

# COMS W4115 Fall 2025: Programming Assignment 1

## 1 Overview of the Programming Project

This is the first of four programming assignments (I–IV) where you will design and build a compiler for **MiniLang**. Each assignment covers one compiler phase:

1. Lexical Analysis
2. Parsing
3. Semantic Analysis
4. Code Generation

For this assignment, you will implement a **lexical analyzer (scanner)** in **C++**.

This work must be done **individually**.

Link to the lexical specification document - [MiniLang specs - Lexical](#)

## 2 Files Provided

1. **lexer.h** – Header file containing the **Lexer** class definition.
2. **lexer.cpp** – File to implement the **Lexer** class (to be completed by you).
3. **exception.h** – Header for custom exceptions thrown during compilation.
4. **token.h** – Header that defines the **Token** class. Tokens include type, text, line, and column.
5. **Makefile** – Used to build and test the project.
6. **main.cpp** – Driver code for executing your lexer.

You are also provided with a directory **test/** which contains some sample inputs and outputs.

## 3 Deliverables

You must complete the method:

Token **Lexer::getNextToken();**

## Requirements:

- The method should read bytes from the input file stream managed by the lexer.

- Each call to `getNextToken()` should scan and return exactly one token, according to the MiniLang lexical specification.
- The lexer must track and update its internal line and column members, which maintain the current reading position in the source file. These will be used to determine where the next `getNextToken()` call continues scanning.
- The created token must be stored in the lexer's `tokens` vector, preserving the order in which tokens appear.
- Each `Token` object must accurately contain the following:
  - Token Type (e.g., Keyword, Identifier, Integer, Operator, Delimiter, EndOfFile)
  - Token Value (the lexeme string)
  - Line Number (the line where the token starts)
  - Column Number (the column where the token starts)
- Whitespace characters and comments should not produce any tokens; they are to be skipped entirely.
- If an unrecognized token or character is encountered at any time, immediately throw a `ParserException` with the token's start line and column position. No further scanning should be performed after the exception.
- Once the end of the input stream is reached, `getNextToken()` should return an EndOfFile (EOF) token with the current line and column numbers.
- The lexer should not emit irrelevant or extra tokens beyond those specified, as points will be deducted for extra tokens.
- The `getNextToken()` method must strictly follow the MiniLang language lexical specification available [here](#).

## Grading Criteria:

- You will be graded on the correctness of each token generated, including token type, lexeme, line number, and column number.
- Each token parameter carries equal weightage in the score.
- The total score will be calculated as a weighted average over all test cases.
- Accuracy and precision in token boundaries and reporting are essential for full credit.

## DESIGN.md

In addition to your code, submit a DESIGN.md file discussing:

1. What you did in your lexical scanner implementation.
2. The challenges you faced while designing or coding it.
3. What you learned during this assignment.
4. Provide at least three distinct sample test cases you created in addition to the provided samples. These do not have to be syntactically correct but should go beyond the examples in the assignment.

There are no single correct answers for the design questions; their purpose is to help the teaching staff assess your understanding of the assignment requirements and the lexical scanning phase of a compiler.

---

## 4 Submission

- Submit **lexer.cpp** and **DESIGN.md** on **Gradescope**. The filenames should match exactly.
- After submission, the autograder will provide feedback, which you can use to make required modifications.
- Hidden test cases will also be run against your submission. Full score on autograder cannot guarantee full score on the assignment, hence test extensively.

## How does the autograder work?

When you try to upload your assignment, you will see the following screen:

### Submit Programming Assignment

Upload all files for your submission

\* Required field

Submission Method

☒ Upload ☐ GitHub ☐ Bitbucket

Add files via Drag & Drop or [Browse Files](#).

Name	Size	Progress	✕
DESIGN.md	0 b	<div></div>	✕
lexer.cpp	3.9 KB	<div></div>	✕

Leaderboard Name \*  
SC

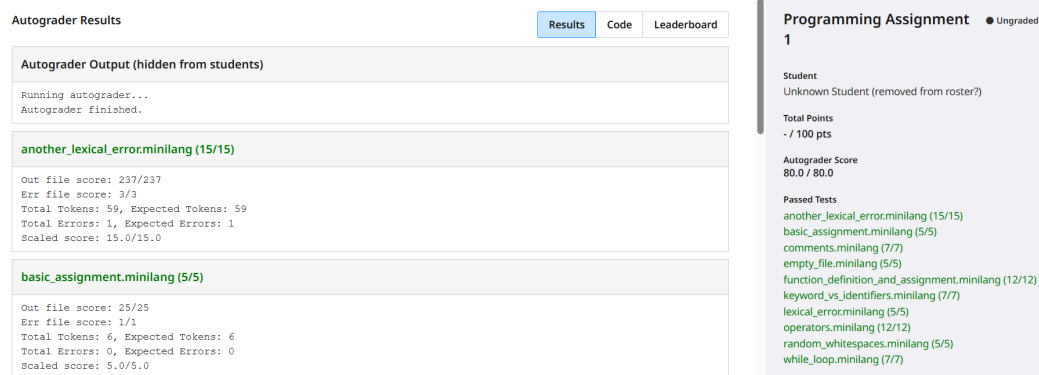
Submitting For  
Shivam Chaturvedi

Cancel

Upload

You just need to upload the **lexer.cpp** and **DESIGN.md** files, with these exact names.

After uploading, you should see the results of the autograder run in some time:



Autograder Results

Results Code Leaderboard

Autograder Output (hidden from students)

Running autograder...  
Autograder finished.

another\_lexical\_error.minilang (15/15)

Out file score: 237/237  
Err file score: 3/3  
Total Tokens: 59, Expected Tokens: 59  
Total Errors: 1, Expected Errors: 1  
Scaled score: 15.0/15.0

basic\_assignment.minilang (5/5)

Out file score: 25/25  
Err file score: 1/1  
Total Tokens: 6, Expected Tokens: 6  
Total Errors: 0, Expected Errors: 0  
Scaled score: 5.0/5.0

Programming Assignment 1

Student  
Unknown Student (removed from roster?)

Total Points  
- / 100 pts

Autograder Score  
80.0 / 80.0

Passed Tests

another\_lexical\_error.minilang (15/15)  
basic\_assignment.minilang (5/5)  
comments.minilang (7/7)  
empty\_file.minilang (5/5)  
function\_definition\_and\_assignment.minilang (12/12)  
keyword\_vs\_identifiers.minilang (7/7)  
lexical\_error.minilang (5/5)  
operators.minilang (12/12)  
random\_whitespace.minilang (5/5)  
while\_loop.minilang (7/7)

There will be some feedback at test cases level. The autograder scores out of 80. Total points will be shown as - / 100 points, which is expected, since we would be manually grading the DESIGN.md files.

Use the feedback from the autograder to fix your lexer implementation.

## 5 Testing

To aid in your understanding of the requirements, we have provided some sample inputs and corresponding sample output and error streams. To run these tests, use the following *make* target:

*make test*

This will run your code against the inputs present under *test/input* and generate results under *test/result*. You can use this testing framework to create your own test inputs and outputs. Below, I am attaching an example of a sample input and corresponding output:

### Input:

```
var y := 20;
```

### Output:

```
Keyword('var') at 1:1  
Identifier('y') at 1:5  
AssignOp(':=') at 1:7  
Integer('20') at 1:10  
Delimiter(';') at 1:12  
EOF('') at 1:12
```

## 6 Logistics

1. *test/* directory has some sample tests and their outputs

2. Utilize EdStem and TA office hours for any doubts and questions.
3. No late submissions are accepted.
4. Please refer to the deadlines as available on GradeScope.
5. Start early.

## 7 Hints

1. Increment column on each character, increment line on newline and reset column to 0. This ensures precise token or error locations.
2. Ignore all whitespace characters and newlines in tokenization; they only serve to separate tokens and do not produce tokens themselves.
3. **MiniLang** uses single-line comments starting with '#', which extend to the end of the line and should be completely ignored during tokenization.
4. Feel free to choose your own workflow for your scanner. You are not required to perform Regex, NFA or DFA conversions for generating tokens.
5. **Maximal munch** means always matching the longest possible token at each step — for example, parse 3456 as one integer token rather than several single-digit tokens.
6. Add a final EndOfFile token to clearly mark the end of input.
7. <https://cplusplus.com/reference/istream/istream/get/> For character reading, consider `istream::get()` for straightforward stream processing.

This assignment was designed by Shivam Chaturvedi, Teaching Assistant, COMSW4115, Fall 2025.