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**

Student Name Student ID

**Under Supervision**

Name of Doctor

Name of T. A.

## **How to Build a Lexical Analyzer**

A Lexical Analyzer is the first phase in the compilation process, converting source code into a stream of tokens. Here are the steps to build one:

1. Define Basic Concepts

**What is a Lexical Analyzer?**

* It reads source code character by character and converts it into tokens such as keywords, identifiers, numbers, operators, and delimiters.
* It removes whitespace and comments and detects errors like invalid characters.

|  |  |
| --- | --- |
| TYPE | EXAMPLES |
| Keywords | int, if, return |
| Identifiers | variableName, x, y |
| Operators | +, -, \*, / |
| Constants | 100, 3.14, 'a' |
| Delimiters | ;, ,, {}, () |

2. Steps to Build a Lexical Analyzer

**Step 1: Define Patterns Using Regular Expressions (Regex)**

* Use regex to define different token types like keywords and identifiers.

// Example source code

std::string code = "int value = 100;";

This line is divided into:

* int → Keyword
* value → Identifier
* = → Operator
* 100 → Constant
* ; → Delimiter

**Step 2: Build DFA or NFA**

* Represent patterns as Finite Automata to improve efficiency.

**Step 3: Write the Code**

**Example in C++ to Build a Lexical Analyzer**

#include <iostream>

#include <string>

#include <cctype>

enum TokenType {

KEYWORD,

IDENTIFIER,

NUMBER,

OPERATOR,

DELIMITER,

UNKNOWN

};

struct Token {

TokenType type;

std::string value;

};

class Lexer {

public:

Lexer(const std::string& input) : input(input), pos(0) {}

Token getNextToken() {

while (pos < input.size() && std::isspace(input[pos])) {

pos++;

}

if (pos >= input.size()) {

return {UNKNOWN, ""};

}

char c = input[pos];

if (std::isdigit(c)) {

std::string num;

while (pos < input.size() && std::isdigit(input[pos])) {

num += input[pos];

pos++;

}

return {NUMBER, num};

} else if (std::isalpha(c)) {

std::string id;

while (pos < input.size() && (std::isalnum(input[pos]) || input[pos] == '\_')) {

id += input[pos];

pos++;

}

if (id == "int" || id == "if" || id == "return") {

return {KEYWORD, id};

} else {

return {IDENTIFIER, id};

}

} else if (c == '+' || c == '-' || c == '\*' || c == '/') {

pos++;

return {OPERATOR, std::string(1, c)};

} else if (c == ';' || c == ',' || c == '{' || c == '}' || c == '(' || c == ')') {

pos++;

return {DELIMITER, std::string(1, c)};

} else {

pos++;

return {UNKNOWN, std::string(1, c)};

}

}

private:

std::string input;

size\_t pos;

};

int main() {

std::string code = "int x = 10;";

Lexer lexer(code);

Token token;

while ((token = lexer.getNextToken()).type != UNKNOWN) {

std::cout << "Token Type: ";

switch (token.type) {

case KEYWORD:

std::cout << "Keyword";

break;

case IDENTIFIER:

std::cout << "Identifier";

break;

case NUMBER:

std::cout << "Number";

break;

case OPERATOR:

std::cout << "Operator";

break;

case DELIMITER:

std::cout << "Delimiter";

break;

default:

std::cout << "Unknown";

break;

}

std::cout << ", Value: " << token.value << std::endl;

}

return 0;

}

3. Error Handling

* If the analyzer encounters an invalid character, it displays an error message with the line and position.

4. Tools to Help Build a Lexical Analyzer

**Using Flex**

%%

int { printf("KEYWORD: %s\n", yytext); }

[a-zA-Z]+ { printf("IDENTIFIER: %s\n", yytext); }

[0-9]+ { printf("NUMBER: %s\n", yytext); }

%%

5. Performance Optimization

* Use DFA instead of NFA for better speed and efficiency.

**Comparison Table Between NFA and DFA**

|  |  |  |
| --- | --- | --- |
| Criterion | NFA | DFA |
| Speed | Slower | Faster |
| Memory Usage | Less | More |
| Complexity | Easier to build | More complex |

6. Final Output

The output is a stream of tokens sent to the parser for further processing.

By following these steps, you can build a lexical analyzer for a specific language using various tools and techniques.