📌 1. Conceito de Análise Léxica (Lexical Analysis)

A **análise léxica** é a **primeira fase** de um compilador. Sua função principal é **ler a sequência de caracteres de um programa-fonte** e **agrupar esses caracteres em unidades significativas chamadas *tokens***.

💡 O que são *tokens*?

Um **token** representa uma **categoria de símbolo léxico**, ou seja, uma unidade básica da linguagem de programação, como:

- **Palavras-chave**: `if`, `while`, `return`
- **Identificadores**: `x`, `total`, `soma_media`
- **Operadores**: `+`, `-`, `*`, `==`
- **Delimitadores**: `(`, `)`, `{`, `}`, `;`
- **Literais**: `123`, `3.14`, `'A'`, `"texto"`

📚 2. Funções da Análise Léxica

1. **Remover espaços em branco e comentários** irrelevantes para a sintaxe.
2. **Reconhecer padrões de tokens** usando expressões regulares ou autômatos.
3. **Informar erros léxicos**, como identificadores com caracteres inválidos ou números mal formados.
4. **Fornecer tokens para o analisador sintático**, geralmente por meio de uma interface que permite o próximo token (`get_next_token()`).

🔠 3. Diferença entre símbolo, lexema e token

Termo	Definição
Símbolo	Um caractere individual da entrada (`a`, `1`, `=`)
Lexema	Uma sequência concreta de caracteres (`int`, `x1`, `123`)
Token	A representação abstrata da categoria do lexema (`INT`, `IDENT`, `NUM`)

Exemplo:

```
int x = 10;
```

Lexema	Token
`int`	`KEYWORD_INT`
`x`	`IDENTIFIER`

Lexema	Token
`=`	`ASSIGN_OP`
`10`	`NUMBER`
`;`	`SEMICOLON`

⚙️ 4. Como funciona internamente?

a) Expressões Regulares

A linguagem léxica é definida com **expressões regulares**. Exemplos:

- Identificador: `[a-zA-Z_][a-zA-Z0-9_]*`
- Número inteiro: `[0-9]+`
- Espaços: `[ \t\n]+`

b) Autômatos Finitos

As expressões regulares são convertidas em **autômatos finitos determinísticos (DFA)** ou **não determinísticos (NFA)**, que são usados para percorrer o texto e identificar os tokens.

🧠 Importância da Análise Léxica

- **Simplifica** o trabalho das fases posteriores, especialmente a análise sintática.
- **Garante a clareza** e robustez do processo de tradução.
- **Permite modularização**, separando a lógica de reconhecimento de símbolos da lógica gramatical.

🔍 5. Etapas Internas da Análise Léxica

A análise léxica pode ser dividida em várias **subetapas** que ocorrem em sequência ou paralelamente durante o processo de leitura do código-fonte:

5.1. Leitura de Caracteres

A entrada é lida caractere por caractere, muitas vezes usando um **buffer** para melhorar a performance. O analisador mantém ponteiros para indicar:

- Início do lexema atual.
- Posição do caractere que está sendo analisado no momento.

5.2. Reconhecimento de Padrões

O analisador tenta **casar a sequência de caracteres lida com algum padrão definido** (geralmente via expressões regulares). Exemplo:

- Se a entrada for `while`, o analisador tenta casar com o padrão de palavra-chave.

Se houver mais de um casamento possível, geralmente se aplica a **regra do maior prefixo** (longest match rule) — ou seja, escolhe-se o maior lexema possível que forme um token válido.

5.3. **Geração de Token**

Uma vez reconhecido o padrão, o analisador **cria um token**, que normalmente inclui:

- O **tipo do token** (ex: `IDENTIFIER`, `KEYWORD`, `NUMBER`, etc.)
- O **valor léxico** (ex: o nome da variável ou o valor numérico)
- Opcionalmente: posição no código (linha, coluna)

5.4. **Tratamento de Erros Léxicos**

Quando o analisador encontra uma sequência inválida (como `@x` ou `1var`), ele pode:

- Ignorar o caractere e seguir.
- Sinalizar o erro e tentar recuperação.
- Parar a compilação.

🤖 6. **Formalismo Matemático: Linguagens Regulares**

Do ponto de vista teórico, a análise léxica trabalha com linguagens formais chamadas **linguagens regulares**, que são aquelas que podem ser descritas por:

- **Expressões regulares** (regex)
- **Autômatos finitos determinísticos (DFA)**
- **Autômatos finitos não determinísticos (NFA)**

Relação entre esses modelos:

- Toda **expressão regular** pode ser transformada em um **NFA**.
- Todo **NFA** pode ser convertido em um **DFA**.
- Todo **DFA** pode ser **minimizado** para obter uma forma mais eficiente.

🧰 7. **Ferramentas para Análise Léxica**

Diversas ferramentas automatizam a construção do analisador léxico com base em expressões regulares e regras definidas. Algumas delas:

7.1. **Lex (UNIX)**

Uma das ferramentas clássicas. Permite definir expressões regulares e ações em C. Usa-se com o `yacc` para análise sintática.

7.2. **Flex (Fast Lex)**

Versão moderna do Lex, mais eficiente.

7.3. **PLY (Python Lex-Yacc)**

Biblioteca em Python que permite criar analisadores léxicos e sintáticos diretamente em código Python.

Exemplo simples com `ply.lex`:

```
import ply.lex as lex

tokens = ('NUMBER', 'PLUS')

t_PLUS = r'\+'
t_NUMBER = r'\d+'

t_ignore = ' \t'

def t_error(t):
    print(f"Caractere ilegal: {t.value[0]}")
    t.lexer.skip(1)

lexer = lex.lex()

lexer.input("3 + 42")

for tok in lexer:
    print(tok)
```

🧠 8. **Desempenho e Otimizações**

- **Buffers duplos** são usados para evitar o custo de I/O a cada caractere.
- **Tabela de símbolos léxicos** pode ser criada durante a análise para armazenar identificadores, constantes, etc.
- **Caching e hashing** são usados para melhorar a busca por lexemas em tokens já reconhecidos.

📌 9. **Resumo dos Papéis do Analisador Léxico**

Papel	Descrição
Filtrar entrada	Remove espaços, tabulações, quebras de linha, comentários.
Agrupar caracteres	Identifica agrupamentos válidos (tokens).
Classificar tokens	Atribui rótulos de tipos de token.
Manter posição	Guarda linha e coluna para mensagens de erro.
Reportar erros léxicos	Detecta e sinaliza entradas malformadas.

📘 10. **Referências Clássicas e Acadêmicas**

A. **Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman**

> Obra: *Compiladores – Princípios, Técnicas e Ferramentas* (também conhecido como "O Livro do Dragão").

Este é o **livro-texto mais citado** na disciplina de compiladores. Os autores definem o processo de análise léxica como:

> “A análise léxica é a fase do compilador responsável por particionar a cadeia de entrada em unidades significativas chamadas tokens.”
> — Aho, Sethi, Ullman (1986)

Eles apresentam formalmente os **autômatos finitos determinísticos (DFA)** como estrutura básica para reconhecimento eficiente de padrões léxicos. No capítulo de análise léxica, os autores também discutem a **tradução de expressões regulares para autômatos finitos**, com algoritmos formais para a construção do autômato a partir da regex (ex: *construção de Thompson*).

B. **Andrew W. Appel**

> Obra: *Modern Compiler Implementation in Java / C / ML*

Appel enfatiza uma abordagem prática de construção de compiladores. Segundo ele:

> “O analisador léxico é um filtro, uma fase que processa os dados da entrada com uma granularidade maior, preparando-os para análise sintática.”
> — Appel (1997)

Appel também introduz o conceito de **geradores de analisadores léxicos** como ferramentas essenciais para engenheiros de compiladores, destacando a importância de separar **análise léxica (reconhecimento)** da **ação semântica (interpretação)**.

C. **Kenneth C. Louden**

> Obra: *Compiler Construction: Principles and Practice*

Louden tem uma abordagem bem didática. Ele explica o processo do scanner (analisador léxico) como:

> “O scanner é responsável por identificar os padrões léxicos, ignorar caracteres irrelevantes, e reconhecer erros primitivos no texto-fonte.”
> — Louden (1997)

Louden detalha implementações passo a passo de um analisador léxico usando C, incluindo técnicas de *backtracking*, *lookahead*, e gerenciamento de erros.

📐 11. **Modelos Formais Aplicados na Análise Léxica**

A teoria da computação serve como base sólida para a análise léxica:

🧩 Linguagens Regulares

A classe de linguagens reconhecíveis por **autômatos finitos** é exatamente a das **linguagens regulares**, que podem ser expressas por:

- Expressões regulares (regex)
- Gramáticas regulares (tipo 3 na hierarquia de Chomsky)

A relação entre essas representações é explorada por autores como:

- **Hopcroft, Motwani e Ullman**

> Obra: *Introduction to Automata Theory, Languages, and Computation*
> Este livro é referência clássica para formalismos como DFA, NFA, expressões regulares, e gramáticas.

🧪 12. **Reconhecimento de Tokens com Expressões Regulares**

No processo de construção de um lexer, cada tipo de token é definido por uma expressão regular:

Token	Expressão Regular
Identificador	`[a-zA-Z_][a-zA-Z0-9_]*`
Número inteiro	`[0-9]+`
Espaços	`[ \t\n\r]+`
Operadores	`\+|\-|\*|\/|==|=`

Estas expressões são **combinadas** em uma **única expressão regular grande**, e um **autômato finito** correspondente é criado para reconhecer os padrões.

🔄 13. **Conversão de Regex → NFA → DFA**

Construção de Thompson (Thompson's Construction):

É um método sistemático para transformar uma expressão regular em um **NFA**. Em seguida, usa-se o **algoritmo de subconjuntos** para converter o NFA em um **DFA** equivalente (determinístico), que é mais eficiente em tempo de execução.

Depois, o DFA pode ser **minimizado** para melhorar ainda mais o desempenho.

⚠ 14. **Tratamento de Erros Léxicos**

A detecção de erros léxicos ocorre quando uma sequência de caracteres **não corresponde a nenhum token válido**.

Autores como Louden e Aho sugerem estratégias como:

- **Pânico**: Ignorar o caractere ou grupo de caracteres até encontrar um separador como `;` ou `}`.
- **Correção de erros**: Tentar pequenas alterações como substituição de um caractere (`=` no lugar de `==`), inserção ou deleção.

O objetivo é **manter a compilação prosseguindo** ao máximo, fornecendo **diagnósticos úteis** ao programador.

📊 15. Tabela de Símbolos e Tokens

Durante a análise léxica, é comum usar uma **tabela de símbolos** para armazenar identificadores e literais (como strings e números). Cada entrada na tabela inclui:

- Nome
- Tipo de dado
- Escopo (em fases posteriores)
- Endereço na memória (em fases de geração de código)

Essa tabela pode ser implementada como uma **tabela hash**.

🧠 16. Resumo Conceitual (com autores)

Conceito	Definição	Autor
Token	Unidade básica de significado léxico	Aho et al.
Lexema	Sequência de caracteres que corresponde a um token	Aho et al.
Expressão regular	Representação de padrões lexicais	Hopcroft et al.
Autômato Finito Determinístico	Máquina de estados que reconhece tokens	Appel, Aho
Gerador léxico	Ferramenta para criar analisadores léxicos	Louden
Tabela de símbolos	Estrutura para armazenar informações de identificadores	Appel, Louden

---

Claro, Luis! Vamos construir um exemplo simples de **analisador léxico (lexer) em C**, **sem usar ferramentas como Lex ou Flex**, para que possamos ver como é o processo "na mão".

Esse analisador reconhecerá:

- **Palavras-chave**: `if`, `else`, `while`, `return`
- **Identificadores**: letras seguidas de letras/dígitos/underscore
- **Números inteiros**
- **Símbolos**: `+`, `-`, `*`, `/`, `=`, `;`, `(`, `)`

✅ Estrutura do Projeto

Vamos organizar o código em um único arquivo `lexer.c` com funções básicas de leitura e classificação de tokens.

🧾 Código completo (lexer.c)

```c
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>

#define MAX_TOKEN_LENGTH 100

// Lista de palavras-chave
const char* keywords[] = {"if", "else", "while", "return"};
const int num_keywords = 4;

typedef enum {
    TOKEN_KEYWORD,
    TOKEN_IDENTIFIER,
    TOKEN_NUMBER,
    TOKEN_SYMBOL,
    TOKEN_UNKNOWN,
    TOKEN_EOF
} TokenType;

typedef struct {
    TokenType type;
    char lexeme[MAX_TOKEN_LENGTH];
} Token;

// Verifica se é uma palavra-chave
int is_keyword(const char* str) {
    for (int i = 0; i < num_keywords; i++) {
        if (strcmp(str, keywords[i]) == 0) {
            return 1;
        }
    }
    return 0;
}

// Lê próximo token
Token get_next_token(FILE* fp) {
    char ch;
    Token token;
    int i = 0;

    // Pular espaços em branco
    while ((ch = fgetc(fp)) != EOF && isspace(ch));
```

```
    if (ch == EOF) {
        token.type = TOKEN_EOF;
        strcpy(token.lexeme, "EOF");
        return token;
    }

    // Identificadores ou palavras-chave
    if (isalpha(ch) || ch == '_') {
        token.lexeme[i++] = ch;
        while ((ch = fgetc(fp)) != EOF && (isalnum(ch) || ch == '_')) {
            token.lexeme[i++] = ch;
        }
        token.lexeme[i] = '\0';
        ungetc(ch, fp); // devolve o último caractere lido

        if (is_keyword(token.lexeme)) {
            token.type = TOKEN_KEYWORD;
        } else {
            token.type = TOKEN_IDENTIFIER;
        }
        return token;
    }

    // Números
    if (isdigit(ch)) {
        token.lexeme[i++] = ch;
        while ((ch = fgetc(fp)) != EOF && isdigit(ch)) {
            token.lexeme[i++] = ch;
        }
        token.lexeme[i] = '\0';
        ungetc(ch, fp);
        token.type = TOKEN_NUMBER;
        return token;
    }

    // Símbolos simples
    if (strchr("+-*/=;()", ch)) {
        token.lexeme[0] = ch;
        token.lexeme[1] = '\0';
        token.type = TOKEN_SYMBOL;
        return token;
    }

    // Token desconhecido
    token.lexeme[0] = ch;
    token.lexeme[1] = '\0';
    token.type = TOKEN_UNKNOWN;
    return token;
}

const char* token_type_to_string(TokenType type) {
    switch (type) {
```

```
        case TOKEN_KEYWORD: return "KEYWORD";
        case TOKEN_IDENTIFIER: return "IDENTIFIER";
        case TOKEN_NUMBER: return "NUMBER";
        case TOKEN_SYMBOL: return "SYMBOL";
        case TOKEN_EOF: return "EOF";
        default: return "UNKNOWN";
    }
}

int main() {
    FILE* fp = fopen("entrada.txt", "r");
    if (!fp) {
        printf("Erro ao abrir arquivo.\n");
        return 1;
    }

    Token token;
    do {
        token = get_next_token(fp);
        printf("<%s, '%s'>\n", token_type_to_string(token.type),
token.lexeme);
    } while (token.type != TOKEN_EOF);

    fclose(fp);
    return 0;
}
```

📝 Exemplo de entrada (`entrada.txt`)

```
if (x1 == 42) {
    return y + 5;
}
```

🧾 Saída esperada

```
<KEYWORD, 'if'>
<SYMBOL, '('>
<IDENTIFIER, 'x1'>
<SYMBOL, '='>
<SYMBOL, '='>
<NUMBER, '42'>
<SYMBOL, ')'>
<KEYWORD, 'return'>
<IDENTIFIER, 'y'>
<SYMBOL, '+'>
<NUMBER, '5'>
```

```
<SYMBOL, ';'>
<SYMBOL, '}' >
<EOF, 'EOF'>
```

📌 Observações

- O lexer lida apenas com **símbolos de um caractere** (ex: ele quebra `==` em dois `=`).
- Podemos expandir facilmente para símbolos compostos como `==`, `<=`, `!=` com **lookahead**.
- A função `ungetc()` permite "devolver" um caractere para o fluxo de leitura.
- A estrutura é ideal para fins educacionais, sendo totalmente customizável.

Boa pergunta, Luis! Reconhecer **operadores compostos** como `==`, `!=`, `<=`, `>=`, `&&`, `||` exige uma **leitura com *lookahead*** — ou seja, ao ver um caractere como `=`, a gente precisa olhar o próximo para decidir se é `=` ou `==`.

Vou te mostrar isso na prática, adicionando esse recurso ao **analisador léxico em C** que te mostrei antes.

✅ Operadores compostos que vamos reconhecer:

Operador	Significado
`==`	Igualdade
`!=`	Diferente
`<=`	Menor ou igual
`>=`	Maior ou igual
`&&`	E lógico
`||`	Ou lógico
`=`	Atribuição
`<`, `>`	Relacionais

🔧 Ajuste na função `get_next_token()`

Aqui está o trecho alterado da função para suportar isso:

```c
// Símbolos simples e compostos
if (strchr("=<>!&|", ch)) {
    char next = fgetc(fp);
    token.lexeme[0] = ch;
    token.lexeme[1] = '\0';
```

```
    // Compara operadores compostos
    if ((ch == '=' && next == '=') ||
        (ch == '!' && next == '=') ||
        (ch == '<' && next == '=') ||
        (ch == '>' && next == '=') ||
        (ch == '&' && next == '&') ||
        (ch == '|' && next == '|')) {
        token.lexeme[1] = next;
        token.lexeme[2] = '\0';
    } else {
        ungetc(next, fp); // não era parte de operador composto
    }

    token.type = TOKEN_SYMBOL;
    return token;
}
```

O que essa lógica faz?

1. Se o caractere lido for `=`, `<`, `>`, `!`, `&` ou `|`, a gente suspeita que **pode ser um operador composto**.
2. Lemos o **próximo caractere**.
3. Verificamos se o par forma um operador válido.
4. Se não formar, usamos `ungetc()` para devolver o caractere extra e ficamos com o símbolo simples.

🧪 Exemplo: entrada de teste

```
if (x1 != 42 && y >= 10 || z == 0) {
    x = 1;
}
```

Saída esperada:

```
<KEYWORD, 'if'>
<SYMBOL, '('>
<IDENTIFIER, 'x1'>
<SYMBOL, '!='>
<NUMBER, '42'>
<SYMBOL, '&&'>
<IDENTIFIER, 'y'>
<SYMBOL, '>='>
<NUMBER, '10'>
<SYMBOL, '||'>
<IDENTIFIER, 'z'>
<SYMBOL, '=='>
```

```
<NUMBER, '0'>
<SYMBOL, ')'>
<SYMBOL, '{'>
<IDENTIFIER, 'x'>
<SYMBOL, '='>
<NUMBER, '1'>
<SYMBOL, ';'>
<SYMBOL, '}' >
<EOF, 'EOF'>
```

📦 Dica bônus: macro para facilitar comparação

Você pode definir uma macro para simplificar a checagem:

```
#define MATCH_COMPOSITE(op1, op2) (ch == op1 && next == op2)
```

E aí no `if` usaria:

```
if (MATCH_COMPOSITE('=', '=') || MATCH_COMPOSITE('!', '=') || ...)
```
