Lexical Analysis

Mini-C Compiler

CS4031 - Compiler Construction | Fall 2025

Team Members

Subhan (22K-4316)

Sadiq (22K-4303)

Muhammad Ahmed Haque (22K-4232)

Muhammad Irtiza (22K-4638)

Umer (22K-4160)

Junaid (22K-4369)

December 2025

1. Overview

The lexer (lexical analyzer) is the first phase of the Mini-C compiler. It converts the source code (a stream of characters) into a stream of tokens that are passed to the parser.

Location: src/frontend/lexer/lexer.py

2. Token Definitions

2.1 Token Categories

Category	Token Name	Pattern	Example
Keywords	int_keyword	int\b	int
Keywords	return_keyword	return\b	return
Keywords	if_keyword	\bif\b	if
Keywords	while	\bwhile\b	while
Keywords	for	\bfor\b	for
Literals	Constant	[0-9]+	42
Literals	float_constant	[0-9]*\.[0-9]+	3.14
Literals	char_constant	'...'	'a'
Operators	Addition	\+	+
Operators	Equal	==	==
Operators	LessOrEqual	<=	<=
Delimiters	Semicolon	;	;
Delimiters	Open_brace	{	{
Identifiers	Identifier	[a-zA-Z_]\w*	sum

3. DFA (Deterministic Finite Automaton)

3.1 Main DFA States Description

The lexer uses a DFA-based approach where:

- START state: Initial state, dispatches based on first character
- IDENTIFIER states (q1): Accumulates alphanumeric characters
- NUMBER states (q2-q5): Handles integers, floats, suffixes
- STRING/CHAR states: Handles quoted literals with escapes
- OPERATOR states: Handles multi-character operators

3.2 Transition Table for Identifiers

Current State	Input	Next State	Action
START	[a-zA-Z_]	q1	Begin identifier
q1	[a-zA-Z0-9_]	q1	Continue identifier
q1	other	ACCEPT	Check if keyword

3.3 Transition Table for Numbers

Current State	Input	Next State	Action
START	[0-9]	q2	Begin integer
q2	[0-9]	q2	Continue integer
q2	.	q3	Begin float
q2	[uU]	ACCEPT_UINT	Unsigned int
q2	[lL]	q4	Long suffix
q3	[0-9]	q3	Continue float
q3	[eE]	q5	Begin exponent

4. DFA State Diagram

The following shows the DFA structure for the Mini-C lexer:

```
START ----[a-zA-Z_]----> q1 (IDENTIFIER) --[a-zA-Z0-9_]*--> ACCEPT_ID
  |                                                  |
  +----[0-9]---------> q2 (INTEGER) --[0-9]*---> ACCEPT_INT
  |                       |
  |                       +--[.]-----> q3 (FLOAT) --> ACCEPT_FLOAT
  |                       +--[uU]----> ACCEPT_UINT
  |                       +--[lL]----> q4 -----> ACCEPT_LONG
  |
  +----[']-----------> q5 (CHAR_START) --> ACCEPT_CHAR
  +----["]-----------> q6 (STRING_START) --> ACCEPT_STRING
  +----[+]-----------> ACCEPT_PLUS
  +----[-]-----------> q7 --[-]--> ACCEPT_DECREMENT
  |                       +--[>]--> ACCEPT_ARROW
  +----[<]-----------> q8 --[=]--> ACCEPT_LE
  |                       +------> ACCEPT_LT
  +----[=]-----------> q9 --[=]--> ACCEPT_EQ
                          +------> ACCEPT_ASSIGN
```

5. Lexer Implementation

5.1 Algorithm

```
def lex(source_code):
    tokens = []
    position = 0

    while position < len(source_code):
        # Skip whitespace
        if is_whitespace(source_code[position]):
            position += 1
            continue

        # Try each pattern in priority order
        for token_type, pattern in patterns:
            match = regex_match(pattern, source_code, position)
            if match:
                tokens.append((token_type, match.group()))
                position = match.end()
                break
        else:
            raise LexicalError(f"Invalid character")

    return tokens
```

5.2 Token Priority

Patterns are matched in the following priority order:

- Keywords (matched before identifiers)
- Multi-character operators (<=, >=, ==, !=, &&, ||, ->)
- Single-character operators
- Literals (numbers, strings, characters)
- Identifiers

6. Example Lexical Analysis

Input:

```
int factorial(int n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}
```

Output Token Stream:

```
(int_keyword, 'int')
(Identifier, 'factorial')
(Open_parenthesis, '(')
(int_keyword, 'int')
(Identifier, 'n')
(Close_parenthesis, ')')
(Open_brace, '{')
(if_keyword, 'if')
(Open_parenthesis, '(')
(Identifier, 'n')
(LessOrEqual, '<=')
(Constant, '1')
(Close_parenthesis, ')')
(return_keyword, 'return')
(Constant, '1')
(Semicolon, ';')
(return_keyword, 'return')
(Identifier, 'n')
(Multiplication, '*')
(Identifier, 'factorial')
(Open_parenthesis, '(')
(Identifier, 'n')
(Negation, '-')
(Constant, '1')
(Close_parenthesis, ')')
(Semicolon, ';')
(Close_brace, '}')
```

7. Error Handling

Error Type	Example	Error Message
Invalid character	@	Unexpected character '@'
Unterminated string	"hello	Unterminated string literal
Unterminated char	'a	Unterminated character constant
Invalid escape	'\z'	Invalid escape sequence