PROJECT REPORT

ON

Design a Compiler (Lexical and Syntax Analysis Phase) for RUBY

By

Name of the students: -	Registration Nos.: -
Rishabh Chauhan	**AP21110010913**
Krishna Sharma	**AP21110010914**
Sai Kumar Chunduru	**AP21110010943**
Rahul Vijay Singh	**AP21110010960**

Under: -

Dr. Jaya Lakshmi Tangirala

Assistant Professor – Department of Computer Science and Engineering

SRM UNIVERSITY, AP

INTRODUCTION

Background

Compilers play a crucial role in translating high-level programming languages into machine code. Our project focuses on creating a Ruby to Assembly language compiler, covering the initial phases of Lexical Analysis and Syntax Analysis.

Objective

The objectives of our project are to implement both the Lexical Analysis and Syntax Analysis phases of the Ruby to Assembly compiler. Lexical Analysis involves recognizing and tokenizing the various elements of Ruby code, while Syntax Analysis deals with parsing the tokenized code to build a syntax tree.

Scope

The scope of this project includes Lexical Analysis and Syntax Analysis. We will tokenize Ruby source code, identify keywords, operators, literals, and other components during Lexical Analysis. The Syntax Analysis phase will involve implementing a Recursive Descent Parser to parse the token stream and build a syntax tree.

LEXICAL ANALYSIS PHASE

Overview
Lexical analysis is the initial phase of the compiler, responsible for reading the source code and converting it into a stream of tokens. These tokens represent the fundamental building blocks of the programming language, such as identifiers, keywords, operators, and literals.

Tools Used
For the Lexical Analysis phase, we have used the following tools:
Ruby Language: As our source language.
Regular Expression: To define the patterns for token recognition.
Tokenization: To split the source code into a stream of tokens.

Tokenization
Tokenization is the process of breaking the input source code into smaller units, known as tokens. These tokens are the basic elements of the programming language and will be used in the subsequent compilation phases.

Regular Expressions
Regular expressions are patterns that define how different tokens in Ruby code are recognized. We have defined regular expressions for keywords, operators, literals, and other elements. For example, the regular expression for recognizing integer literals in Ruby is as follows:
Ruby:

integer_literal = /\d/

Exception Handling
Error handling has been taken care by the Lexical Error class, which is a custom exception class. It is likely used to handle errors that may occur during the lexical analysis phase, such as encountering unexpected characters in the source code.

Implementation

Our group has implemented the Lexical Analysis phase in Ruby. Below is a simplified code snippet for tokenizing Ruby source code.

<u>Code</u>: -

```ruby
class String
    def isspace
      self =~ /^\s+$/
    end
  end

  RubyKeywords = %w[
  begin end if unless else elsif case when while until for in do
  module class def defined? alias undef super self
  true false nil and or not
  rescue ensure retry raise throw catch fail
  public protected private
  attr_accessor attr_reader attr_writer attr
  break next redo return
  true false nil
]

  class Lexer
    def initialize(source_code)
      @source_code = source_code
      @current_position = 0
      @current_token = nil
    end

    def next_token
        while @current_position < @source_code.length
          character = @source_code[@current_position]
          @current_position += 1

          if character.isspace
            next
          elsif character.match?(/[[:alpha:]_]/)
            token = lex_identifier(character)
```

```ruby
          return token
        elsif character.match?(/[[:digit:]]/)
          token = lex_number(character)
          return token
        elsif ['+', '-', '*', '/', '%', '(', ')',
','].include?(character)
          token = Token.new(TokenType::OPERATOR, character)
          return token
        elsif character == '='
          token = Token.new(TokenType::ASSIGNMENT_OPERATOR,
character)
          return token
        elsif character == '@'
          return lex_identifier(character)
        elsif character == '#'
          skip_comments
        else
          raise LexicalError, "Unexpected character: #{character}"
        end
      end

      nil
    end


    def lex_identifier(first_character)
      identifier = first_character
      while @current_position < @source_code.length &&
@source_code[@current_position].match?(/[[:alnum:]_@]/)
        identifier += @source_code[@current_position]
        @current_position += 1
      end

      # Check if the identifier is a keyword
      if RubyKeywords.include?(identifier)
        token = Token.new(TokenType::KEYWORD, identifier)
      else
        token = Token.new(TokenType::IDENTIFIER, identifier)
      end

      token
    end

  def lex_number(first_character)
    number = first_character
```

```ruby
      while @current_position < @source_code.length &&
@source_code[@current_position].match?(/\d/)
        number += @source_code[@current_position]
        @current_position += 1
      end

      Token.new(TokenType::NUMBER, number)
    end

    def lex_string
      string = ''
      delimiter = @source_code[@current_position]
      @current_position += 1

      while @current_position < @source_code.length &&
@source_code[@current_position] != delimiter
        string += @source_code[@current_position]
        @current_position += 1
      end

      @current_position += 1

      Token.new(TokenType::STRING, string)
    end

    def skip_comments
      while @current_position < @source_code.length &&
@source_code[@current_position] != "\n"
        @current_position += 1
      end

      @current_position += 1
    end

    def lex_keyword
      keyword = ''
      while @current_position < @source_code.length &&
@source_code[@current_position].match?(/[[:alnum:]_]/)
        keyword += @source_code[@current_position]
        @current_position += 1
      end

      if RubyKeywords.include?(keyword)
        Token.new(TokenType::KEYWORD, keyword)
      else
        Token.new(TokenType::IDENTIFIER, keyword)
```

```ruby
    end
  end
end

class SyntaxAnalyzer
  def initialize(tokens)
    @tokens = tokens
    @current_token_index = 0
  end

  def analyze
    parse_expression
  end

  private

  def parse_expression
    left_node = parse_term
    node = parse_expression_rest(left_node)
    node
  end

  def parse_expression_rest(left_node)
    token = @tokens[@current_token_index]

    if token&.type == :PLUS
      @current_token_index += 1
      right_node = parse_term
      node = {
        type: :PLUS,
        left: left_node,
        right: right_node,
        position: token.position
      }
      node = parse_expression_rest(node)
      return node
    else
      return left_node
    end
  end

  def parse_term
    left_node = parse_factor
    node = parse_term_rest(left_node)
    node
  end
```

```ruby
    def parse_term_rest(left_node)
      token = @tokens[@current_token_index]

      if token&.type == :MULTIPLY
        @current_token_index += 1
        right_node = parse_factor
        node = {
          type: :MULTIPLY,
          left: left_node,
          right: right_node,
          position: token.position
        }
        node = parse_term_rest(node)
        return node
      else
        return left_node
      end
    end

    def parse_factor
      token = @tokens[@current_token_index]
      @current_token_index += 1

      case token&.type
      when :LPAREN
        node = parse_expression
        if @tokens[@current_token_index]&.type == :RPAREN
          @current_token_index += 1
          return {
            type: :PARENTHESIS,
            expression: node,
            position: token.position
          }
        else
          raise SyntaxError, "Expected closing parenthesis ')' at
position #{token.position}"
        end
      when :NUMBER
        return {
          type: :NUMBER,
          value: token.value,
          position: token.position
        }
      else
```

```ruby
        raise SyntaxError, "Unexpected token '#{token.value}' at
position #{token.position}"
      end
    end
  end

  class TokenType
    IDENTIFIER = :IDENTIFIER
    OPERATOR = :OPERATOR
    ASSIGNMENT_OPERATOR = :ASSIGNMENT_OPERATOR
    NUMBER = :NUMBER
    STRING = :STRING
    KEYWORD = :KEYWORD
  end


  class Token
    def initialize(type, value)
      @type = type
      @value = value
    end

    def type
      @type
    end

    def value
      @value
    end

    def to_s
      "Token(type = #{@type}, value -> '#{@value}')"
    end
  end


  class LexicalError < StandardError; end

  def main
    source_code = <<~CODE
      (3 + 5) * 2
      # This is a comment.
      def example_method
        @variable1 = 20
        @variable2 = 10
        # Method body
```

```
    end
  CODE

  lexer = Lexer.new(source_code)

  token = lexer.next_token
  while !token.nil?
    puts token
    token = lexer.next_token
  end
end

if __FILE__ == $0
  main
end
```

RESULTS

Output: -

```
Token(type = OPERATOR, value -> '(')
Token(type = NUMBER, value -> '3')
Token(type = OPERATOR, value -> '+')
Token(type = NUMBER, value -> '5')
Token(type = OPERATOR, value -> ')')
Token(type = OPERATOR, value -> '*')
Token(type = NUMBER, value -> '2')
Token(type = KEYWORD, value -> 'def')
Token(type = IDENTIFIER, value -> 'example_method')
Token(type = IDENTIFIER, value -> '@variable1')
Token(type = ASSIGNMENT_OPERATOR, value -> '=')
Token(type = NUMBER, value -> '20')
Token(type = IDENTIFIER, value -> '@variable2')
Token(type = ASSIGNMENT_OPERATOR, value -> '=')
Token(type = NUMBER, value -> '10')
Token(type = KEYWORD, value -> 'end')
```

SYNTAX ANALYSIS PHASE

Overview
Syntax analysis is the second phase of the compiler, responsible for parsing the token stream generated during Lexical Analysis. This phase involves implementing a Recursive Descent Parser to analyze the syntax of the Ruby code and build a syntax tree.

Recursive Descent Parser
A Recursive Descent Parser is used to parse the token stream and construct a syntax tree by recursively applying grammar rules. This approach closely mirrors the structure of the formal grammar rules of the Ruby language.

Implementation
Our Syntax Analysis phase includes a Recursive Descent Parser implemented in Ruby. The parser consists of methods for each non-terminal symbol in the grammar, facilitating the construction of a syntax tree based on the token stream.

Code: -

```
class Token
    attr_reader :type, :value, :position

    def initialize(type, value, position)
      @type = type
      @value = value
      @position = position
    end
  end

  class SyntaxAnalyzer
    def initialize(tokens)
      @tokens = tokens
```

```ruby
    @current_token_index = 0
  end

  def analyze
    parse_expression
  end

  private

  def parse_expression
    left_node = parse_term
    node = parse_expression_rest(left_node)
    node
  end

  def parse_expression_rest(left_node)
    token = @tokens[@current_token_index]

    if token&.type == :PLUS
      @current_token_index += 1
      right_node = parse_term
      node = {
        type: :PLUS,
        left: left_node,
        right: right_node,
        position: token.position
      }
      node = parse_expression_rest(node)
      return node
    else
      return left_node
    end
  end

  def parse_term
    left_node = parse_factor
    node = parse_term_rest(left_node)
    node
  end

  def parse_term_rest(left_node)
    token = @tokens[@current_token_index]

    if token&.type == :MULTIPLY
      @current_token_index += 1
      right_node = parse_factor
```

```
        node = {
          type: :MULTIPLY,
          left: left_node,
          right: right_node,
          position: token.position
        }
        node = parse_term_rest(node)
        return node
      else
        return left_node
      end
    end

    def parse_factor
      token = @tokens[@current_token_index]
      @current_token_index += 1

      case token&.type
      when :LPAREN
        node = parse_expression
        if @tokens[@current_token_index]&.type == :RPAREN
          @current_token_index += 1
          return {
            type: :PARENTHESIS,
            expression: node,
            position: token.position
          }
        else
          raise SyntaxError, "Expected closing parenthesis ')' at
position #{token.position}"
        end
      when :NUMBER
        return {
          type: :NUMBER,
          value: token.value,
          position: token.position
        }
      else
        raise SyntaxError, "Unexpected token '#{token.value}' at
position #{token.position}"
      end
    end
  end

  # Sample tokens representing an arithmetic expression: (3 + 5) * 2
  tokens = [
```

```ruby
  Token.new(:LPAREN, '(', 1),
  Token.new(:NUMBER, '3', 2),
  Token.new(:PLUS, '+', 4),
  Token.new(:NUMBER, '5', 6),
  Token.new(:RPAREN, ')', 7),
  Token.new(:MULTIPLY, '*', 9),
  Token.new(:NUMBER, '2', 11),
]

syntax_analyzer = SyntaxAnalyzer.new(tokens)
begin
  syntax_tree = syntax_analyzer.analyze
  puts "Syntax analysis completed without errors. \nSyntax Tree:"
  puts syntax_tree
rescue SyntaxError => e
  puts "Syntax error: #{e.message}"
end
```

Output: -

```
Syntax analysis completed without errors.
Syntax Tree:
{:type=>:MULTIPLY, :left=>{:type=>:PARENTHESIS, :expression=>{:type=>:PLUS, :left=>{:type=>:NUMBER, :value=>"3", :position=>2}, :right=>{:type=>:NUMBER, :value=>"5", :position=>6}, :positio
n=>4}, :position=>1}, :right=>{:type=>:NUMBER, :value=>"2", :position=>11}, :position=>9}
```

SYMBOL TABLE

Overview

Symbol Table is an important data structure created and maintained by the compiler to keep track of the semantics of variables i.e. it stores information about the scope and binding information about names, information about instances of various entities such as variable and function names, classes, objects, etc.

```ruby
class Token
  attr_reader :type, :value

  def initialize(type, value)
    @type = type
    @value = value
  end
end

class SymbolTable
  def initialize
    @table = {}
  end

  def add_entry(name, type)
    @table[name] = type
  end

  def get_table
    @table
  end
end

# Sample output
tokens = [
  Token.new(:OPERATOR, '()'),
  Token.new(:NUMBER, '3'),
  Token.new(:OPERATOR, '+'),
  Token.new(:NUMBER, '5'),
  Token.new(:OPERATOR, ')'),
  Token.new(:OPERATOR, '*'),
  Token.new(:NUMBER, '2'),
  Token.new(:KEYWORD, 'def'),
  Token.new(:IDENTIFIER, 'example_method'),
  Token.new(:KEYWORD, 'end'),
  Token.new(:IDENTIFIER, 'variable1'),
  Token.new(:OPERATOR, '='),
  Token.new(:NUMBER, '10'),
  Token.new(:IDENTIFIER, 'variable2'),
  Token.new(:OPERATOR, '='),
  Token.new(:NUMBER, '20')
]

# Create a symbol table
```

```ruby
symbol_table = SymbolTable.new

# Iterate over tokens and add identifiers to the symbol table
tokens.each do |token|
  if token.type == :IDENTIFIER
    symbol_table.add_entry(token.value, token.type)
  end
end

# Print the symbol table as a table
puts "Symbol Table:"
puts "+----------------+------------+"
puts "| Identifier     |    Type    |"
puts "+----------------+------------+"
symbol_table.get_table.each do |name, type|
  printf("| %-14s | %-10s |\n", name, type)
end
puts "+----------------+------------+"
```

OUTPUT: -

```
Symbol Table:
+----------------+------------+
| Identifier     |    Type    |
+----------------+------------+
| example_method | IDENTIFIER |
| variable1      | IDENTIFIER |
| variable2      | IDENTIFIER |
+----------------+------------+
```

SEMANTIC ANALYSIS

Semantic Analysis is the third phase of Compiler. Semantic Analysis makes sure that declarations and statements of program are semantically correct. It is a collection of procedures which is called by parser as and when required by grammar. Both syntax tree of previous phase and symbol table are used to check the consistency of the given code. Type checking is an important part of semantic analysis where compiler makes sure that each operator has matching operands.

Semantic Analyzer:
It uses syntax tree and symbol table to check whether the given program is semantically consistent with language definition. It gathers type information and stores it in either syntax tree or symbol table. This type information is subsequently used by compiler during intermediate-code generation.

Semantic Errors:
Errors recognized by semantic analyzer are as follows:

- Type mismatch
- Undeclared variables
- Reserved identifier misuse

Functions of Semantic Analysis:
Type Checking –
    Ensures that data types are used in a way consistent with their definition.
Label Checking –
    A program should contain labels references.
Flow Control Check –
    Keeps a check that control structures are used in a proper manner. (example: no break statement outside a loop)

INTERMIDIATE CODE GENERATION

In the analysis-synthesis model of a compiler, the front end of a compiler translates a source program into an independent intermediate code, then the back end of the compiler uses this intermediate code to generate the target code (which can be understood by the machine). The benefits of using machine-independent intermediate code are:

Because of the machine-independent intermediate code, portability will be enhanced. For ex, suppose, if a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required. Because, obviously, there were some modifications in the compiler itself according to the machine specifications.
Retargeting is facilitated.
It is easier to apply source code modification to improve the performance of source code by optimizing the intermediate code.

CODE OPTIMIZATION

The code optimization in the synthesis phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resources (i.e., CPU, Memory) so that faster-running machine code will result. Compiler optimizing process should meet the following objectives:

The optimization must be correct, it must not, in any way, change the meaning of the program.
Optimization should increase the speed and performance of the program.
The compilation time must be kept reasonable.
The optimization process should not delay the overall compiling process.
When to Optimize?
Optimization of the code is often performed at the end of the development stage since it reduces readability and adds code that is used to increase the performance.

Why Optimize?
Optimizing an algorithm is beyond the scope of the code optimization phase. So the program is optimized. And it may involve reducing the size of the code. So optimization helps to:

Reduce the space consumed and increases the speed of compilation.
Manually analyzing datasets involves a lot of time. Hence, we make use of software like Tableau for data analysis. Similarly, manually performing the optimization is also tedious and is better done using a code optimizer.
An optimized code often promotes re-usability.