Documentation

Overview :

The assignment introduced us to the Parser which is a component of the Compiler. The program receives a source file as input and breaks it down into tokens , where a token, such as "program", "int", etc) is defined by the grammar of the custom language ‘X’. The goal of this assignment was to help us understand how the Parser processes various types of tokens by checking if the syntax and structure is correct and also constructs the Abstract syntax tree, both textually and visually.

Development Environment and Execution Instructions :

This program was developed on IntelliJ IDE and JDK 8.

Following steps are used to compile and run the program :

Go inside the directory :

Compile all files with the command : **javac */*.java**

Run the compiler : **java compiler/Compiler sample_files/<file-name> for eg : java compiler/ Compiler sample_files/simple.x**

Scope of Work :

Requirement 1 : `Parser` was successfully modified to include the new production rules to integrate the new tokens.

Requirement 2 : All unnecessary debug statements were removed including the listing of tokens from the Assignment 2 output.

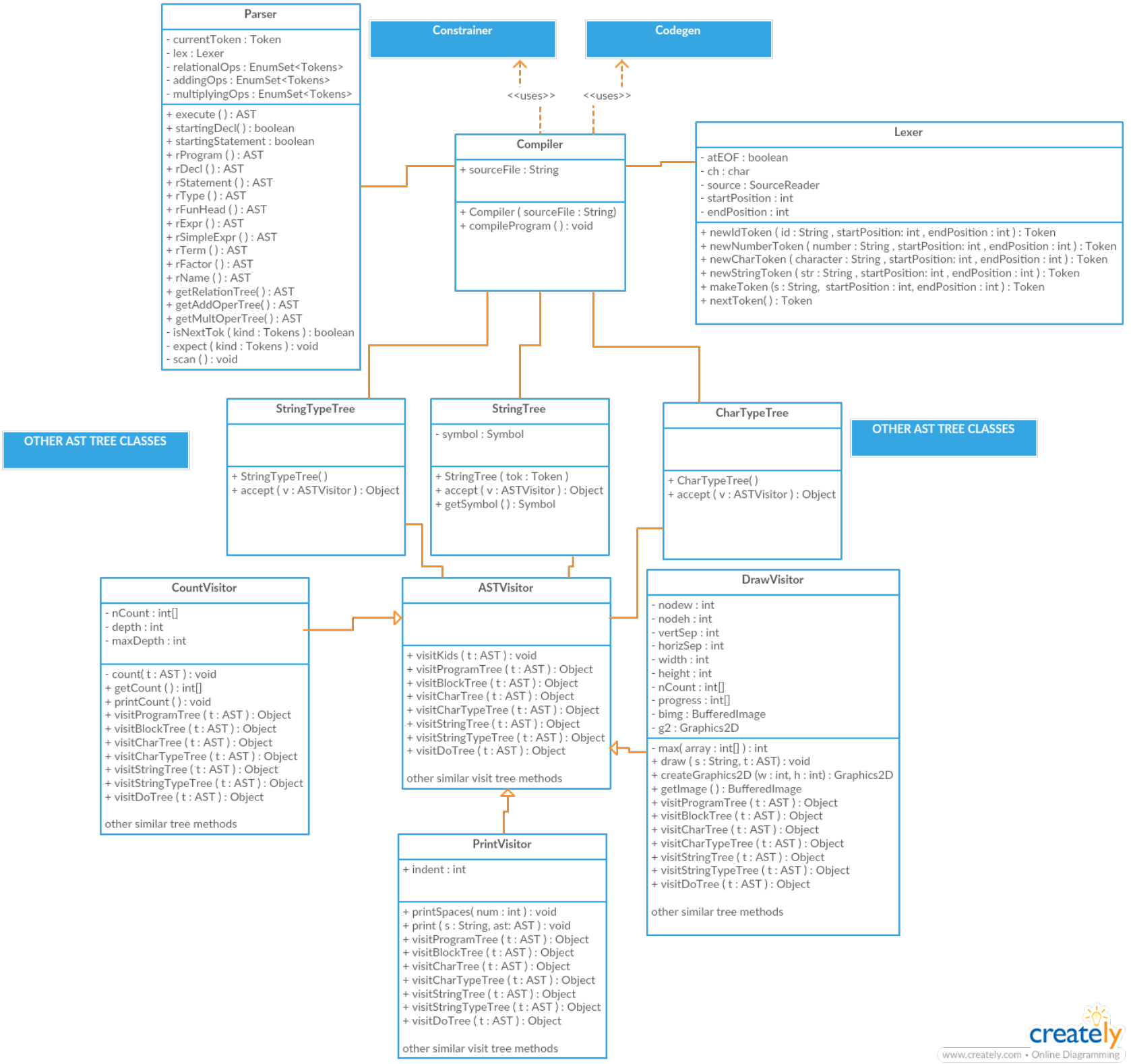
Requirement 3 : Properly aligned Abstract Syntax Tree is displayed and saved.

Assumptions :

The source file will be “x” file. Since the grammar used is of “x” language, the source file has .x extension.

Implementation Discussion :

Class Diagram :



Compiler – The Compiler.java class was used to first call the Lexer to get the stream of tokens and then the Parser was called and finally Abstract Syntax Tree was drawn.

Parser – The Parser class performs recursive-descent parsing; as a by-product it will build the Abstract Syntax Tree representation for the source program. It checks if the program is syntactically and structurally correct or else it throws Syntax Error exception. Following are the methods that were modified to include the corresponding tokens :

Char token : **rType(), startingDecl()**

STring token : **rType(), startingDecl()**

if statement (without else) : **rStatement(), startingStatement()**

do while : **rStatement(), startingStatement()**

Greater token : **private EnumSet<Tokens> relationalOps**

GreaterEqual token : **private EnumSet<Tokens> relationalOps**

Character literals : **rFactor()**

String literals : **rFactor()**

ASTVisitor – ASTVisitor class is the root of the Visitor hierarchy for visiting various AST's; each visitor asks each node in the AST it is given to accept its visit; each subclass must provide all of the visitors mentioned in this class; after visiting a tree the visitor can return any

Object of interest. For example when the constrainer visits an expression tree it will return a reference to the type tree representing the type of the expression.

PrintVisitor Class – PrintVisitor is used to visit an AST and print it using appropriate indentation. It also holds objects of each type of tree and what they must print out.

Code Organisation :

compiler package includes the file Compiler.java that handles the overall compilation process. **ast**

package includes classes for various types of trees such as CharTree, CharTypeTree, StringTree,

StringTypeTree etc. **visitor package** includes ASTVisitor class (abstract class) which is the

superclass of other Visitor classes or processors such as PrintVisitor(), CountVisitor() and DrawVisitor().

Results :

I learnt the mechanics of the Parser and how it checks the structure and syntax of the stream of tokens received from the Lexer. I also learnt how to construct an Abstract Syntax Tree using GUI libraries. I learnt how the Visitor design pattern uses Double Dispatch when linking the appropriate accept() and visit() method at runtime. Overall, I gained a thorough understanding of how the Parser plays an important role in the compilation process.