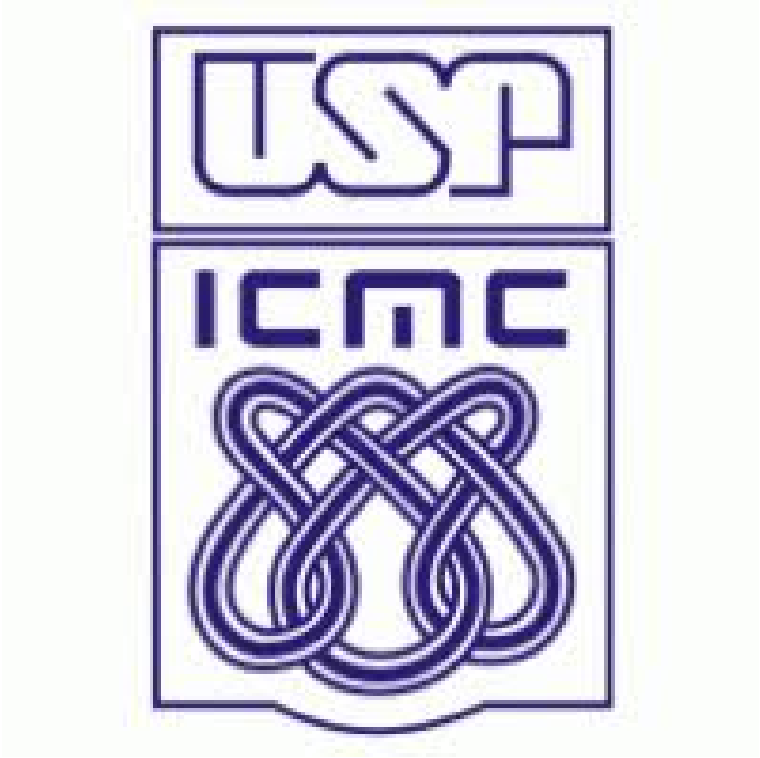

Construção de um Analisador Léxico: Estrutura e Funcionamento

Alunos:

- Giovanni dos Santos – 13695341
- Guilherme Augusto Fincatti da Silva – 13676986
- Marcelo Eduardo Reginato – 13676965
- Pedro Guilherme de Barros Zenatte – 13676919
- Rhayna Christiani Vasconcelos Marques Casado – 13676429

Professor: Thiago Alexandre Salgueiro Pardo

São Carlos, 2025

Conteúdo

1 Objetivo

O principal objetivo deste trabalho é desenvolver um analisador léxico para a linguagem PL/0, capaz de ler um programa fonte escrito em um arquivo .txt e produzir como saída outro arquivo .txt contendo, linha a linha, o par token-classe correspondente a cada elemento reconhecido. O analisador também deve ser capaz de identificar e reportar eventuais erros léxicos encontrados no programa de entrada, por exemplo, o erro de identificador mal formado.

Para alcançar esse objetivo, foi necessário projetar e implementar um autômato de estados finitos (AFD) capaz de reconhecer os diferentes padrões da linguagem, bem como definir uma tabela de transição de estados que orientasse o processamento dos símbolos lidos, além de ser a nossa tabela de transições para a implementação do próprio código. Além disso, decisões importantes de projeto foram tomadas, tais como a estratégia de retrocesso no automato (lookahead), a forma de representar as classes dos tokens e o tratamento adequado para símbolos inválidos.

A correta implementação do analisador léxico é essencial para a construção de um compilador, pois representa a primeira fase da análise do código-fonte, preparando a estrutura de tokens que será utilizada nas etapas seguintes de análise sintática e semântica.

2 Materiais e Métodos

Para o desenvolvimento do analisador léxico, foi utilizada a linguagem de programação C, uma vez que possui elevada eficiência e controle de baixo nível sobre arquivos e estruturas de dados. O ambiente de desenvolvimento utilizado foi o Visual Studio Code junto ao GitHub para controle de versões e colaboração entre os membros da equipe. Esse repositório online permitiu organizar o código, acompanhar as alterações realizadas e integrar o trabalho de forma eficiente e segura.

Além disso, para auxiliar na construção e validação dos autômatos finitos determinísticos (AFDs) com saída, utilizamos a ferramenta JFLAP, que permitiu a criação gráfica dos autômatos e facilitou a verificação dos comportamentos esperados durante a leitura dos tokens. Sendo assim, a partir dos autômatos modelados foi possível construir a tabela de transições, a qual

descreve as mudanças de estado para cada símbolo lido.

A implementação do analisador léxico foi desenvolvida com base nesses autômatos, realizando a leitura do arquivo de entrada caractere por caractere e direcionando-o ao estado apropriado conforme o símbolo lido. Para lidar com símbolos compostos e ambiguidades, foram adotadas estratégias como o retrocesso (lookahead), permitindo a análise correta de padrões que exigem a leitura do próximo caractere para a tomada de decisão. Um exemplo disso são os identificadores, no qual é necessário analisar o último caractere, mas sem incluí-lo no par token-classe.

Cada token identificado é associado a uma classe descritiva, e o par token-classe é escrito no arquivo de saída. Dessa forma, em casos de erro léxico, como a presença de símbolos inválidos na sequência de um determinado estado, o programa registra o erro na saída com a anotação apropriada, por exemplo, `ERRO_NUMERO_MAL_FORMADO` em caso de letras quando estamos formando dígitos.

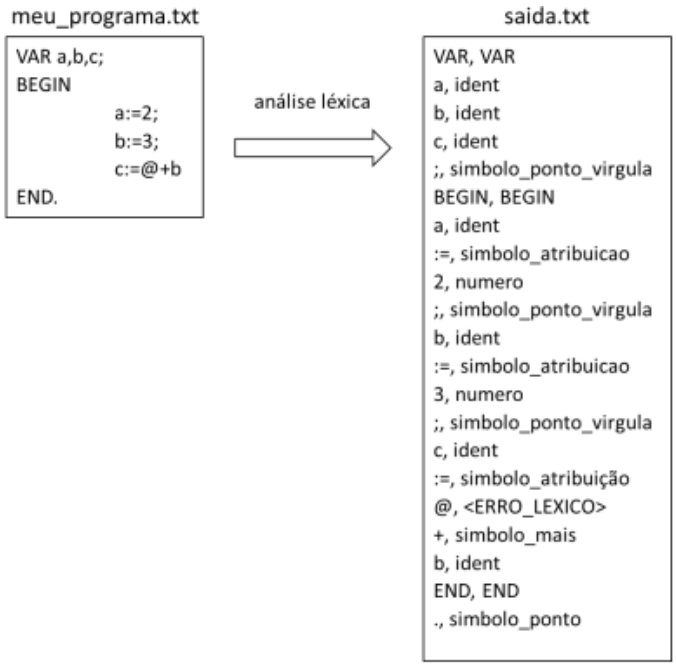

Figura 1: Exemplo de token-classe na saída do programa

Note que, para os lexemas correspondentes a palavras reservadas, isto é, as palavras brutas escritas em maiúsculo, como VAR, o par token-classe é formado pelo próprio lexema.

Durante o projeto, foram necessárias diversas decisões de implementação para lidar com questões desse tipo, como a definição do tratamento das pala-

vras reservadas, a implementação do autômato em si, o formato da tabela de transições, a criação de vetores auxiliares para seu correto funcionamento, a estratégia de armazenamento dos tokens lidos e a estrutura de controle dos estados do autômato, sempre visando garantir clareza, facilidade de manutenção e robustez no tratamento dos dados.

Em relação às palavras reservadas, optamos por não criar um autômato específico para cada uma delas, o que tornaria o autômato excessivamente grande, poluído e ineficiente. Em vez disso, implementamos uma tabela hash, aproveitando sua alta eficiência de busca. Assim sendo, sempre que o programa identifica um token como um identificador, ele consulta a tabela para verificar se o lexema corresponde a uma palavra reservada. Caso corresponda, o token identificado é então classificado corretamente como uma palavra reservada ao invés de um idenficador.

Quanto à implementação do autômato, decidimos utilizar uma matriz de transição para evitar uma solução "ad hoc"e tornar o código o mais elegante e estruturado possível.

Quando nos referimos à tal matriz de transição no código, estamos falando de uma matriz de inteiros, onde cada linha representa um estado e cada coluna representa um símbolo. No entanto, ao invés de criar uma matriz gigantesca, incluindo todos os caracteres do alfabeto (a-z, A-Z), todos os dígitos (0-9) e todos os símbolos especiais, o que, além de trazer mais trabalho para associar um caractere a um inteiro, deixaria o código muito poluído, optamos por uma solução que consideramos mais eficiente e que torna o código mais organizado, isto é, criamos um vetor de caracteres contendo apenas os símbolos que o analisador léxico reconhece.

Desse modo, com o apoio de uma função auxiliar, associamos cada símbolo desse vetor a uma coluna específica da matriz de transição, economizando espaço.

Por exemplo, o vetor de símbolos, um vetor de caracteres, guarda todos os caracteres reconhecidos. As posições de 0 até 51 no vetor correspondem às letras (a-z e A-Z), mas todas essas letras são tratadas como pertencentes à mesma coluna (coluna 0) na matriz de transição, uma matriz de inteiros. Dessa forma, a função auxiliar é responsável por mapear essas posições do vetor para o índice 0 da matriz. O mesmo raciocínio é aplicado aos dígitos e símbolos especiais.

Contudo, para cada símbolo especial, por sua vez, temos apenas um ajuste, isto é, sua posição no vetor é deslocada (shiftada) de forma que, por exemplo, o símbolo ':' que ocupa a posição 62 no vetor, é associado à coluna 2 na matriz de transição. Esse processo continua para todos os símbolos reconhecidos, garantindo uma representação compacta e eficiente.

Portanto, com a matriz de transição podemos garantir um código mais in-

dependente do autômato e mais bem organizado, tomando sempre o cuidado de colocar o vetor de símbolos em sincronia com a matriz de transição.

Uma última decisão importante foi o tratamento dos estados finais. Para isso, adotamos a convenção de representaá-los como constantes negativas. Dessa forma, sempre que a função de transição retorna um valor menor que zero, sabemos que chegamos a um estado final. Para distinguir entre os diferentes estados finais, associamos cada constante negativa a uma posição específica em um vetor de strings que descreve os estados finais. Por exemplo, o estado `TK_ID` é representado pelo valor -3, o qual corresponde à posição 2 no vetor de estados finais (calculada como -3 * (-1) - 1).

3 Resultados

O desenvolvimento do analisador léxico envolveu diversas etapas que se complementam para a construção de um sistema capaz de identificar corretamente os elementos da gramática da linguagem PL/0. Inicialmente, foram elaborados os autômatos responsáveis pelo reconhecimento dos diferentes padrões léxicos, considerando as particularidades específicas de cada token. A partir desses autômatos, consolidou-se uma especificação formal dos tokens, reunindo em uma tabela suas expressões regulares, representações e categorias associadas.

Sendo assim, com base na estrutura dos autômatos e nessas especificações dos tokens, foi possível construir a tabela de transição de estados, inicialmente elaborada como uma tabela separada, antes de ser implementada no código, a fim de melhor organização e visualização, que orienta a leitura e o processamento dos símbolos durante a análise. Por fim, com essas ferramentas foi viável implementar o código e, após a implementação do analisador, exemplos práticos de entrada e saída ilustram o funcionamento do sistema, evidenciando a identificação correta dos tokens e a detecção adequada de erros léxicos.

3.1 Descrição dos Autômatos

A base do funcionamento do analisador léxico é formada pelos autômatos finitos determinísticos com saída, neste caso utilizamos as Máquinas de Moore, elaborados para reconhecer os diferentes tokens da linguagem PL/0. Cada autômato foi projetado considerando os padrões léxicos a serem reconhecidos pela gramática e validado com o apoio da ferramenta JFLAP.

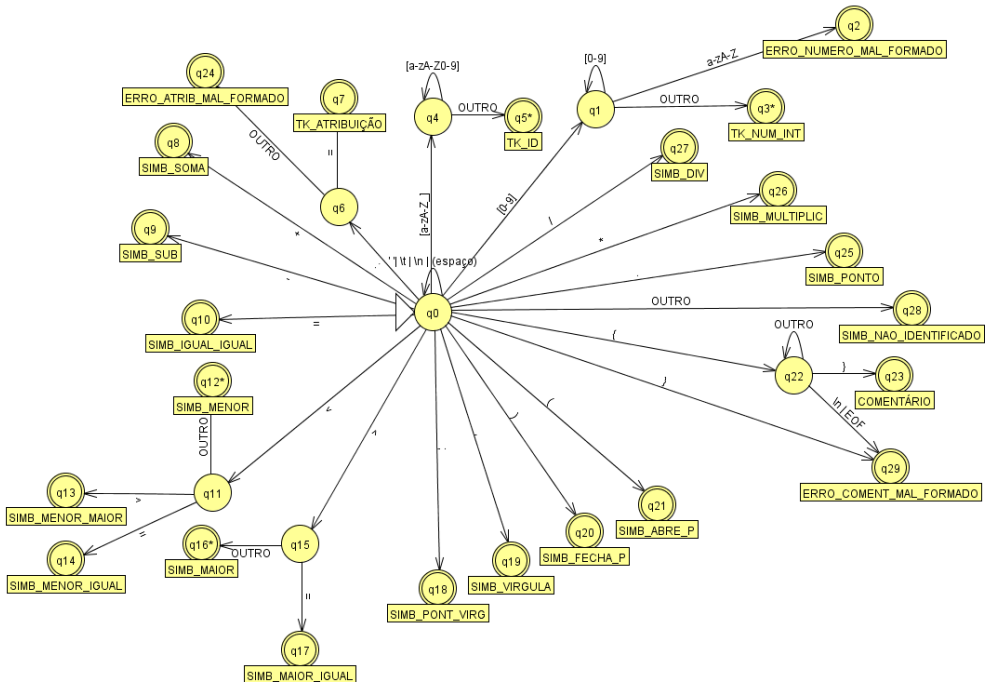

Figura 2: Autômato para Analisador Léxico

3.2 Especificação dos Tokens

Com os autômatos definidos, foi possível formalizar a especificação dos tokens aceitos.

A tabela a seguir apresenta as expressões regulares associadas a cada token, sua representação simbólica e a categoria correspondente, servindo como referência para a análise léxica.

Expressão Regular	Token	Categoria do Token
D = [0-9]	———	
L = [a-zA-Z_]	———	
NUM_INT = D+	TK_NUM_INT	Número inteiro (número)
IDENT = L (L | D)*	TK_ID	Identificador
CONST = CONST	TK_CONSTANTE	Palavra reservada
VAR = VAR	TK_VAR	Palavra reservada
PROCEDURE = PROCEDURE	TK_PROCEDURE	Palavra reservada
CALL = CALL	TK_CALL	Palavra reservada
BEGIN = BEGIN	TK_BEGIN	Palavra reservada
IF = IF	TK_IF	Palavra reservada
THEN = THEN	TK_THEN	Palavra reservada
WHILE = WHILE	TK_WHILE	Palavra reservada
DO = DO	TK_DO	Palavra reservada
ODD = ODD	TK_ODD	Palavra reservada
:=	TK_ATRIBUIÇÃO	Operador de atribuição
+	SIMB_SOMA	Operador aritmético unitário
-	SIMB_SUB	Operador aritmético unitário
=	SIMB_IGUAL_IGUAL	Operador relacional
<>	SIMB_MENOR_MAIOR	Operador relacional
<	SIMB_MENOR	Operador relacional
<=	SIMB_MENOR_IGUAL	Operador relacional
>	SIMB_MAIOR	Operador relacional
>=	SIMB_MAIOR_IGUAL	Operador relacional
(	SIMB_ABRE_P	Símbolo especial
)	SIMB_FECHA_P	Símbolo especial
,	SIMB_VIRGULA	Símbolo especial
;	SIMB_PONT_VIRG	Símbolo especial
.	SIMB_PONTO	Símbolo especial
*	SIMB_MULTIPLIC	Símbolo especial
/	SIMB_DIV	Símbolo especial
COMENTARIO = {.*}	———	Comentário

Tabela 1: Especificação dos Tokens

3.3 Tabela de Transição de Estados

A partir da estrutura dos autômatos, foi construída uma tabela de transição de estados, que define o comportamento do analisador a cada novo símbolo lido.

Essa tabela orienta a movimentação entre estados e garante a correta identificação dos padrões esperados, além de auxiliar o programador a enxergar melhor o que está acontecendo na matriz de transição elaborada na implementação.

Para melhor visualização da tabela, voce pode acessar o link da tabela por aqui: Tabela de Transições

Como exemplo, podemos analisar a função de transição para a fita **AAB+**:

Estado (δ)	LETRA	DÍGITO	:	=	+	-	<	>	;	,	(	)	{	}	'	\t	\n	espaço	.	*	/	EOF	Caractere Estrangeiro (!, ...)
0	4	1	6	10	8	9	11	15	18	19	21	20	22	29	0	0	0	0	25	26	27	0	28
1	2	1	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
2	ERRO_NUMERO_MAL_	ERRO_NUMERO_MAL_FORMA	ERRO_NUMER	ERRO_NUMER	ERRO_NUMER	ERRO_NUMER	ERRO_NUMER	ERRO_NUMER	ERRO_NUMER	ERRO_NUMER	ERRO_NUMER	ERRO_NUMER	ERRO_NUMER	ERRO_NUMER	ERRO_NUMER	ERRO_NUMERO_M	ERRO_NUMER	ERRO_NUMER	ERRO_NUME	ERRO_NUME	ERRO_NUMERO	ERRO_NUMERO	ERRO_NUMERO_MAL_FORMADO
3	TK_NUM_INT	TK_NUM_INT	TK_NUM_INT	TK_NUM_INT	TK_NUM_INT	TK_NUM_INT	TK_NUM_INT	TK_NUM_INT	TK_NUM_INT	TK_NUM_INT	TK_NUM_INT	TK_NUM_INT	TK_NUM_INT	TK_NUM_INT	TK_NUM_INT	TK_NUM_INT	TK_NUM_INT	TK_NUM_INT	TK_NUM_INT	TK_NUM_INT	TK_NUM_INT	TK_NUM_INT	TK_NUM_INT
4	4	4	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
5	TK_ID	TK_ID	TK_ID	TK_ID	TK_ID	TK_ID	TK_ID	TK_ID	TK_ID	TK_ID	TK_ID	TK_ID	TK_ID	TK_ID	TK_ID	TK_ID	TK_ID	TK_ID	TK_ID	TK_ID	TK_ID	TK_ID	TK_ID
6	24	24	24	7	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24
7	TK_ATRIBUIÇÃO	TK_ATRIBUIÇÃO	TK_ATRIBUIÇÃ	TK_ATRIBUIÇÃ	TK_ATRIBUIÇÃ	TK_ATRIBUIÇÃ	TK_ATRIBUIÇÃ	TK_ATRIBUIÇÃ	TK_ATRIBUIÇÃ	TK_ATRIBUIÇÃ	TK_ATRIBUIÇÃ	TK_ATRIBUIÇÃ	TK_ATRIBUIÇÃ	TK_ATRIBUIÇÃ	TK_ATRIBUIÇÃ	TK_ATRIBUIÇÃO	TK_ATRIBUIÇÃ	TK_ATRIBUIÇÃ	TK_ATRIBUIÇÃ	TK_ATRIBUIÇ	TK_ATRIBUIÇÃO	TK_ATRIBUIÇÃO	TK_ATRIBUIÇÃO
8	SIMB_SOMA	SIMB_SOMA	SIMB_SOMA	SIMB_SOMA	SIMB_SOMA	SIMB_SOMA	SIMB_SOMA	SIMB_SOMA	SIMB_SOMA	SIMB_SOMA	SIMB_SOMA	SIMB_SOMA	SIMB_SOMA	SIMB_SOMA	SIMB_SOMA	SIMB_SOMA	SIMB_SOMA	SIMB_SOMA	SIMB_SOMA	SIMB_SOMA	SIMB_SOMA	SIMB_SOMA	SIMB_SOMA
9	SIMB_SUB	SIMB_SUB	SIMB_SUB	SIMB_SUB	SIMB_SUB	SIMB_SUB	SIMB_SUB	SIMB_SUB	SIMB_SUB	SIMB_SUB	SIMB_SUB	SIMB_SUB	SIMB_SUB	SIMB_SUB	SIMB_SUB	SIMB_SUB	SIMB_SUB	SIMB_SUB	SIMB_SUB	SIMB_SUB	SIMB_SUB	SIMB_SUB	SIMB_SUB
10	SIMB_IGUAL_IGUAL	SIMB_IGUAL_IGUAL	SIMB_IGUAL_	SIMB_IGUAL_	SIMB_IGUAL_	SIMB_IGUAL_	SIMB_IGUAL_	SIMB_IGUAL_	SIMB_IGUAL_	SIMB_IGUAL_	SIMB_IGUAL_	SIMB_IGUAL_	SIMB_IGUAL_	SIMB_IGUAL_	SIMB_IGUAL_	SIMB_IGUAL_IGU	SIMB_IGUAL_	SIMB_IGUAL_	SIMB_IGUAL_	SIMB_IGUAL_	SIMB_IGUAL_IG	SIMB_IGUAL_IG	SIMB_IGUAL_IGUAL
11	12	12	12	14	12	12	12	13	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12
12	SIMB_MENOR	SIMB_MENOR	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENOR	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENOR	SIMB_MENOR	SIMB_MENOR
13	SIMB_MENOR_MAIO	SIMB_MENOR_MAIOR	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENOR_MA	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENOR_M	SIMB_MENOR_M	SIMB_MENOR_MAIOR
14	SIMB_MENOR_IGUA	SIMB_MENOR_IGUAL	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENOR_IG	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENO	SIMB_MENOR_I	SIMB_MENOR_I	SIMB_MENOR_IGUAL
15	16	16	16	17	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16
16	SIMB_MAIOR	SIMB_MAIOR	SIMB_MAIOR	SIMB_MAIOR	SIMB_MAIOR	SIMB_MAIOR	SIMB_MAIOR	SIMB_MAIOR	SIMB_MAIOR	SIMB_MAIOR	SIMB_MAIOR	SIMB_MAIOR	SIMB_MAIOR	SIMB_MAIOR	SIMB_MAIOR	SIMB_MAIOR	SIMB_MAIOR	SIMB_MAIOR	SIMB_MAIOR	SIMB_MAIOR	SIMB_MAIOR	SIMB_MAIOR	SIMB_MAIOR
17	SIMB_MAIOR_IGUAL	SIMB_MAIOR_IGUAL	SIMB_MAIOR	SIMB_MAIOR	SIMB_MAIOR	SIMB_MAIOR	SIMB_MAIOR	SIMB_MAIOR	SIMB_MAIOR	SIMB_MAIOR	SIMB_MAIOR	SIMB_MAIOR	SIMB_MAIOR	SIMB_MAIOR	SIMB_MAIOR	SIMB_MAIOR_IGU	SIMB_MAIOR	SIMB_MAIOR	SIMB_MAIOR	SIMB_MAIOR	SIMB_MAIOR_IG	SIMB_MAIOR_IG	SIMB_MAIOR_IGUAL
18	SIMB_PONT_VIRG	SIMB_PONT_VIRG	SIMB_PONT_	SIMB_PONT_	SIMB_PONT_	SIMB_PONT_	SIMB_PONT_	SIMB_PONT_	SIMB_PONT_	SIMB_PONT_	SIMB_PONT_	SIMB_PONT_	SIMB_PONT_	SIMB_PONT_	SIMB_PONT_	SIMB_PONT_VIR	SIMB_PONT_	SIMB_PONT_	SIMB_PONT_	SIMB_PONT_	SIMB_PONT_VIR	SIMB_PONT_VIR	SIMB_PONT_VIRG
19	SIMB_VIRGULA	SIMB_VIRGULA	SIMB_VIRGU	SIMB_VIRGU	SIMB_VIRGU	SIMB_VIRGU	SIMB_VIRGU	SIMB_VIRGU	SIMB_VIRGU	SIMB_VIRGU	SIMB_VIRGU	SIMB_VIRGU	SIMB_VIRGU	SIMB_VIRGU	SIMB_VIRGU	SIMB_VIRGULA	SIMB_VIRGU	SIMB_VIRGU	SIMB_VIRGU	SIMB_VIRGU	SIMB_VIRGULA	SIMB_VIRGULA	SIMB_VIRGULA
20	SIMB_FECHA_P	SIMB_FECHA_P	SIMB_FECHA	SIMB_FECHA	SIMB_FECHA	SIMB_FECHA	SIMB_FECHA	SIMB_FECHA	SIMB_FECHA	SIMB_FECHA	SIMB_FECHA	SIMB_FECHA	SIMB_FECHA	SIMB_FECHA	SIMB_FECHA	SIMB_FECHA_P	SIMB_FECHA	SIMB_FECHA	SIMB_FECHA	SIMB_FECHA	SIMB_FECHA_P	SIMB_FECHA_P	SIMB_FECHA_P
21	SIMB_ABRE_P	SIMB_ABRE_P	SIMB_ABRE_	SIMB_ABRE_	SIMB_ABRE_	SIMB_ABRE_	SIMB_ABRE_	SIMB_ABRE_	SIMB_ABRE_	SIMB_ABRE_	SIMB_ABRE_	SIMB_ABRE_	SIMB_ABRE_	SIMB_ABRE_	SIMB_ABRE_	SIMB_ABRE_P	SIMB_ABRE_	SIMB_ABRE_	SIMB_ABRE_	SIMB_ABRE_	SIMB_ABRE_P	SIMB_ABRE_P	SIMB_ABRE_P
22	22	22	22	22	22	22	22	22	22	22	22	22	22	23	22	22	29	22	22	22	22	29	22
23	COMENTARIO	COMENTARIO	COMENTARI	COMENTARI	COMENTARI	COMENTARI	COMENTARI	COMENTARI	COMENTARI	COMENTARI	COMENTARI	COMENTARI	COMENTARI	COMENTARI	COMENTARI	COMENTARIO	COMENTARI	COMENTARI	COMENTARI	COMENTARI	COMENTARIO	COMENTARIO	COMENTARIO
24	ERRO_ATRIB_MAL_F	ERRO_ATRIB_MAL_FORMA	ERRO_ATRIB	ERRO_ATRIB	ERRO_ATRIB	ERRO_ATRIB	ERRO_ATRIB	ERRO_ATRIB	ERRO_ATRIB	ERRO_ATRIB	ERRO_ATRIB	ERRO_ATRIB	ERRO_ATRIB	ERRO_ATRIB	ERRO_ATRIB	ERRO_ATRIB_MA	ERRO_ATRIB	ERRO_ATRIB	ERRO_ATRIB	ERRO_ATRIB	ERRO_ATRIB_M	ERRO_ATRIB_M	ERRO_ATRIB_MAL_FORMADO
25	SIMB_PONTO	SIMB_PONTO	SIMB_PONTO	SIMB_PONTO	SIMB_PONTO	SIMB_PONTO	SIMB_PONTO	SIMB_PONTO	SIMB_PONTO	SIMB_PONTO	SIMB_PONTO	SIMB_PONTO	SIMB_PONTO	SIMB_PONTO	SIMB_PONTO	SIMB_PONTO	SIMB_PONTO	SIMB_PONTO	SIMB_PONTO	SIMB_PONTO	SIMB_PONTO	SIMB_PONTO	SIMB_PONTO
26	SIMB_MULTIPLIC	SIMB_MULTIPLIC	SIMB_MULTI	SIMB_MULTI	SIMB_MULTI	SIMB_MULTI	SIMB_MULTI	SIMB_MULTI	SIMB_MULTI	SIMB_MULTI	SIMB_MULTI	SIMB_MULTI	SIMB_MULTI	SIMB_MULTI	SIMB_MULTI	SIMB_MULTIPLIC	SIMB_MULTI	SIMB_MULTI	SIMB_MULTI	SIMB_MULTI	SIMB_MULTIPLIC	SIMB_MULTIPLIC	SIMB_MULTIPLIC
27	SIMB_DIV	SIMB_DIV	SIMB_DIV	SIMB_DIV	SIMB_DIV	SIMB_DIV	SIMB_DIV	SIMB_DIV	SIMB_DIV	SIMB_DIV	SIMB_DIV	SIMB_DIV	SIMB_DIV	SIMB_DIV	SIMB_DIV	SIMB_DIV	SIMB_DIV	SIMB_DIV	SIMB_DIV	SIMB_DIV	SIMB_DIV	SIMB_DIV	SIMB_DIV
28	SIMB_NÃO_IDENTIF	SIMB_NÃO_IDENTIFICADO	SIMB_NÃO_I	SIMB_NÃO_I	SIMB_NÃO_I	SIMB_NÃO_I	SIMB_NÃO_I	SIMB_NÃO_I	SIMB_NÃO_I	SIMB_NÃO_I	SIMB_NÃO_I	SIMB_NÃO_I	SIMB_NÃO_I	SIMB_NÃO_I	SIMB_NÃO_I	SIMB_NÃO_IDENT	SIMB_NÃO_I	SIMB_NÃO_I	SIMB_NÃO_I	SIMB_NÃO_I	SIMB_NÃO_IDEN	SIMB_NÃO_IDEN	SIMB_NÃO_IDENTIFICADO
29	ERRO_COMENT_MA	ERRO_COMENT_MAL_FORM	ERRO_COME	ERRO_COME	ERRO_COME	ERRO_COME	ERRO_COME	ERRO_COME	ERRO_COME	ERRO_COME	ERRO_COME	ERRO_COME	ERRO_COME	ERRO_COME	ERRO_COME	ERRO_COMENT_	ERRO_COME	ERRO_COME	ERRO_COME	ERRO_COME	ERRO_COMENT	ERRO_COMENT	ERRO_COMENT_MAL_FORMADO

Figura 3: Tabela de Transição

$$
\begin{aligned}
\delta^*(q_0, \mathrm{AAB}+) &= \delta^*(\delta(q_0, \mathrm{A}), \mathrm{AB}+) \\
&= \delta^*(q_4, \mathrm{AB}+) \\
&= \delta^*(\delta(q_4, \mathrm{A}), \mathrm{B}+) \\
&= \delta^*(q_4, \mathrm{B}+) \\
&= \delta^*(\delta(q_4, \mathrm{B}), +) \\
&= \delta^*(q_4, +) \\
&= \delta^*(\delta(q_4, +), \varepsilon) \\
&= \delta^*(q_5, \varepsilon).
\end{aligned}
$$

O estado q_5 é o responsável por retornar o token `TK_ID`.

3.4 Resultado da Implementação

Com a implementação concluída, foram realizados testes utilizando programas de exemplo.

O analisador léxico gerou corretamente os pares token-classe no arquivo de saída, além de identificar e reportar eventuais erros léxicos, conforme ilustrado nos exemplos a seguir.

Exemplo 1: Comparação entre Arquivo de Entrada e Arquivo de Saída

Arquivo de Entrada:

```
VAR n, fat;
PROCEDURE fatorial;
BEGIN
  fat:1;
  WHILE n>=1 DO
    BEGIN
      fat:=fat!n;
      n:=n-1@;
    END
END;
BEGIN
  n:=4;
  CALL fatorial
END.
```

Arquivo de Saída:

```
VAR, VAR
n, TK_ID
„ SIMB_VIRGULA
fat, TK_ID
;, SIMB_PONT_VIRG
PROCEDURE, PROCEDURE
fatorial, TK_ID
;, SIMB_PONT_VIRG
BEGIN, BEGIN
fat, TK_ID
:,
ERRO_ATRIB_MAL_FORMADO
1, TK_NUM_INT
;, SIMB_PONT_VIRG
WHILE, WHILE
n, TK_ID
>=, SIMB_MAIOR_IGUAL
1, TK_NUM_INT
DO, DO
BEGIN, BEGIN
fat, TK_ID
:=, TK_ATRIBUICAO
fat, TK_ID
!,
SIMB_NAO_IDENTIFICADO
n, TK_ID
;, SIMB_PONT_VIRG
n, TK_ID
:=, TK_ATRIBUICAO
n, TK_ID
-, SIMB_SUB
Eu 1, TK_NUM_INT
@,
SIMB_NAO_IDENTIFICADO
;, SIMB_PONT_VIRG
END, END
END, END
;, SIMB_PONT_VIRG
BEGIN, BEGIN
n, TK_ID
:=, TK_ATRIBUICAO
4, TK_NUM_INT
;, SIMB_PONT_VIRG
CALL, CALL
fatorial, TK_ID
END, END
., SIMB_PONTO
```

Exemplo 2: Comparação entre Arquivo de Entrada e Arquivo de Saída

Arquivo de Entrada:

```
VAR n, fat;
BEGIN
  n:45!;
  fat:=1;
  WHILE n<>1 DO
    BEGIN
      fat:=fat*n;
      n:=n-1;
      n := n*1;
    END {manfiosos}
END.
```

Arquivo de Saída:

```
VAR, VAR
n, TK_ID
„ SIMB_VIRGULA
fat, TK_ID
;, SIMB_PONT_VIRG
BEGIN, BEGIN
n, TK_ID
:,
ERRO_ATRIB_MAL_FORMADO
45, TK_NUM_INT
!,
SIMB_NAO_IDENTIFICADO
;, SIMB_PONT_VIRG
fat, TK_ID
:=, TK_ATRIBUICAO
1, TK_NUM_INT
;, SIMB_PONT_VIRG
WHILE, WHILE
n, TK_ID
<>, SIMB_MENOR_MAIOR
1, TK_NUM_INT
DO, DO
BEGIN, BEGIN
fat, TK_ID
:=, TK_ATRIBUICAO
fat, TK_ID
*, SIMB_MULTIPLIC
n, TK_ID
;, SIMB_PONT_VIRG
n, TK_ID
:=, TK_ATRIBUICAO
n, TK_ID
-, SIMB_SUB
1, TK_NUM_INT
;, SIMB_PONT_VIRG
n, TK_ID
:=, TK_ATRIBUICAO
n, TK_ID
*, SIMB_MULTIPLIC
1, TK_NUM_INT
;, SIMB_PONT_VIRG
END, END
{manfiosos}, COMENTARIO
END, END
., SIMB_PONTO
```

3.5 Execução do Analisador Léxico

Após a implementação do analisador léxico, definimos um procedimento simples para a sua execução utilizando Makefile.

Atenção

O programa foi desenvolvido em linguagem C e deve ser compilado com um compilador compatível, como o `GCC`.

Para compilar e executar o programa, basta seguir os passos descritos abaixo no terminal Linux:

- Acesse o diretório do projeto: `cd Compiladores/Trabalho1/Código_Compiladores`
- Caso deseje utilizar um arquivo de entrada diferente, coloque o arquivo na pasta `/Código_Compiladores` e altere o caminho do arquivo `.txt` diretamente na função `main` antes da compilação.
- Compile o programa utilizando o comando: `make`
- Execute o programa com: `make run`
- **Observação:** caso existam arquivos de compilação antigos, recomenda-se executar o comando: `make clean` antes de compilar novamente.

Observação

O resultado da execução será gerado no arquivo `output.txt`.

4 Discussão e Conclusão

O desenvolvimento do analisador léxico exigiu a superação de diversos desafios que surgiram ao longo do projeto. Desde o início, ficou claro que a construção dos autômatos para o reconhecimento dos tokens seria uma etapa central e fundamental para todo o desenvolvimento do projeto. Sendo assim, foi necessário compreender em detalhes as especificações da linguagem PL/0 e traduzir seus padrões léxicos em formalismos operacionais bem definidos.

Nesse processo, foi de suma importância a utilização da ferramenta JFLAP, pois possibilitou a modelagem gráfica dos autômatos e a validação de seus comportamentos antes da implementação no código, apesar que alguns dos símbolos essa ferramenta não reconhece, como a-zA-Z.

A partir dos autômatos construídos, elaboramos a tabela de transição de estados em uma planilha, o que permitiu organizar de maneira sistemática a lógica de movimentação entre estados a cada caractere lido, como podemos observar no exemplo realizado anteriormente com a fita **AAB**+. Essa etapa foi essencial para garantir que, durante a implementação da tabela, o programador pudesse visualizar com clareza o comportamento de cada etapa do processo.

Durante a implementação, novos desafios se apresentaram, como a implementação de uma tabela hash para o tratamento de palavras reservadas, a necessidade de ter um olhar a frente a cada caractere e a própria função de transição.

De maneira geral, o projeto proporcionou uma experiência prática rica sobre a importância da análise léxica no processo de construção de compiladores e sobre como essa análise se desenvolve. Além disso, também houve a necessidade constante de planejamento e de atenção aos detalhes, especialmente relacionados à gramática da linguagem PL/0, o que evidenciou a relevância de decisões técnicas sólidas para a robustez do sistema. Nesse contexto, torna-se plausível depreender que a qualidade de um analisador léxico está diretamente ligada à firmeza de seus fundamentos. Portanto, como resultado desse trabalho, foi possível alcançar plenamente os objetivos propostos, obtendo-se um analisador funcional, eficiente e capaz de lidar de maneira apropriada com os diferentes elementos da linguagem.

5 Fontes

As principais referências utilizadas para a realização deste trabalho foram os materiais disponibilizados em sala de aula, isto é, os slides fornecidos pelo professor Thiago Alexandre Salgueiro Pardo, que abordam os conceitos de análise léxica e autômatos finitos.

Linguagem PL/0

Gramática da linguagem PL/0

Fonte: Wirth, Niklaus. (1976). *Algorithms + Data Structures = Programs.* Prentice Hall, Inc. (mais detalhes em Wikipedia).

<programa> ::= <bloco> .

<bloco> ::= <declaracao> <comando>

<declaracao> ::= <constante> <variavel> <procedimento>

<constante> ::= CONST ident = numero <mais_const> ; | λ

<mais_const> ::= , ident = numero <mais_const> | λ

<variavel> ::= VAR ident <mais_var> ; | λ

<mais_var> ::= , ident <mais_var> | λ

<procedimento> ::= PROCEDURE ident ; <bloco> ;
<procedimento> | λ

<comando> ::= ident := <expressao>
| CALL ident
| BEGIN <comando> <mais_cmd> END
| IF <condicao> THEN <comando>
| WHILE <condicao> DO <comando>
| λ

<mais_cmd> ::= ; <comando> <mais_cmd> | λ

<expressao> ::= <operador_unario> <termo> <mais_termos>

<operador_unario> ::= - | + | λ

<termo> ::= <fator> <mais_fatores>

<mais_termos> ::= - <termo> <mais_termos> | + <termo>
<mais_termos> | λ

<fator> ::= ident | numero | (<expressao>)

<mais_fatores> ::= * <fator> <mais_fatores> | / <fator>
<mais_fatores> | λ

<condicao> ::= ODD <expressao>
| <expressao> <relacional> <expressao>

<relacional> ::= = | <> | < | <= | > | >=

Notas:

- Comentários são de única linha, entre chaves `{ }`.
- Identificadores são formados por letras e dígitos, começando por uma letra.
- Só há números inteiros, formados por um ou mais dígitos (de 0 a 9).