

# **COVER PAGE**

## **CS323 Programming Assignments**

**Fill out all entries 1 - 7. If not, there will be deductions!**

1. Names [ 1. Sebastien Minassian ] Section [2]  
[ 2. Sean DeFrank ] Section [2]  
[ 3. Gabriel Pacquing ] Section [2]

2. Assignment Number [1]

3. Due Date [ 11/9/2025 ]

4. Submission Date [ 11/9/2025 ]

5. Executable File name [parser.exe]

**(A file that can be executed without compilation by the instructor, such as .exe, .jar, etc - NOT a source file such as .cpp )**

6. Names of the testcase files -      input test file      output test file  
test 1. [Rat25f.txt]      [      output1.txt      ]  
test 2. [Rat25f2.txt]      [      output2.txt      ]  
test 3. [Rat25f3.txt]      [      output3.txt      ]

7. Operating System [ Windows ]

**(Window – preferred)**

---

**To be filled out by the Instructor:**

Comments and Grade:

# CS323 Documentation

About 2 pages

## 1. Problem Statement

Our goal for this assignment is to design and then implement a syntax analyzer/parser for the Rat25f programming language.

For this stage of the compiler, the parser should verify that the sequence of tokens produced by the lexical analyzer (that we made in Assignment 1) follows the structure detailed by the Rat25f grammar.

Our objectives for this assignment are to:

1. Rewrite the grammar to remove left recursion and apply left factorization where necessary, parsing using a top-down approach.
2. Applying our lexer function created in Assignment 1 to obtain tokens from the input source file.
3. Developing a top-down parser that reads these tokens, prints each token/lexeme pair, then lists the production rules used to analyze them.
4. Error handling for each missed grammar rule is extremely important.
5. Printing everything into an output file that records all tokens, lexemes, production rules, and syntax errors produced during the processing of the test file.

## 2. How to use your program

The first step is to ensure all project files are in the same directory, including:

Syntax\_Analyzer.cpp

Lexical\_Analyzer.cpp

Lexical\_Analyzer.h

Rat25f.txt, Rat25f2.txt, or Rat25f3.txt (Our input)

Compile the program using any standard C++ compiler, we used:

```
g++ Syntax_Analyzer.cpp Lexical_Analyzer.cpp -o rat25f.exe
```

Prepare your input file in Rat25f.txt, Rat25f2.txt, or Rat25f3.txt

Then run the executable, ./rat25f.

Lexical analysis will then be performed using the lexer from assignment 1, the tokens will be parsed according to Rat25f grammar, and then everything will be printed to an output.txt file.

### 3. Design of your program

First off, we have Lexical Analyzer.cpp and .h.

These files tokenize the input source program and categorizes each lexeme into token types, Identifier, Operator, Separator, Integer, Real, etc.

Then returns a token struct containing two vectors, one for token types, and one for lexemes.

Next, we have Syntax\_Analyzer.cpp, which implements a recursive descent parser for the Rat25f grammar, as well as recursive functions that correspond to each grammar nonterminal.

Program → StatementList

StatementList → Statement StatementList | ε

Statement → Assign | Compound

Assign → Identifier = Expression ;

Expression → Term Expression'

Expression' → + Term Expression' | - Term Expression' | ε

Term → Factor Term'

Term' → \* Factor Term' | / Factor Term' | ε

Factor → ( Expression ) | Identifier | Integer | Real

The parser function prints the production rule (if turned on) that it is currently applying. Our error handling also prints a clear syntax error message as to what the parser expected to see in that spot, as well as what was actually there, then terminates the program. Everything is written to an output.txt file that has all the steps for the syntax analyzer.

Data Structures Used:

- Vectors to store token and lexeme lists.
- Struct Tok holds the token type.
- Recursive functions represent the grammar nonterminals and parsing order.

#### **4. Any Limitation**

When using the executable, the text “Starting lexical analysis...” prints twice.

#### **5. Any shortcomings**

The syntax analysis does not print in the terminal but is fully written out in the output.txt files.