## CS5800 – theory of foundations

## Western Michigan University

Instructor: Dr. Elise de Doncker

TA: Edwin Jose

Team: Sri Harsha Godavarthi, Venu Modugula

Sathish Kumar madiga Mallayolla

**1. Introduction**

This document details the design of a mini Java compiler built using Python 3 and the PLY library. The compiler will be able to analyze and understand a predefined subset of the Java programming language.

**2. Tools and Technologies**

* **Programming Language:** Python 3
* **Library:** PLY (Python Lex-Yacc)
  + Lexical Analyzer (Lexer): Processes input code into tokens (keywords, identifiers, operators, etc.) using regular expressions.
  + Parser: Analyzes tokens based on grammar rules to determine the code's structure. This project will utilize the LALR(1) parsing method.

**3. Grammars and Definitions**

* **Context-Free Grammar (CFG):** A set of rules defining valid structures within the mini Java language.
* **Abstract Syntax Tree (AST):** A tree-like representation of the parsed code, capturing its hierarchical structure.

**4. Design Phases:**

**Phase 1: Lexical Analysis**

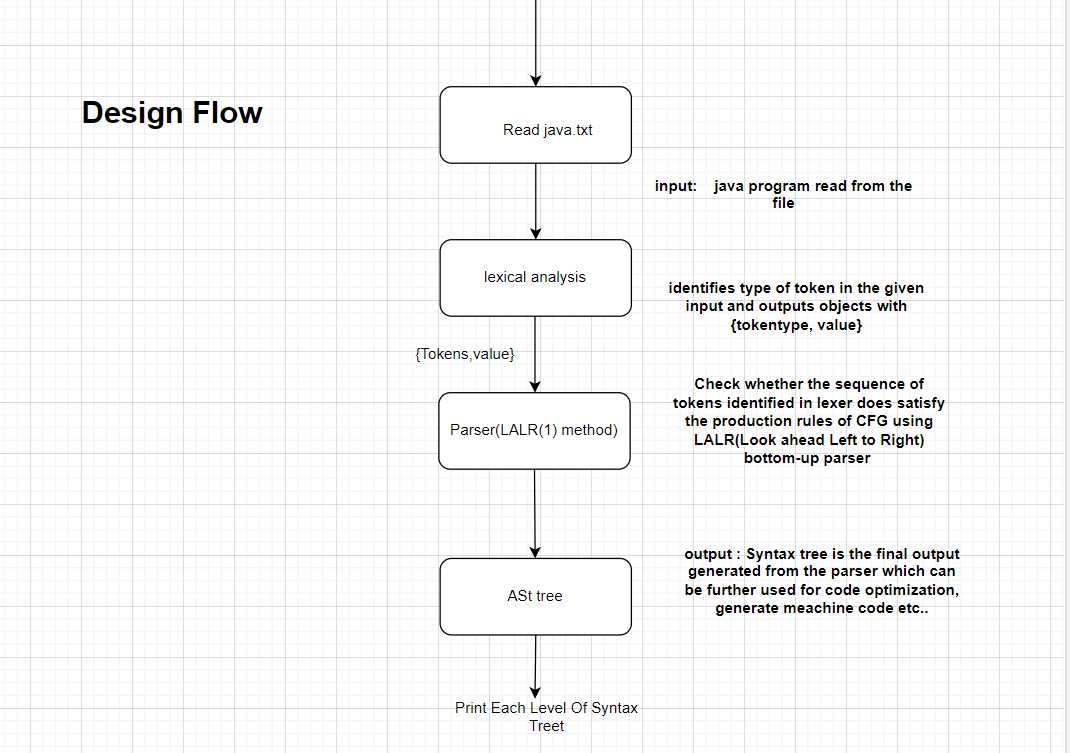
1. **Token Identification:** Define regular expressions using PLY's tokens function to identify various tokens like keywords, identifiers, operators, delimiters, and literals.
2. **Lexer Implementation:** Develop the lexer module using lex.py in PLY to perform tokenization based on the defined regular expressions.

**Phase 2: Parsing**

1. **Grammar Specification:** Define the mini Java grammar using PLY's yacc.py module, employing Backus-Naur Form (BNF) notation. This grammar will specify valid syntax constructs within the mini Java language.
2. **Parser Implementation:** Develop the parser module to analyze the token stream generated by the lexer and verify if it adheres to the defined grammar rules. The LALR(1) built-in parsing method will be used for this purpose.
3. **Error Handling:** Integrate error handling mechanisms within the parser to identify and report syntax errors encountered during the parsing process.

**Phase 3: Abstract Syntax Tree Generation**

1. **AST Node Creation:** Define classes for different AST node types representing various language constructs (e.g., expressions, statements, declarations).
2. **AST Building:** During parsing, construct the AST by creating appropriate AST nodes based on the matched grammar rules and attaching them in a hierarchical manner reflecting the code structure.



**5. Project Deliverables**

* **Python source code:** Complete source code for the mini Java compiler, including lexer, parser, and AST generation modules.
* **Documentation:** User guide explaining how to use the compiler and interpret the generated AST.
* **Test cases:** A set of test cases covering various functionalities of the mini Java language to ensure the compiler's correctness.

**6. Evaluation Criteria**

* **Functionality:** Ability to successfully parse valid mini Java code and generate the corresponding AST.
* **Error Handling:** Effective identification and reporting of syntax errors during the parsing process.(it is not included in design flow but will be handled in parser)
* **Code Quality:** Well-structured, documented, and efficient code implementation.
* **Test Coverage:** Comprehensiveness of test cases covering different aspects of the mini Java language.

**7. Future Enhancements**

* Expanding the supported subset of the Java language.
* Implementing semantic analysis to check for type compatibility and other semantic errors.
* Code generation phase to translate the parsed code using AST into a lower-level language (e.g., assembly code)

**8 Modules outline for implementation:**

This Project has 2 module which will be written as classes

* 8.1.1 Tokenizer:

i) identifier holds regular Expression for each token

ii) methods for doing lexical analysis

iii) create lexical analyzer using ply

iv) display token and it’s type for each one in source code

* 8.1.2 Parser:

i) pass tokens, source code, lexer (i.e. build during before lexical analysis)

ii) create a grammar for parsing the input source code

iii) build the parser

iv) display shift and reduce steps based on grammar or if error is present display where the error occured

v) call display\_ast to display each level of Parse Tree

* 8.1.3 AstNode:

i) Assign nodetype,children,value which are properties of node for Ast Tree

ii) call the class constructor and pass the aurguments

* 8.1.3 Main:
* where everything is integrated