Code Generation for Assignment 6

AST Node Type	AST rule	Constraints and Actions
Prog	Program ::= IDENT Dec* Stmt*	Set up main method Assign slots for local variables to represent predefined ints x and y Visit children Finalize main method Visit local vars x and y Return bytecode
Dec	Dec ::= Type IDENT	Create a static or local variable (your choice) for this variable. If it is an image, instantiate a PLPImage object.
AlternativeStmt	Stmt ::= Expr $(Stmt^*)_{ifStmtList}$ $(Stmt^*)_{elseStmtList}$	Visit Expr to generate code to leave its value on top of the stack. IFEQ elseLabel visit statements in ifStmtmList GOTO endOfAlternativeLabel elseLabel: visit statements in elseStmtList endOfAlternativeLabel:
AssignExprStmt	Stmt ::= IDENT Expr	Visit the expression to generate code to leave its value on top of the stack. Store it in the variable on the left hand side.
AssignPixelStmt	Stmt ::= IDENT Pixel	*Change your TypeCheckVisitor to allow either an image or a pixel on the left hand side.* If the lhs is a pixel • visit pixel to generate code to leave pixel value on top of the stack • store in pixel indicated by lhs else, the lhs is an image • there is an implicit loop over x and y. *See below.*
FileAssignStmt	Stmt ::= IDENT FileName	Acquire a BufferedImage indicated by the filename or URL using the PLPImage.loadImage method and store it into the PLPImage object
ScreenLocationAssignmentStmt	Stmt ::= IDENT $Expr_{xScreenExpr}$ $Expr_{yScreenExpr}$	Visit the expressions. Store the values in the appropriate fields of the image indicated by the IDENT. Invoke the image's updateFrame method.

SetVisibleAssignmentStmt	Stmt ::= IDENT Expr	Visit the expression. Store the value in the isVisible field of the image indicated by the IDENT. Invoke the image's updateFrame method.
ShapeAssignment Stmt	Stmt ::= IDENT $Expr_{width}$ $Expr_{height}$	Visit the expressions. Store the values in the width and height fields of the image indicated by the IDENT. Invoke the image's updateImageSize method. Invoke the image's updateFrame method.
SinglePixelAssignmentStatement	Stmt ::= IDENT $Expr_{xExpr}$ $Expr_{yExpr}$ Pixel	Visit the expressions to generate code to evaluate expressions indicating a location in the image indicated by the IDENT. Visit the Pixel to generate code to pack it into an integer. Update the pixel in the image and invoke the image's updateFrame method.
SingleSampleAssignmentStmt	Stmt ::= IDENT $Expr_{xExpr}$ $Expr_{yExpr}$ COLOR $Expr_{rhsExpr}$	Visit the expressions to generate code to leave their values on top of the stack. Use the PLPImage.setSample method to update the given sample of the indicated image. Invoke the image's updateFrame method (left out of assignment 5 by mistake)
IterationStatement	Stmt ::= Expr Stmt*	GOTO guardLabel bodyLabel: visit statements in stmtList guardLabel: visit expr to generate code to leave its value on top of the stack IFNE bodyLabel
PauseStatement	Stmt ::= Expr	Visit the expression to generate code to leave its value on top of the stack. Invoke the PLPImage.pause method
BinaryExpr	Expr ::= $Expr_{e0}$ Op $Expr_{e1}$	Visit the expressions to generate code to leave their values on top of the stack. Evaluate the binary expression and leave its value on top of the stack. Implement the following operators on ints: +,-,*,/,%, ≪, ≫. Implement remaining operators. See below
BooleanLitExpr	Expr ::= BooleanLit	Generate code to leave the value (0 or 1) of the Boolean literal on top of the stack.

ConditionalExpr	Expr ::= $Expr_{condition}$ $Expr_{trueValue}$ $Expr_{falseValue}$	Visit condition to generate code to leave value of condition on top of stack. IFEQ falseConditionLabel visit trueValue to generate code to leave the value on top of the stack GOTO endOfExprLabel falseConditionLabel: visit falseValue to generate code to leave the value on top of the stack endOfExprLabel:
IdentExpr	Expr ::= IDENT	Generate code to load the value of the given variable on top of the stack.
ImageAttributeExpr	Expr ::= IDENT SELECTOR	Generate code to load the appropriate attribute of the image indicated by the ident on top of the stack. You may use getter/setter methods of the PLPImage class or just access the fields directly.
IntLitExpr	Expr ::= INT_LIT	Generate code to leave the int value of the literal on top of the stack.
PreDefExpr	Expr ::= CONSTANT_LIT	Generate code to load the value on top of the stack. Z is defined in ImageConstants SCREEN_SIZE is defined in PLPImage.[1] x and y are local variables.
SampleExpr	Expr ::= IDENT $Expr_{xLoc}$ $Expr_{yLoc}$ COLOR	Visit the expressions to generate code to leave their values on top of the stack. Use the getSample method to return the value of the sample and leave it on top of the stack.
Pixel	Pixel ::= $Expr_{redExpr}$ $Expr_{greenExpr}$ $Expr_{blueExpr}$	Visit the expressions and invoke the Pixel.makePixel method to pack it into an int.

[1] It would have made more sense to make this a pair instead of a single value. It gets the size, in pixels of the smallest of the width and height of the available screen (not including task bars, etc.)

Using labels

1. Declare a label before you refer to it: Label label0 = new Label();
2. You can now refer to the label in an instruction: mv.visitJumpInsn(GOTO, label0);
3. Visit the label at the place you want it to be: mv.visitLabel(label0);

AssignPixelStatement

This statement is perhaps the most interesting one in our language. It allows us to define an interesting image algebraically with a single line of code.

For example, here is a program that takes a given image cfl and creates and displays a new image ud that is cfl upside down.

```
upside_down {
image cfl;
image ud;
int Y;
int X;

cfl = "http://www.cise.ufl.edu/news/NA00166/image.png";

ud.shape  = [cfl.width, cfl.height];  //set the shape of ud to be the same as cfl
Y = cfl.height-1;
X = cfl.width-1;

ud = {{  cfl[X-x, Y-y]red, cfl[X-x, Y-y]green, cfl[X-x, Y-y]blue }};
ud.visible = true;

}
```

The highlighted statement is equivalent to the following:

```
for (x = 0; x < ud.width; ++x){
   for (y = 0; x < ud.height; ++y){
      ud[x,y] = {{ cfl[X-x, Y-y]red, cfl[X-x, Y-y]green, cfl[X-x, Y-y]blue }};
   }
}
update the frame
```

where the assignment ud[x,y] = … is implemented by loading x and y onto the stack, visiting the pixel on the RHS, then invoking setPixel.

It is your job to figure out the necessary branching logic to implement this.

Boolean Valued Binary Expressions

In assignment 5, you implemented integer valued binary expressions by loading the two arguments on top of the stack and applying an operator, for example iadd for +. The JVM does not have similar instructions for operators with Boolean values. Instead it has branching instructions. So to get the value

of, say, the expression a>b, which will be 0 or 1 depending on what the values of a and b are, the Java compiler generates code such as the following

```
load a onto stack
load b onto stack
IF_ICMGT    gtLabel
ldc 0
GOTO endOfExprLabel
gtLabel:
ldc 1
endOfExprLabel:
```

You may do this, or if you prefer, you may add a new class written in Java with methods to perform these operations, for example

```
public static boolean gt(int a, int b) { return a>b;}
```