Complier Design Project

This project creates a **Graphical User Interface (GUI) application** for analyzing and visualizing different components of a compiler's functionality using Python and the Tkinter library.

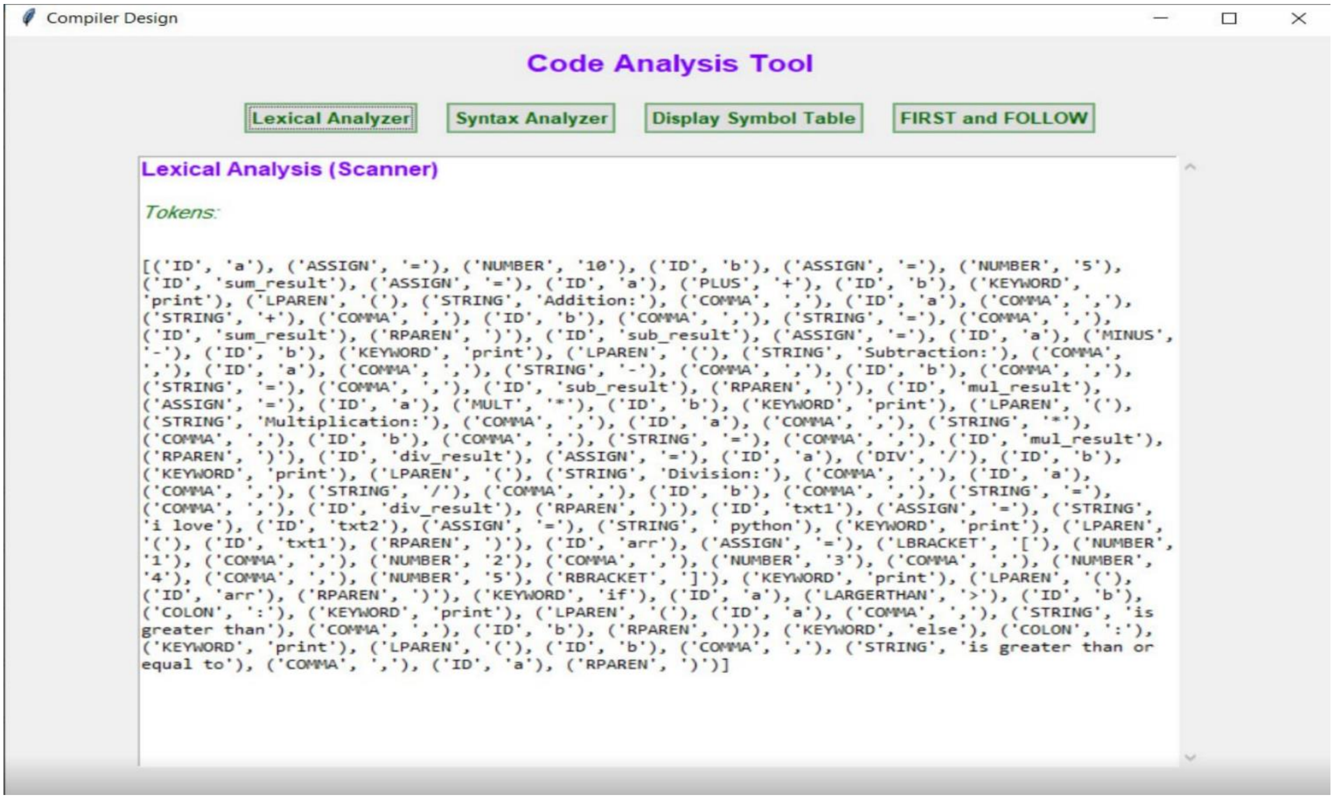
LEXER :This code is a **lexer** that converts source code into tokens for a compiler or interpreter. It:

```python
TOKENS = [
    ("ASSIGN", "="),
    ("PLUS", "+"),
    ("MINUS", "-"),
    ("MULT", "*"),
    ("DIV", "/"),
    ("LARGERTHAN", ">"),
    ("LESSTHAN", "<"),
    ("EQUALS", "=="),
    ("NOTEQUALS", "!="),
    ("LPAREN", "("),
    ("RPAREN", ")"),
    ("LBRACE", "{"),
    ("RBRACE", "}"),
    ("LBRACKET", "["),
    ("RBRACKET", "]"),
    ("COLON", ":"),
    ("COMMA", ","),
    ("NUMBER", "0123456789"),
    ("WHITESPACE", " \t\n"),
]

KEYWORDS = {"if", "else", "for", "print", "in", "range"}

def is_identifier_start(char):
    return char.isalpha() or char == "_"

def is_identifier_part(char):
    return char.isalnum() or char == "_"
```

```python
def tokenize(code):
    tokens = []
    i = 0
    line = 1
    col = 1

    while i < len(code):
        char = code[i]

        if char in " \t\n":
            if char == "\n":
                line += 1
                col = 1
            else:
                col += 1
            i += 1
            continue

        if is_identifier_start(char):
            identifier = char
            i += 1
            col += 1
            while i < len(code) and is_identifier_part(code[i]):
                identifier += code[i]
                i += 1
                col += 1
            if identifier in KEYWORDS:
                tokens.append(("KEYWORD", identifier))
            else:
                tokens.append(("ID", identifier))
            continue
```

```python
            # quote_type = ' \ "
        if char in "'\"":
            quote_type = char
            string_literal = ""
            i += 1
            col += 1
            while i < len(code) and code[i] != quote_type:
                # مافيش قافله استرينج
                if code[i] == "\n":
                    raise ValueError(f"Unterminated string literal at line {line}, col {col}")
                string_literal += code[i]
                i += 1
                col += 1
            if i >= len(code) or code[i] != quote_type:
                raise ValueError(f"Unterminated string literal at line {line}, col {col}")
            i += 1
            col += 1
            tokens.append(("STRING", string_literal))
            continue

        #multi-character operators: == , !=
        if i + 1 < len(code) and code[i:i + 2] in {"==", "!="}:
            tokens.append(("OPERATOR", code[i:i + 2]))
            i += 2
            col += 2
            continue

        # single-character tokens
        for type, character in TOKENS:
            if char in character:
                if type == "NUMBER":
                    number = char
                    i += 1
                    col += 1
                    while i < len(code) and code[i] in "0123456789":
                        number += code[i]
                        i += 1
                        col += 1
                    tokens.append((type, number))
                else:
                    tokens.append((type, char))
                    i += 1
                    col += 1
                break
        else:
            raise ValueError(f"Unexpected character '{char}' at line {line}, col {col}")

    return tokens
```

The result is a list of tokens for further processing in the compilation or interpretation process.



Defines Tokens: Recognizes operators (+, =), symbols ({, }), numbers, strings, and keywords (if, else).

Processes Code:

Skips whitespace.

Identifies keywords or variable names (identifiers).

Detects numbers and string literals.

Handles multi-character (==, !=) and single-character tokens.

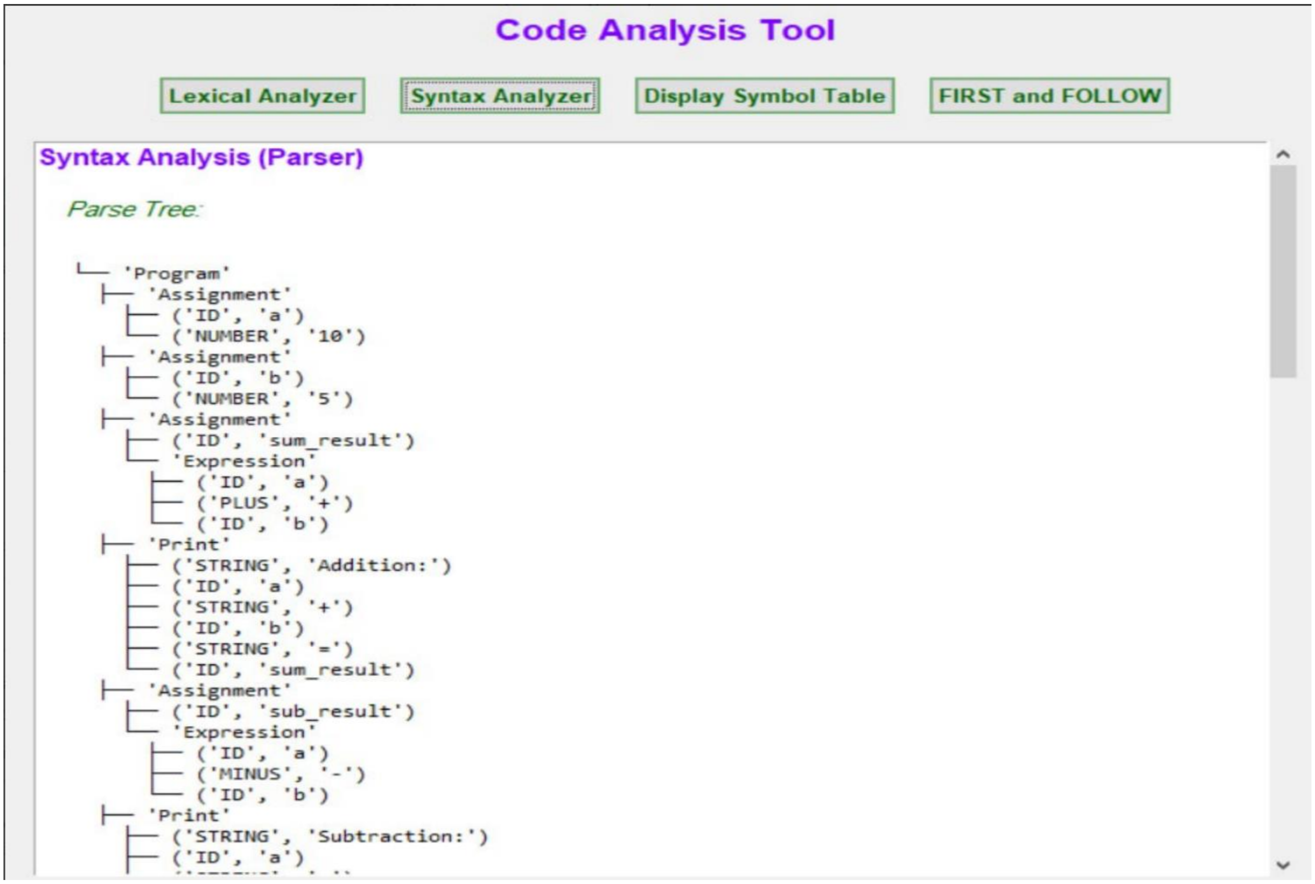
Error Handling: Raises an error for unexpected characters or unterminated strings.

PARSER : This code defines a parser that takes a list of tokens (produced by a lexer) and constructs a parse tree representing the structure of a program. Here's a brief explanation:

```python
class ParseNode:
    def __init__(self, value, children=None):
        self.value = value
        self.children = children or []

    def __str__(self, level=0, is_last=True):
        edge = "└── " if is_last else "├── "
        ret = "    " * level + edge + repr(self.value) + "\n"

        for i, child in enumerate(self.children):
            ret += child.__str__(level + 1, is_last=(i == len(self.children) - 1))

        return ret


class Parser:
    def __init__(self, tokens):
        self.tokens = tokens
        self.pos = 0

    def current_token(self):
        return self.tokens[self.pos] if self.pos < len(self.tokens) else None

    def match(self, expected_type):
        token = self.current_token()
        if token and token[0] == expected_type:
            self.pos += 1
            return token
        else:
            raise SyntaxError(f"Expected {expected_type}, found {token}")

    def parse_program(self):
        return ParseNode("Program", self.parse_statements())

    def parse_statements(self):
        statements = []
        while self.current_token():
            statements.append(self.parse_statement())
        return statements

    def parse_statement(self):
        token = self.current_token()
        if token[0] == "ID":
            return self.parse_assignment()
        elif token[0] == "KEYWORD" and token[1] == "if":
            return self.parse_if_statement()
        elif token[0] == "KEYWORD" and token[1] == "else":
            return self.parse_else_statement()
        elif token[0] == "KEYWORD" and token[1] == "print":
            return self.parse_print_statement()
        else:
            raise SyntaxError(f"Unexpected statement: {token}")

    def parse_assignment(self):
        id_token = self.match("ID")
        self.match("ASSIGN")
        expr = self.parse_expression()
        return ParseNode("Assignment", [ParseNode(id_token), expr])

    def parse_else_statement(self):
        self.match("KEYWORD")
        self.match("COLON")
        else_body = self.parse_statements()
        return ParseNode("Else", else_body)

    def parse_if_statement(self):
        self.match("KEYWORD")
        condition = self.parse_expression()
        self.match("COLON")
        body = self.parse_statements()

        else_node = None
        if self.current_token() and self.current_token()[0] == "KEYWORD" and self.current_token()[1] == "else":
            else_node = self.parse_else_statement()

        children = [
            condition,
            ParseNode("Body", body)
        ]

        if else_node:
            children.append(ParseNode("Else", [else_node]))  # Only add the Else node if it exists

        return ParseNode("IfStatement", children)
```

```python
    def parse_print_statement(self):
        self.match("KEYWORD")
        self.match("LPAREN")
        arguments = []

        while self.current_token() and self.current_token()[0] != "RPAREN":
            if self.current_token()[0] == "COMMA":
                self.match("COMMA")
            else:
                if self.current_token()[0] == "STRING":
                    arguments.append(ParseNode(self.match("STRING")))
                elif self.current_token()[0] == "ID":
                    arguments.append(ParseNode(self.match("ID")))
                else:
                    raise SyntaxError(f"Unexpected token in print statement: {self.current_token()}")

        self.match("RPAREN")
        return ParseNode("Print", arguments)

    def parse_expression(self):
        left = self.parse_term()
        while self.current_token() and self.current_token()[0] in {"PLUS", "MINUS"}:
            op = self.match(self.current_token()[0])
            right = self.parse_term()
            left = ParseNode("Expression", [left, ParseNode(op), right])

        while self.current_token() and self.current_token()[0] in {"LARGERTHAN", "LESSTHAN", "EQUALS", "NOTEQUALS"}:
            op = self.match(self.current_token()[0])
            right = self.parse_term()
            left = ParseNode("RelationalExpression", [left, ParseNode(op), right])

        return left

    def parse_term(self):
        left = self.parse_factor()
        while self.current_token() and self.current_token()[0] in {"MULT", "DIV"}:
            op = self.match(self.current_token()[0])
            right = self.parse_factor()
            left = ParseNode("Term", [left, ParseNode(op), right])
        return left

    def parse_factor(self):
        token = self.current_token()
        if token[0] == "NUMBER":
            return ParseNode(self.match("NUMBER"))
        elif token[0] == "ID":
            return ParseNode(self.match("ID"))
        elif token[0] == "STRING":
            return ParseNode(self.match("STRING"))
        elif token[0] == "LPAREN":
            self.match("LPAREN")
            expr = self.parse_expression()
            self.match("RPAREN")
            return expr
        elif token[0] == "LBRACKET":
            self.match("LBRACKET")
            elements = []
            while self.current_token() and self.current_token()[0] != "RBRACKET":
                if self.current_token()[0] == "COMMA":
                    self.match("COMMA")
                else:
                    elements.append(self.parse_expression())
            self.match("RBRACKET")
            return ParseNode("Array", elements)
        else:
            raise SyntaxError(f"Unexpected factor: {token}")
```

Produces a structured parse tree for the input tokens, suitable for further processing like interpretation or code generation.



How It Works

parse_program:

Starts parsing the entire program as a series of statements.

parse_statements:

Parses multiple statements (e.g., assignments, if blocks, print).

Specific Parsers:

parse_statement: Identifies and dispatches to the correct statement parser.

parse_assignment: Parses variable assignments (e.g., x = 5).

parse_if_statement: Parses if conditions, including optional else.

parse_print_statement: Parses print with arguments.

parse_expression: Handles arithmetic (+, -) and relational (<, ==) expressions.

parse_term / parse_factor: Parses more granular expressions, numbers, identifiers, and strings.

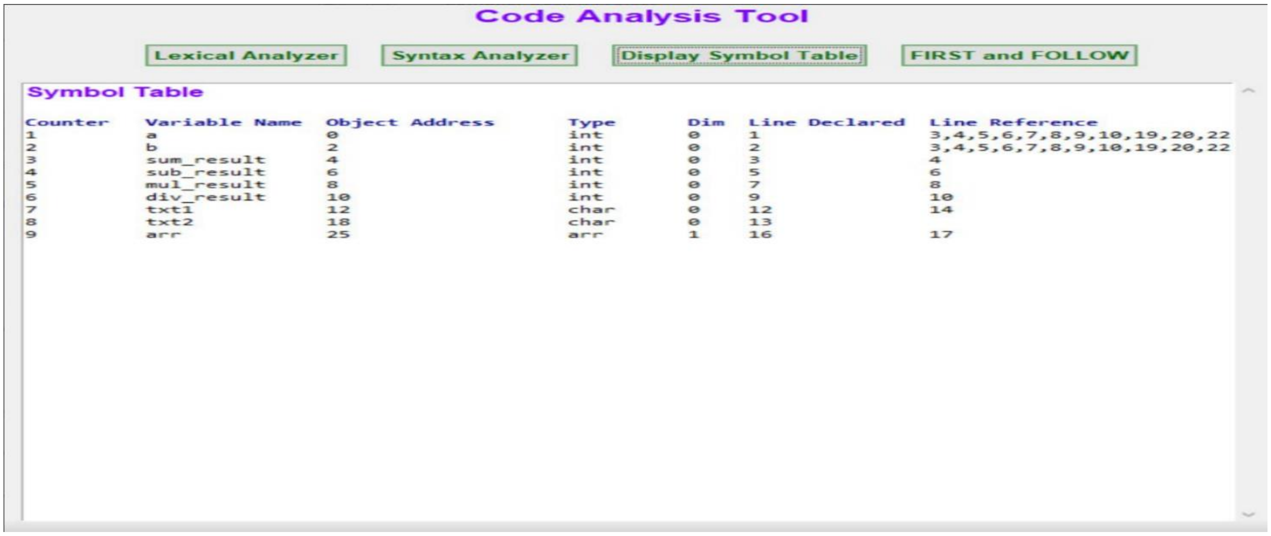
Error Handling:

Detects invalid tokens or mismatched structures (e.g., missing : or)).

symbol table generator : commonly used in compilers to store information about variables declared in the source code. It analyzes the code to extract metadata about variables, including their names, types, memory addresses, declaration, and references.

```python
import re

TYPE_SIZES = {
    'int': 2,
    'float': 4,
    'char': 1,
}

def generate_symbol_table(code):
    pattern = r"(\w+)\s*=\s*(.*)"
    lines = code.strip().split('\n')
    symbol_table = []
    current_address = 0

    def get_var_type_and_size(var_value):
        if re.match(r'^[\d\+\-\*/\(\) ]+$', var_value):
            return 'int', TYPE_SIZES['int']
        elif var_value.startswith('[') and var_value.endswith(']'):
            num_elements = len(re.findall(r'\d+', var_value))
            return 'arr', TYPE_SIZES['int'] * num_elements
        elif '"' in var_value:
            return 'char', TYPE_SIZES['char'] * len(var_value.replace('"', ''))
        else:
            return 'int', TYPE_SIZES['int']

    for line_num, line in enumerate(lines, 1):
        match = re.match(pattern, line.strip())
        if match:
            var_name, var_value = match.groups()
            var_type, var_size = get_var_type_and_size(var_value)
            symbol_table.append({
                'counter': len(symbol_table) + 1,
                'Variable Name': var_name,
                'Object Address': current_address,
                'Type': var_type,
                'Dim': 0 if var_type in ['int', 'float', 'char'] else 1,
                'Line Declared': line_num,
                'Line Reference': []
            })
            current_address += var_size

    for line_num, line in enumerate(lines, 1):
        for var in symbol_table:
            if var['Variable Name'] in line and var['Line Declared'] != line_num:
                if "print" in line.strip():
                    if var['Variable Name'] in re.findall(r'\w+', line):
                        var['Line Reference'].append(line_num)
                elif "[" not in line.strip():
                    var['Line Reference'].append(line_num)

    return symbol_table
```

Returns a list of dictionaries, where each dictionary represents a variable and its metadata.



 Input and Symbol Table Initialization:

Takes code (a multiline string with variable declarations).

Initializes symbol_table to store metadata about variables.

Starts current_address at 0 to calculate memory addresses.

 Type Detection (get_var_type_and_size):

Determines the type and size of variables based on their values:

Integer (int): Numeric expressions like 1 + 2.

Array (arr): Square-bracketed values like [1, 2, 3].

Character (char): Strings enclosed in quotes like "hello".

 Variable Parsing:

Uses a regex pattern (r"(\w+)\s*=\s*(.*)") to match variable assignments.

Extracts the variable name, value, type, size, and other metadata:

Dim: Dimensions (1 for arrays, 0 for scalars).

Line Declared: Line number where the variable is declared.

Updates current_address for each variable based on its size.

 Reference Tracking:

For each line, checks if declared variables are referenced later in the code.

Updates Line Reference with line numbers where the variable is used.

the text file we used :

Code.txt

File Edit View

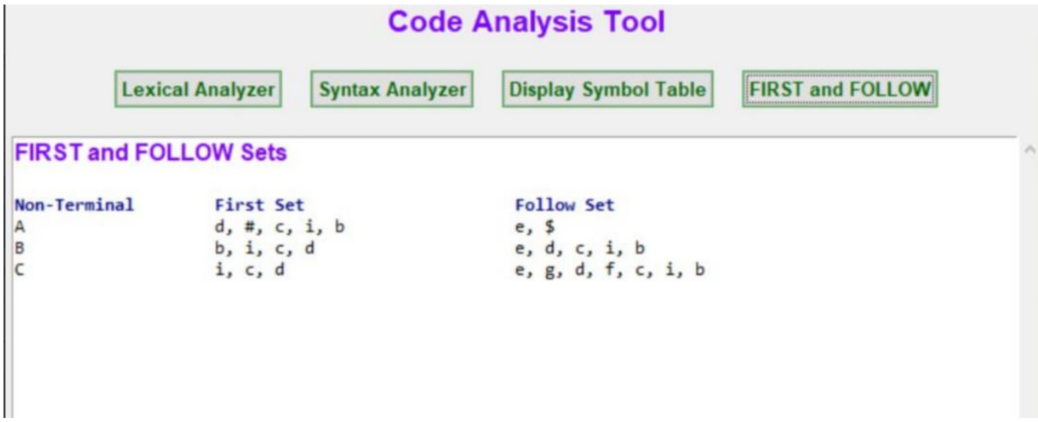
```
a = 10
b = 5
sum_result = a + b
print("Addition:", a, "+", b, "=", sum_result)
sub_result = a - b
print("Subtraction:", a, "-", b, "=", sub_result)
mul_result = a * b
print("Multiplication:", a, "*", b, "=", mul_result)
div_result = a / b
print("Division:", a, "/", b, "=", div_result)

txt1 = "i love"
txt2 = " python"
print(txt1)

arr = [1, 2, 3, 4, 5]
print(arr)

if a > b:
     print(a, "is greater than", b)
else:
     print(b, "is greater than or equal to", a)
```

first and follow :This code processes context-free grammar (CFG) rules to compute **FIRST** and **FOLLOW** sets while eliminating left recursion and performing left factoring. Here's a brief breakdown of its functionality:

```python
import re

start_symbol = ""
productions = {}
first_table = {}
follow_table = {}

def creatProduction(file_path):
    global start_symbol, productions
    with open(file_path, "r") as file:
        for production in file:
            lhs, rhs = re.split(r"->", production)
            rhs = re.split(r"\||\n", rhs)
            productions[lhs.strip()] = set(rhs) - {''}
            if not start_symbol:
                start_symbol = lhs.strip()

def isNonterminal(symbol):
    return symbol.isupper()

def eliminateLeftRecursion():
    global productions
    new_productions = {}
    for nt, rules in productions.items():
        alpha_rules = set()
        beta_rules = set()
        for rule in rules:
            if rule.startswith(nt):
                alpha_rules.add(rule[len(nt):])
            else:
                beta_rules.add(rule)

        if alpha_rules:
            nt_new = nt + "'"
            while nt_new in productions:
                nt_new += "'"
            new_productions[nt] = {beta + nt_new for beta in beta_rules}
            new_productions[nt_new] = {alpha + nt_new for alpha in alpha_rules} | {"#"}
        else:
            new_productions[nt] = rules

    productions = new_productions

def performLeftFactoring():
    global productions
    new_productions = {}
    for nt, rules in productions.items():
        prefix_map = {}
        for rule in rules:
            prefix = rule[0]
            if prefix not in prefix_map:
                prefix_map[prefix] = []
            prefix_map[prefix].append(rule)

        new_rules = set()
        for prefix, grouped_rules in prefix_map.items():
            if len(grouped_rules) > 1:
                nt_new = nt + "'"
                while nt_new in productions:
                    nt_new += "'"
                new_rules.add(prefix + nt_new)
                new_productions[nt_new] = {rule[1:] or "#" for rule in grouped_rules}
            else:
                new_rules.add(grouped_rules[0])
        new_productions[nt] = new_rules

    productions.update(new_productions)

def firstFunc(symbol):
    if symbol in first_table:
        return first_table[symbol]

    if isNonterminal(symbol):
        first = set()
        for production in productions[symbol]:
            if production:
                first_element = production[0]
                fst = firstFunc(first_element)
                first = first.union(fst)
        return first
    else:
        return set(symbol)
```

```python
def followFunc(symbol):
    if symbol not in follow_table:
        follow_table[symbol] = set()
    for nt in productions.keys():
        for rule in productions[nt]:
            pos = rule.find(symbol)
            if pos != -1:
                if pos == (len(rule) - 1):
                    if nt != symbol:
                        follow_table[symbol] = follow_table[symbol].union(followFunc(nt))
                else:
                    first_next = set()
                    for next in rule[pos + 1:]:
                        fst_next = firstFunc(next)
                        first_next = first_next.union(fst_next - {'#'})
                        if '#' not in fst_next:
                            break
                    if '#' in fst_next:
                        if nt != symbol:
                            follow_table[symbol] = follow_table[symbol].union(followFunc(nt))
                            follow_table[symbol] = follow_table[symbol].union(first_next) - {'#'}
                    else:
                        follow_table[symbol] = follow_table[symbol].union(first_next)
    return follow_table[symbol]

def compute_first_follow(file_path):
    global first_table, follow_table
    creatProduction(file_path)
    eliminateLeftRecursion()
    performLeftFactoring()

    for nt in productions:
        first_table[nt] = firstFunc(nt)
    follow_table[start_symbol] = set('$')
    for nt in productions:
        follow_table[nt] = followFunc(nt)


    result = {}
    for nt in productions:
        result[nt] = (list(first_table[nt]), list(follow_table[nt]))
    return result

```

Produces a dictionary with non-terminals as keys, and their respective **FIRST** and **FOLLOW** sets as values.

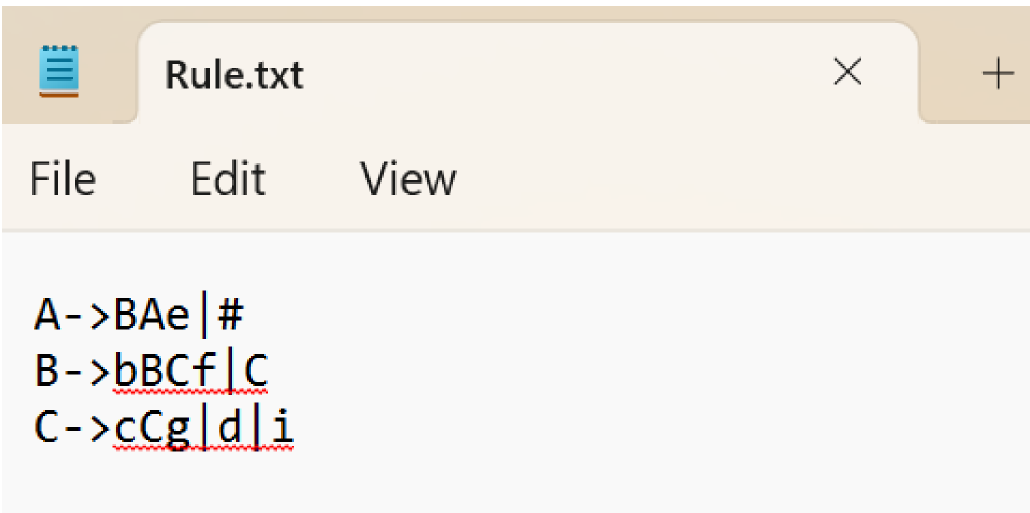


Key Features:

1. **Input**: Reads CFG from a file where each line defines a production, e.g., S -> Aa | b.
2. **Elimination of Left Recursion**:
   - Identifies rules with left recursion (e.g., A -> Aa | b).
   - Rewrites them into non-left-recursive form using auxiliary non-terminals.
3. **Left Factoring**:
   - Detects common prefixes in production rules (e.g., A -> ab | ac).
   - Refactors them to remove ambiguity (e.g., A -> aA', A' -> b | c).
4. **FIRST Function**:
   - Computes the set of terminal symbols that can appear at the start of strings derived from a given symbol.
5. **FOLLOW Function**:

- Computes the set of terminal symbols that can appear immediately after a given non-terminal in any derivation.
- Starts with FOLLOW(start_symbol) = {'$'} (end-of-input marker).

the rule we used is :



The Mian code :

```python
def analyze_lexical():
    with open("Code.txt", "r") as file:
        code = file.read()

    result_text.delete(1.0, tk.END)
    result_text.insert(tk.END, "Lexical Analysis (Scanner)\n\n", "header")
    tokens = tokenize(code)
    result_text.insert(tk.END, "Tokens:\n", "subheader")
    result_text.insert(tk.END, "\n\n")
    result_text.insert(tk.END, str(tokens) + "\n")

    result_text.tag_configure("header", foreground="#7F00FF", font=("Arial", 12, "bold"))
    result_text.tag_configure("subheader", foreground="darkgreen", font=("Arial", 11, "italic"))

def analyze_parser():
    with open("Code.txt", "r") as file:
        code = file.read()

    result_text.delete(1.0, tk.END)
    result_text.insert(tk.END, "Syntax Analysis (Parser)\n\n", "header")
    tokens = tokenize(code)
    parser = Parser(tokens)
    parse_tree = parser.parse_program()
    result_text.insert(tk.END, "    Parse Tree:\n", "subheader")
    result_text.insert(tk.END, "\n\n")
    result_text.insert(tk.END, str(parse_tree) + "\n", "indented")

    result_text.tag_configure("header", foreground="#7F00FF", font=("Arial", 12, "bold"))
    result_text.tag_configure("subheader", foreground="darkgreen", font=("Arial", 11, "italic"))
    result_text.tag_configure("indented", lmargin1=20, font=("Consolas", 10))

def display_symbol_table():
    with open("Code.txt", "r") as file:
        code = file.read()

    symbol_table = generate_symbol_table(code)

    result_text.delete(1.0, tk.END)
    result_text.insert(tk.END, "Symbol Table\n\n", "header")
    result_text.insert(tk.END, f"{'Counter':<10}{'Variable Name':<15}{'Object Address':<20}{'Type':<10}{'Dim':<5}{'Line Declared':<15}{'Line Reference'}\n", "table_header")

    for entry in symbol_table:
        line = f"{entry['counter']:<10}{entry['Variable Name']:<15}{entry['Object Address']:<20}{entry['Type']:<10}{entry['Dim']:<5}{entry['Line Declared']:<15}{','.join(map(str, entry['Line Reference']))
        result_text.insert(tk.END, line)

    result_text.tag_configure("header", foreground="#7F00FF", font=("Arial", 12, "bold"))
    result_text.tag_configure("table_header", foreground="darkblue", font=("Consolas", 10, "bold"))

def display_first_follow():
    result_text.delete(1.0, tk.END)
    result_text.insert(tk.END, "FIRST and FOLLOW Sets\n\n", "header")

    first_follow = compute_first_follow('Rule.txt')

    result_text.insert(tk.END, f"{'Non-Terminal':<20}{'First Set':<30}{'Follow Set'}\n", "table_header")
    for nt, (first_set, follow_set) in first_follow.items():
        line = f"{nt:<20}{', '.join(first_set):<30}{', '.join(follow_set)}\n"
        result_text.insert(tk.END, line)

    result_text.tag_configure("header", foreground="#7F00FF", font=("Arial", 12, "bold"))
    result_text.tag_configure("table_header", foreground="darkblue", font=("Consolas", 10, "bold"))

root = tk.Tk()
root.title("Compiler Design")
root.geometry("900x700")

frame = ttk.Frame(root, padding=(90, 0, 0, 0))
frame.grid(row=0, column=0, sticky=(tk.W, tk.E, tk.N, tk.S))

header_label = ttk.Label(frame, text="Code Analysis Tool", font=("Arial", 16, "bold"), foreground="#7F00FF")
header_label.grid(row=0, column=1, pady=10)

style = ttk.Style()
style.configure(
    "Custom.TButton",
    background="#4CAF50",
    foreground="darkgreen",
    height=3,
    font=("Arial", 10, "bold")
)

button_frame = ttk.Frame(frame)
button_frame.grid(row=1, column=0, columnspan=3, pady=10, padx=10)

lexical_button = ttk.Button(button_frame, text="Lexical Analyzer", command=analyze_lexical, style="Custom.TButton")
lexical_button.grid(row=0, column=0, padx=10)

parser_button = ttk.Button(button_frame, text="Syntax Analyzer", command=analyze_parser, style="Custom.TButton")
parser_button.grid(row=0, column=1, padx=10)

symbol_table_button = ttk.Button(button_frame, text="Display Symbol Table", command=display_symbol_table, style="Custom.TButton")
symbol_table_button.grid(row=0, column=2, padx=10)

first_follow_button = ttk.Button(button_frame, text="FIRST and FOLLOW", command=display_first_follow, style="Custom.TButton")
first_follow_button.grid(row=0, column=3, padx=10)

result_text = scrolledtext.ScrolledText(frame, wrap=tk.WORD, width=100, height=35, font=("Consolas", 10))
result_text.grid(row=2, column=0, columnspan=3, pady=10)

root.mainloop()
```

Team members :

Hanan Amer Abo Abdo ID:804621462

Rahma Mohamed Abo Shaheen ID:804611059

Arwa Hassan Abo ganba ID:804610150