give ppt based on content provided

## 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

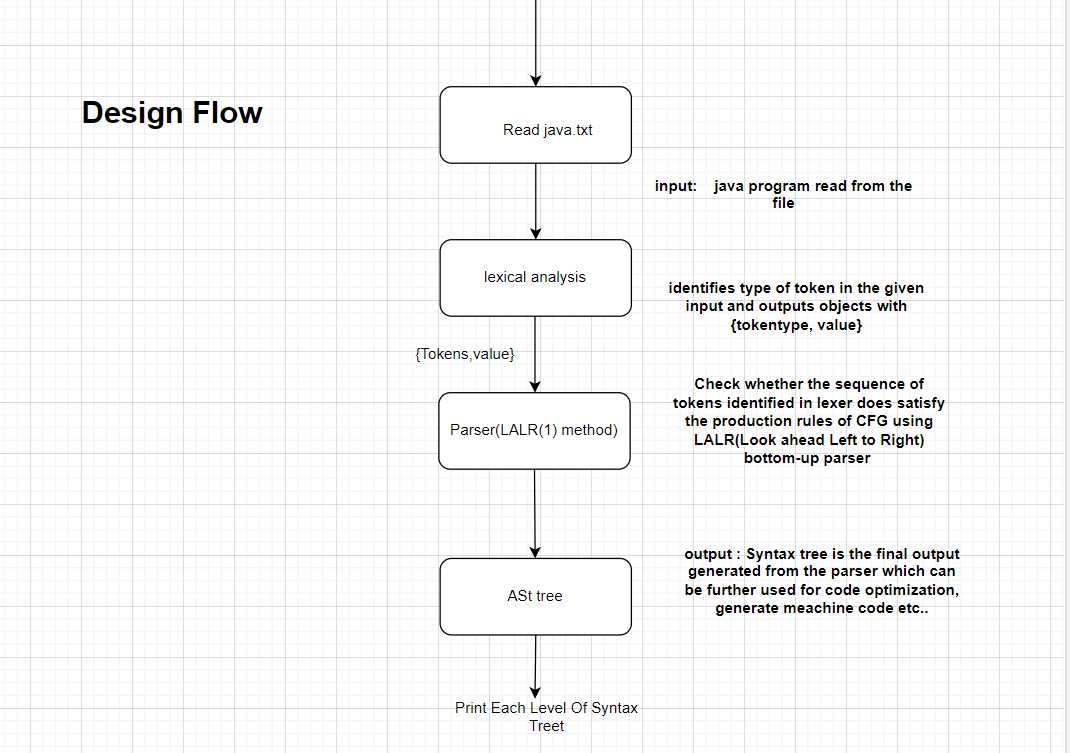
LPAR //left parenthesis

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

**Lexical Analysis Phase**

Objective: The lexical analysis phase, often referred to as the scanner, transforms a sequence of characters into a sequence of tokens. Tokens are categorized according to their role in the language's grammar, such as keywords, identifiers, operators, separators, and others. This phase also handles the detection of illegal characters.

**Test Cases:**

**Keywords Identification**

Input: Class, while, for, constructor

Output: keywords

Explanation: Reserved words like Class, while, for, and constructor are predefined by the language's grammar and should be recognized as keywords, indicating they have a special meaning in the language.

**Identifiers**

**Input:** Variables used by the programmer (e.g., sum, list)

**Output:** identifiers

Explanation: Identifiers are names given to elements like variables and functions. This test case ensures that the scanner can distinguish user-defined names from other token classes.

Arithmetic Operators

Input: Plus (+), Minus (-), Multiplication (\*), Division (/)

Output: Arithmetic Operator String (e.g., PLUS for +, DIV for /)

Explanation: Arithmetic operators should be recognized and labeled according to their function in mathematical expressions.

Logical Operators

Input: logical\_AND (&&), logical\_OR (||) etc..

Output: LOGICAL\_<operator\_name>

Explanation: Logical operators used in conditional statements and boolean expressions must be correctly identified, such as LOGICAL\_AND for &&.

Separators

Input: ;, {}, (), []

Output: SEPARATOR

Explanation: Separators like semicolons, brackets, and parentheses play a critical role in defining the structure of statements and blocks in the program.

Illegal Characters

Input: cl<invalidsymbol\_in\_language>ass

Output: Illegal character: <character symbol>

Explanation: The presence of characters not recognized by the language's grammar should trigger an error, identifying the specific illegal character.

**Parser Phase**

Objective: The parser phase analyzes the sequence of tokens from the lexical phase according to the grammar of the language, constructing a parse tree or abstract syntax tree (AST). This phase checks the syntactic structure of the program and handles syntax errors.

Test Cases:

Valid Syntax

Input: Keyword <identifier> LPR (e.g., int identifier = number;)

Output: AST representing left-most derivation, displaying each level.

Explanation: This case tests the parser's ability to recognize and correctly parse declarations and assignments, building an appropriate AST.

Invalid Syntax

Input: identifier expression

Output: invalid syntax

Explanation: When the input deviates from the expected grammatical structure, the parser should report an error. This case checks the parser's error-reporting mechanism for unexpected or misplaced tokens.

Syntax Error Handling

Missing Delimiters

Case1: A missing semicolon at the end of a statement.

Explanation: The parser should detect and report missing delimiters that are critical for the correct separation of statements.

Incorrect Use of Keywords

Case2: Using class keyword instead of void.

Explanation: The misuse of reserved keywords in incorrect contexts should be flagged as a syntax error.

Unbalanced Parentheses or Braces

Explanation: The parser must ensure that every opening parenthesis or brace has a corresponding closing counterpart. Imbalance in these symbols often indicates a syntactic error.

Additional Test Cases

Reading Input from File

Input: Reading input.txt containing Java code in string format.

Output: Java code as a string.

Explanation: This case tests the system's ability to read and correctly interpret the contents of a file as input for further processing.

Evaluating Expressions

Input: x+y/z+(k+z)

Output: Output based on operator precedence given in the program.

Explanation: This test ensures that the parser correctly applies operator precedence rules when constructing the AST, which is crucial for accurate expression evaluation

**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**

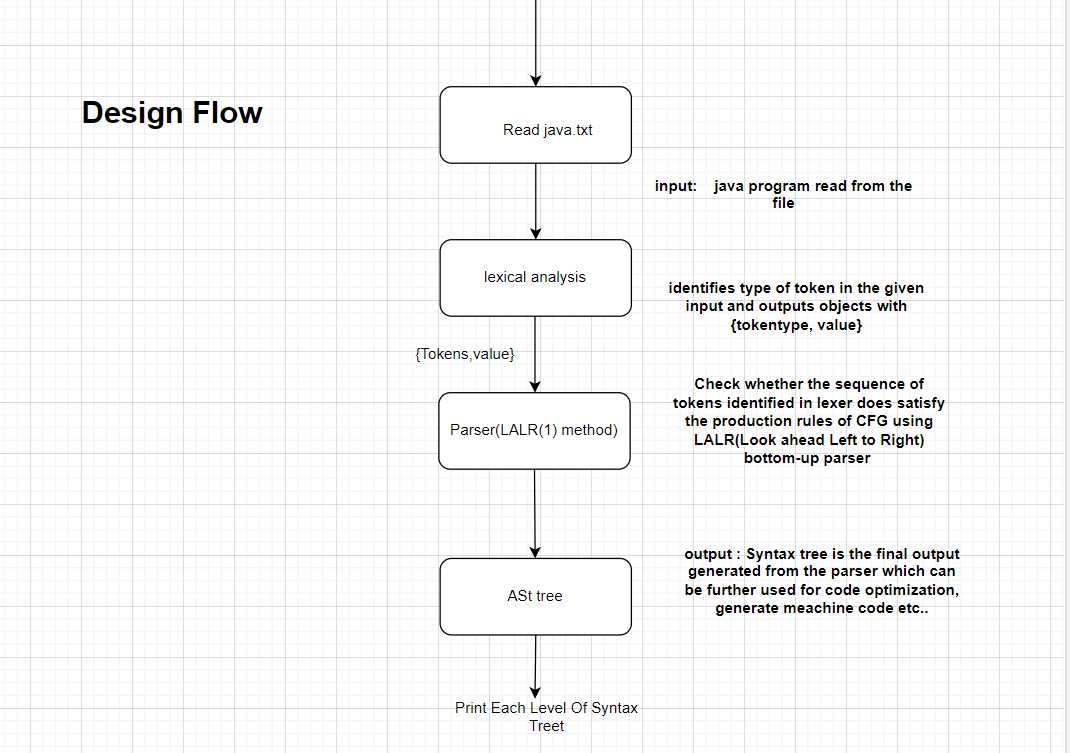
1. **Token Identification:** Define regular expressions using PLY's tokens function to identify various tokens like keywords, identifiers, operators, delimiters, and literals.
2. **Lexer Implementation:** Develop the lexer module using lex.py in PLY to perform tokenization based on the defined regular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

**References:**

* <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/document/6382048>
* <https://ieeexplore.ieee.org/document/6382040>
* **https://ieeexplore.ieee.org/document/6382042**