CENTRO UNIVERSITÁRIO DE BRASÍLIA - CEUB

CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Aluno: Gustavo Czarneski 22252072
Aluno: Miguel Pereira 22204046
Professor: Leonardo Pol
Disciplina: Compiladores

TRABALHO DE CONCLUSÃO DA DISCIPLINA DE COMPILADORES

PROJETO A: LABORATÓRIO DE PESQUISA DE COMPILADORES (ANALISADOR LÉXICO)

Brasília - DF 2025

Sumário

1. Introdução

Este trabalho apresenta o desenvolvimento da primeira etapa de um compilador: o Analisador Léxico (Scanner). O objetivo deste projeto foi implementar um software capaz de ler códigos-fonte escritos em uma linguagem baseada em Portugol, identificar os elementos básicos (tokens) e gerar uma tabela de símbolos, conforme as especificações do Projeto A da disciplina de Compiladores.

A implementação foi realizada na linguagem C++, escolhida por sua eficiência na manipulação de caracteres e arquivos, atendendo aos requisitos de linguagem permitidos para o trabalho.

2. Embasamento teórico

2.1 O Processo de Compilação

A compilação é o processo de tradução de uma linguagem de alto nível para a linguagem de máquina. O Analisador Léxico (ou *Scanner*) é a primeira etapa do *front-end* do compilador.

2.2 Definição de Token e Lexema

A análise léxica consiste na varredura do código fonte,visando agrupar caracteres em unidades lógicas chamadas **lexemas**. Para cada lexema, o analisador produz um **token** correspondente.

- **Lexema:** É a sequência real de caracteres encontrada no código (ex: variavel, 10, se).
- **Token:** É a classificação abstrata desse lexema. Por exemplo, o lexema 10 gera um token do tipo CONST_INT.

2.3 Tabela de Símbolos

Estrutura de dados essencial para armazenar informações sobre os identificadores (variáveis) do programa. Neste projeto, utilizou-se um vetor dinâmico (std::vector) onde o índice do vetor representa o endereço do símbolo. Palavras reservadas não entram nesta tabela, pois possuem significados fixos.

2.4 Modelo Computacional (AFD)

O funcionamento do analisador baseia-se no modelo de **Autômatos Finitos Determinísticos (AFD)**. O autômato parte de um estado inicial e, a cada caractere lido do arquivo de entrada, transita para novos estados até reconhecer um padrão válido (como um número, uma palavra reservada ou um operador).

Conforme a especificação do projeto, o analisador deve operar sob demanda, retornando um token por vez sempre que solicitado pelo analisador sintático, garantindo maior flexibilidade ao processo de compilação.

3. Detalhes de implementação

A solução foi implementada em **C++** devido à sua eficiência e capacidade de manipulação de arquivos e strings.

3.1 Estruturas de Dados Utilizadas

- **Enumeração (enum):** Para definição dos códigos dos tokens;
- **Mapa (std::map):** Utilizado para verificar palavras reservadas com complexidade **O(logn)**, garantindo rapidez na consulta se um identificador é, na verdade, um comando como se ou para;
- **Vetor (std::vector):** Utilizado para a Tabela de Símbolos, armazenando apenas os identificadores de usuário.

3.2 Tratamento de Ambiguidade

Implementou-se a técnica de lookahead (espiar à frente) para resolver operadores ambíguos. Por exemplo, ao ler <, o sistema verifica o próximo caractere: se for -, gera um token de atribuição (<-); se for =, gera menor ou igual (<=); caso contrário, é apenas menor (<)8.

3.3 Entrada e Saída

Diferente de abordagens que fixam o código na memória, este analisador lê de um arquivo externo (teste.por), simulando um ambiente real de compilação. A saída é formatada em colunas para facilitar a visualização dos tokens e seus respectivos índices na tabela de símbolos.

4. Código fonte

Segue abaixo a implementação completa do analisador léxico, desenvolvida na linguagem C++. O código foi documentado para explicar a lógica de transição de estados e o gerenciamento da tabela de símbolos.

```
#include <iostream>
#include <fstream>
#include <string>
#include <map>
#include <vector>
#include <cctype>
#include <iomanip> //deixar a tabela de saída alinhada

using namespace std;

//1. DEFINIÇÃO DOS CÓDIGOS DOS TOKENS
//Usamos um enum para facilitar a identificação de cada token no
código
enum TokenType {
    TOK_EOF = 0,
    TOK_ERRO,

    //Palavras Reservadas da linguagem
    TOK_INICIO, TOK_FIM, TOK_SE, TOK_ENTAO, TOK_SENAO, TOK_FIMSE,
    TOK_PARA, TOK_ATE, TOK_PASSO, TOK_FIMPARA,
    TOK_INTEIRO_TYPE, TOK_LEIA, TOK_IMPRIMA,

    //Tipos de Dados e Identificadores
    TOK_ID,
    TOK_NUM_INT,
    TOK_LITERAL,

    //Operadores Aritméticos e Lógicos
    TOK_MAIS, TOK_MENOS, TOK_MULT, TOK_DIV,
    TOK_E, TOK_OU, TOK_NAO,

    //Operadores Relacionais e Atribuição
    TOK_MAIOR, TOK_MENOR, TOK_MAIOR_IGUAL, TOK_MENOR_IGUAL,
    TOK_IGUAL_REL, TOK_DIFERENTE,
    TOK_ATRIBUICAO,

    //Pontuação/Delimitadores
    TOK_ABRE_PAR, TOK_FECHA_PAR, TOK_PONTO_VIRGULA,
TOK_DOIS_PONTOS
};
```

```
//ESTRUTURA DO TOKEN
struct Token {
    TokenType tipo;
    string lexema;
    int linha;
    int indiceTabela;
};

//CLASSE DO ANALISADOR LÉXICO
class AnalisadorLexico {
private:
    ifstream arquivo;
    int linhaAtual;

    //Mapa para busca rápida de palavras reservadas (Complexidade
O(log n))
    map<string, TokenType> palavrasReservadas;

    //Tabela de Símbolos: Armazena apenas os nomes das variáveis
criadas pelo usuário
    vector<string> tabelaSimbolos;

    //Configura o mapa com as palavras da linguagem
    void inicializarTabela() {
        palavrasReservadas["inicio"] = TOK_INICIO;
        palavrasReservadas["fim"] = TOK_FIM;
        palavrasReservadas["se"] = TOK_SE;
        // Tratamento para aceitar versões com e sem acento
(robustez)
        palavrasReservadas["entao"] = TOK_ENTAO;
palavrasReservadas["então"] = TOK_ENTAO;
        palavrasReservadas["senao"] = TOK_SENAO;
palavrasReservadas["senão"] = TOK_SENAO;
        palavrasReservadas["fim_se"] = TOK_FIMSE;
        palavrasReservadas["para"] = TOK_PARA;
        palavrasReservadas["ate"] = TOK_ATE;
palavrasReservadas["até"] = TOK_ATE;
        palavrasReservadas["passo"] = TOK_PASSO;
        palavrasReservadas["fim_para"] = TOK_FIMPARA;
        palavrasReservadas["inteiro"] = TOK_INTEIRO_TYPE;
        palavrasReservadas["leia"] = TOK_LEIA;
        palavrasReservadas["imprima"] = TOK_IMPRIMA;
        palavrasReservadas["escreva"] = TOK_IMPRIMA; // Sinônimo
aceito
        palavrasReservadas["E"] = TOK_E;
        palavrasReservadas["OU"] = TOK_OU;
        palavrasReservadas["NAO"] = TOK_NAO;
    }
```

```
    //Adiciona uma nova variável na tabela de símbolos se ela
ainda não existir
    int gerenciarTabelaSimbolos(string lexema) {
        for (size_t i = 0; i < tabelaSimbolos.size(); i++) {
            if (tabelaSimbolos[i] == lexema) return i;
        }
        tabelaSimbolos.push_back(lexema); //Nova variável
        return tabelaSimbolos.size() - 1;
    }

public:
    AnalisadorLexico(string nomeArquivo) {
        arquivo.open(nomeArquivo);
        linhaAtual = 1;
        inicializarTabela();

        if (!arquivo.is_open()) {
            cout << "Erro: Nao foi possivel abrir o arquivo '" <<
nomeArquivo << "'" << endl;
            exit(1);
        }
    }

    ~AnalisadorLexico() {
        if (arquivo.is_open()) arquivo.close();
    }

    //acessa a tabela de símbolos final para impressão na main
    vector<string> getTabelaSimbolos() {
        return tabelaSimbolos;
    }

    //FUNÇÃO PRINCIPAL: O AUTOMATO FINITO
    Token proximoToken() {
        char car;

        //1. Ignorar espaços em branco e contar linhas
        while (arquivo.get(car)) {
            if (car == '\n') linhaAtual++;
            else if (!isspace(car)) break;
        }

        //Verifica se o arquivo acabou
        if (arquivo.eof()) return {TOK_EOF, "EOF", linhaAtual,
-1};
```

```
        //2. Autômato para Identificadores e Palavras Reservadas
(Começa com Letra)
        if (isalpha(car)) {
            string lexema = "";
            lexema += car;

            //continua lendo enquanto for letra, número ou
underline
            while (arquivo.get(car)) {
                if (isalnum(car) || car == '_' || (unsigned
char)car > 127) {
                    lexema += car;
                } else {
                    arquivo.unget();
                    break;
                }
            }

            //Verifica: É palavra reservada ou variável?
            if (palavrasReservadas.count(lexema)) {
                return {palavrasReservadas[lexema], lexema,
linhaAtual, -1};
            } else {
                //Se não é reservada, é um ID. Salva na tabela.
                int idx = gerenciarTabelaSimbolos(lexema);
                return {TOK_ID, lexema, linhaAtual, idx};
            }
        }

        //3. Autômato para Números Inteiros
        if (isdigit(car)) {
            string lexema = "";
            lexema += car;
            while (arquivo.get(car)) {
                if (isdigit(car)) {
                    lexema += car;
                } else {
                    arquivo.unget();
                    break;
                }
            }
            return {TOK_NUM_INT, lexema, linhaAtual, -1};
        }

        //4. Autômato para Literais (Strings entre aspas)
        if (car == '"') {
            string lexema = "";
            while (arquivo.get(car)) {
```

```
                if (car == '"') break;
                lexema += car;
            }
            //Retorna com as aspas para indicar que é string
            return {TOK_LITERAL, "\"" + lexema + "\"", linhaAtual,
-1};
        }

        //5. Autômato para Símbolos e Operadores (Tratamento de
Ambiguidade)
        string lexema = string(1, car);
        switch (car) {
            case '+': return {TOK_MAIS, lexema, linhaAtual, -1};
            case '-': return {TOK_MENOS, lexema, linhaAtual, -1};
            case '*': return {TOK_MULT, lexema, linhaAtual, -1};
            case '/': return {TOK_DIV, lexema, linhaAtual, -1};
            case '(': return {TOK_ABRE_PAR, lexema, linhaAtual,
-1};
            case ')': return {TOK_FECHA_PAR, lexema, linhaAtual,
-1};
            case ';': return {TOK_PONTO_VIRGULA, lexema,
linhaAtual, -1};
            case ':': return {TOK_DOIS_PONTOS, lexema, linhaAtual,
-1};

            case '>':
                //Lookahead: Espia o próximo para ver se é >=
                if (arquivo.peek() == '=') {
                    arquivo.get(car); // Consome o '='
                    return {TOK_MAIOR_IGUAL, ">=", linhaAtual,
-1};
                }
                return {TOK_MAIOR, ">", linhaAtual, -1};

            case '<':
                //Lookahead complexo: Pode ser <=, <- (atribuição)
ou <> (diferente)
                if (arquivo.peek() == '=') {
                    arquivo.get(car);
                    return {TOK_MENOR_IGUAL, "<=", linhaAtual,
-1};
                } else if (arquivo.peek() == '-') {
                    arquivo.get(car);
                    return {TOK_ATRIBUICAO, "<-", linhaAtual, -1};
                } else if (arquivo.peek() == '>') {
                    arquivo.get(car);
                    return {TOK_DIFERENTE, "<>", linhaAtual, -1};
                }
```

```
                return {TOK_MENOR, "<", linhaAtual, -1};

            case '=': return {TOK_IGUAL_REL, "=", linhaAtual, -1};

            default:
                return {TOK_ERRO, lexema, linhaAtual, -1};
        }
    }
};

//Função auxiliar que traduz o enum em texto na hora de imprimir
string nomeToken(TokenType t) {
    switch(t) {
        case TOK_EOF: return "EOF";
        case TOK_ERRO: return "ERRO";
        case TOK_ID: return "ID";
        case TOK_NUM_INT: return "CONST_INT";
        case TOK_LITERAL: return "CONST_LIT";
        case TOK_INICIO: return "KW_INICIO";
        case TOK_FIM: return "KW_FIM";
        case TOK_SE: return "KW_SE";
        case TOK_ENTAO: return "KW_ENTAO";
        case TOK_SENAO: return "KW_SENAO";
        case TOK_FIMSE: return "KW_FIM_SE";
        case TOK_PARA: return "KW_PARA";
        case TOK_ATE: return "KW_ATE";
        case TOK_PASSO: return "KW_PASSO";
        case TOK_FIMPARA: return "KW_FIM_PARA";
        case TOK_INTEIRO_TYPE: return "KW_INTEIRO";
        case TOK_LEIA: return "KW_LEIA";
        case TOK_IMPRIMA: return "KW_IMPRIMA";

        case TOK_MAIS: return "OP_SOMA";
        case TOK_MENOS: return "OP_SUB";
        case TOK_MULT: return "OP_MULT";
        case TOK_DIV: return "OP_DIV";

        case TOK_MAIOR: return "OP_REL_MAIOR";
        case TOK_MENOR: return "OP_REL_MENOR";
        case TOK_MAIOR_IGUAL: return "OP_MAIOR_IGUAL";
        case TOK_MENOR_IGUAL: return "OP_MENOR_IGUAL";
        case TOK_DIFERENTE: return "OP_DIFERENTE";
        case TOK_IGUAL_REL: return "OP_REL_IGUAL";
        case TOK_ATRIBUICAO: return "OP_ATRIBUICAO";

        case TOK_ABRE_PAR: return "DELIM_ABRE_PAR";
        case TOK_FECHA_PAR: return "DELIM_FECHA_PAR";
        case TOK_PONTO_VIRGULA: return "DELIM_P_VIRGULA";
```

```
        case TOK_DOIS_PONTOS: return "DELIM_DOIS_PONTOS";

        default: return "TOKEN_OUTRO";
    }
}


//MAIN: (Simulador do Sintático)
int main() {
    //O arquivo teste.por deve estar na mesma pasta do executável
    AnalisadorLexico lexico("teste.por");
    Token token;

    cout << "=== ANALISADOR LEXICO - RESULTADO ===" << endl;
    cout << left << setw(10) << "LINHA"
         << setw(20) << "TIPO TOKEN"
         << setw(20) << "LEXEMA"
         << setw(10) << "IND. TAB" << endl;
    cout << string(60, '-') << endl;

    //Loop principal: chama o próximo token até acabar o arquivo
    do {
        token = lexico.proximoToken();

        if (token.tipo != TOK_EOF) {
            cout << left << setw(10) << token.linha
                 << setw(20) << nomeToken(token.tipo)
                 << setw(20) << token.lexema;

            //Se for ID, mostra o índice na tabela de símbolos
            if (token.indiceTabela != -1) cout << setw(10) <<
token.indiceTabela;
            else cout << setw(10) << "-";

            cout << endl;
        }

    } while (token.tipo != TOK_EOF);

    //Tabela de Símbolos Final
    cout << "\n=== TABELA DE SIMBOLOS (VARIAVEIS) ===" << endl;
    vector<string> tab = lexico.getTabelaSimbolos();
    for (size_t i = 0; i < tab.size(); i++) {
        cout << "Indice " << i << ": " << tab[i] << endl;
    }

    return 0;
}
```

5. Resultados

Para validar a integridade e o funcionamento do Analisador Léxico, foram realizados testes de execução simulando o comportamento do analisador sintático. O objetivo foi verificar se o software é capaz de:

1. Ler o arquivo de entrada corretamente.
2. Ignorar espaços em branco e tabulações.
3. Classificar corretamente Palavras Reservadas, Operadores e Literais.
4. Gerenciar a Tabela de Símbolos, inserindo apenas os identificadores de usuário.

5.1 Arquivo de Entrada

O arquivo de teste utilizado (teste.por) foi elaborado com base nos exemplos fornecidos na especificação do projeto. O código fonte contém estruturas condicionais (se/senão), laços de repetição (para), declaração de variáveis e comandos de entrada/saída, cobrindo todo o escopo da gramática proposta.

Conteúdo do arquivo ***teste.por:***

```
inicio
  inteiro:a;
  inteiro:b;
  imprima "Teste do Analisador";
  leia(a);

  se a = 5
  entao
    escreva(a);
  senao
    b <- 10;
    escreva(b);
  fim_se

  para b=0 ate a passo 2
    imprima(b);
  fim_para
fim
```

5.2 Execução e Saída

Devido à extensão da saída gerada pelo teste completo, os resultados são apresentados em duas partes. A Figura 1 demonstra o início do processamento e a formatação da tabela de tokens.

```
PS C:\Users\gut_c\Documents\ProjetoCompilador> .\analisador.exe
=== ANALISADOR LEXICO - RESULTADO ===
LINHA     TIPO TOKEN          LEXEMA              IND. TAB
-------------------------------------------------------------
1         KW_INICIO           inicio              -
2         KW_INTEIRO          inteiro             -
2         DELIM_DOIS_PONTOS   :                   -
2         ID                  a                   0
2         DELIM_P_VIRGULA     ;                   -
3         KW_INTEIRO          inteiro             -
3         DELIM_DOIS_PONTOS   :                   -
3         ID                  b                   1
3         DELIM_P_VIRGULA     ;                   -
4         KW_IMPRIMA          imprima             -
4         CONST_LIT           "Teste do Analisador"-
4         DELIM_P_VIRGULA     ;                   -
5         KW_LEIA             leia                -
5         DELIM_ABRE_PAR      (                   -
5         ID                  a                   0
5         DELIM_FECHA_PAR     )                   -
5         DELIM_P_VIRGULA     ;                   -
7         KW_SE               se                  -
7         ID                  a                   0
7         OP_REL_IGUAL        =                   -
7         CONST_INT           5                   -
8         KW_ENTAO            entao               -
9         KW_IMPRIMA          escreva             -
9         DELIM_ABRE_PAR      (                   -
9         ID                  a                   0
9         DELIM_FECHA_PAR     )                   -
9         DELIM_P_VIRGULA     ;                   -
10        KW_SENAO            senao               -
11        ID                  b                   1
11        OP_ATRIBUICAO       <-                  -
11        CONST_INT           10                  -
11        DELIM_P_VIRGULA     ;                   -
12        KW_IMPRIMA          escreva             -
12        DELIM_ABRE_PAR      (                   -
12        ID                  b                   1
12        DELIM_FECHA_PAR     )                   -
12        DELIM_P_VIRGULA     ;                   -
```

Figura 1: início da análise léxica

A Figura 2 apresenta a conclusão do processamento, destacando a identificação do token de fim de arquivo (EOF) e o estado final da Tabela de Símbolos, contendo os identificadores armazenados.

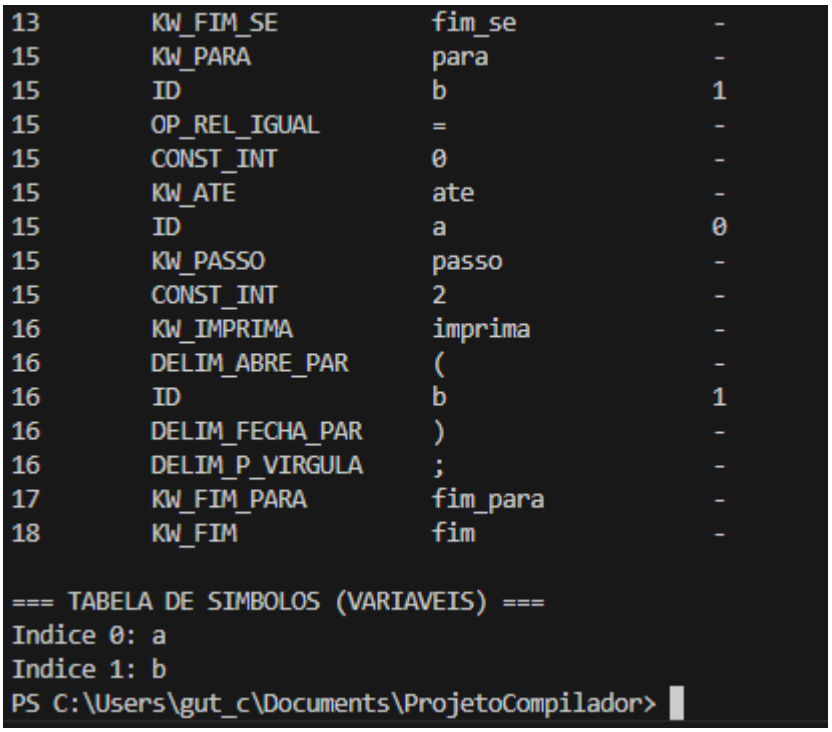

Figura 2: finalização

5.3 Análise dos Resultados

Ao analisar a saída gerada, observa-se o cumprimento dos requisitos do projeto:

- **Distinção de Tokens:** O sistema diferenciou corretamente palavras reservadas (ex: inicio classificado como KW_INICIO) de identificadores. Note que na Tabela de Símbolos (última coluna), as palavras reservadas apresentam valor -, indicando que não são armazenadas, conforme a regra de negócio.
- **Gestão de Identificadores:** As variáveis a e b foram corretamente identificadas como ID. Na coluna "IND. TAB", é possível ver que a recebeu o índice **0** e b recebeu o índice **1**, comprovando que a Tabela de Símbolos está sendo populada dinamicamente e sem duplicatas.
- **Tratamento de Strings:** O texto "Teste do Analisador" foi corretamente agrupado em um único token do tipo CONST_LIT, preservando as aspas duplas, o que valida a lógica do autômato para literais.
- **Operadores Compostos:** O operador de atribuição <- foi identificado corretamente como OP_ATRIBUICAO (linha 9), demonstrando o funcionamento da técnica de *lookahead* para resolver a ambiguidade com o operador relacional <.

6. Conclusão

O desenvolvimento deste analisador léxico permitiu compreender na prática o funcionamento da primeira etapa de um compilador. As maiores dificuldades encontradas foram o tratamento de ambiguidades nos operadores (resolvido com a técnica de *lookahead*) e o gerenciamento da tabela de símbolos para identificadores. O uso da estrutura de dados ***std::map*** mostrou-se eficiente para a verificação de palavras reservadas, atendendo aos requisitos de desempenho e simplicidade do projeto.

7. Bibliografia

1. AHO, A. V. et al. ***Compiladores: princípios, técnicas e ferramentas*** (O "Livro do Dragão"). 2. ed. São Paulo: Pearson, 2008.
2. LOUDEN, K. C. ***Compiladores: princípios e práticas***. São Paulo: Cengage Learning, 2004.
3. Documentação da Linguagem C++. Disponível em: **cplusplus.com.**