## 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

**Purpose:** thisprojectisto create a mini Java compiler using yacc in python3with the help of CFG (Context Free Grammar) and Regular Expressions.

**Introduction:**

Yacc (Yet Another Compiler Compiler) is a powerful tool for generating parsers that can recognize and analyze the structure of text according to a set of grammar rules.

PLY provides a pure Python implementation of lex and yacc, allowing you to build parsers entirely within the Python environment. It revolves around two main components: lexical analysis and parsing

* **Lexical Analysis (Lexer): This process decomposes the input text into a series of meaningful elements known as tokens. These elements include keywords, identifiers, operators, delimiters, and literals. The lex.py module in PLY aids in developing regular expressions that establish the patterns for identifying these tokens.**
* **Parsing (Parser): This step analyzes the tokens produced by the lexer, employing grammar rules to ascertain the input's syntactic arrangement. The yacc.py module in PLY is instrumental in defining this grammar, utilizing Backus-Naur Form (BNF) notation. Through aligning these rules with the stream of tokens, the parser constructs an Abstract Syntax Tree (AST), which depicts the input's hierarchical organization**

**Phase 1: Possible** **Approaches**

there are two different methods for parsing

1. **Top-down parsing techniques:**
2. **Recursive Descent Parsing:** A method that transforms grammar rules into recursive functions, with each function dedicated to a particular non-terminal symbol and recursively invoking other functions for the associated productions
3. **LL(k) Parsing:** This method employs left recursion and leftmost derivation, along with a k-token lookahead. By utilizing a lookahead of k tokens, it makes parsing decisions that ensure the selection of the appropriate path while preventing the need for backtracking.
4. **Predict-Lookahead Parsing:** This method merges aspects of recursive descent and LL(k) parsing, aiming to anticipate the upcoming token by considering the current state of parsing and conducting lookahead only as required.
5. **Bottom-up:**
6. **Shift-Reduce Parsing:** This basic method involves the parser transferring tokens from the input to a stack through a process known as "shifting," and then attempting to "reduce" these tokens by merging them into bigger constructs according to grammar rules. It continues to shift and reduce until it has fully parsed the input and arrives at the starting symbol on the stack.
7. **LR(k) Parsing:** This technique, which is more potent and relies on Lookahead, employs Rightmost derivation with k lookahead. It constructs the parse tree from the bottom up, utilizing lookahead to guide decisions regarding reductions.
8. **LALR(K) parsing:** The LALR parser is a bottom-up parsing method. It constructs the parse tree from the leaves (the input tokens) upwards towards the root. This approach involves combining tokens into larger units according to the grammar rules, progressively building up the structure of the parse until the entire input is reduced to a single start symbol that represents the entire program or input phrase

**Ply Module Syntax and usage:**

1. **Grammar:**

Def p\_rulename(self,p):

‘expression: expression PLUS term ’

//The above one represents a production rule in CFG of the form

**expression -> expression PLUS term**, where expression and term are non-terminals and PLUS is terminal defined as a regular expression which represents a token in a programming language

1. **Regular Expressions:**

**tokens are defined as:**  t\_PLUS = r'\+'

**Token Classes:**

Keywords

Identifiers

Literals

Logical\_Operators

Arthimatic Operators

Comments

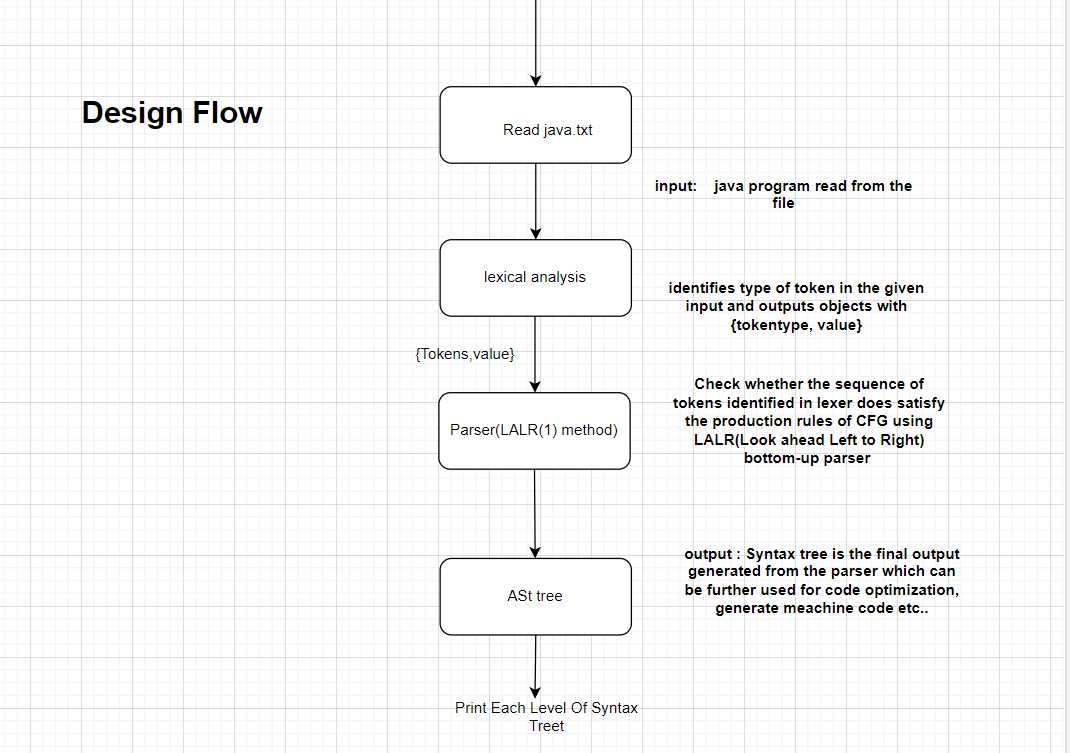
LPAR //left parenthesis

RPAR // right parenthesis

LBRC // left bracket

RBRC // right bracket

**Phase 2: Design Flow**



**Lexical Analysis:** the input sequence of characters (often source code) is transformed into a sequence of tokens using regular expression defined.

**Parsing:** takes tokens provided by the lexer (i.e. from lexical analysis) and checks whether it can generated by the grammar of the language using the LALR method and outputs the AST tree.

**LALR(1):** this method will be used for syntactic analysis in compilers (i.e. for parsing) which uses look ahead from left-right(shift and reduce)

**Parser Tree:** This is the syntax tree the parser generates for further optimization and transformation of source code to machine code. (the tree generated here is called Abstract Syntax Tree)

**Phase 3: Research Summary**

This research delves into the world of compiler construction, specifically focusing on parsing techniques.

One paper explores how Lex and Yacc, powerful tools for pattern recognition and parser generation, can be used to build a compiler for a simple calculator. This demonstrates how to translate a high-level language into machine code for execution.

Other papers delve into different parsing techniques. One explores bottom-up parsing, a method that analyzes source code from the ground up, suitable for specific grammar types and handling potential conflicts that might arise. Another focuses on recursive-descent parsing, a technique for processing grammars, highlighting its capabilities and potential challenges. By understanding these parsing approaches, compiler developers can choose the most appropriate method for their specific language and grammar

**References:**

* [**https://ieeexplore.ieee.org/document/5942077**](https://ieeexplore.ieee.org/document/5942077)
* [**https://ieeexplore.ieee.org/xpl/ebooks/bookPdfWithBanner.jsp?fileName=6382043.pdf&bkn=6381794&pdfType=chapter**](https://ieeexplore.ieee.org/xpl/ebooks/bookPdfWithBanner.jsp?fileName=6382043.pdf&bkn=6381794&pdfType=chapter)
* [**https://ieeexplore.ieee.org/document/6382048**](https://ieeexplore.ieee.org/document/6382048)
* [**https://ieeexplore.ieee.org/document/6382040**](https://ieeexplore.ieee.org/document/6382040)
* **https://ieeexplore.ieee.org/document/6382042**