Compiladores

**Por Luciano Alfredo**

Dedico este artigo à memória da minha mãe, Zinha Daniel Cambinga, cuja presença continua a iluminar meu caminho. Também dedico a todos aqueles que ousam buscar a luz do conhecimento em um mundo muitas vezes envolto pela escuridão da ignorância.

**Indice**

**1. Introdução**

O presente artigo foi elaborado no âmbito da disciplina de Compiladores e tem como objetivo apresentar o estudo e a construção de um compilador para a linguagem C++. Este trabalho representa um desafio enriquecedor, proporcionando uma compreensão mais profunda sobre o funcionamento das linguagens de programação e dos compiladores em geral. Espera-se que esta jornada também seja uma oportunidade valiosa de aprendizado para outros estudantes interessados na área.

**Capítulo I: Conceito de Linguagens Formais e Autómatos**

**1.1 Introdução**

Antes de mergulharmos no estudo e na construção de um compilador, é essencial compreender o funcionamento dos autómatos. Para isso, este capítulo será dedicado à introdução de alguns conceitos fundamentais da disciplina de Linguagens Formais e Autómatos (LFA), com ênfase nos Autómatos Finitos Determinísticos (AFD), que desempenham um papel crucial na análise léxica dos compiladores.

Segundo Menezes (2000), o estudo das Linguagens Formais trata dos aspetos sintáticos e semânticos das linguagens. A **sintaxe** se ocupa das estruturas e regras que definem a forma correta dos programas, permitindo, por exemplo, a verificação gramatical de expressões e comandos. Ela manipula símbolos sem considerar seus significados. Já a **semântica** está voltada à atribuição de significado aos programas, ou seja, à interpretação e ao comportamento que determinadas instruções devem ter durante a execução.

Compreender essas duas dimensões — sintática e semântica — é fundamental para a construção de compiladores, uma vez que ambos os aspetos são manipulados em diferentes fases do processo de compilação, como veremos mais adiante.

**1.2 Autómatos Finitos Determinísticos (AFD)**

Menezes(2000) nos ajuda a conceituar um AFD como uma máquina composta, basicamente, de três partes:

1. Fita. Dispositivo de entrada que contem a informação a ser processada,
2. Unidade de controle. Reflete o estado corrente da máquina.
3. Programa ou Função de Transição. Função que comanda as leituras e define o estado da máquina.

O conceito de autómato não está tão distante da nossa realidade quotidiana. Utilizamos, de forma indireta, autómatos em diversas situações do dia a dia. Um exemplo clássico e bastante intuitivo é o semáforo, que pode ser modelado como um Autómato Finito Determinístico (AFD).

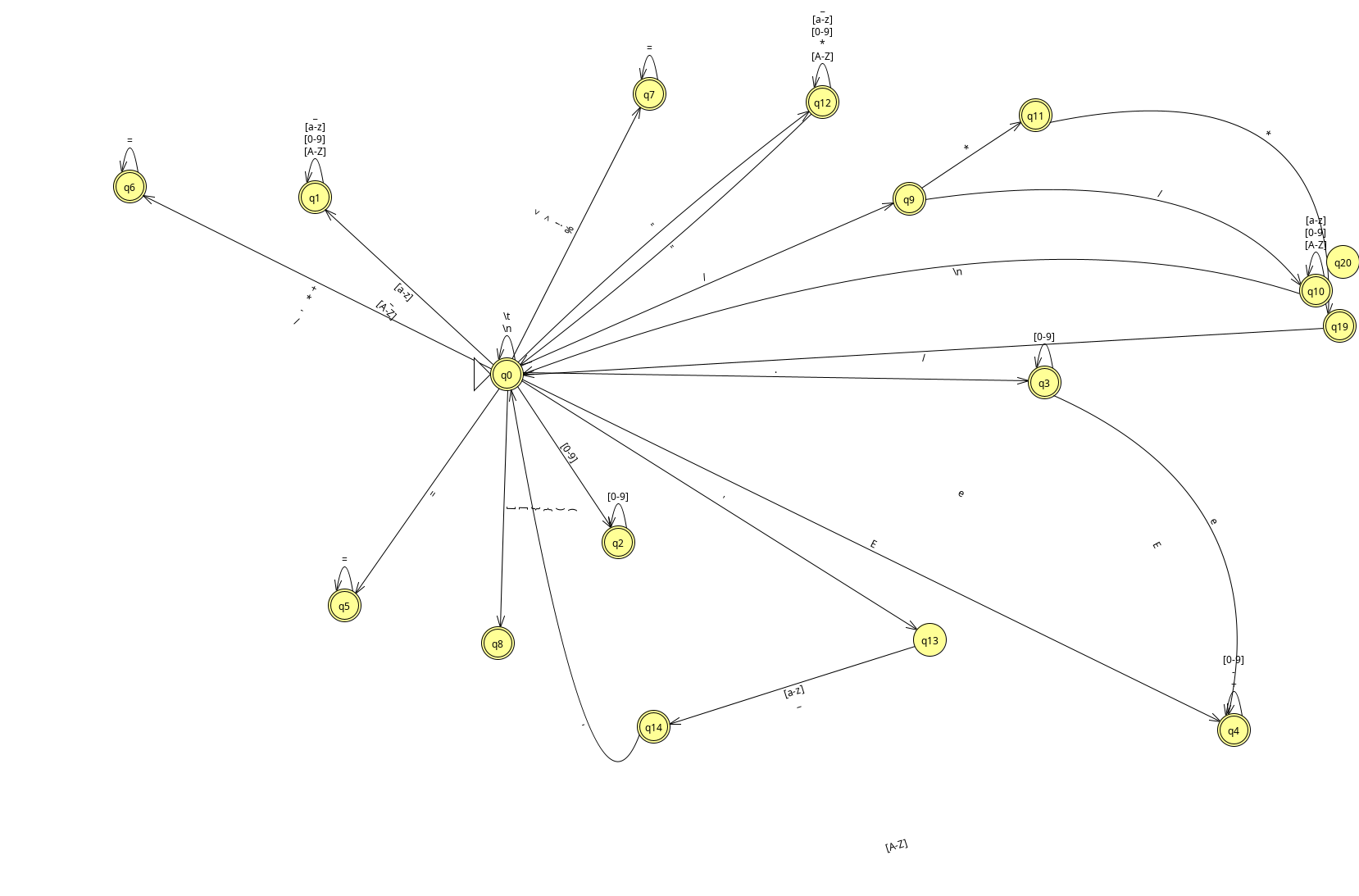
Um semáforo possui três estados possíveis (por exemplo: s0, s1, s2), um alfabeto composto pelas cores {vermelho, amarelo, verde} e uma função de transição que determina a mudança de um estado para outro com base em regras definidas. Em um dado momento, o semáforo está em apenas um estado — ou seja, exibe apenas uma cor por vez. Essa característica ilustra bem o princípio fundamental dos AFDs: a existência de um único estado ativo em cada instante de tempo, com transições bem definidas entre os estados.

Esse exemplo ajuda a visualizar como os autómatos finitos determinísticos funcionam e por que são tão importantes na análise léxica de compiladores, onde é necessário reconhecer padrões de símbolos de maneira precisa e eficiente.

**1.3 Autómato Finito Determinístico (AFD) para a leitura o Analisador léxico.**

Durante o desenvolvimento do analisador léxico, construiremos uma máquina que processa diferentes símbolos e faz a analise lexical dos mesmos.

Abaixo temos o AFD geral do nosso compilador.

****

Neste AFD cada símbolo, termo ou expressão será processado e receberá um significa sintático no programa.

## Capítulo II: Construção do Compilador – Analisador Léxico

### 2.1 Introdução aos compiladores

### 2.1 Visão Geral da Fase Léxica

A primeira fase do compilador é a análise léxica, responsável por transformar o fluxo de caracteres do código-fonte em uma sequência de tokens. Tokens são as menores unidades significativas para o compilador, representando identificadores, palavras-chave, números, operadores, símbolos de pontuação, entre outros.

Esta fase é fundamental pois prepara a entrada da análise sintática, além de alimentar a **tabela de símbolos**, onde são armazenadas informações úteis sobre os identificadores encontrados.

### 2.2 Arquitetura do Analisador Léxico

#### a) Módulos principais

O analisador léxico foi desenvolvido modularmente para facilitar manutenção e expansão:

* O **main.c** : Função principal que abre o arquivo de entrada, chama o analisador e imprime os tokens.
* O **analisador.h**: Cabeçalhos e definições globais.
* O **analisador.h**: Implementação da máquina de estados que reconhece os tokens.

### 2.3 Funcionamento do Analisador: A Máquina de Estados

A leitura do código-fonte é feita **caractere por caractere**. Para cada caractere, o analisador atualiza seu **estado atual** de acordo com a lógica de um **Autômato Finito Determinístico (AFD)**. Cada estado representa uma fase de identificação de um token específico.

### 2.4 Manipulação de Lexemas e Tokens

À medida que a máquina percorre os caracteres, ela **armazena os lexemas** (cadeia de caracteres que forma um token) em um buffer temporário. Quando um estado final é alcançado, o lexema é classificado e transformado em um token com base nas seguintes regras:

* Se o lexema for uma palavra reservada é classificado como **palavra-chave**.
* Se for composto apenas por dígitos, é um **literal numérico**.
* Se começar com uma letra e possuir letras/dígitos, é um **identificador**.
* Se corresponder a símbolos, é um **token especial**.

#### Exemplo de estrutura de token:

typedef struct EntradaTabela {

int token;

char lexema[MAX\_LEXEMA];

char token\_name[MAX\_TOKEN\_NAME];

int linha;

struct EntradaTabela \*prox;

} EntradaTabela;

### 2.5 Controle de Posição no Código-Fonte

Além de reconhecer tokens, o analisador mantém o controle de posição no arquivo (linha e coluna). Isso é útil tanto para depuração quanto para mensagens de erro mais precisas.

char ler\_caractere() {

char c = fgetc(arquivo);

If (c == '\n') {

linha++;

coluna = 1;

} else {

coluna++;}

return c;

}

### 2.6 Implementação de Palavras-Chave

Após o reconhecimento de um possível identificador, o analisador chama a função para verificar se aquele identificador é, na verdade, uma palavra-chave da linguagem.

int e\_palavra\_reservada(const char \*lexema, char \*token\_name) {

struct {

const char \*palavra;

const char \*token\_name;

int token\_val;

} palavras[] = {

{"int", "TOKEN\_INT", 1}, {"float", "TOKEN\_FLOAT", 1},

{"char", "TOKEN\_CHAR", 1}, {"string", "TOKEN\_STRING", 1},

{"bool", "TOKEN\_BOOL", 1}, {"void", "TOKEN\_VOID", 1},

{"if", "TOKEN\_IF", 1}, {"else", "TOKEN\_ELSE", 1},

{"while", "TOKEN\_WHILE", 1}, {"for", "TOKEN\_FOR", 1},

{"return", "TOKEN\_RETURN", 1}, {"break", "TOKEN\_BREAK", 1},

{"continue", "TOKEN\_CONTINUE", 1}, {"switch", "TOKEN\_SWITCH", 1},

{"case", "TOKEN\_CASE", 1}, {"default", "TOKEN\_DEFAULT", 1},

{"do", "TOKEN\_DO", 1}, {"class", "TOKEN\_CLASS", 1},

{"public", "TOKEN\_PUBLIC", 1}, {"private", "TOKEN\_PRIVATE", 1},

{"protected", "TOKEN\_PROTECTED", 1}, {"static", "TOKEN\_STATIC", 1},

{"const", "TOKEN\_CONST", 1}, {"new", "TOKEN\_NEW", 1},

{"this", "TOKEN\_THIS", 1}, {"null", "TOKEN\_NULL", 1},

{"true", "TOKEN\_TRUE", 1}, {"false", "TOKEN\_FALSE", 1},

{NULL, NULL, 0}

};

for (int i = 0; palavras[i].palavra != NULL; i++) {

if (strcmp(lexema, palavras[i].palavra) == 0) {

strcpy(token\_name, palavras[i].token\_name);

return palavras[i].token\_val;

}

}

return 0;

}

### 2.7 Finalização e Integração com o Compilador

Ao término da leitura do arquivo, o analisador retorna um token especial chamado EOF (que deve ser definido em FIM\_DE\_ARQUIVO 9):

while (analex() != FIM\_DE\_ARQUIVO) {

// Continua analisando

}

Esse token informa à função main() ou ao analisador sintático que não há mais tokens a serem processados.

### 2.8Analisador Léxico: Conclusão

A implementação do analisador léxico é uma das partes mais desafiadoras e ricas de um compilador. A partir de um simples fluxo de caracteres, conseguimos construir uma estrutura robusta capaz de reconhecer diferentes componentes de uma linguagem de programação, com a ajuda de autómatos finitos e lógica de máquina de estados.

Com o analisador léxico completo, o próximo passo será construir o **analisador sintático**, que utilizará os tokens gerados para formar estruturas gramaticais e validar a sintaxe do programa.