Shane Waxler

918415347

March 30, 2021

Assignment 3 Documentation

Github Repository

[Link to Assignment 3 Github]

Project Introduction and Overview

For this project we updated the codebase for our compiler in order to include parsing. Parsing takes place after lexing and it is meant to turn the lexed tokens into a tree structure based on a particular grammar. With this, we also created a few tree visitors in order to render a well-structured diagram using 2D java graphics.

Scope of Work

- Update Lexer to accommodate for new tokens (“unless”, ‘:’, etc.)
- Update Parser logic to accommodate for new grammar rules
    - Base Parser file works for some simple X language files, but new rules need to be added
- Remove all debug print statements for clean output
- Create OffsetVisitor and DrawOffsetVisitor files
    - OffsetVisitor visits each node and calculates a graph position hashtable
    - DrawOffsetVisitor will use the hashtable to draw a well-structured 2D graphic tree

Execution and Development Environment

For the development and writing portion of this project I used VSCode as a text editor and ran the program through windows powershell. For creating the Class Diagrams I used IntelliJ’s IDE.

JDK version 15.0.1

Command Line Instructions to Compile and Execute

To compile this project I used Compiler.java. For example, running through Windows command line:

```
javac compiler/Compiler.java
java compiler/Compiler “<insert .x file path here>”
```

Assumptions

When writing this I assumed that the X code files may contain syntax errors. Each function defined in Parser extends SyntaxError which throws an error if the grammar rules are not followed properly. When an error is encountered, SyntaxError will print a simple explanation of the error and gracefully exit.

Implementation

TokenSetup

TokenSetup from Lexer was used to update the Tokens file for the new Tokens. Nothing about the implementation was changed from the previous project.

Parser

Parser was updated to accommodate for the new grammar rules. This includes new lines expecting Tokens such as Switch, Case, CharLit, and Integer. These implementations were simple because the base logic and structure for Parser was already meant to handle similar cases to the ones we implemented. Implementing Switch statements was slightly different because it involved a while loop that checked for at least one Case statement and exited upon finding a right bracket or a single default statement.

AST Folder

In the AST folder a few new ASTs were defined in order to accommodate for new node types. All of these simply extended the logic that was already included in the base files so listing them is unnecessary. For literal-type trees the node definition includes a Token argument in the constructor to store the symbol of that token (value of the token). Any of the other tree types just include an empty constructor and an accept method which takes an ASTVisitor as an argument and returns the visit method according to the tree type with itself as a parameter.

ASTVisitor

ASTVisitor was updated to include new abstract declarations for the tree types. Each of these takes an AST as an argument. The visitKids method visits each of the children for a given node by iterating through an ArrayList of its children.

PrintVisitor

For PrintVisitor, no changes were made to the algorithm; however, it is a subclass of ASTVisitor so each of the visit methods for the different tree types had to be updated. These methods

simply print the name of the trees recursively (as well as the symbol if the tree is type literal), and the algorithm accounts for the white space based on the tree structure created by lexer.

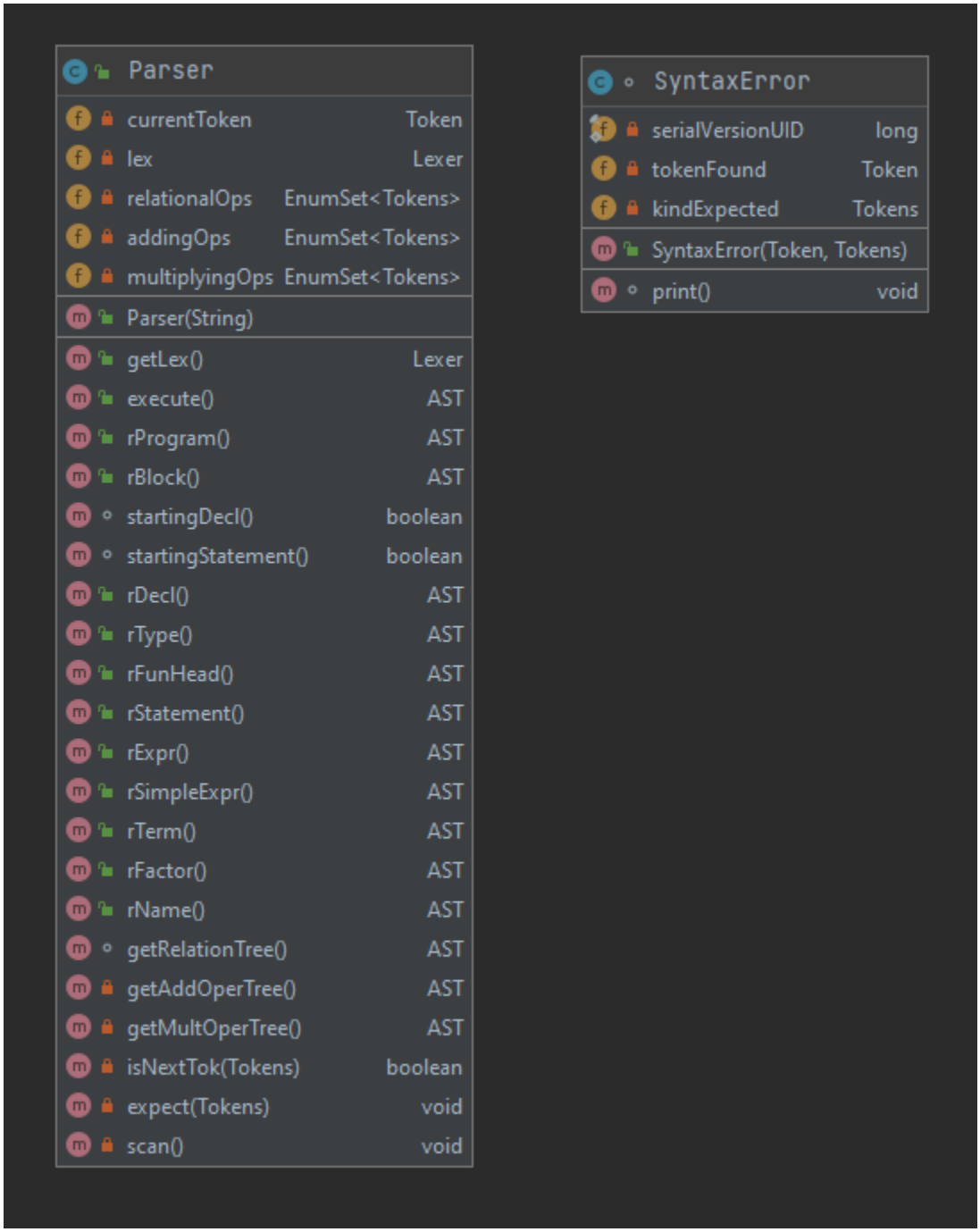
OffsetVisitor

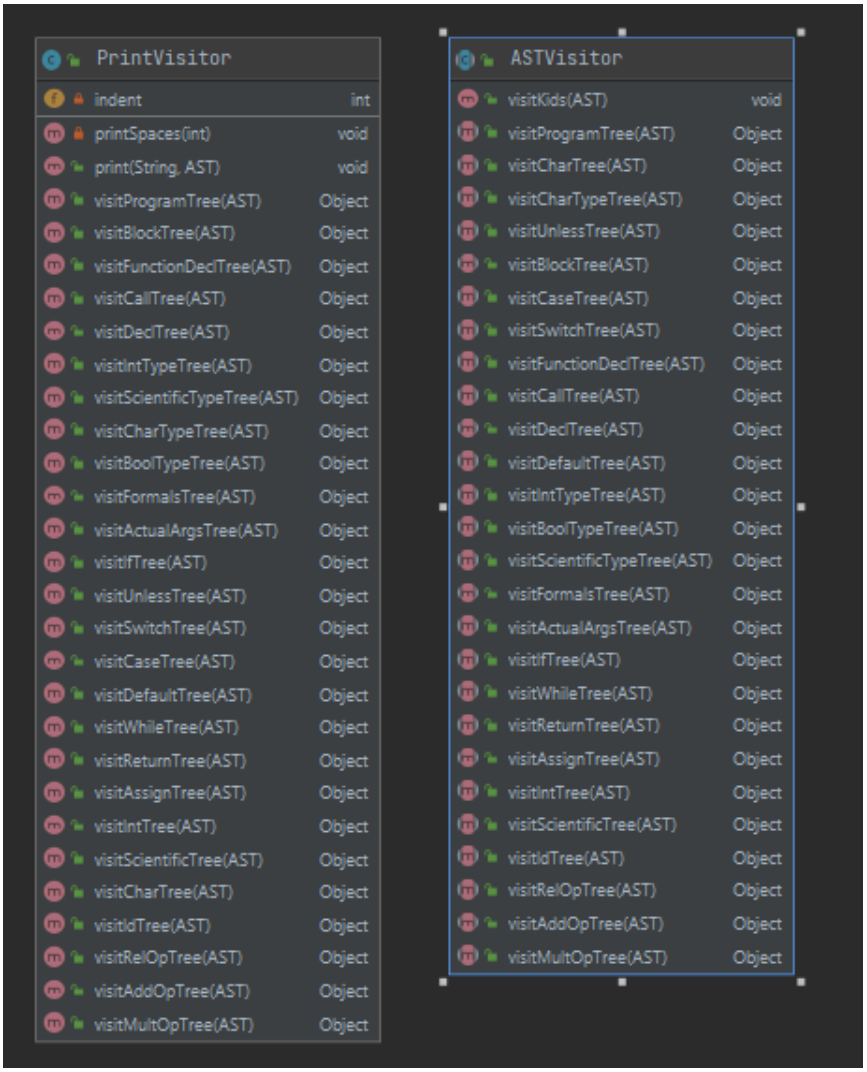
OffsetVisitor is a class which visits each node of the tree recursively through postorder traversal. With a hashmap (Key is type AST and value is an integer array of length 2) it stores each node in the map and sets the first value of the array as the node's Offset and the second value as the depth. This map will be used in DrawOffsetVisitor to create an accurate layout for a map diagram. The offset values are calculated by a simple algorithm which normally iterates each value of the next offset by 2 and updates those values if they do not match up according to the previous offset at that depth.

DrawOffsetVisitor

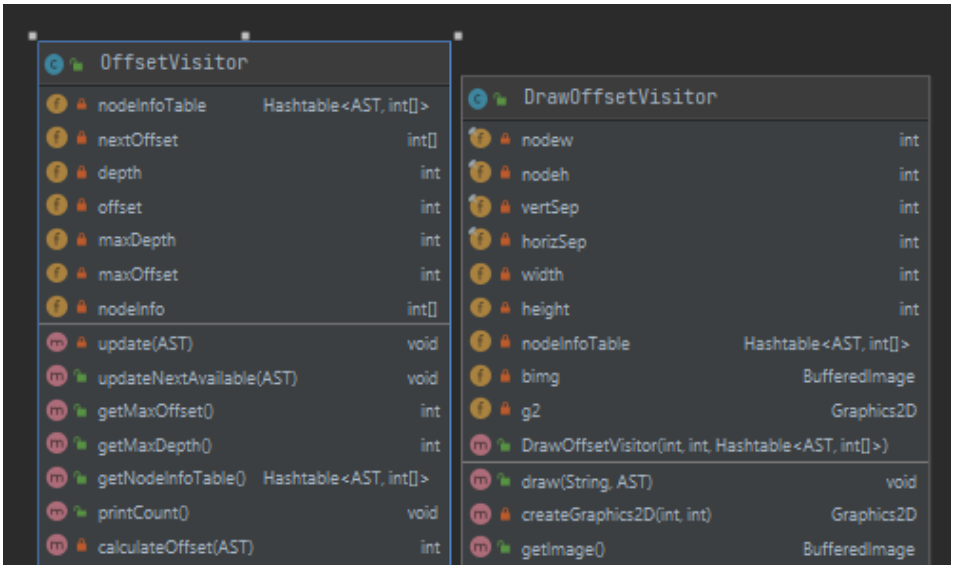
DrawOffsetVisitor takes the hashtable created in OffsetVisitor and iterates through it by recursively visiting each child and calling the method draw. Draw calculates x and y coordinates for each node and draws an oval shape with the name of the tree printed inside. If the tree has any children, lines will be drawn from the bottom of the parent to the top of the child.

Class Diagrams





(All visit methods are hidden in the diagram for DrawOffset and OffsetVisitor to reduce redundancy)



Results and Conclusion

This project gave me a much better understanding of how code is compiled. Working with code grammar and creating a traversable tree makes the process far easier to understand from the perspective of the compiler. Previous to the assignment I had not worked much with tree traversal and therefore had a minimal understanding of how code processing could happen even after tokens were created from the source file.

Challenges

Similar to the previous assignments, the biggest challenge was sorting the task into smaller pieces. Since the codebase is so large it is hard to understand how each part interacts with another without closely examining the program. In addition, tree traversal was something I only understood from a theory perspective, so traversing was a mild challenge at first.

Future Work

In the future I assume we will be working with this lexer in order to make a fully functioning compiler for “.x” code. The parser could certainly be expanded to account for a more complex grammar. This would make the utility of “.x” code far greater. In addition, the graphics rendering could be improved so the window does not take up as much space after running the code.