# Laboratory work nr.3
# Lexer

Course: Formal Languages and Finite Automata

Author: Perlog Boris FAF-221

## Objectives

1. Understand what is lexical analysis and how it works.
2. Implement a lexer and explain how it works.

# 1. The TokenType enum

To describe tokens effectively a **TokenType** custom type is going to be needed. **TokenType** is an enum that stores what type a token is whether it is an operator, literal or keyword.

```
enum TokenType {
	// Punctuation
	LEFT_BRACE, RIGHT_BRACE, LEFT_PAREN, RIGHT_PAREN,
	COMMA, DOT, SEMICOLON,

	// Operators
	PLUS, MINUS, SLASH, STAR, NOT, NOT_EQUAL,
	EQUAL, EQUAL_EQUAL, LESS, LESS_EQUAL,
	GREATER, GREATER_EQUAL,

	// Literals
	IDENTIFIER, STRING, NUMBER,

	// Keywords
	IF, ELSE, FOR, WHILE, TRUE, FALSE, INT, FLOAT, RETURN, THIS
};
```

# 2. The Token class

The **Token** class contains all relevant information about a token like its type, its associated lexeme, and the line in the source code where it appears.

It has two static maps that map from **TokenType** to **string** and vice-versa. These are helper structures that will come in handy later in conjunction with functions like **PrintToken()** or **IsKeyWord()** which tests if an identifier is a reserved keyword, and **GetKeyworkTokenType()** which returns a keyword token type if a compatible lexeme was found.

```
class Token {
public:
	Token(TokenType type, uint32_t line, const std::string& lexeme = "\0")
		: m_Type(type), m_Line(line), m_Lexeme(lexeme) {}

	void PrintToken() { std::cout << "Type: " << TokenTypeToString[m_Type] <<
" Lexeme: " << m_Lexeme
		<< " Line: " << m_Line << "\n"; }
	inline static bool IsKeyWord(std::string str) { return (Keywords.find(str)
!= Keywords.end()); }
	static TokenType GetKeywordTokenType(std::string& keyword) { return
Keywords[keyword]; }
private:
private:
	TokenType m_Type;
	std::string m_Lexeme;
	uint32_t m_Line;

	static std::map<TokenType, std::string> TokenTypeToString;
	static std::map<std::string, TokenType> Keywords;
};
```

# 3. The Lexer class

The **Lexer** class is the core of this laboratory, a **Lexer** object is constructed using a string that represents the source code. The source code is stored in the **m_Source** field and the tokens are to be stored in the **m_Tokens** vector.

Three unsigned integers are used to keep track of the lexer scanning process.

```
class Lexer {
public:
	Lexer(const std::string& source)
		: m_Source(source) {}

	void ScanTokens();

	void PrintTokens();
private:
	void ScanToken();
	void AddToken(TokenType type);
	void AddToken(TokenType type, const std::string& lexeme);

	bool IsAtEnd() { return !(m_CurrentChar < m_Source.length()); }
	bool IsCurrentChar(char c);
	inline char Peek() { return m_Source[m_CurrentChar]; }
	inline char PeekNext() { return m_Source[m_CurrentChar + 1]; }
	void String();
	void Number();
	void Identifier();
private:
	std::string m_Source;
	std::vector<Token> m_Tokens;

	uint32_t m_StartChar = 0;
	uint32_t m_CurrentChar = 0;
	uint32_t m_Line = 1;
};
```

The most important functions are **ScanTokens()** and **ScanToken()**. The first one scans the entire source code calling the former one when a lexeme is encountered. **AddToken()** is responsible for adding a new token into the **m_Tokens** vector.

The function **String()** will be called when a string lexeme is encountered and is responsible of converting that into a string token. Functions **Number()** and **Identifier()** have similar purposes but for numbers and identifiers lexemes respectivelly.

The rest of the functions are small and simple helper functions meant to simplify our core functionality code and make it more readable.

## 4. Lexer's inner workings

The scanning starts with the **ScanTokens()** function.

```
void Lexer::ScanTokens()
{
	while (!IsAtEnd())
	{
		m_StartChar = m_CurrentChar;
		ScanToken();
	}
}
```

The **ScanToken()** function is where each encountered lexeme is analyzed and transformed into a token. Every time it is called it consumes the current character of the source code and starts its analysis in a switch statement.
First, it checks for the punctuators and operators of length one. Then it checks for operators of length two like ==, !=, <=, or >=.
After that, it checks for comments and ignores white spaces, while also incrementing the **m_Line** field when a new line character is encountered.

```
void Lexer::ScanToken()
{
	char c = m_Source[m_CurrentChar];
	m_CurrentChar++;
	switch (c)
	{
	case '{': AddToken(LEFT_BRACE); break;
	case '}': AddToken(RIGHT_BRACE); break;
	case '(': AddToken(LEFT_PAREN); break;
	case ')': AddToken(RIGHT_PAREN); break;
	case ',': AddToken(COMMA); break;
	case '.': AddToken(DOT); break;
	case '-': AddToken(MINUS); break;
	case '+': AddToken(PLUS); break;
	case ';': AddToken(SEMICOLON); break;
	case '*': AddToken(STAR); break;
	case '!':
		AddToken(IsCurrentChar('=') ? NOT_EQUAL : NOT);
		break;
	case '=':
		AddToken(IsCurrentChar('=') ? EQUAL_EQUAL : EQUAL);
		break;
	case '<':
		AddToken(IsCurrentChar('=') ? LESS_EQUAL : LESS);
		break;
	case '>':
		AddToken(IsCurrentChar('=') ? GREATER_EQUAL : GREATER);
		break;
	case '/':
		if (IsCurrentChar('/')) {
			// advace the current char pointer until a new line (ignore
the comment)
			while (m_Source[m_CurrentChar] != '\n' && !IsAtEnd())
				m_CurrentChar++;
		}
		else {
			AddToken(SLASH);
		}
		break;
	case ' ':
```

```
		case '\r':
		case '\t':
			// Ignore whitespace.
			break;
		case '\n':
			m_Line++;
			break;
		case '"': String(); break;
		default:
			if (isdigit(c))
				Number();
			else if (isalpha(c))
				Identifier();
			else
				std::cout << "Unexpected character '" << c << "' at line : "
<< m_Line << "\n";
			break;
		}
}
```

If the current lexeme does not pass any of the described above checks it must be a string, number, identifier, or an invalid character. The invalid character case is handled at the very bottom end of the checklist.

If the lexeme seems to be a string, number, or identifier the task of further analysis is passed onto one of the appropriate functions **String()**, **Number()**, or **Identifier()**.

The **String()** function consumes character until it reaches the ‘ “ ‘ symbol representing the end of the string. If it reaches the end of the source code and no terminating quotes are found an “Unterminated string!” error is displayed.

In case of a valid string encounter a lexeme is constructed and a string type token is created.

```
void Lexer::String()
{
	while (Peek() != '"' && !IsAtEnd())
	{
		if (Peek() == '\n')
			m_Line++;
		m_CurrentChar++;
	}
	if (IsAtEnd()) {
		std::cout << "Unterminated string!\n";
		return;
	}
	// Closing "
	m_CurrentChar++;

	std::string lexeme = m_Source.substr(m_StartChar + 1, m_CurrentChar -
m_StartChar - 2);
	AddToken(STRING, lexeme);
}
```

The **Number()** function consumes all digit characters if it encounters a '.' It starts consuming the fractional part of the number.
Similarly to the **String()** function the lexeme is extracted from the source code and a number type token is created.

```
void Lexer::Number()
{
	while (isdigit(Peek()))
		m_CurrentChar++;
	if (Peek() == '.' && isdigit(PeekNext())) {
		m_CurrentChar++;
		while (isdigit(Peek()))
			m_CurrentChar++;
	}

	std::string lexeme = m_Source.substr(m_StartChar, m_CurrentChar -
m_StartChar);
	AddToken(NUMBER, lexeme);
}
```

The **Identifier()** function is called when an alphabetical character is encountered, however further characters may be digits or the underscore character. After all valid characters are consumed an identifier string is created.
Here the functions and map structures mentioned above in the **The Token class** section come in handy to determine if the identifier is a keyword.
A check is made to determine if the identifier is found in the **Keywords** map that maps strings of keyword names to keyword **TokenType**s. If the identifier is a keyword the **GetKeywordTokenType()** function is used to retrieve the appropriate **TokenType**.
Then a new token is added with the appropriate type.

```
void Lexer::Identifier()
{
	while (isalpha(Peek()) || isdigit(Peek()) || Peek() == '_')
		m_CurrentChar++;

	std::string identifier = m_Source.substr(m_StartChar, m_CurrentChar -
m_StartChar);
	TokenType type = TokenType::IDENTIFIER;
	if (Token::IsKeyWord(identifier))
		type = Token::GetKeywordTokenType(identifier);

	AddToken(type);
}
```

## Main Function

In the **main()** function a **Lexer** object is created with the input source code. Then tokens are scanned and printed to the console.

The input contains three lines of code, a string, two numbers, one invalid character, three keywords, one identifier, and several punctuators and operators.

```
std::string source_code = " \"my string\" text// this is a comment\n(()) {} //
grouping \"inside\" stuff\n!*+-/= <54.12> if else<= # 2==for // operators";
int main()
{
    std::cout << "Source code:\n\n" << source_code << "\n\n";
    Lexer lexer(source_code);
    lexer.ScanTokens();
    lexer.PrintTokens();

    return 0;
}
```

```
Source code:

 "my string" text// this is a comment
(()) {} // grouping "inside" stuff
!*+-/= <54.12> if else<= # 2==for // operators

Unexpected character '#' at line : 3
Type: string Lexeme: my string Line: 1
Type: identifier Lexeme: text Line: 1
Type: ( Lexeme:  Line: 2
Type: ( Lexeme:  Line: 2
Type: ) Lexeme:  Line: 2
Type: ) Lexeme:  Line: 2
Type: { Lexeme:  Line: 2
Type: } Lexeme:  Line: 2
Type: ! Lexeme:  Line: 3
Type: * Lexeme:  Line: 3
Type: + Lexeme:  Line: 3
Type: - Lexeme:  Line: 3
Type: / Lexeme:  Line: 3
Type: = Lexeme:  Line: 3
Type: < Lexeme:  Line: 3
Type: number Lexeme: 54.12 Line: 3
Type: > Lexeme:  Line: 3
Type: if Lexeme: if Line: 3
Type: else Lexeme: else Line: 3
Type: <= Lexeme:  Line: 3
Type: number Lexeme: 2 Line: 3
Type: == Lexeme:  Line: 3
Type: for Lexeme: for Line: 3
```

All of them were scanned successfully and are printed as **Token**s in the console.

# Conclusion

This laboratory work served as an introduction to how a compiler translated code from one language to another, by taking a detailed look at how a lexer works and implementing one.

The lexer is responsible for performing the lexical analysis part of a compilation process and serves to transform the raw source code into meaningful bits of information that can be used by the parser.

I observed how tokens may be of different types while also fitting into one of the broader categories like punctuators, operators, literals, and identifiers.

A custom type **TokenType** was used to keep track of various token types.

The lexer has to perform extensive checks on lexemes to put them into the right category and construct the right type of token while accounting for various language features like comments, and multi-line comments.

I used hash maps to achieve constant look-up time every time the lexer needs to check if an identifier is a keyword or retrieve the appropriate token type.

A lexer is a system that gets more and more complex as the language capabilities are expanded. Good software architecture is a must to ensure the future scalability of such a system.