UNIVERSIDADE DE SÃO PAULO – USP
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO
DEPARTAMENTO DE CIÊNCIAS DE COMPUTAÇÃO CURSO DE ENGENHARIA DE COMPUTAÇÃO

SCC0605 - TEORIA DA COMPUTAÇÃO E COMPILADORES

TRABALHO 1 - ANÁLISE LÉXICA

Prof. Dr. Thiago Alexandre Salgueiro Pardo
Amália Vitória de Melo Nº USP: 13692417
Bárbara Fernandes Madera - Nº USP: 11915032
Letícia Crepaldi da Cunha- Nº USP:11800879

São Carlos
2025

Sumário

1. Introdução

Este relatório tem como objetivo documentar o processo de desenvolvimento de um analisador léxico para a linguagem PL/0, realizado no contexto da disciplina SCC0605 — Teoria da Computação e Compiladores, do Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo. A linguagem PL/0, uma versão simplificada da linguagem Pascal, projetada para fins didáticos, foi escolhida para proporcionar uma abordagem prática ao estudo dos conceitos fundamentais de compiladores.

Ao longo deste documento, são descritas as etapas do trabalho, desde a construção dos autómatos e a implementação do código-fonte em linguagem C até as principais decisões de projeto tomadas pela equipe. Tanto os automatos quanto o código são feitos com comentários com a finalidade de apresentar o raciocínio de uma forma mais clara.Além disso, são apresentadas instruções para compilação e execução do analisador, bem como exemplos de funcionamento. Cada seção busca esclarecer não apenas as soluções adotadas, mas também as razões que motivaram essas escolhas do grupo. Ao final, espera-se obter um analisador léxico estruturado, funcional e coerente com os requisitos estabelecidos, capaz de ler um programa em PL/0 e gerar corretamente a sequência de pares token-classe correspondente.

2. Descrição

A partir da gramática da linguagem P–, disponibilizada no e-Disciplinas e ampliada com a inclusão do comando "for" conforme discutido em aula, deve-se proceder ao desenvolvimento do analisador léxico correspondente.

```
1  <programa> ::= program ident ; <corpo> .
2  <corpo> ::= <dc> begin <comandos> end
3  <dc> ::= <dc_c> <dc_v> <dc_p>
4  <dc_c> ::= const ident = <numero> ; <dc_c> |  lambda
5  <dc_v> ::= var <variaveis> : <tipo_var> ; <dc_v> |  lambda
6  <tipo_var> ::= real | integer
7  <variaveis> ::= ident <mais_var>
8  <mais_var> ::= , <variaveis> |  lambda
9  <dc_p> ::= procedure ident <parametros> ; <corpo_p> <dc_p> |  lambda
10 <parametros> ::= ( <lista_par> ) |  lambda
11 <lista_par> ::= <variaveis> : <tipo_var> <mais_par>
12 <mais_par> ::= ; <lista_par> |  lambda
13 <corpo_p> ::= <dc_loc> begin <comandos> end ;
14 <dc_loc> ::= <dc_v>
15 <lista_arg> ::= ( <argumentos> ) |  lambda
16 <argumentos> ::= ident <mais_ident>
17 <mais_ident> ::= ; <argumentos> |  lambda
18 <pfalsa> ::= else <cmd> |  lambda
19 <comandos> ::= <cmd> ; <comandos> |  lambda
20 <cmd> ::= read ( <variaveis> ) |
21          write ( <variaveis> ) |
22          while ( <condicao> ) do <cmd> |
23          for ident := <expressao> to <expressao> <cmd> |
24          if <condicao> then <cmd> <pfalsa> |
25          ident := <expressao> |
26          ident <lista_arg> |
27          begin <comandos> end
28 <condicao> ::= <expressao> <relacao> <expressao>
29 <relacao> ::= = | <> | >= | <= | > | <
30 <expressao> ::= <termo> <outros_termos>
31 <op_un> ::= + | - |  lambda
32 <outros_termos> ::= <op_ad> <termo> <outros_termos> |  lambda
33 <op_ad> ::= + | -
34 <termo> ::= <op_un> <fator> <mais_fatores>
35 <mais_fatores> ::= <op_mul> <fator> <mais_fatores> |  lambda
36 <op_mul> ::= * | /
37 <fator> ::= ident | <numero> | ( <expressao> )
38 <numero> ::= numero_int | numero_real
```

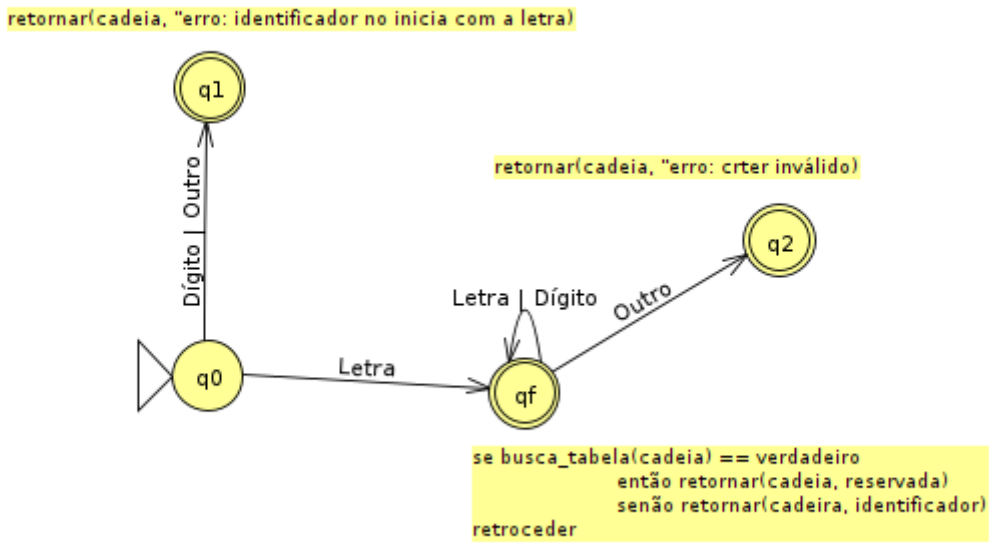
Listing 1 – Linguagem P–

Com base no código apresentado, foram elaborados autômatos utilizando a ferramenta JFLAP para autômatos, assim como o VisualStudio para a implementação dos códigos em linguagem C.

Os autômatos considerados relevantes pela equipe abrangem: identificadores, números inteiros e reais, comentários, parênteses e operadores. Os autômatos foram escolhidos porque representam os principais elementos lexicais que compõem a base sintática de programas escritos na linguagem definida pela gramática. Neste sentido, esses elementos são indispensáveis para a formação de expressões, comandos e declarações. Ademais, todos possuem padrões regulares que podem ser reconhecidos de forma eficiente por autômatos finitos determinísticos (AFDs), o que torna sua modelagem prática e adequada ao contexto de análise léxica.

3. Autômato Identificadores

Figura 1 – Autômato dos Identificadores



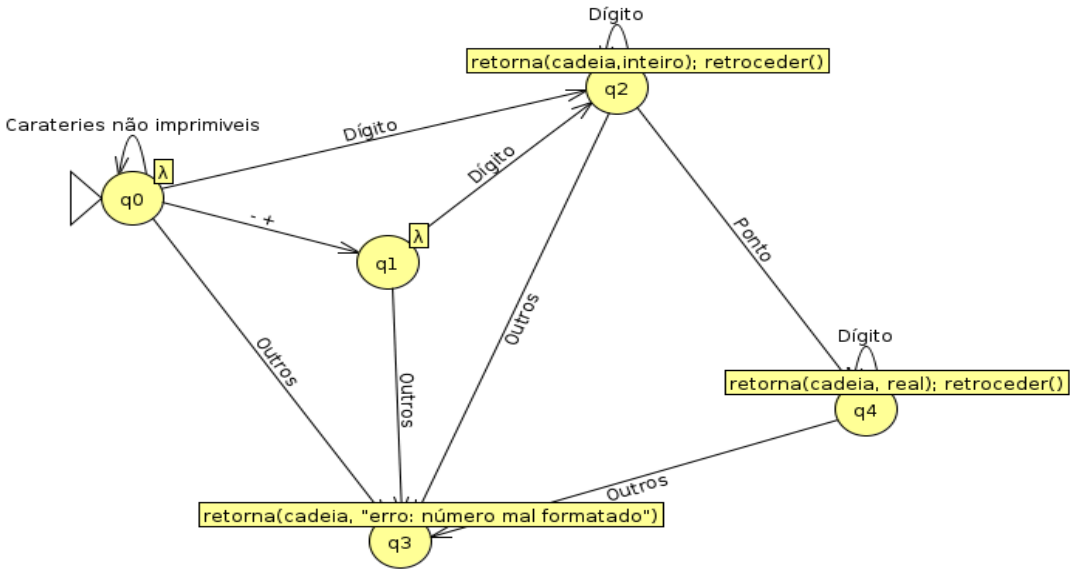
O autômato da Figura 1 tem como objetivo reconhecer identificadores válidos pela linguagem PL0. Um identificador, segundo a maioria das linguagens formais, deve obrigatoriamente começar com uma letra e pode ser seguido por letras e dígitos. Assim, o autômato inicia no estado q0, onde apenas a leitura de uma letra permite a transição para o estado final qf, que representa um identificador válido em construção. Caso o

primeiro caractere seja um dígito ou símbolo, a transição ocorre para q1, caracterizando um erro léxico, dado que o identificador não começa com uma letra.

A partir do estado qf, o autômato aceita a leitura contínua-por isso o loop- de letras e dígitos, permanecendo no mesmo estado. Quando um caractere que não seja letra nem dígito é encontrado, a transição ocorre para q2, indicando o término do identificador e a necessidade de o analisador continuar processando o restante do código. Dessa forma, o autômato implementa as regras para formação de identificadores, distinguindo entradas válidas das inválidas com base nas transições entre seus estados.

4. Autômato Números Inteiros e Reais

Figura 2 – Autômato dos Números Inteiros e Reais



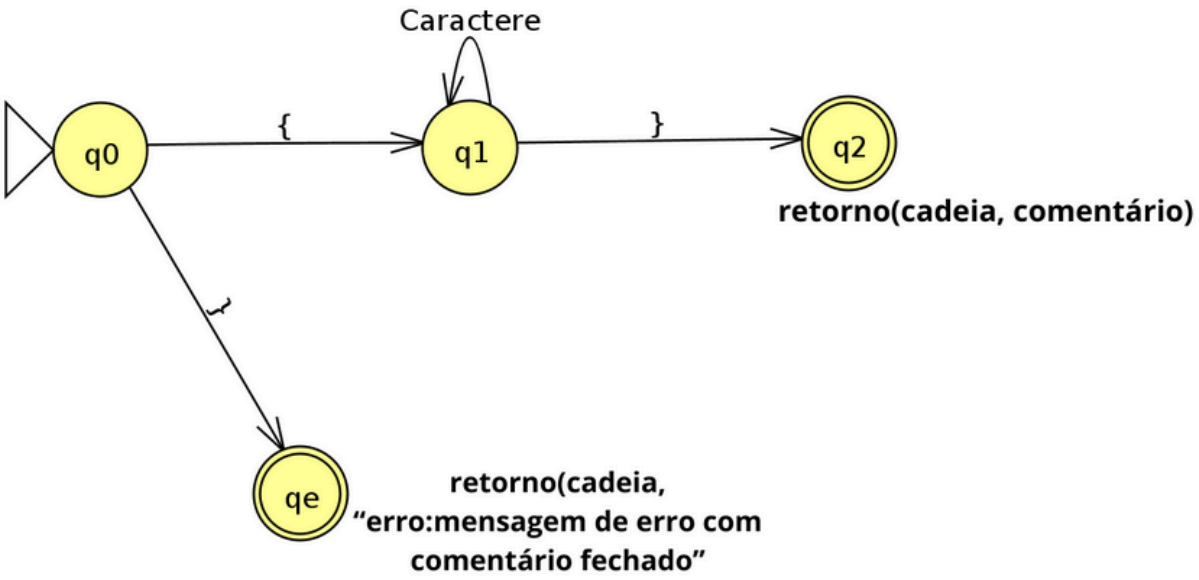
O autômato apresentado tem como objetivo reconhecer números inteiros e reais, validando a sintaxe de acordo com as regras lexicais da linguagem. O estado inicial q0 aceita espaços ou caracteres não imprimíveis e transita para q1 ao encontrar um sinal (+ ou -) ou diretamente para q2 ao encontrar um dígito. O estado q1 possui a finalidade de ser intermediário para aceitar um número com sinal, exigindo que o próximo caractere seja um

dígito para alcançar o estado q2. O estado q2, por sua vez, representa um número inteiro válido, permitindo a leitura contínua de dígitos. A transição de q2 para q4 ocorre quando um ponto (.) é lido, indicando,assim, o início de um número real. Quando o estado q4 aceitar mais dígitos, isso significa o final da leitura, retornando-se, com isso, o número como real.

Caso um caractere inválido seja lido durante qualquer uma das fases de construção do número, o autômato transita para o estado de erro q3, onde a cadeia é rejeitada e retorna uma mensagem de número mal formatado. Estados q2 e q4 são finais, indicando que a leitura do número foi concluída com sucesso — sendo q2 e q4 números inteiro e para reais,respectivamente . Ambos executam a função `retorna(...); retroceder()` para devolver o token ao analisador. Com essa estrutura, o autômato garante a distinção entre números válidos e mal formados, permitindo também a presença opcional de sinais no início.

5. Autômato Comentários

Figura 3 – Autômato dos Comentários



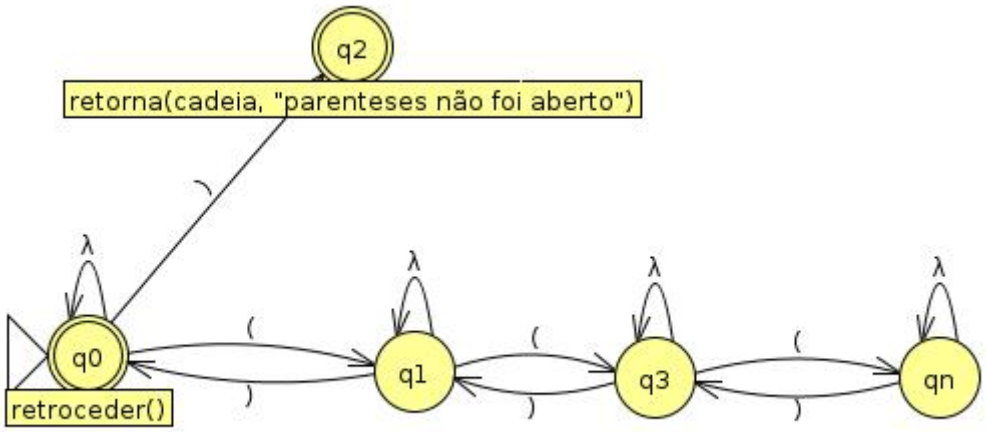
Este autômato tem como objetivo identificar comentários delimitados por chaves {...}. Ele começa no estado inicial q0. Ao encontrar um caractere {, transita para o estado q1, indicando o início de um comentário. Quando no estado q1, ele aceita qualquer caractere (representado genericamente por "Caractere"), permanecendo em q1 até que encontre a

chave de fechamento `}`. Ao identificar o `}`, o autômato transita para o estado `q2`, que é um estado final e representa o reconhecimento correto de um comentário. Nesse ponto, ocorre a ação `retorno(cadeia, comentário)`.

Caso, no estado inicial `q0`, o autômato encontre diretamente um `}` (ou seja, um fechamento de comentário sem abertura), ele transita imediatamente para o estado de erro `qe`. Este estado representa uma situação inválida, onde um comentário foi fechado sem ter sido iniciado. Quando isso ocorre, a ação executada é `retorno(cadeia, "erro: mensagem de erro com comentário fechado")`, informando que houve erro léxico relacionado à estrutura de comentários. Este autômato é útil para garantir que os comentários sejam corretamente abertos e fechados no código-fonte analisado.

6 Autômato Parênteses

Figura 4 – Autômato dos Parênteses



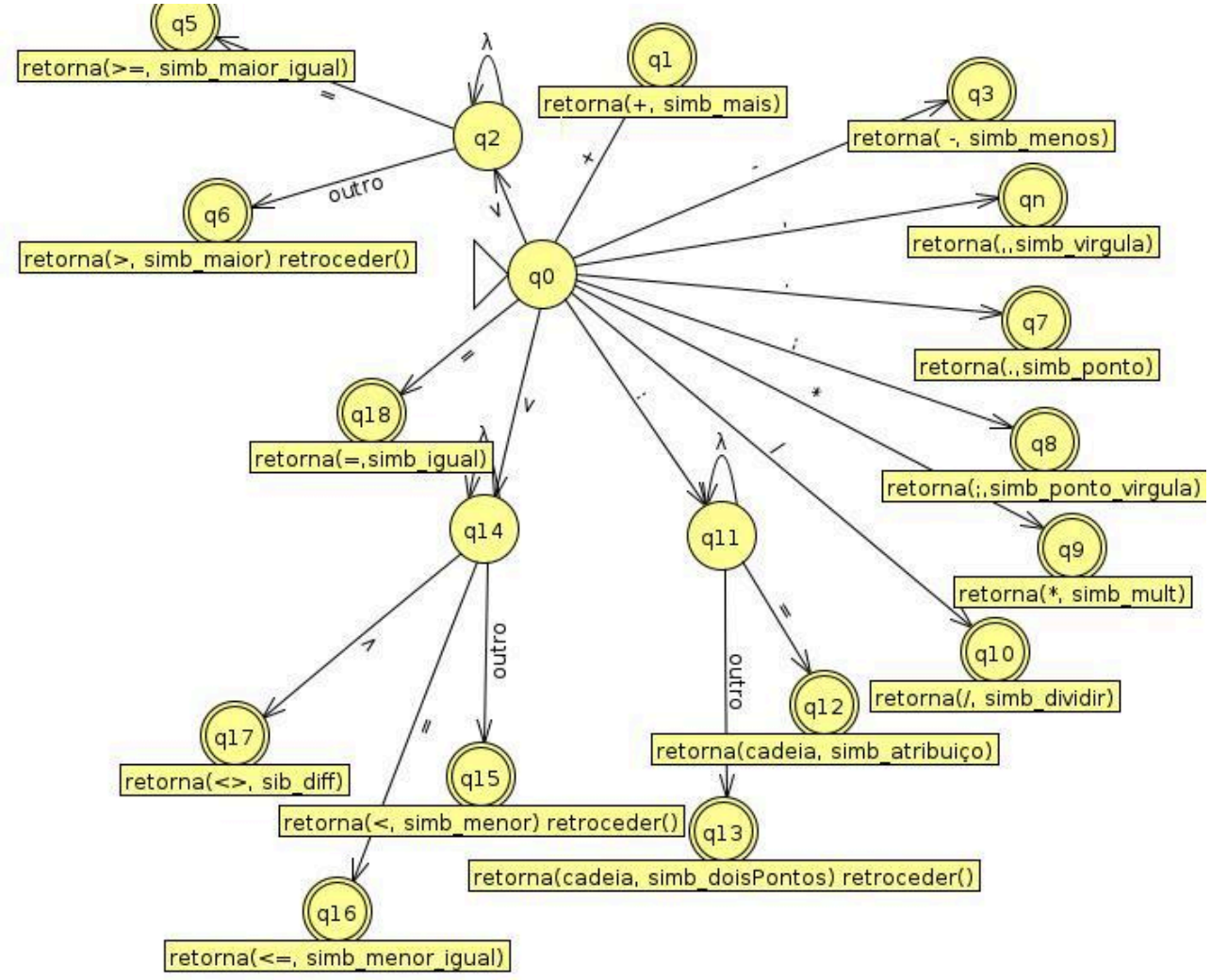
Este autômato tem como objetivo validar se os parênteses estão abertos e fechados em uma cadeia de entrada de forma correta. O processo começa no estado inicial `q0`. Ao encontrar um parêntese de abertura `'('`, o autômato transita para o próximo estado (`q1`, depois `q3`, e assim por diante), indicando que houve uma abertura. Em cada novo estado, ele espera encontrar um fechamento `')'` correspondente. Quando isso ocorre, o autômato volta para o estado anterior, representando o balanceamento de parênteses — isto é, realizando uma simulação de empilhamento.

A cadeia é considerada correta se, após todas as transições, o autômato retorna ao

estado inicial ou permanece em um estado final como qn, representando que todos os parênteses abertos foram fechados. Caso encontre um parêntese de fechamento) logo no início (q0), sem ter visto um de abertura antes, ele vai para o estado de erro q2, e retorna a mensagem: "parênteses não foi aberto", sinalizando um erro léxico. Essa abordagem garante que a estrutura correta de expressões envolvendo agrupamentos seja corretamente implementada, operando-se assim tal qual funções ou operações matemáticas.

7 Autômato Operadores

Figura 5 – Autômato dos Operadores



Este autômato tem como objetivo identificar corretamente os operadores aritméticos, relacionais e símbolos de pontuação usados na linguagem de programação, como +, -, *,

`/`, `:=`, `>=`, `<=`, `<>`, `;`, `,`, `.` entre outros símbolos matemáticos. Neste sentido, ele começa no estado inicial `q0`, onde cada símbolo lido direciona para um estado específico que retorna o token correspondente. Por exemplo, ao ler `+`, transita para `q1`, que retorna o símbolo "simb_mais", analogamente ao ler `-`, vai para `q3`, que retorna "simb_menos". Símbolos simples como `;`, `,`, `*`, `/` e `.` são tratados de forma direta com transições únicas para estados de aceitação.

Símbolos compostos, por outro lado, como `<=`, `>=`, `<>` e `:=`, são tratados com transições múltiplas. Exemplificando, se o autômato lê `<`, ele vai para `q4` e espera um segundo caractere. Se esse caractere for `=`, ele vai para `q16` e retorna "simb_menor_igual". Se for `>`, vai para `q17` e retorna "simb_diff", e se for outro caractere, vai para `q15` e retorna apenas "simb_menor". Isso ocorre de forma semelhante com os casos de `>` (indo para `q5` ou `q6`) e `:` (para `q13` ou `q12`). O estado intermediário `q2` é usado para consumir espaços e verificar próximos caracteres quando há ambiguidade. O autômato cobre todas as possibilidades e garante que cada operador seja corretamente identificado, mesmo que esteja incompleto ou mal formulado, retornando adequadamente com `retroceder()` quando necessário.

8. Análise do código fonte

8.1 Como compilar o código-fonte

Para compilar o analisador léxico escrito em linguagem C, é necessário utilizar um compilador como o `gcc`. Sendo o código salvo em um arquivo chamado `main.c`, é necesário abrir o terminal na pasta onde esse arquivo está localizado e digitar o comando `gcc main.c -o analisador`. Isso criará um executável chamado `analisador` (ou `analisador.exe` no Windows). Com o executável gerado, a execução do programa pode ser feita com o comando `./analisador` no Linux ou `analisador` no Windows. É importante garantir que o arquivo de entrada `compilador.txt` esteja na mesma pasta, pois é ele que será analisado.

8.2 Exemplos de execução

Ao rodar o analisador, ele processará linha por linha do arquivo `compilador.txt`, reconhecendo os diferentes tokens da linguagem. A saída com todos os tokens

encontrados será salva automaticamente no arquivo `saida.txt`, mostrando para cada lexema seu tipo e se foi identificado corretamente ou não.

```
program fatorial;
{exemplo 1}
var x, aux, fat: integer;
begin
read(x);
fat:=1;
for aux:=1 to x do
begin
fat:=fat*aux;
end;
write(fat);
end
```

Entrada Exemplo 01 - Compilador.txt - Fatorial

```
program, ident
fatorial, ident
;, simbolo_ponto_e_virgula
{exemplo 1}, comentario
var, ident
x, ident
,, simbolo_virgula
aux, ident
,, simbolo_virgula
fat, ident
: , simbolo_dois_pontos
integer, ident
;, simbolo_ponto_e_virgula
begin, ident
read, ident
(, parenteses_esquerdo
x, ident
), parenteses_direito
;, simbolo_ponto_e_virgula
fat, ident
:=, simbolo_atribuicao
1, numero
;, simbolo_ponto_e_virgula
for, ident
aux, ident
:=, simbolo_atribuicao
```

Saída Exemplo 01 - Saida.txt - Fatorial

9. Explicação do Código implementado

Esse código em C implementa um analisador léxico para uma linguagem de programação fictícia. Ele lê um arquivo chamado `compilador.txt` e escreve os resultados da análise no arquivo `saida.txt`. A análise léxica reconhece os seguintes tipos de tokens: identificadores, números (inteiros), comentários, operadores relacionais, operadores aritméticos, parênteses e pontuação. Veja o que o programa faz por partes:

9.1. Estrutura geral

- O código define constantes e tipos, incluindo palavras reservadas e símbolos esperados.
- Cada tipo de token (como identificador, número, operador etc.) tem sua própria função, a qual realiza a tentativa de reconhecer esse token no ponto atual da leitura do arquivo.

9.2. Funcionamento

- A função `main` lê o arquivo de entrada caractere por caractere e tenta identificar tokens, chamando cada função de reconhecimento em sequência.
- A verificação é feita multiplicando os retornos das funções (ENCONTRADO = 5, NAO_ENCONTRADO = 2, FIM_DE_ARQUIVO = 3), e se nenhuma verificação reconhecer o token, ele trata como erro léxico.
- Quando um token válido é identificado, ele é impresso no arquivo de saída com seu tipo (ex: `:=, simbolo_atribuicao`).
- Comentários devem estar entre `{` e `}`, e erros como comentário não fechado ou número mal formatado também são registrados com `ERRO_LEXICO`.

Esse código é uma simulação funcional de um autômato léxico, onde cada função representa um subautômato para tipos de tokens específicos.

9.3. Versão Comentada

Há duas versões presentes no arquivo mandado. Uma delas chama-se “main_comentada.c”. Ela foi adicionada para ser mais didática e fornecer mais informações para análise caso haja dúvida sobre algo no código. Sugere-se que

ela seja vista quando possível.

10. Conclusão

O desenvolvimento do analisador léxico para a linguagem PL/0 permitiu consolidar diversos conceitos importantes abordados ao longo da disciplina de Teoria da Computação e Compiladores. Durante a execução do projeto, foi possível aplicar, de maneira prática, o estudo de autômatos, o tratamento de tokens, o reconhecimento de padrões e o manejo de erros léxicos, reforçando o aprendizado de uma estruturação cuidadosa e modularizada do código.

A construção dos autômatos, a definição das transições de estados e a implementação em linguagem C, aliados à utilização de ferramentas de apoio, proporcionaram uma solução funcional e alinhada aos requisitos estabelecidos na proposta. Além disso, as decisões de projeto tomadas ao longo do processo buscaram sempre favorecer a usabilidade do analisador, simulando um ambiente real de desenvolvimento de compiladores.

O projeto resultou em um sistema capaz de reconhecer corretamente os elementos léxicos da linguagem PL/0, gerar a saída no formato especificado e detectar possíveis inconsistências no código de entrada. Dessa forma, cumpre-se o objetivo de construir uma base sólida para as próximas etapas do curso, nas quais a análise sintática e semântica serão incorporadas.