Lexical Analyzer and Parser for TinyPie

Overview

This project implements a Lexical Analyzer and Parser for a simplified programming language, TinyPie, using Python.

It features a graphical user interface (GUI) built with Tkinter that allows users to enter source code, process each line, view tokens, and display basic parse trees.

Objectives

- Tokenize source code into lexemes (keywords, operators, separators, literals, identifiers).
- Validate syntax and build parse trees from the tokens.
- Provide an interactive GUI to support line-by-line lexical and syntax analysis.

Technologies Used

- Python 3
- Tkinter - for the GUI
- Regular Expressions (re) - for tokenizing source code

Features

1. GUI Functionality

The application window is divided into three primary sections:

- Source Code Input: Text area for users to enter TinyPie code.
- Tokens Output: Displays the lexical tokens extracted from the current line.
- Parse Tree Output: Shows a basic tree-structured interpretation of each line's syntax.

2. Lexical Analysis

A custom class LexicalAnalyzer uses regular expressions to identify and classify:

- Keywords: float, int, if, else

- Operators: =, +, >, *

- Separators: (,), :, ;, "

- Literals: floating-point and integer values

- Identifiers: valid variable names

Each line is split into tokens, validated, and displayed in the GUI.

3. Syntax Parsing

A minimal parser (TinyPieParser) generates a rough parse tree:

- Validates the line starts with a keyword.

- Prints the structure in the form of a tree outline.

- The parser contains placeholders for extending logic to full syntax trees (inspired by the "Let's Build a Simple Interpreter" series).

Example Input

```
float mathresult1 = 5*4.3 + 2.1;
float mathresult2 = 4.1 + 2*5.5;
if (mathresult1 > mathresult2):
print("I just built some parse trees");
```

Example Output:

```
Tokens for Line 1:
<key,float>
<id,mathresult1>
<op,=>
```

<lit_int,5>

<op,*>

<lit_float,4.3>

<op,+>

<lit_float,2.1>

<sep,;>

Parse Tree:

Building Tree:

key, float

id, mathresult1

op, =

lit_int, 5

op, *

lit_float, 4.3

op, +

lit_float, 2.1

sep, ;

Limitations & Future Work

- Current parser only builds a flat tree of token types and values.

- Syntax checking is minimal; deeper grammar validation is pending.

- The print statement is not yet parsed fully; additional rules are planned.

- Future enhancements could include:

- Full abstract syntax tree (AST) construction

- Semantic analysis (type checking, variable declaration)

- Code generation or execution simulation

- Saving/loading scripts

Conclusion

This project offers a foundational tool for understanding compiler design components, particularly lexical analysis and basic parsing. The GUI enhances usability, making it easier for beginners to visualize how source code is tokenized and parsed.