

# Phase 1 Report — Lexical Analyzer (Scanner) for the C-minus Compiler

## 1 Introduction

This report describes the implementation of the lexical analyzer (scanner) for the C-minus compiler. The scanner reads characters from an input file, classifies them into tokens, identifies lexical errors, and constructs a symbol table. The driver program `compiler.py` produces the output files:

- `tokens.txt`
- `lexical_errors.txt`
- `symbol_table.txt`

The scanner is implemented as a Python class providing a streaming token interface via `get_next_token()`, used by subsequent compiler phases.

## 2 Lexical Specifications of C-minus

### 2.1 Token Types

The scanner recognizes the following classes:

Token Type	Description
NUM	Digit sequence <code>[0--9]+</code>
ID	Identifier: <code>[A-Za-z] [A-Za-z0-9]*</code>
KEYWORD	<code>if, else, void, int, repeat, break, until, return</code>
SYMBOL	<code>; , [ ] ( ) { } + - * = == &lt; /</code>
COMMENT	<code>/* ... */</code> (ignored)
WHITESPACE	space, tab, newline, carriage return

Whitespace is skipped and comments are consumed entirely.

## 3 Scanner Architecture

### 3.1 Modules

The scanner consists of:

- `tokens_definitions.py`: keyword and symbol definitions.
- `scanner.py`: implements lexing logic.
- `compiler.py`: driver that writes output files.

## 3.2 Tokenization Logic

The scanner:

- processes input one character at a time,
- skips whitespace,
- detects comments of the form `/* ... */`,
- uses lookahead to distinguish `=` from `==`,
- classifies lexemes using helper methods:
  - `_number_or_error()`
  - `_identifier_or_keyword()`
  - `_symbol_or_error()`

Tokens are returned as pairs `(TOKEN_TYPE, lexeme)`.

## 4 Lexical Error Handling

The scanner identifies:

- **Invalid input characters**, e.g. `#`, `@`.
- **Unmatched comment closer `*/`**.
- **Unclosed comments** at end-of-file.
- **Invalid numbers** such as `3a` or `12abc`.
- **Composite invalid lexemes**.

Errors are recorded with line numbers and written by `compiler.py`. If no errors occur, `lexical_errors.txt` contains:

There is no lexical error.

## 5 Symbol Table Construction

Keywords are inserted first with ascending indices. Identifiers are inserted upon first encounter. `symbol_table.txt` lists entries as:

1. `int`
2. `void`
3. `main`
- ...

## 6 Output Files

### 6.1 tokens.txt

For each input line:

```
line_number. (TOKEN_TYPE, lexeme) (TOKEN_TYPE, lexeme) ...
```

This matches the output format in `compiler.py`.

### 6.2 lexical\_errors.txt

Contains formatted error messages or “There is no lexical error.”

### 6.3 symbol\_table.txt

Contains ordered entries of keywords and identifiers.

## 7 Testing and Validation

Test cases included:

- whitespace-less programs,
- malformed comments,
- invalid identifiers and numbers,
- lookahead-dependent symbol sequences.

A comparison tool `compare_tokens.py` was used to compare scanner output with an ANTLR reference.

## 8 Challenges and Solutions

During the development of the scanner, several practical and conceptual challenges were encountered, similar to those described in the original project documentation and internal project report [?]. The most important challenges were:

### 8.1 Correct Token Boundary Detection

Distinguishing between valid tokens and malformed ones (e.g., `23apple`, `2milk`, `var!name`) required carefully designed regular expressions and fallback logic. To address this, the scanner consumes the entire problematic lexeme and categorizes it as an invalid number or invalid token instead of producing multiple partial tokens.

### 8.2 Handling Complex Lexical Errors

Incorrect comment delimiters (such as unmatched `*/`) and unclosed comments were a major source of errors in early versions. We implemented a robust comment consumer that tracks the opening delimiter, scans until closure, and generates the correct error message with a lexeme preview for unclosed comments.

### 8.3 Managing Character-by-Character Scanning

Instead of using fully regex-driven tokenization, the scanner uses a manual pointer-based approach. This required careful tracking of:

- the current index,
- line breaks,
- lookahead for composite symbols (e.g., ==).

This design gives fine-grained control but increases implementation complexity.

### 8.4 Ensuring Consistency With ANTLR Output

Using ANTLR as a reference scanner revealed subtle differences between implementations. The `compare_tokens.py` tool was developed to automatically compare token streams and ensure consistency, helping to catch edge cases missed during manual testing.

### 8.5 Output Formatting and Error Standardization

The output formats for `tokens.txt`, `symbol_table.txt`, and `lexical_errors.txt` had to strictly follow the specification to allow automated grading. Ensuring consistent formatting (e.g., periods after line numbers, error message structure) required multiple iterations and refinements.

## 9 Conclusion

The Phase 1 lexical analyzer for C-minus correctly recognizes tokens, handles comments and whitespace, detects lexical errors, and constructs a symbol table. It produces all required output files and integrates seamlessly with later compiler phases.

## Contributors

Herbod Pourali, Mohammad-Saeed Arvanaghi, and Ilya Farhangfar

## References

- [1] Aho, Alfred V., Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. 2nd Edition, Addison-Wesley.
- [2] Terence Parr. *The ANTLR Parser Generator Documentation*.