CS 3513: Programming Languages

Programming Languages Project

Group 33

05/05/2024

210471F : Perera S.A.I.M.
210434V : Niroshan G.

RPAL Interpreter Implementation

1. Introduction

In this project we are tasked to build a comprehensive implementation of an interpreter for the Right-reference Pedagogic Algorithmic Language (RPAL). The project is structured into several key components, each playing a crucial role in the interpretation process.

This project includes the following components:

- Lexical analyzer
- Parser
- Standerizer
- CSE machine

2. Implementation Details

2.1 Language and Tools Used

This project was fully implemented using the python programming language and this does not require any specific libraries or packages to be installed before the execution.

2.2 Lexical Analyzer

The lexical analyzer follows the rules specified in the RPAL_Lex.pdf. It tokenizes the input RPAL program into lexemes, which are then passed to the parser.

2.3 Parser

The parser, following the grammar defined in RPAL_Grammar.pdf, constructs the Abstract Syntax Tree (AST) from the lexemes generated by the lexical analyzer. We

did not use 'lex' or 'yacc' or any similar tool; instead, we implemented the parsing algorithm manually.

2.4 AST to ST Conversion (Standerizer)

After constructing the AST, we implemented an algorithm to convert it into a Standardized Tree (ST). The ST represents the program in a format suitable for execution by the CSE machine.

2.5 CSE Machine Implementation

We implemented a CSE (Control Structure Evaluation) machine to execute the RPAL programs represented by the ST. The CSE machine evaluates the program by traversing the ST and performing the necessary computations.

2.6 Input and Output

The program reads an input file containing RPAL program code. It accepts the -ast switch to print the Abstract Syntax Tree (AST) of the input program. In addition, -std switch prints the standardized tree which consists of lambdas and gammas.

3. Program Structure

3.1 Lexical Analyzer

```
def tokenize(input_str):
    tokens = []
    keywords = {
        'COMMENT': r'//.*',
        'KEYWORD': r'(let|in|fn|where|aug|or|not|gr|ge|ls|le|eq|ne|true|false|nil|dummy|within|and|rec)\b',
        'IDENTIFIER': r'[a-zA-Z][a-zA-Z0-9_]*',
        'INTEGER': r'\d+',
        'OPERATOR': r'[+\-*<>&.@/:=~|$\#!%^_\[\]{}"\'?]+',
        'STRING': r'\'\'\'(?:\\t|\\n|\\\'|\\\'\'|[\(\);,\'a-zA-Z0-9+\-*<>&.@/:=~|$\#!%^_\[\]{}"\'?\s])+\'\'\'',
        'SPACES': r'[ \t\n]+',
        'PUNCTUATION': r'[();,]'
    }

    while input_str:
        matched = False
        for key, pattern in keywords.items():
            match = re.match(pattern, input_str)
            if match:
                if key != 'SPACES':
                    if key == 'COMMENT':
                        comment = match.group(0)
                        input_str = input_str[match.end():]
                        matched = True
                        break
                    else:
                        token_type = getattr(TokenType, key)  # Get TokenType enum value
                        if not isinstance(token_type, TokenType):
                            raise ValueError(f"Token type '{key}' is not a valid TokenType")
                        tokens.append(MyToken(token_type, match.group(0)))
                        input_str = input_str[match.end():]
                        matched = True
                        break
                input_str = input_str[match.end():]
                matched = True
                break
        if not matched:
            print("Error: Unable to tokenize input")
    return tokens
```

Data Formatting:

- **Input**: The tokenize function expects a string as input, representing the program code to be tokenized.
- **Output**: It produces a list of MyToken objects, where each token contains information about its type (enumerated by TokenType) and value.
- **Parameter Passing**: The input string is passed to the tokenize function by value, ensuring that the function operates on a copy of the original string.
- **Error Handling**: The tokenize function handles errors by printing an error message if it encounters an unrecognized token or fails to tokenize the input string.
- **Return Values**: The return value of the tokenize function is a list of tokens, allowing the calling code to iterate over the tokens and process them further.

3.2 Parser

```
51      def parse(self):
52          self.tokens.append(MyToken(TokenType.END_OF_TOKENS, ""))  # Add an End Of Tokens marker
53          self.E()  # Start parsing from the entry point
54          if self.tokens[0].type == TokenType.END_OF_TOKENS:
55              return self.ast
56          else:
57              print("Parsing Unsuccessful!...........")
58              print("REMAINIG UNPARSED TOKENS:")
59              for token in self.tokens:
60                  print("<" + str(token.type) + ", " + token.value + ">")
61              return None
62
63      def convert_ast_to_string_ast(self):
64          dots = ""
65          stack = []
66
67          while self.ast:
68              if not stack:
69                  if self.ast[-1].no_of_children == 0:
70                      self.add_strings(dots, self.ast.pop())
71                  else:
72                      node = self.ast.pop()
73                      stack.append(node)
74              else:
75                  if self.ast[-1].no_of_children > 0:
76                      node = self.ast.pop()
77                      stack.append(node)
78                      dots += "."
79                  else:
80                      stack.append(self.ast.pop())
81                      dots += "."
82                      while stack[-1].no_of_children == 0:
83                          self.add_strings(dots, stack.pop())
84                          if not stack:
85                              break
86                          dots = dots[:-1]
87                          node = stack.pop()
88                          node.no_of_children -= 1
89                          stack.append(node)
90
91          # Reverse the list
92          self.string_ast.reverse()
93          return self.string_ast
```

Data Formatting:

- **Input**: The parse method takes a list of tokens as input, representing the lexemes generated by the lexical analyzer.
- **Output**: It produces an Abstract Syntax Tree (AST) representing the parsed program.
- **Parameter Passing**: The list of tokens is passed to the parse method by reference, allowing the method to directly manipulate the list during parsing.
- **Error Handling**: If parsing is unsuccessful, the parse method prints an error message and returns None, indicating failure.
- **Return Values**: The return value of the parse method is the constructed AST, enabling the calling code to further process or analyze the parsed program.

3.3 AST to ST Conversion (Standerizer)

```
    def standardize(self):
        if not self.is_standardized:
            for child in self.children:
                child.standardize()

            if self.data == "let":
                temp1 = self.children[0].children[1]
                temp1.set_parent(self)
                temp1.set_depth(self.depth + 1)
                temp2 = self.children[1]
                temp2.set_parent(self.children[0])
                temp2.set_depth(self.depth + 2)
                self.children[1] = temp1
                self.children[0].set_data("lambda")
                self.children[0].children[1] = temp2
                self.set_data("gamma")
            elif self.data == "where":
                temp = self.children[0]
                self.children[0] = self.children[1]
                self.children[1] = temp
                self.set_data("let")
                self.standardize()
            elif self.data == "function_form":
                Ex = self.children[-1]
                current_lambda = NodeFactory.get_node_with_parent("lambda", self.depth + 1, self, [], True)
                self.children.insert(1, current_lambda)

                i = 2
                while self.children[i] != Ex:
                    V = self.children[i]
                    self.children.pop(i)
                    V.set_depth(current_lambda.depth + 1)
                    V.set_parent(current_lambda)
                    current_lambda.children.append(V)

                    if len(self.children) > 3:
                        current_lambda = NodeFactory.get_node_with_parent("lambda", current_lambda.depth + 1, current_lambda, [], True)
                        current_lambda.get_parent().children.append(current_lambda)
```

Note: This code is a part of the whole function. Please review the code snippet provided within the repository.

Data Formatting:

- **Input**: After converting the nodes which were returned by the parser, to another set of nodes which have more attributes and methods (This is done by “ASTFactory”) the root node is pushed to the standardize function.
- **Output**: This produces the standardized AST as a set of nodes.

- **Parameter Passing**: The root node which was returned by the ASTFactory.
- **Return Values**: Constructed standard AST object

3.4 CSE Machine

```
class CSEMachine:
    def __init__(self, control, stack, environment):
        self.control = control
        self.stack = stack
        self.environment = environment

    def execute(self):
        current_environment = self.environment[0]
        j = 1
        while self.control:
            current_symbol = self.control.pop()
            if isinstance(current_symbol, Id):
                self.stack.insert(0, current_environment.lookup(current_symbol))
            elif isinstance(current_symbol, Lambda):
                current_symbol.set_environment(current_environment.get_index())
                self.stack.insert(0, current_symbol)
            elif isinstance(current_symbol, Gamma):
                next_symbol = self.stack.pop(0)
                if isinstance(next_symbol, Lambda):
                    lambda_expr = next_symbol
                    e = E(j)
                    if len(lambda_expr.identifiers) == 1:
                        e.values[lambda_expr.identifiers[0]] = self.stack.pop(0)
                    else:
                        tup = self.stack.pop(0)
                        for i, id in enumerate(lambda_expr.identifiers):
                            e.values[id] = tup.symbols[i]
                    for env in self.environment:
                        if env.get_index() == lambda_expr.get_environment():
                            e.set_parent(env)
                    current_environment = e
                    self.control.append(e)
                    self.control.append(lambda_expr.get_delta())
                    self.stack.insert(0, e)
                    self.environment.append(e)
                elif isinstance(next_symbol, Tup):
                    tup = next_symbol
                    i = int(self.stack.pop(0).get_data())
                    self.stack.insert(0, tup.symbols[i - 1])
                elif isinstance(next_symbol, Ystar):
```

Note: This code is a part of the whole function. Please review the code snippet provided within the repository

Data Formatting:

- **Input:**This takes control, stack and environment which were returned by the "get_cse_machine" which takes the standardized AST as the input.
- **Output:** It produces the final output result for the RPAL program code.
- **Parameter Passing:**
    - **Control Stack**: The control attribute represents the control stack of the CSE machine, containing symbols and instructions to be executed.
    - **Stack**: The stack attribute represents the stack of the CSE machine, used for storing intermediate results during execution.
    - **Environment**: The environment attribute represents the environment of the CSE machine, containing bindings between identifiers and their values.
- **Return Values:**The get_answer method returns the final result after executing the CSE machine.

4.Usage

The interpreter can be run by two methods.

1. Using Direct Python Commands
2. Using Makefile

1. Using Direct Python Commands

1.1. Open the terminal and navigate to the directory where the interpreter is located.

1.2. Run the following commands for output, ast and standardized ast respectively.

```
python myrpal.py input.txt
python myrpal.py input.txt -ast
python myrpal.py input.txt -sast
```

1.3. More inputs can be found in inputs/ directory.

2. Using Makefile

2.1. Open the terminal and navigate to the directory where the interpreter is located.

2.2. Run the following commands for output, ast and standardized ast respectively.

```
make run file=input.txt
make ast file=input.txt
make sast file=input.txt
```

5. Conclusion

In conclusion, we successfully implemented a lexical analyzer, parser, AST to ST conversion algorithm, and CSE machine for the RPAL language. Our program can efficiently parse RPAL programs, construct ASTs, execute them using the CSE machine, and produce accurate results.

Please find the GitHub repository link below for the project:
https://github.com/Irash-Perera/RPAL-Project.git