Compiler Project Cover Page

Project contributors: Brian Wei, Kelvin Nguyen, Suhaib Affaneh

Submission Date: 5/10/2025

Executable File name: run_tests.exe

Test case and output files:

Input file	Output file
test_case1.txt	output_test_case1.txt
test_case2.txt	output_test_case2.txt
test_case3.txt	output_test_case3.txt

Operating System: Windows

Compiler Project Documentation

1. Problem Statement

Assignment 1 requires the development of a lexical analyzer for the RAT25S language. It is meant to convert source code into tokens and lexemes. Each lexeme has a token associated with it, which are identifiers, integers, reals, operators, keywords, and separators. Identifiers must start with a letter followed optionally by either letter(s), digit(s), or underscore(s). A real must begin with digit(s) followed by ".", and then again followed by digit(s). Integers are all digits but without ".". Operators, keywords, and separators are defined according to a predefined list.

Assignment 2 requires making a syntax analyzer for the RAT25S language. There is left recursion in our rules, so the productions must be rewritten. We also have to use left factorization so that there is no backtracking. The lexer needs to be kept in the code and the parser (syntax analyzer) prints out the productions depending on the statement structure.

Assignment 3 requires symbol table handling to track program identifiers and their memory locations; and assembly code generation for a stack-based virtual machine. The compiler now validates

variable declarations, performs type checking, and outputs executable assembly code. The symbol table functionality prevents duplicate declarations and usage of undeclared variables. The code generator translates the simplified Rat25S language (with no function definitions or real types) into machine instructions that can be executed by a virtual stack machine with operations for arithmetic, comparison, input/output, and control flow.

2. How to use the program

Requirements:

- Must have executable, rat25s_parser.py, and the three test case files.

Steps to Execute

1. Unzip and extract the files. Ensure you have the following files in your directory:
    - The executable file
    - Source code file: rat25s_parser.py
    - Test case files containing Rat25S code
        - test_case1.txt
        - test_case2.txt
        - test_case3.txt
2. Run ".\run_tests.exe" in the terminal under the correct directory that the exe file, rat25s_parser.py, and test case files are in. It is recommended to use VS Code.
   You might have to run the cd command to do that to get to the correct directory.
   Also, this assumes you are using the recommended Windows operating system.
3. The program will process each test file and generate corresponding output files (ex: result for test_case1.txt will be in output_test_case1.txt, etc.):
    - output_test_case1.txt
    - output_test_case2.txt
    - output_test_case3.txt
4. The program will also display the results in the console, showing:
    - whether or not it succeeded or failed
    - Number of tokens
    - A summary of the processing results
5. If exe file does not work, then you would have to install Python, and have the three test case files, rat25s_parser.py, and run_tests.py in one directory. Then, in terminal, type in "python run_tests.py". You will see output in corresponding output result files and in the terminal. It is recommended to use VS Code.

` Output Format

Each output file will contain:

- The tokens and lexemes identified in the source code
- Productions identified in the test case
- Any errors found

- The symbol table with all identifiers, their memory locations, and types
- The generated assembly code with instruction numbers

3. Design of your program

Major Components:

1. Token Class
2. Finite State Machine Functions
3. Tokenize Function (lexer())
4. File Processing Functions
5. Parser Class
6. Parsing Functions
7. Scope Functions
8. Print Production Function
9. Type Mismatch Detector
10. Symbol Table Implementation
11. Assembly Instruction Generator

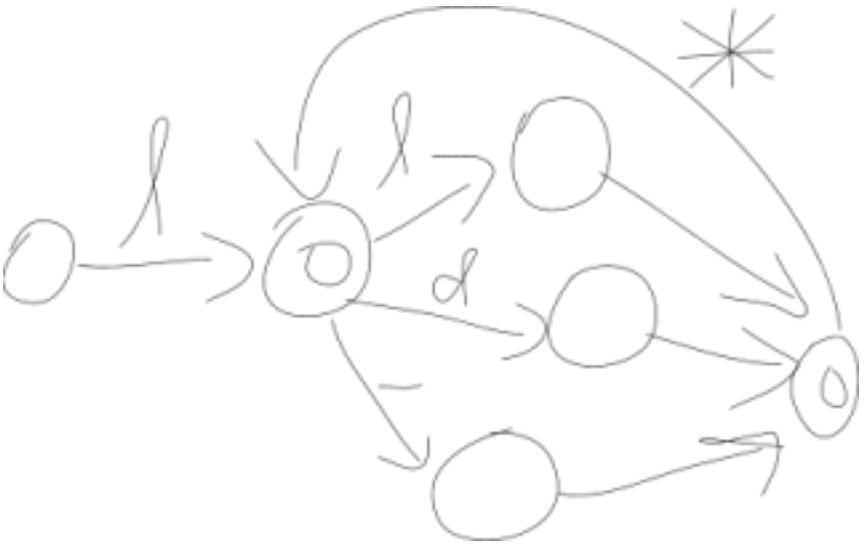
Data Structures

1. Token : a class with attributes for token type, lexeme, and line number
2. Lexer Lists : used to store tokens, keywords, operators, and separators
3. Strings: used to process and manipulate lexemes
4. Parser: a class with tokens attribute
5. Parser lists: stores output lines, scope stacks, function names
6. Symbol Table: stores identifier information (lexeme, memory address, type)
7. Assembly Instruction Array: stores generated assembly code instructions
8. Memory Address Counter: tracks next available memory location
9. Instruction Counter: tracks next instruction number

Algorithms

1. FSM Implementation
    - Each FSM function tracks a state variable that changes based on input characters
    - State transitions follow rules defined by Regular Expressions
    - Final states determine whether input is accepted or rejected
2. Regular Expressions for FSMs
    - Identifier: letter(letter|digit|_)*
    - Integer: digit+
    - Real: digit+ . digit+
3. NFSM Construction (Thompson's Method):

    Identifier:



Integer:

Real:

4. Tokenization Algorithm
    - Skip whitespace
    - Identify and handle comments
    - Check for identifiers and keywords

- Check for integers and real numbers
- Check for operators (two-character operators first)
- Check for separators (including a section that checks specifically for $$ before other separators)
- Handle unrecognized characters as errors

5. Parser Algorithm
    - Looks at what kind of statement it is
    - Calls different parsing functions depending on type of statement (ex: function, if, while, etc.)
    - Checks if it's syntactically correct by checking each token and the statement structure
    - If syntax is incorrect, the type of error is outputted.
    - Checks if variables have been declared before use
    - Checks if the variable is assigned to a value of same data type
        - Checks for arithmetic operations with booleans (ex: true + false, true / 50 would create error)
    - Error count

Rules for Assignment 2 (after removing left recursion and implementing left factorization):

- **<Program> -> <Statement List>**
- **<Statement List> -> <Statement> <Statement List> | ε**
- **<Statement> -> <Compound> | <Assign> | <If> | <Return> | <Print> | <Scan> | <While> | <Declaration>**
- **<Compound> -> { <Statement List> }**
- **<Function> -> function <Identifier> (<Parameter List>) <Compound>**
- **<Scan> -> scan (<IDs>);**
- **<Print> -> print (<Expression>);**
- **<Parameter List> -> <Parameter> <Parameter List Prime> | ε**
- **<Parameter List> -> ε**
- **<Parameter List Prime> -> , <Parameter> <Parameter List Prime> | ε**
- **<Parameter List Prime> -> ε**
- **<Parameter> -> <IDs> <Qualifier>**
- **<Qualifier> -> integer | real | boolean**
- **<If> -> if (<Condition>) <Statement> <IfPrime>**
- **<IfPrime> -> else <Statement> endif | endif**
- **<While> -> while (<Condition>) <Statement List> endwhile**
- **<Condition> -> <Expression> <Relop> <Expression>**

- **<Relop> -> == | != | > | < | <= | >=**
- **<Return> -> return <Expression> ;**
- **<Assign> -> <Identifier> = <Expression> ;**
- **<Expression> -> <Term> <ExpressionPrime>**
- **<ExpressionPrime> -> + <Term> <ExpressionPrime> |**
  - <Term> <ExpressionPrime> | ε
- **<ExpressionPrime> -> ε**
- **<Term> -> <Factor> <TermPrime>**
- **<TermPrime> -> * <Factor> <TermPrime> | / <Factor> <TermPrime> | ε**
- **<TermPrime> -> ε**
- **<Factor> -> <Identifier> | <Number> | (<Expression>) |**
  <Function Call>
- **<Declaration> -> <Qualifier> <IDs> ;**
- **<IDs> -> <Identifier> <IDsPrime>**
- **<IDsPrime> -> , <Identifier> <IDsPrime> | ε**
- **<IDsPrime> -> ε**
- **<Function Call> -> <Identifier> (<Arguments>)**
- **<Arguments> -> <Expression> <ArgumentsPrime> | ε**
- **<Arguments> -> ε**
- **<ArgumentsPrime> -> , <Expression> <ArgumentsPrime> | ε**
- **<ArgumentsPrime> -> ε**

6. Symbol Table Operations:
   - Insert: adds new identifier with memory address
   - Lookup: checks if identifier exists
   - Print: displays all identifiers with their memory addresses

7. Code Generation Algorithms:
   - Expression code generation with operator precedence
   - Conditional statement code generation with jump instructions
   - Loop code generation with conditional and unconditional jumps
   - Variable assignment code generation
   - Input / output code generation

4. Any Limitation

- Maximum line length: The program processes one line at a time, so extremely long lines might cause memory issues
- Comment handling: Nested comments are not supported
- Error recovery: The program identifies errors but does not attempt to recover from them
- Specifically Project 2:
    - The code is around 1200 lines long which makes debugging and understanding the code, tedious.
    - If there is a real syntax error in the test case, then it could cause bogus error notifications to pop up in subsequent lines. But because an option is for the syntax analyzer to stop once an actual error has been met, we didn't think too much about it.

For project 3:

- Memory addresses start at 10,000 and are implemented sequentially
- No optimization of generated assembly code is performed
- No support for function calls in the assembly code generation
- The simplified Rat25S for Assignment 3 does not support function definitions or real types

5. Any shortcomings

For project 1:

None. All the required features have been implemented according to the assignment specifications

- The lexer correctly identifies all the token types: keywords, identifiers, operators, separators, integers, reals.
- FSMs are implemented for identifiers, integers, and real numbers.
- The program processes input files and generates results to the corresponding output files.
- Error detection for invalid tokens is included.
- Multiple test cases are supported. Each test case file has a function and the files pass the "minimum source line" requirement, which is to have < 10 lines in one test file, < 21 lines in another test file, and > 21 lines in the other test file. The test files contain code that follows RAT25S syntax and includes $$ to separate functions from declarations and declarations from statements.

For project 2:

- No major shortcomings; it passes our test cases

For project 3:

None. All the required features have been implemented according to the

assignment specifications:

- Symbol table implementation with all required operations (check, insert, print)
- Memory address tracking for each identifier
- Type checking and validation
- Error detection for duplicate declarations and undeclared variables
- Assembly code generation for all required instruction types
- Correct handling of arithmetic operations
- Proper implementation of comparison operations
- Correct handling of loop and conditional structures
- Input/output operations (SOUT, SIN)