CS 3513 - Programming Languages: Project Report

June 7, 2025

1 Introduction

This report documents the implementation of a lexical analyzer, parser, and CSE machine for the RPAL programming language as part of the CS 3513 Programming Languages course. The project involves reading an RPAL program from an input file, constructing an Abstract Syntax Tree (AST), standardizing it into a Standardized Tree (ST), and executing it using a Control-Stack-Environment (CSE) machine. The implementation is written in Python and supports the `-ast` switch to output the AST. This report includes the program structure, function prototypes, and implementation details.

2 Basic Information

- **Name**: Chamindu Sathsara
- **Student ID**: 220585R
- **Name**: Rasindu Dulshan
- **Student ID**: 220589H

3 Program Structure

The project is organized into several Python files, each handling a specific component of the RPAL interpreter. Below is the structure of the program:

- **main.py**: Entry point of the program. Handles command-line arguments, reads the input file, and initiates parsing.
- **lexer.py**: Contains the `Token` class and tokenization logic (partially implemented in `Parser.get_token`).
- `tree.py`: Defines the `Tree` class for constructing and manipulating the AST and ST.

- `parser.py`: Implements the `Parser` class, which includes lexical analysis, parsing, AST construction, standardization, and CSE machine execution.
- `cse_machine.py`: Defines the `Environment` class for managing variable bindings in the CSE machine.
- `Makefile`: Automates the execution of the program with the appropriate command-line arguments.

The program flow is as follows:

1. `main.py` reads the input file and command-line arguments (`-ast`, `-st`, or `-lex`).
2. The `Parser` class tokenizes the input, builds the AST using a recursive descent parser, and standardizes it into an ST.
3. The ST is converted into control structures, which are executed by the CSE machine to produce the final output.
4. If the `-ast` switch is provided, the AST is printed; otherwise, the program output is displayed.

4 Function Prototypes

Below are the key function prototypes from the main classes, illustrating the structure of the program:

4.1 Token Class (`lexer.py`)

```
class Token:
    def __init__(self, token_type: str = "", token_value: str = ""):
    def set_type(self, token_type: str):
    def set_value(self, token_value: str):
    def get_type(self) -> str:
    def get_value(self) -> str:
```

4.2 Tree Class (`tree.py`)

```
class Tree:
    def __init__(self, value: str = "", node_type: str = ""):
    def set_type(self, node_type: str):
    def set_value(self, value: str):
    def get_type(self) -> str:
    def get_value(self) -> str:
    @staticmethod
    def build_node(value: str, node_type: str) -> 'Tree':
    @staticmethod
    def build_node_from_existing(x: 'Tree') -> 'Tree':
    def display_syntax_tree(self, indentation_level: int = 0):
```

4.3 Parser Class (`parser.py`)

```
class Parser:
   def __init__(self, read_array: str, row: int, size: int, af: int)
      :
   def is_reserved_key(self, string: str) -> bool:
   def is_operator(self, ch: str) -> bool:
   def is_alpha(self, ch: str) -> bool:
   def is_digit(self, ch: str) -> bool:
   def is_binary_operator(self, op: str) -> bool:
   def is_number(self, s: str) -> bool:
   def read(self, val: str, type: str):
   def buildTree(self, val: str, type: str, child: int):
   def get_token(self) -> Token:
   def parse(self):
   def cse_parse(self):
   def isParsingComplete(self) -> bool:
   def MST(self, t: Tree):
   def makeStandardTree(self, t: Tree):
   def build_control_structures(self, x: Tree, controlNodeArray):
   def build_node(self, x: Tree, value: str, node_type: str):
   def cse_machine(self, control_struct):
   def procedure_E(self):
   def procedure_Ew(self):
   def procedure_T(self):
   def procedure_Ta(self):
   def procedure_Tc(self):
   def procedure_B(self):
   def procedure_Bt(self):
   def procedure_Bs(self):
   def procedure_Bp(self):
   def procedure_A(self):
   def procedure_At(self):
   def procedure_Af(self):
   def procedure_Ap(self):
   def procedure_R(self):
   def procedure_Rn(self):
   def procedure_D(self):
   def procedure_Da(self):
   def procedure_Dr(self):
   def procedure_Db(self):
   def procedure_Vb(self):
   def procedure_Vl(self):
```

4.4 Environment Class (`cse_machine.py`)

```
class Environment:
   def __init__(self, name: str = "", prev: 'Environment' = None):
```

4.5 Main Function (`main.py`)

```
1 def main():
```

5 Implementation Details

The lexical analyzer is implemented within the `Parser.get_token` method, which processes the input character stream and generates tokens (`INT`, `ID`, `KEYWORD`, `OPERATOR`, `PUNCTUATION`, `STR`, `COMMENT`). The parser uses a recursive descent approach, with procedures (`procedure_E`, `procedure_T`, etc.) corresponding to the RPAL grammar rules. The AST is built using the `Tree` class and standardized using the `makeStandardTree` method, which transforms constructs like `let`, `where`, and `rec` into lambda-based forms. The CSE machine executes the control structures, managing environments and variable bindings via the `Environment` class. The program supports the `-ast` switch to print the AST and produces the final output as specified.

6 Conclusion

The project successfully implements a lexical analyzer, parser, and CSE machine for RPAL, meeting the input/output and submission requirements. The code is modular, well-commented, and follows the specified grammar and execution model. The report provides a clear overview of the program structure and function prototypes, facilitating understanding and grading.