A black background with grey leaves

AI-generated content may be incorrect.

Lexical Analyzer

Build Scanner

A grey logo on a black background

AI-generated content may be incorrect.

**Prepared By**

Omar Mahmoud Ibrahim

Id:200042847

**Under Supervision**

Dr. Nahal Abdel Salam

**Lexical Analyzer Report**

**1. Introduction**

A compiler is a program that translates source code from a high-level programming language into machine code. The compilation process consists of multiple phases, with Lexical Analysis being one of the most critical. This report presents an implementation of a lexical analyzer that processes a subset of the C++ programming language.

**1.1 Phases of Compiler**

A compiler operates in multiple phases, each responsible for a different aspect of translation.

|  |  |
| --- | --- |
| Phase | Description |
| Lexical Analysis | Converts source code into tokens. |
| Syntax Analysis | Checks grammatical correctness and builds a parse tree. |
| Semantic Analysis | Ensures valid meaning and detects type errors. |
| Intermediate Code Generation | Produces an intermediate representation. |
| Code Optimization | Improves performance by reducing redundancies. |
| Code Generation | Converts optimized code to machine code. |
| Error Handling | Identifies and reports errors. |

**2. Lexical Analyzer**

A Lexical Analyzer reads the source code character by character and groups them into meaningful tokens.

**Table 2: Token Categories**

|  |  |
| --- | --- |
| Token Type | Examples |
| Keywords | int, float, if, else, return |
| Identifiers | variable names like x, myFunction |
| Numbers | 10, 3.14 |
| Strings | "Hello World" |
| Operators | +, -, \*, /, =, <, >, ==, != |
| Special Symbols | {, }, (, ), [, ], ;, , |

**3. Software Tools**

**3.1 Computer Program**

- Python 3.x: Used for implementing the lexical analyzer.  
- Regular Expressions (re module): For pattern matching.  
- Text Editor (VS Code, PyCharm, etc.): For writing and testing the code.

**3.2 Programming Language**

The lexical analyzer is designed to analyze the c++ lexiems into token using Python programming language

**4. Implementation of a Lexical Analyzer**

**4.1 Lexical Analyzer Code in Python**

import re  
  
TOKEN\_TYPES = {  
 "KEYWORD": r"\b(int|float|if|else|while|for|return|void|char|string|double)\b",  
 "IDENTIFIER": r"\b[a-zA-Z\_][a-zA-Z0-9\_]\*\b",  
 "NUMBER": r"\b\d+(\.\d+)?\b",  
 "STRING": r'"[^"]\*"',  
 "OPERATOR": r"[+\-\*/=<>!]+",  
 "SPECIAL\_SYMBOL": r"[{}()\[\],;]",  
}  
  
def lexical\_analyzer(code):  
 tokens = []  
 combined\_regex = "|".join(f"(?P<{name}>{pattern})" for name, pattern in TOKEN\_TYPES.items())  
 for match in re.finditer(combined\_regex, code):  
 token\_type = match.lastgroup  
 lexeme = match.group(token\_type)  
 tokens.append((lexeme, token\_type))  
 return tokens  
  
cpp\_code = """  
int main() {  
 int x =3;  
 for(int i =0; i<3;i++){  
 x++;  
 }  
 cout<<x;  
 return 0;  
}  
"""

tokens = lexical\_analyzer(cpp\_code)  
print("\nTokens:")  
for lexeme, token\_type in tokens:  
 print(f"{lexeme} --> {token\_type}")

**Explanation of Lexical Analyzer Code**

**1. Importing Required Module**

The `re` module provides support for regular expressions, which are used for pattern matching to identify different types of tokens.

```python

import re

```

**2. Defining Token Patterns**

Each token type is defined using a regular expression pattern:

- \*\*KEYWORD\*\*: Matches C++ reserved words like `int`, `float`, `return`, etc.

- \*\*IDENTIFIER\*\*: Matches variable names and function names.

- \*\*NUMBER\*\*: Matches integers and floating-point numbers.

- \*\*STRING\*\*: Matches text within double quotes.

- \*\*OPERATOR\*\*: Matches operators such as `+`, `-`, `\*`, `=`, etc.

- \*\*SPECIAL\_SYMBOL\*\*: Matches punctuation symbols used in C++.

```python

TOKEN\_TYPES = {

"KEYWORD": r"\b(int|float|if|else|while|for|return|void|char|string|double)\b",

"IDENTIFIER": r"\b[a-zA-Z\_][a-zA-Z0-9\_]\*\b",

"NUMBER": r"\b\d+(\.\d+)?\b",

"STRING": r'"[^"]\*"',

"OPERATOR": r"[+\-\*/=<>!]+",

"SPECIAL\_SYMBOL": r"[{}()\[\],;]",

}

```

**3. Lexical Analyzer Function**

The function takes code as input and uses the regex pattern to identify all tokens.

```python

def lexical\_analyzer(code):

tokens = []

combined\_regex = "|".join(f"(?P<{name}>{pattern})" for name, pattern in TOKEN\_TYPES.items())

for match in re.finditer(combined\_regex, code):

token\_type = match.lastgroup

lexeme = match.group(token\_type)

tokens.append((lexeme, token\_type))

return tokens

```

**4. Sample Code Input**

This sample C++ code is passed to the lexical analyzer.

```python

cpp\_code = '''

int main() {

int x = 10;

float y = 3.14;

if (x < y) {

x = x + 1;

}

return 0;

}

'''

```

**5. Output Tokens**

The output prints each token in the form of `lexeme --> token\_type`.

```python

tokens = lexical\_analyzer(cpp\_code)

print("\nTokens:")

for lexeme, token\_type in tokens:

print(f"{lexeme} --> {token\_type}")

**4.2 Sample Output**

Tokens:

int --> KEYWORD

main --> IDENTIFIER

( --> SPECIAL\_SYMBOL

) --> SPECIAL\_SYMBOL

{ --> SPECIAL\_SYMBOL

int --> KEYWORD

x --> IDENTIFIER

= --> OPERATOR

3 --> NUMBER

; --> SPECIAL\_SYMBOL

for --> KEYWORD

( --> SPECIAL\_SYMBOL

int --> KEYWORD

i --> IDENTIFIER

= --> OPERATOR

0 --> NUMBER

; --> SPECIAL\_SYMBOL

i --> IDENTIFIER

< --> OPERATOR

3 --> NUMBER

; --> SPECIAL\_SYMBOL

i --> IDENTIFIER

++ --> OPERATOR

) --> SPECIAL\_SYMBOL

{ --> SPECIAL\_SYMBOL

x --> IDENTIFIER

++ --> OPERATOR

; --> SPECIAL\_SYMBOL

} --> SPECIAL\_SYMBOL

cout --> IDENTIFIER

<< --> OPERATOR

x --> IDENTIFIER

; --> SPECIAL\_SYMBOL

return --> KEYWORD

0 --> NUMBER

; --> SPECIAL\_SYMBOL

} --> SPECIAL\_SYMBOL

**5. References**

1. Compilers: Principles, Techniques, and Tools – Aho, Lam, Sethi, Ullman.  
2. Python Regular Expressions (re Module) – Python Documentation.  
3. Lexical Analysis in C++ and Python – GeeksforGeeks.  
4. Compiler Design – Lexical Analysis – TutorialsPoint.