AFLL Project - Implementation of Constructs using PLY Library

Objective

The primary objective of this assignment is to perform syntax validation of Python programming constructs by implementing their context-free grammar and tokenization using PLY tools (Python Lex-Yacc).

Assignment Context and Guidelines

- Tools Used: PLY (Python implementation of Lex-Yacc for parsing/tokenizing) and Tkinter(GUI)
- Requirements and Features:
    - Python installed and PLY library configured in IDE virtual environment (download from website and use powershell to run setup.py in a venv)
    - GUI Based output using tkinter-python
    - Syntax error and invalid character handling

Explanation of Code Design

1. `lexer.py` (Lexical Analyzer)

- Purpose: Scans input and breaks it into meaningful tokens, including IDs, numbers, strings, operators, and Python keywords (if, else, def).

2. `parser.py` (Grammar Parser)

- Purpose: Defines the grammar rules for valid Python statements and expressions.
- Method: Grammar established for constructs—
  Variable declaration (`ID = value`), If-else condition , Function definitions, Lists declaration and Dictionary assignment

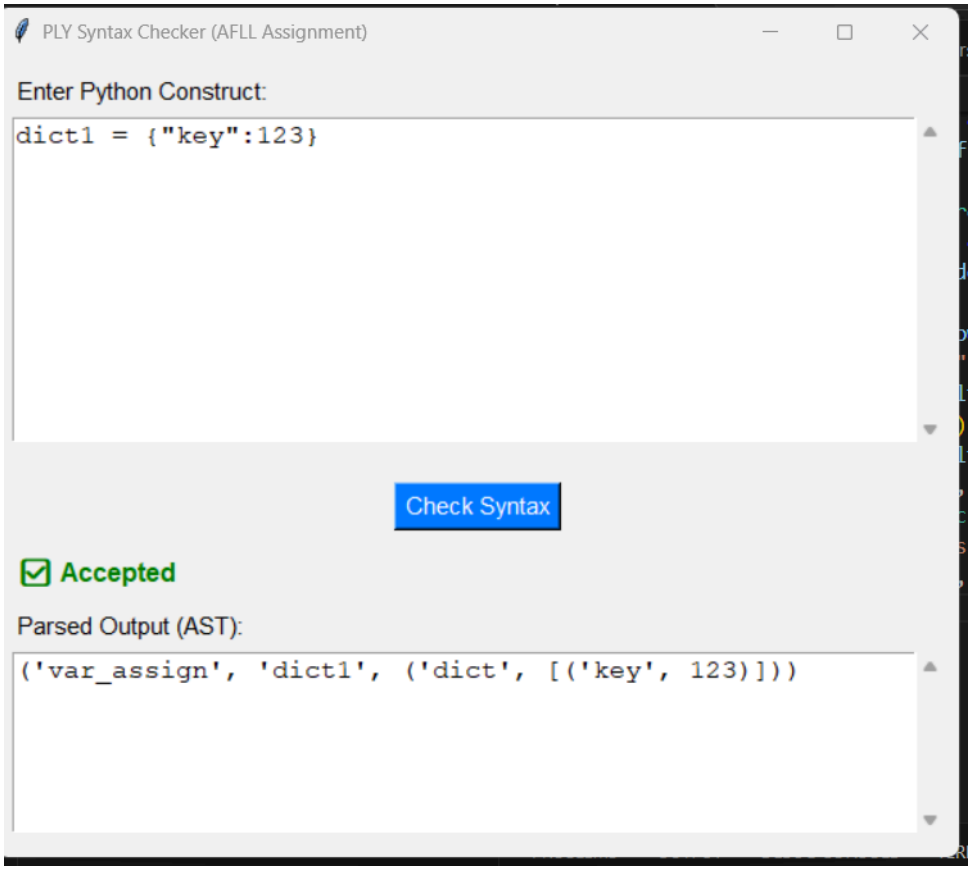
3. `main.py` (Interactive Syntax Checker)

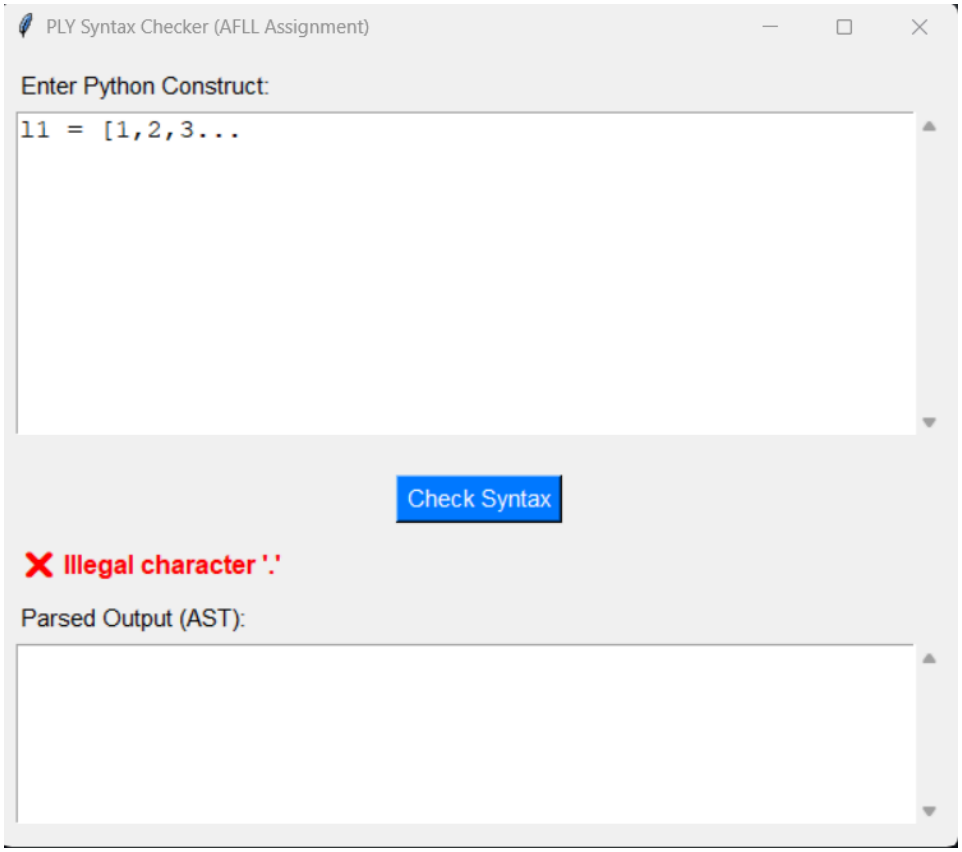
- Purpose: Provides a user interface via Tkinter for input and output checking.
- Features: GUI components styled for usability and clarity, stateful output handling.

Walkthrough and Output

- When the user submits Python code snippets via the GUI, the system validates syntax using the grammar defined.
- Correct input results in "Accepted" with a parsed representation (shown below); incorrect input triggers specific syntax error details.
- Examples:
    - Input: `x = [3,4,5]` → Output: AST tuple for list assignment.
    - Input: `def f(x, y): x + y` → Output: tuple for function definition.
    - Input: `x= [1,2` → Output: Syntax error at the end of input

Working Screenshots


PLY Syntax Checker (AFLL Assignment)
Enter Python Construct:
dict1 = {"key":123}
Check Syntax
Accepted
Parsed Output (AST):
('var_assign', 'dict1', ('dict', [('key', 123)]))



Code

1) lexer.py

```
import ply.lex as lex
tokens = (
    'ID',
    'NUMBER',
    'STRING',
    'IF',
    'ELSE',
    'DEF',
    'EQUALS',      # =
    'LPAREN',      # (
    'RPAREN',      # )
    'LBRACKET',    # [
    'RBRACKET',    # ]
    'LBRACE',      # {
    'RBRACE',      # }
    'COLON',       # :
```

```
    'COMMA',       # ,
    'PLUS',        # +
    'MINUS',       # -
    'TIMES',       # *
    'DIVIDE',      # /
    'GT',          # >
    'LT',          # <
    'EQEQ',        # ==
)
reserved = {
    'if': 'IF',
    'else': 'ELSE',
    'def': 'DEF',
}
t_EQUALS = r'='
t_LPAREN = r'\('
t_RPAREN = r'\)'
t_LBRACKET = r'\['
t_RBRACKET = r'\]'
t_LBRACE = r'\{'
t_RBRACE = r'\}'
t_COLON = r':'
t_COMMA = r','
t_PLUS = r'\+'
t_MINUS = r'-'
t_TIMES = r'\*'
t_DIVIDE = r'/'
t_GT = r'>'
t_LT = r'<'
t_EQEQ = r'=='
def t_ID(t):
    r'[a-zA-Z_][a-zA-Z_0-9]*'
    t.type = reserved.get(t.value, 'ID')
    return t
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t
def t_STRING(t):
    r'(\"[^\"]*\")|(\'[^\']*\')'
```

```
    t.value = t.value[1:-1]
    return t
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)
t_ignore = ' \t'
def t_error(t):
    message = f"Illegal character '{t.value[0]}'"
    raise SyntaxError(message)
lexer = lex.lex()
```

2) parser.py

```
# parser.py
import ply.yacc as yacc
from lexer import tokens
start = 'statement'
def p_statement(p):
    '''statement : variable_declaration
                 | if_else_statement
                 | function_definition
                 | expression
                 | list
                 | dictionary
    '''
    p[0] = p[1]
def p_variable_declaration(p):
    'variable_declaration : ID EQUALS value'
    p[0] = ('var_assign', p[1], p[3])
def p_value(p):
    '''value : expression
             | STRING
             | list
             | dictionary
             | NUMBER
             | ID
    '''
    p[0] = p[1]
def p_if_else_statement(p):
    '''if_else_statement : IF condition COLON statement ELSE COLON
statement
```

```
                        | IF condition COLON statement
    '''
    if len(p) == 8:
        p[0] = ('if-else', p[2], p[4], p[7])
    else:
        p[0] = ('if', p[2], p[4])
def p_condition(p):
    '''condition : expression GT expression
                 | expression LT expression
                 | expression EQEQ expression
                 | expression
    '''
    if len(p) == 4:
        p[0] = (p[2], p[1], p[3])
    else:
        p[0] = p[1]
def p_function_definition(p):
    'function_definition : DEF ID LPAREN arg_list RPAREN COLON
statement'
    p[0] = ('func_def', p[2], p[4], p[7])
def p_arg_list(p):
    '''arg_list : ID
                | ID COMMA arg_list
                | empty
    '''
    if len(p) == 2:
        p[0] = [p[1]]
    elif len(p) == 4:
        p[0] = [p[1]] + p[3]
    else:
        p[0] = []
def p_list(p):
    'list : LBRACKET item_list RBRACKET'
    p[0] = ('list', p[2])

def p_item_list(p):
    '''item_list : value
                 | value COMMA item_list
                 | empty
    '''
```

```
    if len(p) == 2:
        p[0] = [p[1]]
    elif len(p) == 4:
        p[0] = [p[1]] + p[3]
    else:
        p[0] = []
def p_dictionary(p):
    'dictionary : LBRACE pair_list RBRACE'
    p[0] = ('dict', p[2])

def p_pair_list(p):
    '''pair_list : pair
                 | pair COMMA pair_list
                 | empty
    '''
    if len(p) == 2:
        p[0] = [p[1]]
    elif len(p) == 4:
        p[0] = [p[1]] + p[3]
    else:
        p[0] = []
def p_pair(p):
    '''pair : STRING COLON value
            | NUMBER COLON value
    '''
    p[0] = (p[1], p[3])
def p_expression(p):
    '''expression : term PLUS term
                  | term MINUS term
                  | term
    '''
    if len(p) == 4:
        p[0] = (p[2], p[1], p[3])
    else:
        p[0] = p[1]
def p_term(p):
    '''term : factor TIMES factor
            | factor DIVIDE factor
            | factor
    '''
```

```
    if len(p) == 4:
        p[0] = (p[2], p[1], p[3])
    else:
        p[0] = p[1]
def p_factor(p):
    '''factor : NUMBER
              | ID
              | LPAREN expression RPAREN
    '''
    if len(p) == 4:
        p[0] = p[2]
    else:
        p[0] = p[1]
def p_empty(p):
    'empty :'
    pass
def p_error(p):
    if p:
        message = f"Syntax error at token '{p.value}' (type:
{p.type})"
    else:
        message = "Syntax error at end of input"
    raise SyntaxError(message)
parser = yacc.yacc()
```

3) main.py
4) main.py

```
import tkinter as tk
from tkinter import scrolledtext, font
from lexer import lexer
from parser import parser
def check_syntax():
    data = input_text.get("1.0", tk.END)
    result_label.config(text="")
    output_text.config(state="normal")
    output_text.delete("1.0", tk.END)
    output_text.config(state="disabled")
    if not data.strip():
        result_label.config(text="Please enter some code to check.",
fg="#555")
```

```
        return
    try:
        parsed = parser.parse(data, lexer=lexer)
        if parsed is not None:
            result_label.config(text="✅ Accepted", fg="green")
            output_text.config(state="normal")
            output_text.insert(tk.END, str(parsed))
            output_text.config(state="disabled")
        else:
            result_label.config(text="No valid construct found.",
fg="#555")

    except SyntaxError as e:
        result_label.config(text=f"❌ {e}", fg="red")
    except Exception as e:
        result_label.config(text=f"An unexpected error occurred:
{e}", fg="orange")
window = tk.Tk()
window.title("PLY Syntax Checker (AFLL Assignment)")
window.geometry("600x500")
main_font = font.Font(family="Arial", size=11)
code_font = font.Font(family="Courier New", size=12)
result_font = font.Font(family="Arial", size=12, weight="bold")
text_frame = tk.Frame(window, padx=10, pady=10)
text_frame.pack(fill="both", expand=True)
input_label = tk.Label(text_frame, text="Enter Python Construct:",
font=main_font)
input_label.pack(anchor="w")
input_text = scrolledtext.ScrolledText(text_frame, wrap=tk.WORD,
font=code_font, height=10)
input_text.pack(fill="both", expand=True, pady=(5, 10))
check_button = tk.Button(window, text="Check Syntax",
font=main_font, command=check_syntax, bg="#007bff", fg="white")
check_button.pack(pady=5)
result_frame = tk.Frame(window, padx=10, pady=5)
result_frame.pack(fill="both", expand=True)
result_label = tk.Label(result_frame, text="", font=result_font,
pady=5)
result_label.pack(anchor="w")
```

```
output_label = tk.Label(result_frame, text="Parsed Output (AST):",
font=main_font)
output_label.pack(anchor="w", pady=(5,0))
output_text = scrolledtext.ScrolledText(result_frame, wrap=tk.WORD,
font=code_font, height=5)
output_text.config(state="disabled")
output_text.pack(fill="both", expand=True, pady=(5, 10))
window.mainloop()
```

References

- Python Lex-Yacc Tutorials and Documentation.
- AFLL Assignment Description and PLY Features sent via email.
- Official PLY documentation.

Thank you.