A Parser

Author(s): Hareena Chowdary Polavaram
Date: 25 Nov. 23

A PARSER

Phase 1: INTRODUCTION

- The provided Python code implements a parser for the C- programming language using the PLY (Python Lex-Yacc) library. C- is a minimalistic language that serves as a subset of the C programming language. This parser is designed to perform lexical and syntax analysis on C- source code, producing an Abstract Syntax Tree (AST) as its output.
- The primary goal of this parser is to recognize the structure of valid C- programs, identifying tokens such as identifiers, constants, operators, and keywords, and then organizing them into a hierarchical tree structure that represents the syntactic structure of the input code. The resulting AST can be further utilized for various stages in the compilation process.

Phase 2: SPECIFICATION

To implement a Parser analyzer for a self-designed Cminus programming language (a tiny simplified subset of the C language) using Python.

Objectives that we should follow in order to write a code:

2.1: Language Specification:

- Cminus is a case-sensitive language using ASCII characters.
- It supports int and float data types.
- Supports arithmetic, logical, relational, and assignment operators.
- Control flow statements include if-else, while, and compound statements.
- Supports single-line and multi-line comments (/* ... */).
- Identifiers start with a letter and can contain alphanumeric characters.
- Literal strings are enclosed in single quotes.
- Keywords (reserved words): int, float, if, else, exit, while, read, write, return.

2.2: Lexical Analyzer Specification:

- Read source code line by line and store it in a buffer.
- Scan the buffer to identify tokens based on language specification.
- Create a token object with type and value for each token.
- Append the token object to a list of tokens.
- Continue scanning until the end of the buffer or an error is encountered.
- Return the list of tokens as output or display an error message for encountered errors.

2.3: Parser (Syntax Analysis):

- Explain the role of the parser in the compilation process.
- Provide an overview of the grammar rules using BNF-like notation.
- Discuss the structure of the Abstract Syntax Tree (AST).
- Include a sample output of the parser for a given input.

Phase 3: DESIGN

1. Lexical Analysis:

- The lexer utilizes the PLY library to define token names and corresponding regular expression rules. It handles reserved keywords, comments, and tracks line numbers.
- The `test_lexer` function demonstrates the lexer's ability to tokenize input code and prints the resulting tokens.

2. Parser Syntax Analysis:

- The parser uses PLY's yacc module to define grammar rules for the C-language. It employs a bottom-up parsing approach to generate an AST.
- The grammar rules cover program structure, procedures, declarations, statements, expressions, and more.
- The `print_tree` function is responsible for visualizing the generated AST.

3. Error Handling:

- The parser includes error-handling rules to detect and report syntax errors during both lexical and syntax analysis.
- The `p_error` function is invoked when a syntax error is encountered, providing information about the location of the error.

4. Testing and Example:

- The provided code includes a file input mechanism to test the parser with sample C- code (loaded from a file).
- The `test_lexer` function demonstrates the lexical analysis output, and the `print_tree` function displays the resulting AST.

5. Future Work:

- Future enhancements could involve extending the parser to handle more features of the C- language or improving error recovery mechanisms.
- Consider incorporating additional optimization phases or generating intermediate code from the AST.

Phase 4: RISK ANALYSIS

The code is a Parser Analyzer. No specific risks were identified in this project. However, potential issues might include:

1.Testing Coverage:

The code includes a basic test of the lexer, but it may lack comprehensive testing to cover various input scenarios and edge cases.

2.Portability:

The code is written for a specific use case and input file ('cminusinput.txt'). It may not be easily portable to different projects without modification.

3.Dependency on PLY:

The code relies on the PLY library. If PLY's support or compatibility changes, it could affect the code's functionality.

4.Regular Expression Complexity:

The regular expressions used to define token patterns are relatively complex. There's a risk of regex errors or inefficiencies. While the provided regular expressions seem to be valid, complex regular expressions can be challenging to debug and maintain.

Phase 5: VERIFICATION

All the functions in the code are tested with multiple expressions and satisfied the required output.

Phase 6: CODING

The code of this program is included in the zip file.

Phase 7: TESTING

7.1 Input & Output Specifications:

1.cminusinput1.c (input):

```
/* This is a multi-line comment */
int max() {
   int a, b;
   read(a);
   read(b);
   if (a < b) {
       return b;
   } else {
       return a;
   }
}
```

Output:

```
### Lex Output:
('INT', 'int')
('IDENTIFIER', 'max')
('LP', '(')
('RP', ')')
('LBR', '{')
('INT', 'int')
('IDENTIFIER', 'a')
('CM', ',')
('IDENTIFIER', 'b')
('SC', ';')
('READ', 'read')
('LP', '(')
('IDENTIFIER', 'a')
('RP', ')')
('SC', ';')
('READ', 'read')
('LP', '(')
('IDENTIFIER', 'b')
('RP', ')')
('SC', ';')
('IF', 'if')
('LP', '(')
('IDENTIFIER', 'a')
('LT', '<')
('IDENTIFIER', 'b')
('RP', ')')
('LBR', '{')
('RETURN', 'return')
('IDENTIFIER', 'b')
('SC', ';')
('RBR', '}')
('ELSE', 'else')
('LBR', '{')
('RETURN', 'return')
('IDENTIFIER', 'a')
('SC', ';')
('RBR', '}')
('RBR', '}')
### YACC Output-AST:
Program :
  Procedures :
    ProcedureDecl :
      ProcedureHead :
        FunctionDecl :
          Type :
            int
          max
        DeclList :
          Type :
            int
          IdentifierList :
            IdentifierList :
              VarDecl :
                a
            VarDecl :
              b
      ProcedureBody :
        StatementList :
          StatementList :
            StatementList :
              Statement :
                IOStatement :
                  READ
                  Variable :
                    a
            Statement :
              IOStatement :
```

2.cminusinput2.c(input):

```
int max () {
int a,b ;
read(a); read(b);
if (a < b) {
return b;
}
else {
return a;
}
}
```

OUTPUT:

```
hareena@Hareenas-Air pa4-parser-Hareena-Polavaram % /usr/local/bin/python3 /Users/hareena/Documents/CS5950/pa4-parser-Hareena-Polavaram/cminus_parser.py
### Lex Output:
('INT', 'int')
('IDENTIFIER', 'max')
('LP', '(')
('RP', ')')
('LBR', '{')
('INT', 'int')
('IDENTIFIER', 'a')
('CM', ',')
('IDENTIFIER', 'b')
('SC', ';')
('READ', 'read')
('LP', '(')
('IDENTIFIER', 'a')
('RP', ')')
('SC', ';')
('READ', 'read')
('LP', '(')
('IDENTIFIER', 'b')
('RP', ')')
('SC', ';')
('IF', 'if')
('LP', '(')
('IDENTIFIER', 'a')
('LT', '<')
('IDENTIFIER', 'b')
('RP', ')')
('LBR', '{')
('RETURN', 'return')
('IDENTIFIER', 'b')
('SC', ';')
('RBR', '}')
('ELSE', 'else')
('LBR', '{')
('RETURN', 'return')
('IDENTIFIER', 'a')
('SC', ';')
('RBR', '}')
('RBR', '}')
### YACC Output-AST:
Program :
  Procedures :
    ProcedureDecl :
      ProcedureHead :
        FunctionDecl :
          Type :
            int
          max
        DeclList :
          Type :
            int
          IdentifierList :
            IdentifierList :
              VarDecl :
                a
            VarDecl :
              b
      ProcedureBody :
        StatementList :
          StatementList :
            StatementList :
              Statement :
                IOStatement :
                  READ
                Variable :
                  a
            Statement :
              IOStatement :
```

```
hareena@Hareenas-Air pa4-parser-Hareena-Polavaram % /usr/local/bin/python3 /Users/hareena/Documents/CS5950/pa4-parser-Hareena-Polavaram/cminus_parser.py
### Lex Output:
('INT', 'int')
('IDENTIFIER', 'max')
('LP', '(')
('RP', ')')
('LBR', '{')
('INT', 'int')
('IDENTIFIER', 'a')
('CM', ',')
('IDENTIFIER', 'b')
('SC', ';')
('READ', 'read')
('LP', '(')
('IDENTIFIER', 'a')
('RP', ')')
('SC', ';')
('READ', 'read')
('LP', '(')
('IDENTIFIER', 'b')
('RP', ')')
('SC', ';')
('IF', 'if')
('LP', '(')
('IDENTIFIER', 'a')
('LT', '<')
('IDENTIFIER', 'b')
('RP', ')')
('LBR', '{')
('RETURN', 'return')
('IDENTIFIER', 'b')
('SC', ';')
('RBR', '}')
('ELSE', 'else')
('LBR', '{')
('RETURN', 'return')
('IDENTIFIER', 'a')
('SC', ';')
('RBR', '}')
('RBR', '}')
### YACC Output-AST:
Program :
  Procedures :
    ProcedureDecl :
      ProcedureHead :
        FunctionDecl :
          Type :
            int
          max
        DeclList :
          Type :
            int
          IdentifierList :
            IdentifierList :
              VarDecl :
                a
            VarDecl :
              b
      ProcedureBody :
        StatementList :
          StatementList :
            StatementList :
              Statement :
                IOStatement :
                  READ
                Variable :
                  a
            Statement :
              IOStatement :
```

3.cminusinput3(input):

```
int max () {

int a,b,c,d ;
```

```
read(a); read(b); read(c); read(d);
}
```

OUTPUT:

Phase 8: REFINING THE PROGRAM

We can make few improvements for better code structure, efficiency, and usability. Here are some refinements:

1.Error Handling:

The error handling mechanism in the code is limited to printing an error message and skipping the character. This is not very robust. More sophisticated error handling, such as logging errors and providing better feedback to the user, would be preferable.

2.Performance and Efficiency:

Complex regular expressions, especially in large input files, can impact performance. Depending on the use case, performance may need to be optimized.

Phase 9: MAINTAINANCE

This application can be futher improved.