Duccio Rocca
ID = 922254031
March 29th, 2022

Assignment 3 Documentation

Github Repository

Assignment three repository.
or
https://github.com/sfsu-csc-413-fall-2022-roberts/assignment-3---parser-RINO-GAELICO

Project Introduction and Overview

This project required me to complete the Compiler program for a made-up language ‘x’, building on the modification to Lexer that we made in the previous assignment to recognize new tokens. It required me to modify the Parser introducing new production rules based on the new added token and a few other rules, including the production rule for a switch statement.

As a second step, it required me to create two new Visitor classes. An OffsetVisitor to calculate the positioning of each node so that the final result would be a well-balanced (not in a technical sense) tree. A second Visitor would draw the tree using the positions calculated by the OffsetVisitor.

Execution and Development Environment

As IDE I used Visual Studio Code on my MacBook Pro and I tested in both IDE and terminal through the command line.

JDK is Openjdk, version "17.0.2" 2022-01-18. The OpenJDK Runtime Environment is build 17.0.2+8-86, and OpenJDK 64-Bit Server VM is build 17.0.2+8-86 (mixed mode, sharing).

Assumptions

I assumed that the Lexer that I designed in the previous assignment was working properly and that the compiler was already equipped to deal with any syntax error it might encounter while parsing each token.

I also assumed that the TokenSetup class was correctly implemented to generate the two classes TokenType.java and Symbol.java.

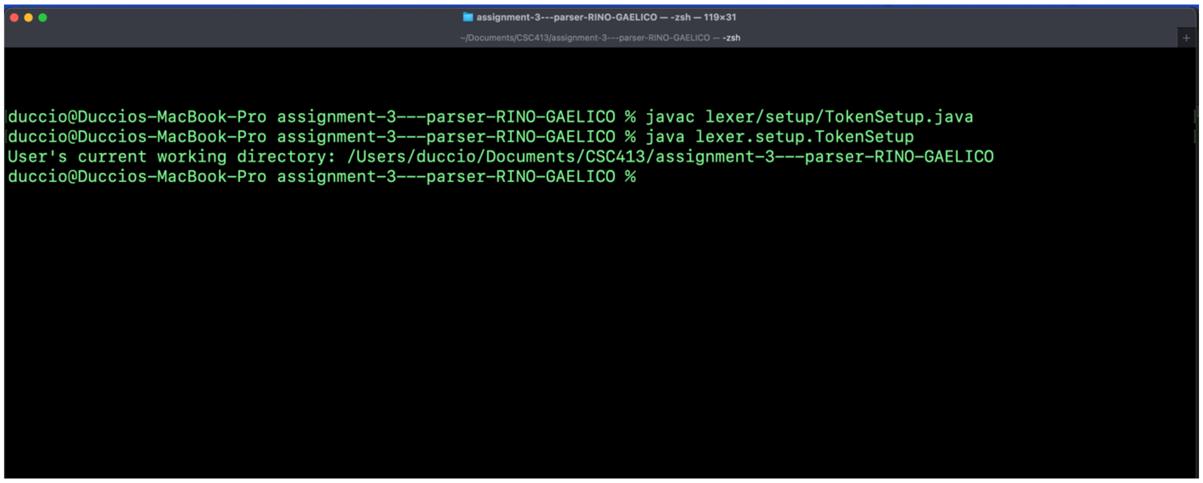
Scope of Work

Task		Completed
Modify the Parser, introducing new production rules:		X
	TYPE → ‘utf16string’	X
	TYPE → ‘timestamp’	X
	F → <utf16string>	X
	F → <timestamp>	X
	E → SE ‘>=’ SE	X
	E → SE ‘>’ SE	X
	S → ‘if’ E ‘then’ BLOCK	X
	S → ‘switch’ ‘(‘ <id> ‘)’ CASEBLOCK	X
	CASEBLOCK → ‘{‘ CASESTATEMENT+ DEFAULTSTATEMENT? ‘}’	X
	CASESTATEMENT → ‘case’ CASELIST ‘#’ S	X
	CASELIST → ‘[‘ (E list ‘,’)? ‘]’	X
	DEFAULTSTATEMENT → ‘default’ ‘#’ S	X
Copy Lexer from Assignment 2 in this directory		X
Add additional tokens		X
	Add ‘switch’ token	X
	Add ‘default’ token	X
	Add ‘case’ token	X
	Add ‘#’ token	X
Remove all debug statements		X
Create OffsetVisitor		X
	Data structure: HashMap	X
	Recursive method to calculate position of each Node	X
	Recursive method to adjust position of children if necessary	X
Create DrawOffsetVisitor		X
	Recursive method to render the tree	X

Compilation Result

First, I recompiled and reran the TokenSetup class so that a new table with the newly added token could be created:

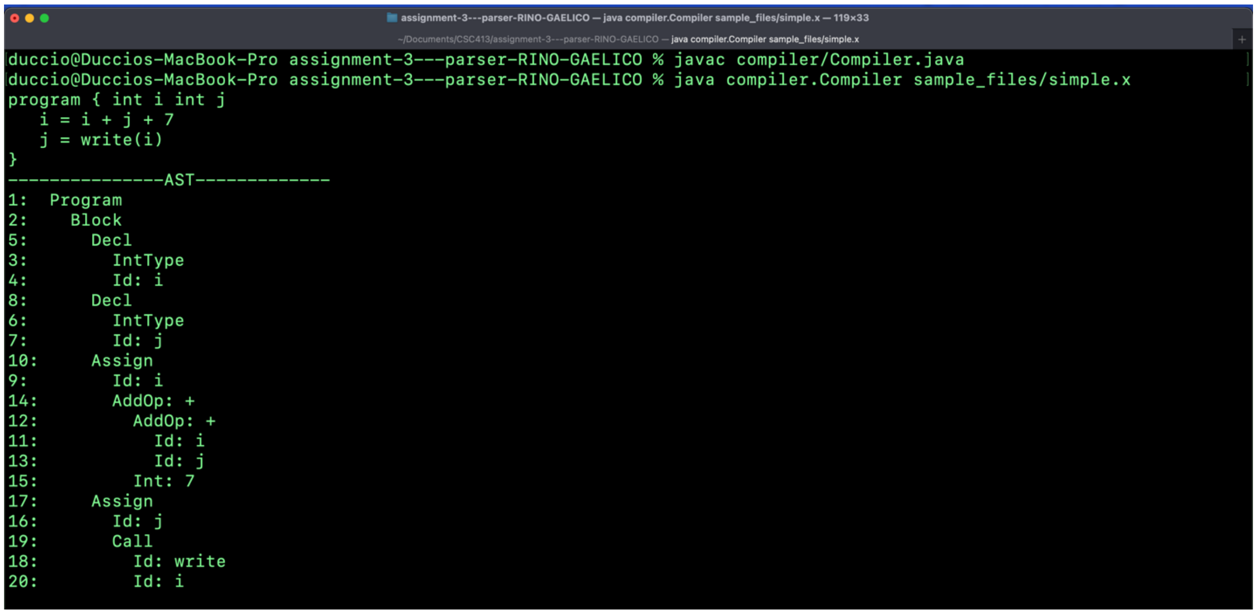
```
javac lexer/setup/TokenSetup.java
java lexer.setup.TokenSetup
```



```
duccio@Duccios-MacBook-Pro assignment-3---parser-RINO-GAELICO % javac lexer/setup/TokenSetup.java
duccio@Duccios-MacBook-Pro assignment-3---parser-RINO-GAELICO % java lexer.setup.TokenSetup
User's current working directory: /Users/duccio/Documents/CSC413/assignment-3---parser-RINO-GAELICO
duccio@Duccios-MacBook-Pro assignment-3---parser-RINO-GAELICO %
```

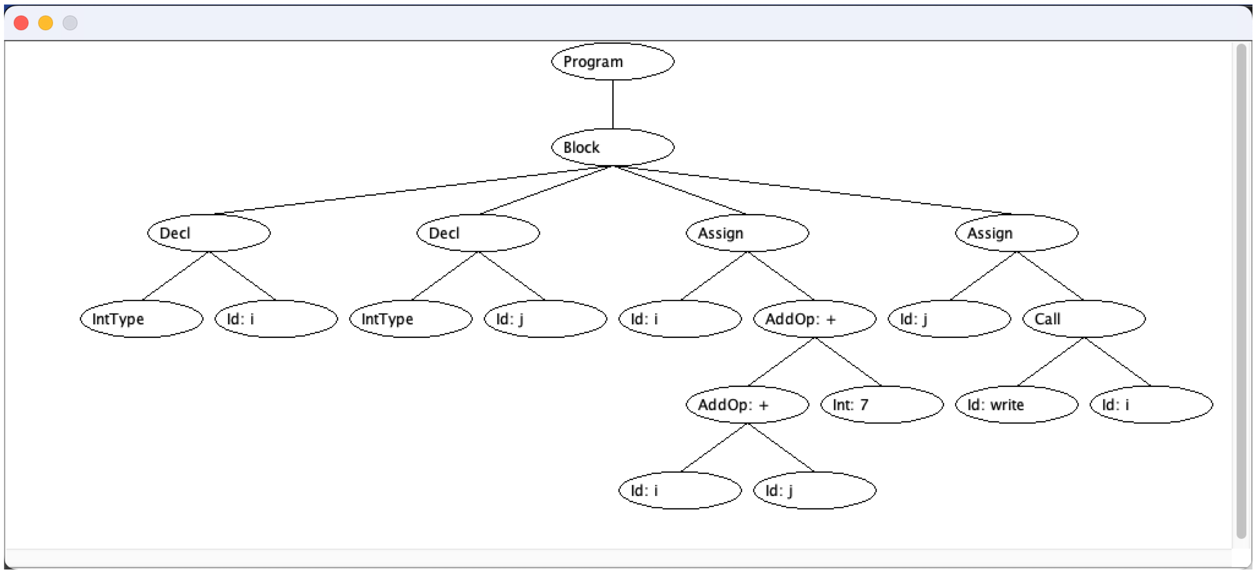
Then, I compiled and ran the Compiler Class using simple.x file as a test. No error messages or warnings were displayed, and the application ran as expected.

```
javac compiler/Compiler.java
java compiler.Compiler sample_files/simple.x
```



```
duccio@Duccios-MacBook-Pro assignment-3---parser-RINO-GAELICO % javac compiler/Compiler.java
duccio@Duccios-MacBook-Pro assignment-3---parser-RINO-GAELICO % java compiler.Compiler sample_files/simple.x
program { int i int j
   i = i + j + 7
   j = write(i)
}
---------------AST-------------
1:  Program
2:    Block
5:      Decl
3:        IntType
4:        Id: i
8:      Decl
6:        IntType
7:        Id: j
10:     Assign
9:        Id: i
14:       AddOp: +
12:         AddOp: +
11:           Id: i
13:           Id: j
15:         Int: 7
17:     Assign
16:       Id: j
19:       Call
18:         Id: write
20:         Id: i
```

Here is the result from DrawOffsetvisitor:



Implementation

Production Rules: Parser class and AST

The first requirement for this assignment was to introduce new production rules. Production rules replace nonterminal symbols (on the left-hand side of the production rule) with other nonterminal or terminal symbols (on the right-hand side of the production).
We needed new production rules to integrate the tokens added with the previous assignment and, furthermore, we had to introduce new production rules for a switch statement and modify the existing if-statement.
All production rules are implemented within the Parser class.
Obviously, for each new token, I had to create the corresponding AST class, so that the Parser could generate the corresponding Tree node while parsing.

rType, rFactor, rExpr, and rStatement

The first category of production rules was the one that allow the production of a terminal ‘utf16string’ type and a ‘timestamp’ type. To do that I had to modify the rType method within the Parser class. This method simply checks if the next token to be parsed is a token of the types accepted for the language ‘x’. Before the modification, the only two types were int and Boolean. So to create the required production rules I had simply to add the two aforementioned types within this method. It is important to mention that this method returns an AST tree of the type corresponding to the token’s type, in our case TimeStampTypeTree UstringTypeTree.

As a second category of production rules, I had to implement the corresponding terminal literals for the recently added tokens ‘utf16string’ and ‘timestamp’. To do that I had to modify the rFactor method that simply checks if the next token to parse is one of the types available for factors, in our case I had to add to the possible factors the Ustring literal and the timestamp literal. The returning value for this method is an AST object. In our case, the two possible returning AST were UtfStringLitTree and TimeStampLitTree.

A very similar step required me to update the parser to be able to parse the new tokens ‘>=’ and ‘>’. The corresponding AST tree, which in our case was the RelOpTree, was generated by the rExpr method. This method used a helper called getRelationTree, which was checking if the symbol encountered was contained in an EnumSet. I had simply to update the EnumSet with the two recently added tokens: ‘>=’ and ‘>’.

One more requirement was to modify the existing production rule for the if-statement so that it could also parse if-statement without an ELSE. To do that I had to modify the rStatement method. The AST generated by this method, in case no Else was found, was an IfTree with two more kids added to it: an Expression tree and a Block tree. To implement this production rule I had to simply add the possibility that no else token was found and consequently an ifTree was returned with the two children added to it.

Switch Statement

This set of production rules was the most articulated one and it required several modifications to the Parser class and the creation of new tokens within the list of valid tokens.

First, I had to modify the rStatement method so that in case a ‘switch’ token was found it could create a SwitchTree with two children, one for idTree and one for CaseBlock Tree.
For the case block, I had to implement a new method, rCaseBlock. This method returned a CaseBlockTree with two children, one CaseStatement tree and one DefaultStatement tree. Each of these trees was returned by the corresponding methods (rCaseStatement and rDefaultAStatement). The rDefaultAStatement was pretty simple since it returned a DefaultStatementTree with a StatementTree as a child, it also checked that before the statement a hash token was found.

The rCaseStatement was a little more articulated. This method returns a CaseStatementTree, but before returning the AST tree it calls the addKid method twice passing rCaseList and rStatement as parameters. rCaseList is a method that I had to add and implement. It returns a CaseListTree with zero to an infinite number of children. In fact, on the right-hand side of the production rule for Caselist, there is a list of expressions. Obviously, for each expression, it calls the rExpr method and checks that the next token is a comma. It does this in a loop until the next token is something different from a comma, at that point a right parenthesis is expected.

This set of production rules was the more articulated but it was easy because the other production rules that were already present in the parser class offered a good source of inspiration for the implementation of the switch statement.

OffsetVisitor

Data Structures: HashMap and array

One of the requirements was to implement a new visitor called OffsetVisitor.
This visitor would traverse the tree generated by the parser and calculate the position of each node so that the final result would be a tree well balanced and each child would be aligned with the parent node, which would be centered on top of the children.
One of the most delicate parts of this implementation was to figure out how to retain the information relative to the position of each node. I decide to use a hashMap that would use the node number as a key and the correspondent value would be an ArrayList of Integers. In the array, the program would store two entries: the depth where the node was found and the offset. This allowed me to use this information in the drawing visitor as the x and the y coordinates for each node. Since the node number is unique for each node I could traverse the tree again and match the node number with the corresponding x and y values stored in the ArrayList.
For the information relative to the next available offset for each depth level, I decided to use a simple array of 100 entries (the maximum depth indicated in the instructions). To each entry corresponded a value that represented the next available offset for that specific depth level.

Recursive method Offset

The method that I implemented emulated the design of the other visitors already present in the package. Each visitor class has a method for each type of tree AST and within this method, there is a method that can be called recursively when visiting another node. In my case, I called the recursive method *offset*. This method checks if the visited node has more children, if there are children then the method calls the visitKids method which will call for each child the accept method and pass the visitor as a parameter. This will make the visitor call the corresponding method appropriate for the type of tree node visited. Once this series of calls is executed, the control is back to the offset method that would again operate the check of the existence of more children to visit. If the node visited doesn't have any children then the method would go ahead and calculate the correct offset position, store the values into the hashmap and return. Once the recursive call would start to reclimb the tree towards higher levels of depth, this same method would calculate the offset position of the parent node.
As we can see, up to this point, this is a typical post-order traversal. The tricky part comes when the offset calculated for the parent node is smaller than the next available value. In this case, an adjustment is needed and I decide, following the single-responsibility principle, to assign this task to a different method.

Recursive method adjustChildrenOffset

I called the method that operates the adjustment of the children's offset value adjustChildrenOffset. I pass into this method two parameters: the node visited, and the value of the shift calculated as per instructions based on the position of the parent node.
Also, this method is a recursive method and every time is called checks if the visited node has children and it creates an array of children nodes. Each node is visited and the same method is called passing the shift value and the node visited. When no more children are found it starts to

calculate the shift adding it to the value already assigned to the node. It also updates the corresponding entry in the array that keeps track of the next available offset.
This method is relatively simple and the use of the hashMap has helped me to access easily the corresponding values to be updated.

DrawOffsetVisitor

The last piece of the program that I had to implement was the DrawOffsetVisitor. This part of the project seemed pretty smooth. In part, because a DrawVisitor was already present and it served as a template for the DrawOffset Visitor, but also because the data structure that I chose helped me to have the right information available with little effort for each visited node.
For this class, I used a traverse technique similar to the one utilized in the OffsetVisitor, but with a simpler logic.
This visitor has the same structure as other visitors. It has a method for each type of tree and each of those methods calls a recursive method called *draw*.
Every node visited, using the recursive method, is drawn by the visitor accessing the corresponding values stored in the hashMap. It also checks if the node has children and in case there are children, it draws the corresponding lines that end at the midpoint of the invisible rectangle around the children's oval.
To render the tree in a more presentable way and to make sure the space was used wisely and the proportions were respected, I declared some constants at the beginning of the class. I declared the width of each node, its height, the gap between each node, some more padding to add between each node.
Furthermore, I calculated the size of the image using the information available through the hashMap. To retrieve this information I used a functional approach taking advantage of the Stream API introduced with JAVA 8. Once I obtained the max depth and max offset (which also gave me an indication of the max number of nodes in a row), I multiplied these values for the vertical step (node height + vertical gap) and horizontal step (node width + horizontal gap) respectively, plus some values to create some margin on the sides.
For the part that deals with the Graphics2D object, I followed the same logic contained in the DrawVisitor class.

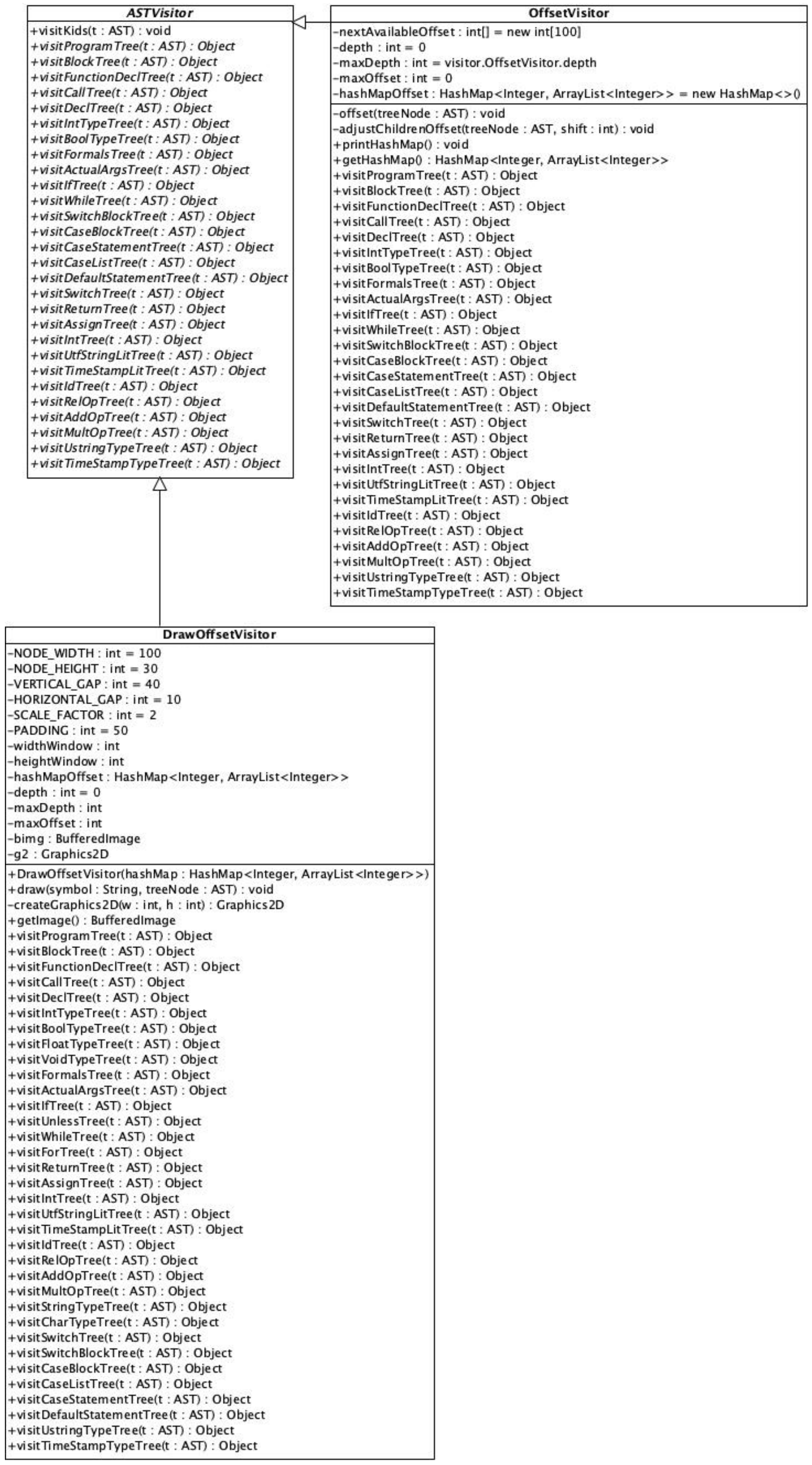
Code Organization

The classes were already neatly organized into packages, I only added the new classes into the correspondent packages and create a new folder for documentation.

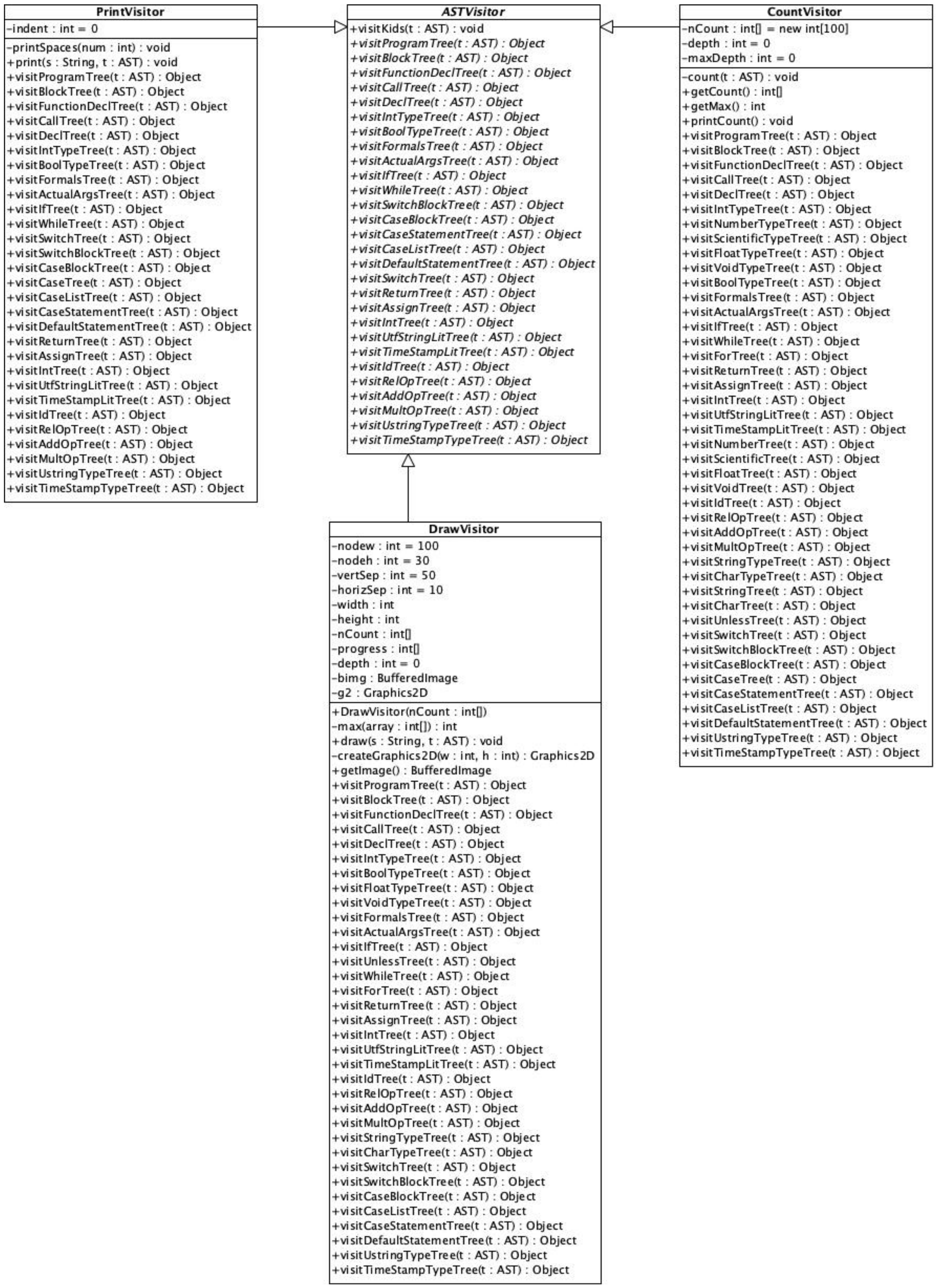
Class Diagram

The class' diagrams shown below represent all the classes that I modified in this project.

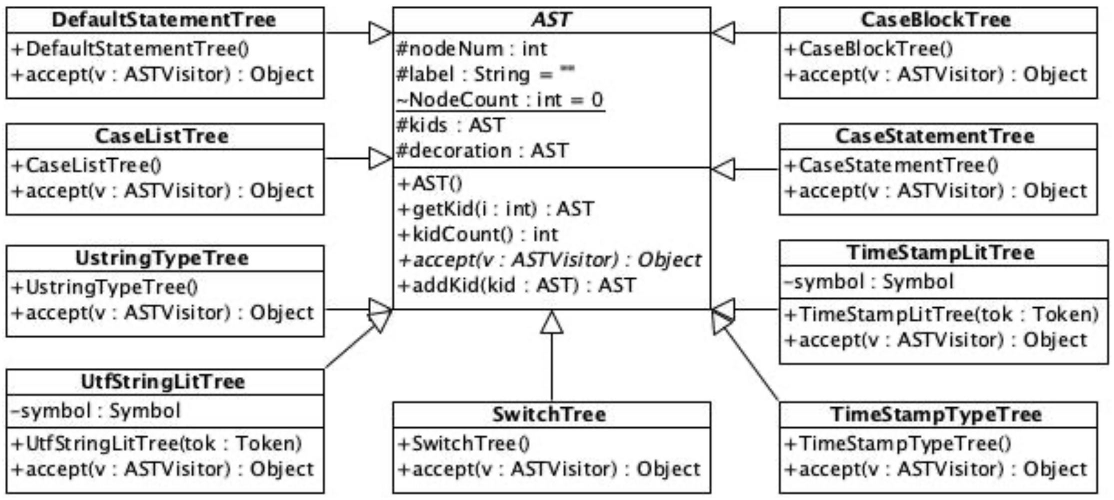
Here is the diagram for the two visitor classes I implemented:


ASTVisitor
+visitKids(t : AST) : void
+visitProgramTree(t : AST) : Object
+visitBlockTree(t : AST) : Object
+visitFunctionDeclTree(t : AST) : Object
+visitCallTree(t : AST) : Object
+visitDeclTree(t : AST) : Object
+visitIntTypeTree(t : AST) : Object
+visitBoolTypeTree(t : AST) : Object
+visitFormalsTree(t : AST) : Object
+visitActualArgsTree(t : AST) : Object
+visitIfTree(t : AST) : Object
+visitWhileTree(t : AST) : Object
+visitSwitchBlockTree(t : AST) : Object
+visitCaseBlockTree(t : AST) : Object
+visitCaseStatementTree(t : AST) : Object
+visitCaseListTree(t : AST) : Object
+visitDefaultStatementTree(t : AST) : Object
+visitSwitchTree(t : AST) : Object
+visitReturnTree(t : AST) : Object
+visitAssignTree(t : AST) : Object
+visitIntTree(t : AST) : Object
+visitUtfStringLitTree(t : AST) : Object
+visitTimeStampLitTree(t : AST) : Object
+visitIdTree(t : AST) : Object
+visitRelOpTree(t : AST) : Object
+visitAddOpTree(t : AST) : Object
+visitMultOpTree(t : AST) : Object
+visitUstringTypeTree(t : AST) : Object
+visitTimeStampTypeTree(t : AST) : Object
OffsetVisitor
-nextAvailableOffset : int[] = new int[100]
-depth : int = 0
-maxDepth : int = visitor.OffsetVisitor.depth
-maxOffset : int = 0
-hashMapOffset : HashMap<Integer, ArrayList<Integer>> = new HashMap<>()
-offset(treeNode : AST) : void
-adjustChildrenOffset(treeNode : AST, shift : int) : void
+printHashMap() : void
+getHashMap() : HashMap<Integer, ArrayList<Integer>>
+visitProgramTree(t : AST) : Object
+visitBlockTree(t : AST) : Object
+visitFunctionDeclTree(t : AST) : Object
+visitCallTree(t : AST) : Object
+visitDeclTree(t : AST) : Object
+visitIntTypeTree(t : AST) : Object
+visitBoolTypeTree(t : AST) : Object
+visitFormalsTree(t : AST) : Object
+visitActualArgsTree(t : AST) : Object
+visitIfTree(t : AST) : Object
+visitWhileTree(t : AST) : Object
+visitSwitchBlockTree(t : AST) : Object
+visitCaseBlockTree(t : AST) : Object
+visitCaseStatementTree(t : AST) : Object
+visitCaseListTree(t : AST) : Object
+visitDefaultStatementTree(t : AST) : Object
+visitSwitchTree(t : AST) : Object
+visitReturnTree(t : AST) : Object
+visitAssignTree(t : AST) : Object
+visitIntTree(t : AST) : Object
+visitUtfStringLitTree(t : AST) : Object
+visitTimeStampLitTree(t : AST) : Object
+visitIdTree(t : AST) : Object
+visitRelOpTree(t : AST) : Object
+visitAddOpTree(t : AST) : Object
+visitMultOpTree(t : AST) : Object
+visitUstringTypeTree(t : AST) : Object
+visitTimeStampTypeTree(t : AST) : Object
DrawOffsetVisitor
-NODE_WIDTH : int = 100
-NODE_HEIGHT : int = 30
-VERTICAL_GAP : int = 40
-HORIZONTAL_GAP : int = 10
-SCALE_FACTOR : int = 2
-PADDING : int = 50
-widthWindow : int
-heightWindow : int
-hashMapOffset : HashMap<Integer, ArrayList<Integer>>
-depth : int = 0
-maxDepth : int
-maxOffset : int
-bimg : BufferedImage
-g2 : Graphics2D
+DrawOffsetVisitor(hashMap : HashMap<Integer, ArrayList<Integer>>)
+draw(symbol : String, treeNode : AST) : void
-createGraphics2D(w : int, h : int) : Graphics2D
+getImage() : BufferedImage
+visitProgramTree(t : AST) : Object
+visitBlockTree(t : AST) : Object
+visitFunctionDeclTree(t : AST) : Object
+visitCallTree(t : AST) : Object
+visitDeclTree(t : AST) : Object
+visitIntTypeTree(t : AST) : Object
+visitBoolTypeTree(t : AST) : Object
+visitFloatTypeTree(t : AST) : Object
+visitVoidTypeTree(t : AST) : Object
+visitFormalsTree(t : AST) : Object
+visitActualArgsTree(t : AST) : Object
+visitIfTree(t : AST) : Object
+visitUnlessTree(t : AST) : Object
+visitWhileTree(t : AST) : Object
+visitForTree(t : AST) : Object
+visitReturnTree(t : AST) : Object
+visitAssignTree(t : AST) : Object
+visitIntTree(t : AST) : Object
+visitUtfStringLitTree(t : AST) : Object
+visitTimeStampLitTree(t : AST) : Object
+visitIdTree(t : AST) : Object
+visitRelOpTree(t : AST) : Object
+visitAddOpTree(t : AST) : Object
+visitMultOpTree(t : AST) : Object
+visitStringTypeTree(t : AST) : Object
+visitCharTypeTree(t : AST) : Object
+visitSwitchTree(t : AST) : Object
+visitSwitchBlockTree(t : AST) : Object
+visitCaseBlockTree(t : AST) : Object
+visitCaseListTree(t : AST) : Object
+visitCaseStatementTree(t : AST) : Object
+visitDefaultStatementTree(t : AST) : Object
+visitUstringTypeTree(t : AST) : Object
+visitTimeStampTypeTree(t : AST) : Object

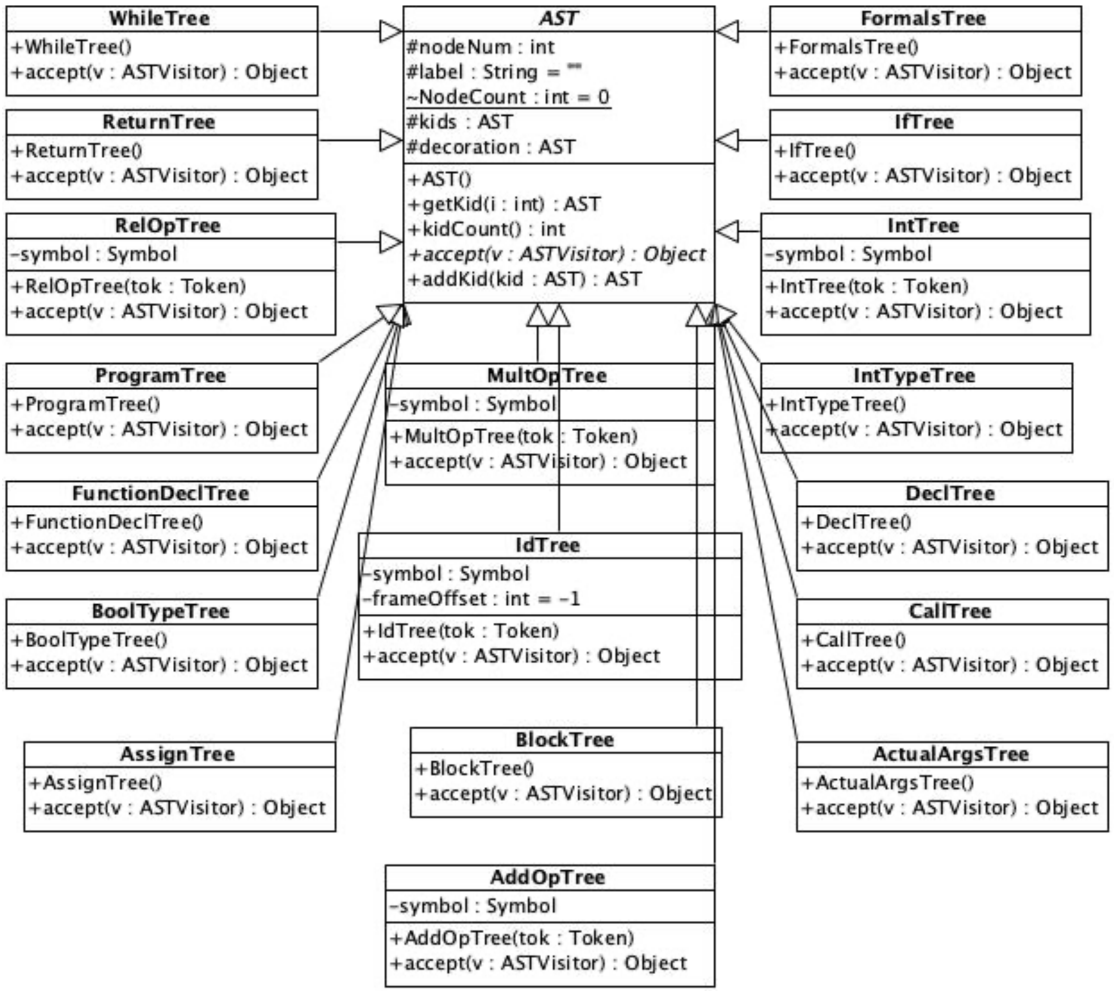
Here is a diagram for the other visitor classes:


PrintVisitor
-indent : int = 0
-printSpaces(num : int) : void
+print(s : String, t : AST) : void
+visitProgramTree(t : AST) : Object
+visitBlockTree(t : AST) : Object
+visitFunctionDeclTree(t : AST) : Object
+visitCallTree(t : AST) : Object
+visitDeclTree(t : AST) : Object
+visitIntTypeTree(t : AST) : Object
+visitBoolTypeTree(t : AST) : Object
+visitFormalsTree(t : AST) : Object
+visitActualArgsTree(t : AST) : Object
+visitIfTree(t : AST) : Object
+visitWhileTree(t : AST) : Object
+visitSwitchTree(t : AST) : Object
+visitSwitchBlockTree(t : AST) : Object
+visitCaseBlockTree(t : AST) : Object
+visitCaseTree(t : AST) : Object
+visitCaseListTree(t : AST) : Object
+visitCaseStatementTree(t : AST) : Object
+visitDefaultStatementTree(t : AST) : Object
+visitReturnTree(t : AST) : Object
+visitAssignTree(t : AST) : Object
+visitIntTree(t : AST) : Object
+visitUtfStringLitTree(t : AST) : Object
+visitTimeStampLitTree(t : AST) : Object
+visitIdTree(t : AST) : Object
+visitRelOpTree(t : AST) : Object
+visitAddOpTree(t : AST) : Object
+visitMultOpTree(t : AST) : Object
+visitUstringTypeTree(t : AST) : Object
+visitTimeStampTypeTree(t : AST) : Object
ASTVisitor
+visitKids(t : AST) : void
+visitProgramTree(t : AST) : Object
+visitBlockTree(t : AST) : Object
+visitFunctionDeclTree(t : AST) : Object
+visitCallTree(t : AST) : Object
+visitDeclTree(t : AST) : Object
+visitIntTypeTree(t : AST) : Object
+visitBoolTypeTree(t : AST) : Object
+visitFormalsTree(t : AST) : Object
+visitActualArgsTree(t : AST) : Object
+visitIfTree(t : AST) : Object
+visitWhileTree(t : AST) : Object
+visitSwitchBlockTree(t : AST) : Object
+visitCaseBlockTree(t : AST) : Object
+visitCaseStatementTree(t : AST) : Object
+visitCaseListTree(t : AST) : Object
+visitDefaultStatementTree(t : AST) : Object
+visitSwitchTree(t : AST) : Object
+visitReturnTree(t : AST) : Object
+visitAssignTree(t : AST) : Object
+visitIntTree(t : AST) : Object
+visitUtfStringLitTree(t : AST) : Object
+visitTimeStampLitTree(t : AST) : Object
+visitIdTree(t : AST) : Object
+visitRelOpTree(t : AST) : Object
+visitAddOpTree(t : AST) : Object
+visitMultOpTree(t : AST) : Object
+visitUstringTypeTree(t : AST) : Object
+visitTimeStampTypeTree(t : AST) : Object
CountVisitor
-nCount : int[] = new int[100]
-depth : int = 0
-maxDepth : int = 0
-count(t : AST) : void
+getCount() : int[]
+getMax() : int
+printCount() : void
+visitProgramTree(t : AST) : Object
+visitBlockTree(t : AST) : Object
+visitFunctionDeclTree(t : AST) : Object
+visitCallTree(t : AST) : Object
+visitDeclTree(t : AST) : Object
+visitIntTypeTree(t : AST) : Object
+visitNumberTypeTree(t : AST) : Object
+visitScientificTypeTree(t : AST) : Object
+visitFloatTypeTree(t : AST) : Object
+visitVoidTypeTree(t : AST) : Object
+visitBoolTypeTree(t : AST) : Object
+visitFormalsTree(t : AST) : Object
+visitActualArgsTree(t : AST) : Object
+visitIfTree(t : AST) : Object
+visitWhileTree(t : AST) : Object
+visitForTree(t : AST) : Object
+visitReturnTree(t : AST) : Object
+visitAssignTree(t : AST) : Object
+visitIntTree(t : AST) : Object
+visitUtfStringLitTree(t : AST) : Object
+visitTimeStampLitTree(t : AST) : Object
+visitNumberTree(t : AST) : Object
+visitScientificTree(t : AST) : Object
+visitFloatTree(t : AST) : Object
+visitVoidTree(t : AST) : Object
+visitIdTree(t : AST) : Object
+visitRelOpTree(t : AST) : Object
+visitAddOpTree(t : AST) : Object
+visitMultOpTree(t : AST) : Object
+visitStringTypeTree(t : AST) : Object
+visitCharTypeTree(t : AST) : Object
+visitStringTree(t : AST) : Object
+visitCharTree(t : AST) : Object
+visitUnlessTree(t : AST) : Object
+visitSwitchTree(t : AST) : Object
+visitSwitchBlockTree(t : AST) : Object
+visitCaseBlockTree(t : AST) : Object
+visitCaseTree(t : AST) : Object
+visitCaseStatementTree(t : AST) : Object
+visitCaseListTree(t : AST) : Object
+visitDefaultStatementTree(t : AST) : Object
+visitUstringTypeTree(t : AST) : Object
+visitTimeStampTypeTree(t : AST) : Object
DrawVisitor
-nodew : int = 100
-nodeh : int = 30
-vertSep : int = 50
-horizSep : int = 10
-width : int
-height : int
-nCount : int[]
-progress : int[]
-depth : int = 0
-bimg : BufferedImage
-g2 : Graphics2D
+DrawVisitor(nCount : int[])
-max(array : int[]) : int
+draw(s : String, t : AST) : void
-createGraphics2D(w : int, h : int) : Graphics2D
+getImage() : BufferedImage
+visitProgramTree(t : AST) : Object
+visitBlockTree(t : AST) : Object
+visitFunctionDeclTree(t : AST) : Object
+visitCallTree(t : AST) : Object
+visitDeclTree(t : AST) : Object
+visitIntTypeTree(t : AST) : Object
+visitBoolTypeTree(t : AST) : Object
+visitFloatTypeTree(t : AST) : Object
+visitVoidTypeTree(t : AST) : Object
+visitFormalsTree(t : AST) : Object
+visitActualArgsTree(t : AST) : Object
+visitIfTree(t : AST) : Object
+visitUnlessTree(t : AST) : Object
+visitWhileTree(t : AST) : Object
+visitForTree(t : AST) : Object
+visitReturnTree(t : AST) : Object
+visitAssignTree(t : AST) : Object
+visitIntTree(t : AST) : Object
+visitUtfStringLitTree(t : AST) : Object
+visitTimeStampLitTree(t : AST) : Object
+visitIdTree(t : AST) : Object
+visitRelOpTree(t : AST) : Object
+visitAddOpTree(t : AST) : Object
+visitMultOpTree(t : AST) : Object
+visitStringTypeTree(t : AST) : Object
+visitCharTypeTree(t : AST) : Object
+visitSwitchTree(t : AST) : Object
+visitSwitchBlockTree(t : AST) : Object
+visitCaseBlockTree(t : AST) : Object
+visitCaseListTree(t : AST) : Object
+visitCaseStatementTree(t : AST) : Object
+visitDefaultStatementTree(t : AST) : Object
+visitUstringTypeTree(t : AST) : Object
+visitTimeStampTypeTree(t : AST) : Object

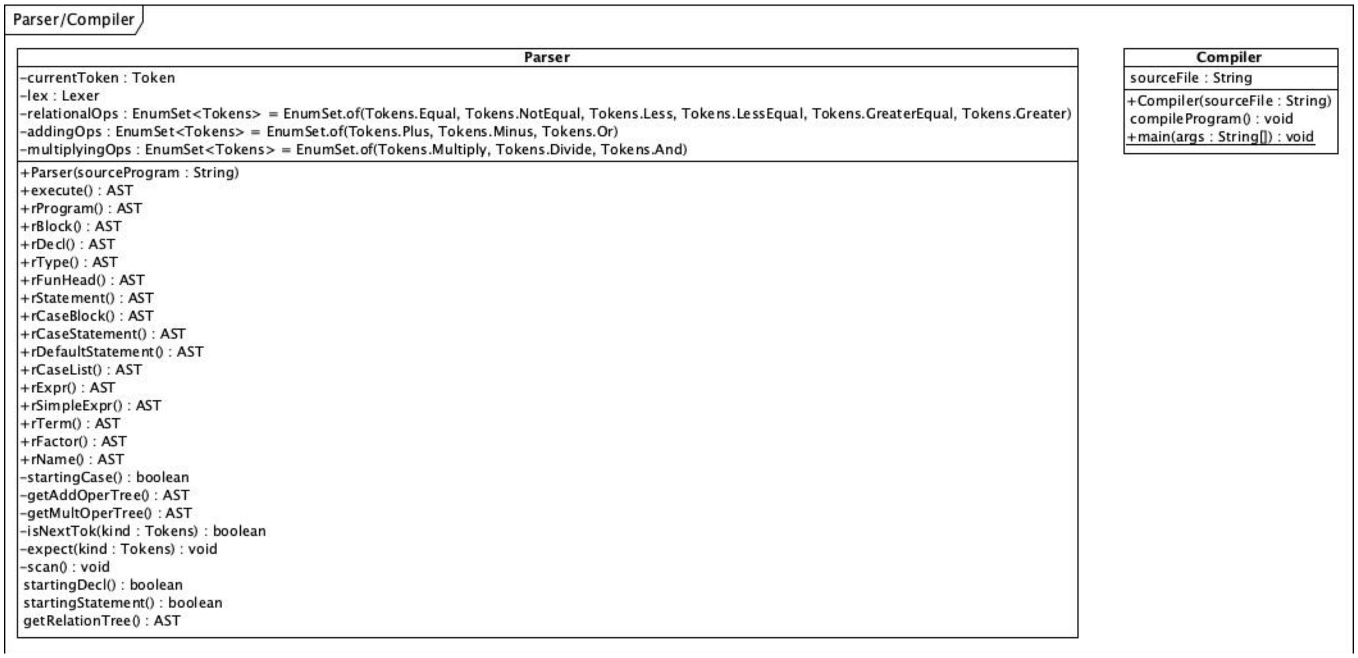
Here is a diagram for the AST classes that I implemented:



Here is a diagram for the other AST classes:



Here is the diagram for the Parser and Compiler classes:


Parser/Compiler
Parser
-currentToken : Token
-lex : Lexer
-relationalOps : EnumSet<Tokens> = EnumSet.of(Tokens.Equal, Tokens.NotEqual, Tokens.Less, Tokens.LessEqual, Tokens.GreaterEqual, Tokens.Greater)
-addingOps : EnumSet<Tokens> = EnumSet.of(Tokens.Plus, Tokens.Minus, Tokens.Or)
-multiplyingOps : EnumSet<Tokens> = EnumSet.of(Tokens.Multiply, Tokens.Divide, Tokens.And)
+Parser(sourceProgram : String)
+execute() : AST
+rProgram() : AST
+rBlock() : AST
+rDecl() : AST
+rType() : AST
+rFunHead() : AST
+rStatement() : AST
+rCaseBlock() : AST
+rCaseStatement() : AST
+rDefaultStatement() : AST
+rCaseList() : AST
+rExpr() : AST
+rSimpleExpr() : AST
+rTerm() : AST
+rFactor() : AST
+rName() : AST
-startingCase() : boolean
-getAddOperTree() : AST
-getMultOperTree() : AST
-isNextTok(kind : Tokens) : boolean
-expect(kind : Tokens) : void
-scan() : void
startingDecl() : boolean
startingStatement() : boolean
getRelationTree() : AST
Compiler
sourceFile : String
+Compiler(sourceFile : String)
compileProgram() : void
+main(args : String[]) : void

Results and Conclusion

This project was particularly interesting because it built on the previous work done with the lexer. Parsing consists of the following step in the process of compilation.
It was very stimulating to work on existing production rules and implement some new ones.
This part of the work reminded me of the MU puzzle introduced by Hofstadter in his book Goedel Escher and Bach. The puzzle consisted in producing a particular string using some transformation rules provided in the instructions.
The rest of the work was also particularly interesting because it offered a great opportunity to better understand how an abstract syntax tree is generated and its usefulness when it comes to checking syntax errors in the process of compilation.
I found the rendering part also interesting but less complicated. Probably because the data structure that I decide to use offered a good tool to retrieve all the information needed to render each node in the right position within the image.

Challenges

At the very beginning, it took me a while to understand how exactly the parser was working and the generation of the abstract syntax tree. The lectures and the material of the slides helped me to have a clear picture of the whole process.
The first part of the assignment, related to the production rules, didn't pose many obstacles. It was more a lengthy process than a very complicated one. Especially the switch statement required more time than the other production rules. Most of the time I checked how the same

issue was solved by the parser for similar production rules and I emulated the logic to solve the issue I was working on. Probably the most tricky part was the possibility to include an infinite number of expressions in form of a list. The example offered by other production rules helped me to create the right logic for this particular case.
When it came to designing the offsetVisitor I found the most challenging part the case in which the children had to be adjusted if the parent node was shifted to a different offset than the one calculated. I believe the recursive mechanism was a great tool and being able to understand how it works helped me to design the correct implementation for this part of the program.
The data structure to be passed as a result from the offsetVisitor to the DrawOffsetVisitor wasn’t particularly challenging but it represented the most crucial element for this second part of the assignment. Choosing the right data structures helped to store and retrieve the information very easily, which resulted in a smooth process of rendering the tree.

Future Work

I would like to be able to expand the Parser a little more to include a few other production rules, for example, more complicated statements or a for-loop. They might be not too complicated to implement by it would be interesting to have a more complete set of production rules at disposal.
When it comes to manipulating and working with the abstract syntax tree, I would like to be able to better understand how the interpreter integrates with the work done by the parser. I guess this is what the next assignment is about, that is why I am looking forward to it.

Summary of Technical Work

Category	Description		Notes
Code Quality (15)		✓ Code is clean, well formatted (appropriate white space and indentation)	
		✓ Classes, methods, and variables are meaningfully named (no comments exist to explain functionality - the identifiers serve that purpose)	
		✓ Methods are small and serve a single purpose	
		✓ Code is well organized into a meaningful file structure	
Documentation (30)		✓ A PDF is submitted that contains:	
		✓ Full name/Student ID	
		✓ A link to the github repository	
	Overview	✓ Project introduction	
	Overview	✓ Summary of technical work	
	Overview	✓ Execution and development environment described	
	Overview	✓ Scope of work described (including what work was completed)	
		✓ Command line instructions to compile and execute	
		✓ Assumptions made	
	Implementation discussion	✓ Class diagram with hierarchy	
	Implementation discussion	✓ Implementation decisions	
	Implementation discussion	✓ Code organization	
		✓ Results/Conclusions	
		✓ Formatting	
Requirement 1 (40)	Parser updates	✓	
Requirement 2 (15)	Tree rendering	✓	