
	`"0x" + Integer.toHexString(Color.BLUE.getRGB())`
	
	Which is
	`0xff0000ff`

| BooleanLitExpr	true or false
ExpandedPixelExpr	
ConditionalExpr ::= $Expr_{GuardExpr}$ $Expr_{TrueExpr}$ $Expr_{FalseExpr}$	(_ $Expr_{GuardExpr}$_ ? _ $Expr_{TrueExpr}$ _ : _ $Expr_{FalseExpr}$ _)
BinaryExpr ::= $Expr_{leftExpr}$ op $Expr_{rigthExpr}$	If $Expr_{leftExpr}$.type is string and op is EQ _ $Expr_{leftExpr}$_ .equals(_ $Expr_{rigthExpr}$ _) If op is EXP ((int)Math.round(Math.pow(_ $Expr_{leftExpr}$ _ , _

	Expr$_{rigthExpr}$ _))) Otherwise (_ Expr$_{leftExpr}$ _ _op_ _ Expr$_{rigthExpr}$ _) See notes below for handling Pixel and Image types.
UnaryExpr ::= op Expr	(_op_ _Expr_) ~~Note: you do not need to handle width and height in this assignment~~ If the op is RES_width or RES_height, we know that the type is image. Invoke the getWidth or getHeight methods from BufferedImage on the image. If op == RES_height (__Expr __ .getHeight()) If op == RES_width (__Expr __ .getWidth())
PostfixExpr::= Expr PixelSelector? ChannelSelector?	If Expr.type is Pixel _ChannelSelector_ (_Expr_) Otherwise it is an image If PixelSelector != null && ChannelSelector ==null Generate code to get the value of the pixel at the indicated location. ImageOps.getRGB(_Expr_ , _PixelSelector _) If PixelSelector != null && ChannelSelector != null, generate code to get the value of the pixel at the indicated location and to invoke PixelOps.red, PixelOps.green, or PixelOps.blue. (You may want to visit the ChannelSelector, passing info that this is in the context of an expression as indicated here, or you may want to just get the value from visitPostfixExpr) _ChannelSelector_ (ImageOps.getRGB(_Expr_ , _PixelSelector_))

	If PixelSelector == null && ChannelSelector != null, generate code to invoke the ImageOPs.extractRed,extractGreen, or extractBlue method to return a new image containing the indicated color channel. ImageOps.extractRed(_Expr_) (or extractBlue or extractGreen)
ChannelSelector ::= red | green | blue	See PostfixExpr rule for how to handle this in context of an expression. See LValue for how to handle in context of an LValue
PixelSelector ::= $Expr_{xExpr}$ $Expr_{yExpr}$	__$Expr_{xExpr}$__ , __$Expr_{yExpr}$__
ExpandedPixelExpr ::= $Expr_{red}$ $Expr_{green}$ $Expr_{blue}$	PixelOps.pack(_$Expr_{red}$_, _$Expr_{green}$_, _$Expr_{blue}$_)
Dimension ::= $Expr_{width}$ $Expr_{height}$	__$Expr_{width}$__ , __$Expr_{height}$ __
LValue ::= IDENT PixelSelector? ChannelSelector?	_Ident_.getNameDef().getJavaName() (PixelSelector and ChannelSelector if present, must be visited. It may be easier to invoke this methods from the parent AssignmentStatement.)
Statement::=	
AssignmentStatement |	
WriteStatement |	
DoStatement |	
IfStatement |	
ReturnStatement |	
StatementBlock	
AssignmentStatement ::= LValue Expr	If LValue.varType == image. See explanation below. If LValue.varType == pixel and LValue.ChannelSelector != null PixelOps.setRed(_LValue_ , _Expr_) or setGreen or SetBlue Otherwise _LValue_ = _Expr_

WriteStatement ::= Expr	ConsoleIO.write(_Expr_) The ConsoleIO class includes an overloaded method write for each Java type that represents a PLC Language type. Thus, you can simply generate code to call the write method and let the Java compiler determine which overloaded version to use. The exception is that int and pixel in PLC Language are both represented by a Java int. When the type of Expr is pixel, you need to use the writePixel method.
DoStatement ::= GuardedBlock$^+$	See below
IfStatement ::= GuardedBlock$^+$	See below
GuardedBlock := Expr Block	Handling depends on the context. See description of DoStatement and IfStatement below.
ReturnStatement ::= Expr	return _Expr_
StatementBlock ::= Block	_Block_

Binary Expressions with Images and Pixels

Binary operations on these types are generally carried out component-wise.

For example, for images im0 and im1, the expression (e0+e1) yields a new image with the individual pixels added together componentwise. Pixels in turn are also added componentwise. For pixels p0, and p1, the expression (p0 + p1) yields a pixel with individual color channel values added together. Any time the value of a color channel goes out of the range [0,255], it is truncated to the maximum or minimum value.

When a binary operation has one operand with a “larger” type than the other, the value of the smaller type is replicated. So for example, im0/2 would divide each pixel in im0 by 2. For each pixel, each individual color channel value is divided by 2.

Several routines in edu.ufl.cise.cop4020fa23.ImageOps have been provided to implement binary expressions with images and pixels. See binaryImageIMageOp (combine two images), binaryImagePixelOp (combine an image with a pixel), binaryImageIntOp (combine an image with an int), binaryPackedPixelPixelOP (combine two pixels), binaryPackedPixelIntOp (combine a pixel with an int) and binaryPackedPixelBooleanOP (compare two pixels Boolean operator).

Semantics of IfStatement

You will need to figure out the details yourself.

The semantics are similar to Dijkstra's guarded command if statement except our version is not non-deterministic (i.e. is deterministic). The guarded blocks are evaluated starting from the top. One other difference is that Dijkstra's version requires that at least one guard be true. In our version, if none of the guards are true, nothing will happen.

In other words, if the guards are G0, G1, .. Gn, and the corresponding blocks are B0, B1,..,Bn, Guards are evaluated in turn, starting with G0. When a guard, say Gi, is true, execute the corresponding Block Bi. That is the end of the if statement. The Java code would look something like "if (G0) {B0;} else if (G1) {B1;}… else if (Gn) {Bn;}"

Semantics of DoStatement

You will need to figure out the details yourself.

The semantics are like Dijkstra's guarded command do-od statement except our version is not non-deterministic (i.e. is deterministic). In each iteration, the guarded blocks are evaluated starting from the top. (Note that this semantic choice was made for ease of implementation in a class project. There are alternatives that would probably be more useful in practice.) The loop terminates when none of the guards are true.

In other words, if the guards are G0, G1, .. Gn, and the corresponding blocks are B0, B1,..,Bn, Guards are evaluated in turn, starting with G0. When a guard, say Gi is true, execute the corresponding Block Bi. That is an iteration. Repeat, starting at the top with G0 again, for each iteration. The statement terminates when none of the guards are true.

Assignment statements where LValue.VarType == IMAGE

There are several cases:

- PixelSelector == null and ChannelSelector == null (i.e. something like im0 = expr;)
    - If Expr.type = IMAGE, use ImageOps.copyInto to copy the Expr image into the LValue image
    - If Expr.type = PIXEL, use ImageOps.setAllPixels to update each pixel of the LValue image with the given pixel value

   - If Expr.type = STRING, load the image from the URL in the string, and resizing to the size of the LValue image. Then copy the pixels of the loaded image into the LValue image. Use FileURLIO.readImage and ImageOps.copyInto

- ChannelSelector != null (i.e im0:red = expr or im0[x,y]:red = expr)

  To simplify the project a little, you may skip this case.
  Throw an UnsupportedOperationException instead

- PixelSelector != null and ChannelSelector == null (i.e. something like im[x1,x2] =expr)

This is the most interesting part of the language. Recall that we could write statements like im0[x,y] = expr where x and/or y are not visible in the current scope. In this case, we added a SyntheticNameDef object for the variable. For each variable whose nameDef is actually a SyntheticNameDef object, we generate code for an implicit loop over the values of that variable.

For example, suppose we have a statement

Im0[x,y] = im1[y,x]

where x and y are not previously declared and thus their NameDef objects in the AST are SyntheticNameDef. Generate code to loop over x and y from 0 to im0.getWidth() and from 0 to im0.getHeight(), respectively and update each pixel with the value of the expression on the right side.

See the generated code in the starter Junit test cases for example.

Notes on Junit Test cases

Many of the test cases have an example of generated source code. (Sorry about the formatting). It is OK if your generated code does not exactly match mine as long as the semantics are correctly implemented.

The show(BufferedImage) routine will request and wait for console input if WAIT_FOR_INPUT is true. This prevents the IDE from immediately closing the image frame before you have a chance to see it. Change the value of WAIT_FOR_INPUT to false if you just want to run all the test cases as quickly as possible.