Universidade Católica de Brasília

Pró-Reitoria Acadêmica
Curso de Bacharelado em Ciência da Computação
Trabalho da Disciplina Linguagens Formais, Autômatos e Compiladores .

Analisador Léxico

Autores:
João Victor Ribeiro Oliveira;
Juan Alberto Bezerra Jeronimo;
Júlia Gabriela Gomes Da Silva;
Lara Ewellen De Carvalho Rocha
Leandro Veras De Souza.
Orientador: Prof. MARCELO EUSTAQUIO SOARES DE LIMA JUNIOR

Brasília - DF
2024

JOÃO VICTOR RIBEIRO OLIVEIRA;
JUAN ALBERTO BEZERRA JERONIMO;
JÚLIA GABRIELA GOMES DA SILVA;
LARA EWELLEN DE CARVALHO ROCHA;
LEANDRO VERAS DE SOUZA.

.

Analisador Léxico

Documento apresentado ao Curso de graduação de Bacharelado em Ciência da Computação da Universidade Católica de Brasília, como requisito parcial para obtenção da aprovação na disciplina de Linguagens Formais, autômatos e compiladores.

Orientador: Prof. MARCELO EUSTAQUIO SOARES DE LIMA JUNIOR

Brasília
2024

RESUMO

Esse trabalho fala sobre a criação de um analisador léxico para a linguagem MicroPascal. Ele identifica e classifica tokens (como identificadores, números e operadores) no código fonte, armazenando essas informações em um arquivo .lex e organizando os símbolos em uma tabela. Testes foram feitos com programas corretos e incorretos para verificar o funcionamento. O trabalho conta com uma figura de um autômato finito determinístico (AFD) para reconhecer tokens.

Palavras-chave: Análise léxica, MicroPascal, tokens, tabela de símbolos, compilador.

ABSTRACT

This report discusses the creation of a lexical analyzer for the MicroPascal language. It identifies and classifies tokens (such as identifiers, numbers, and operators) in the source code, storing this information in a .lex file and organizing the symbols in a table. Tests were conducted with both correct and incorrect programs to verify its functionality. The work includes a figure of a deterministic finite automaton (DFA) to recognize tokens.

Keywords: *Lexical analysis, MicroPascal, tokens, symbol table, compiler.*

SUMÁRIO

INTRODUÇÃO

Neste relatório, apresentamos o desenvolvimento de um analisador léxico para a linguagem MicroPascal, implementado na linguagem C. O principal objetivo deste projeto é identificar e classificar tokens, como identificadores, números e operadores, presentes no código fonte. As informações coletadas são armazenadas em um arquivo .lex e organizadas em uma tabela de símbolos.

Para validar a funcionalidade do analisador, foram realizados testes com programas tanto corretos quanto incorretos. Além disso, o trabalho inclui uma figura de um autômato finito determinístico (AFD) utilizado para o reconhecimento dos tokens.

OBJETIVO

1.1 OBJETIVO GERAL

Este projeto tem como objetivo principal desenvolver um analisador léxico para a linguagem de programação MicroPascal (μ-Pascal), conforme descrito em sala de aula. O analisador léxico desempenha um papel fundamental no processo de compilação, sendo o responsável por interpretar o código-fonte, separá-lo em componentes léxicos (tokens), como identificadores, números, operadores e palavras-chave, e realizar uma análise preliminar em busca de erros léxicos.

A construção de um analisador léxico eficiente é um passo essencial no desenvolvimento de compiladores. Ele garante que o código seja bem estruturado e livre de erros básicos antes que etapas mais complexas, como a análise sintática e semântica, sejam executadas.

1.2 OBJETIVOS ESPECÍFICOS

1. Implementação de um Autômato Finito Determinístico (AFD): O analisador léxico será baseado em um AFD, que identificará lexemas no código-fonte e retornará uma struct contendo o token correspondente.

2. Reconhecimento de Tokens: O sistema reconhecerá e classificará corretamente tokens da linguagem MicroPascal, como palavras-chave, operadores, identificadores, números e símbolos de pontuação.

3. Utilização de uma Tabela de Símbolos (TS): A TS armazenará identificadores e palavras reservadas da linguagem, evitando duplicação de registros e facilitando a reutilização.

4. Gerenciamento de Erros Léxicos: O analisador detectará erros léxicos, como caracteres inválidos, strings não fechadas e comentários não finalizados, indicando linha e coluna dos erros.

5. Geração de Arquivo de Saída: O sistema gerará um arquivo ".lex" com uma lista detalhada dos tokens reconhecidos, incluindo nome, lexema, linha e coluna de ocorrência.

6. Descarte de Elementos Não Relevantes: Espaços em branco, tabulações, quebras de linha e comentários serão ignorados, focando apenas nos elementos relevantes do código.

Relatório técnico

Estruturas de Dados

1. Token

```
typedef struct {
    char tipo[TAMANHO_MAX_LEXEMA];
    char valor[TAMANHO_MAX_LEXEMA];
    int linha;
    int coluna;
} Token;
```

- ➔ **Tipo:**
  Armazena o tipo do token (ex: IDENTIFICADOR, RESERVADA).
- ➔ **Valor:**
  Armazena o valor do token reconhecido (ex: o texto do identificador).
- ➔ **Linha e Coluna:**
  Indicam a posição do token no arquivo de entrada.

2. Símbolo

```
typedef struct {
    char lexema[TAMANHO_MAX_LEXEMA];
    char tipo[TAMANHO_MAX_LEXEMA];
} Simbolo;
```

- ➔ **Lexema:**
  Palavra-chave ou texto do identificador.

Tipo:
O tipo de símbolo (ex: IDENTIFICADOR).

Variáveis Globais

- ➔ **tabela símbolos:**
  Armazena identificadores e palavras-chave reconhecidas.
- ➔ **contador simbolos:**
  Localiza o número de símbolos armazenados.
- ➔ **linha e coluna:**
  Acha a posição atual de leitura no arquivo.

Palavras-Chave

```
const char *palavras_chave[] = {
    "program", "var", "integer", "real", "begin", "end",
    "if", "then", "else", "while", "do", "write", "read", NULL
};
```

São as palavras reconhecidas pelo analisador.

Funções

1. adicionar tabela símbolos

```
void adicionar_tabela_simbolos(const char *lexema, const char *tipo) {
    for (int i = 0; i < contador_simbolos; i++) {
        if (strcmp(tabela_simbolos[i].lexema, lexema) == 0) {
            return;
        }
    }
    strcpy(tabela_simbolos[contador_simbolos].lexema, lexema);
    strcpy(tabela_simbolos[contador_simbolos].tipo, tipo);
    contador_simbolos++;
}
```

Adiciona um novo símbolo na tabela, se ele já não existir. O strcmp é para que não sejam duplicados.

2. palavra chave

```
int e_palavra_chave(const char *lexema) {
    for (int i = 0; palavras_chave[i] != NULL; i++) {
        if (!strcmp(lexema, palavras_chave[i])) return 1;
    }
    return 0;
}
```

Se o lexema for uma palavra-chave retorna `1` e `0` se não for.

3. exibir erro léxico

```
void exibir_erro_lexico() {
    printf("Erro léxico na linha %d, coluna %d\n", linha, coluna);
}
```

Mostra no terminal para o usuário que foi encontrado um erro léxico e onde foi encontrado.

4. próximo token

```
Token proximo_token(FILE *arquivo) {
}
```

Essa função é responsável por ler o arquivo caractere por caractere, ignorar espaços em branco, e reconhecer identificadores, números, operadores e símbolos. Toda vez que reconhece um token, armazena as informações (tipo, valor, linha, coluna).

5. realizar análise

```
void realizar_analise(FILE *arquivo, FILE *saida) {
}
```

Chama a função `proximo_token` até o final do programa. Armazena os tokens em um arquivo de saída e gera tabela de símbolos.

Resumindo

- ➔ O programa recebe um arquivo .pas como argumento de entrada com o arquivo para analisar.
- ➔ A função `proximo_token` lê o arquivo e identifica tokens, atualizando a posição de linha e coluna. Os tokens são armazenados em um arquivo de saída e os símbolos em uma tabela.
- ➔ Um arquivo com extensão `.lex` é gerado, com os tokens reconhecidos e a tabela de símbolos.

Testes realizados

Programas corretos

1- SomaSimples

```
program SomaSimples;
var
    a, b, resultado: integer;
begin
    a := 10;
    b := 20;
    resultado := a + b;
    writeln('O resultado da soma é: ', resultado);
end.
```

2-VerificaNumero

```
program VerificaNumero;
var
    numero: integer;
begin
    numero := 5;

    if numero > 0 then
        writeln('Numero positivo')
    else
        writeln('Numero negativo');

    while numero < 10 do
    begin
        numero := numero + 1;
        writeln('Numero: ', numero);
    end;
end.
```

3- MultiplicacaoValores

```
program MultiplicaValores;
var
    x, y, z: integer;
begin
    x := 2;
    y := 3;
    if x < y then
    begin
```

```
        z := x * y;
        writeln('Resultado: ', z);
    end;
end.
```

Programas incorretos

Erro1

```
program Erro1;
var
    a, b, resultado: integer;
begin
    a := 5;
    b := 3;
    resultado := a %% b;
    writeln('Resultado: ', resultado);
end.
```

O erro nesse caso é de sintaxe, o operador %% não existe no micropascal

Erro2

```
program Erro2
var
  1variavel: integer;
begin
  1variavel := 5;
end.
```

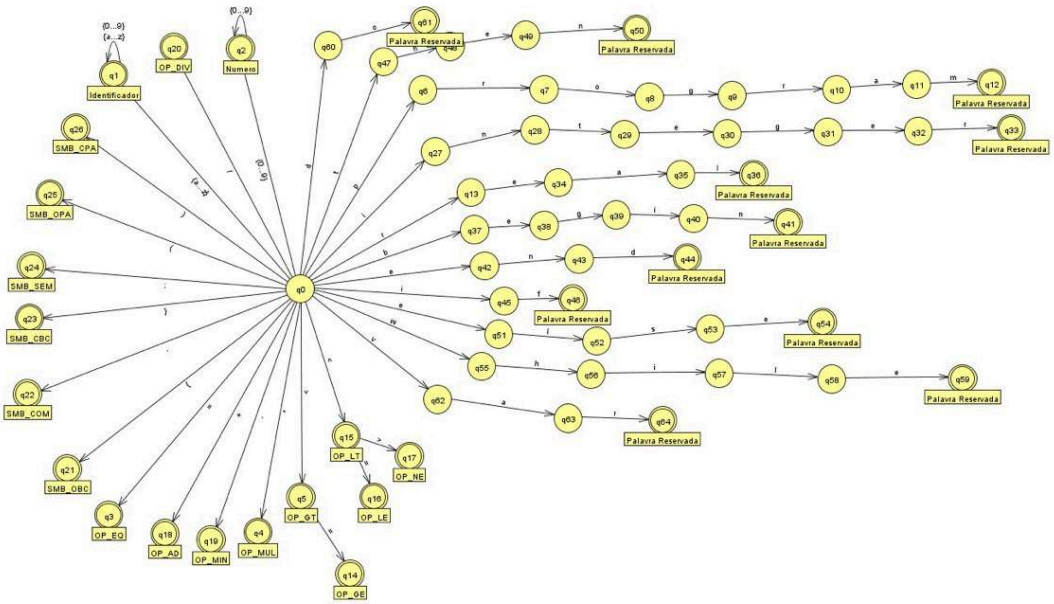
O erro nesse caso é que o identificador não pode começar com um número

Erro3

```
program Erro3;
var
  valor: integer;
begin
  valor := 10 @ 2;
end.
```

O erro nesse caso é que o @ não faz parte da gramática do pascal.

FIGURA (Autômato Finito Determinístico para reconhecimento dos tokens)



CONCLUSÃO

A implementação do analisador léxico para a linguagem MicroPascal demonstrou a importância dessa etapa na construção de compiladores. Com a utilização de um autômato finito determinístico, foi possível reconhecer corretamente os lexemas e classificá-los em tokens. A criação de uma Tabela de Símbolos eficiente permitiu o armazenamento e a reutilização de identificadores e palavras reservadas, contribuindo para a otimização do processo de análise.

A habilidade do analisador em identificar erros léxicos foi outro aspecto crucial. Problemas como caracteres inválidos, strings não fechadas e comentários mal formatados foram detectados com precisão, possibilitando a correção de erros ainda nas fases iniciais da compilação. Isso reduz significativamente o tempo de depuração e evita que erros simples avancem para fases mais complexas, onde seriam mais difíceis de corrigir.

Ao final, o analisador gerou um arquivo detalhado com todos os tokens reconhecidos, juntamente com suas respectivas posições no código-fonte. Essa funcionalidade foi essencial para oferecer uma visão clara do comportamento do analisador e do fluxo de interpretação do código.

Em suma, o projeto alcançou seus objetivos, resultando em uma ferramenta robusta e eficiente para auxiliar no processo de compilação de programas em MicroPascal.