UNIVERSIDADE FEDERAL DE JUIZ DE FORA
CIÊNCIA DA COMPUTAÇÃO

Daniel Ribeiro Lavra
Rafael Freesz Resende Correa
Thiago Teixeira Guimarães

Analisador Léxico para a linguagem C–

Juiz de Fora
2022

Daniel Ribeiro Lavra
Rafael Freesz Resende Correa
Thiago Teixeira Guimarães

Analisador Léxico para a linguagem C–

Trabalho referente a construção da etapa 1,
um Analisador Léxico de um compilador.

Professor: Marcelo Bernardes Vieira

Juiz de Fora

2022

SUMÁRIO

1 VISÃO GERAL

Este trabalho tem como objetivo o desenvolvimento do Analisador Léxico de um compilador para a linguagem de programação C–, uma versão simplificada da linguagem C++ (Stroustrup 1986).

O Analisador Léxico compõe o *Front End* do compilador, sendo a primeira etapa do processo de compilação, trabalhando diretamente com o código fonte, como mostra a Imagem 1. Sua função é receber os *streams* de caracteres, identificando padrões e elementos da linguagem, transformando-os em *tokens* para serem utilizados futuramente pelo *parser*, bem como o reconhecimento de erros advindos do código de origem (Appel 1997, Aho et al. 2007).

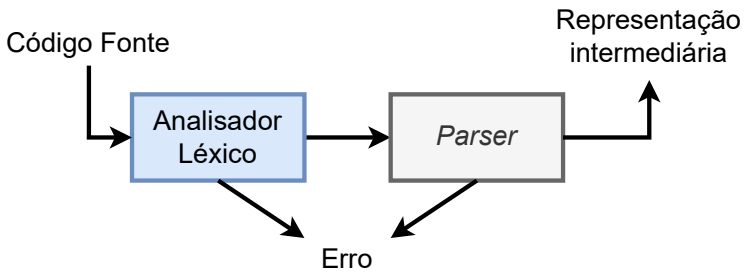

Figura 1 – Representação do *Front End* do compilador.

O algoritmo deste trabalho foi desenvolvido em C, sendo dividido em quatro módulos, como mostra a Imagem 2. A vantagem da divisão da implementação em módulos se dá pela possibilidade de definir os escopos de delegações, encapsulamento de atributos privados e legibilidade do código.

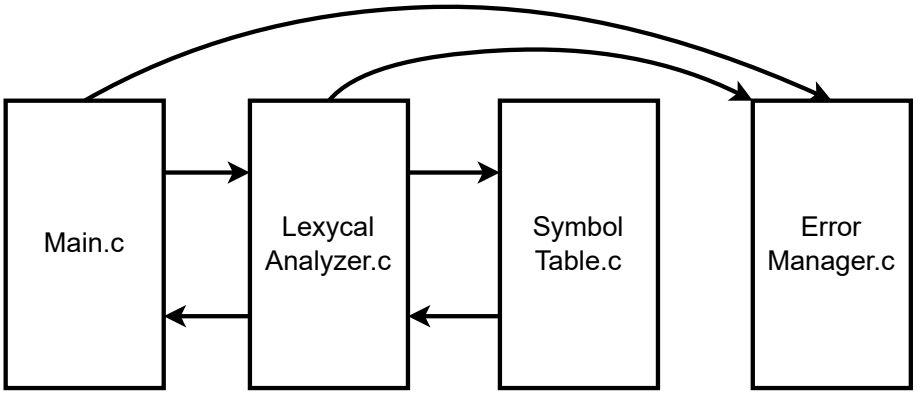

Figura 2 – Modularização do código para desenvolvimento do Analisador Léxico.

O módulo principal, *Main.c*, é responsável pela recepção dos atributos de entrada, como o nome do arquivo fonte, e pela invocação e manipulação do analisador léxico, fazendo uso de suas funções, como a solicitação de um novo token. O módulo *LexycalAnalyzer.c*

implementa o próprio analisador léxico. Suas principais funções são: disponibilizar a estrutura do token, gerenciar o arquivo de entrada, alimentar o *buffer* com novas *strings* quando necessário e retornar novos *tokens* quando solicitado. *SymbolTable.c* implementa a Tabela de Símbolos, cumprindo a função de guiar os tokens de entrada, disponibilizados pelo analisador léxico, para sua representação equivalente. É função deste módulo também manter o *buffer* de lexemas, para aqueles *tokens* que apresentam lexemas dinâmicos.

O módulo principal e o analisador léxico estão sujeitos a diversos erros. Estes podem ser classificados em Erros de Sistema, representando insucesso ao abrir arquivos bem como extensões incompatíveis do mesmo; e Erros Léxicos, quando se encontram conflitos ou inconsistências em relação à linguagem de origem, contidos no arquivo fonte. Para todos os tipos de erros, o módulo *ErrorManager.c* é responsável pelo reporte do erro, bem como sua identificação.

Os demais capítulos estão dispostos da seguinte forma: O Capítulo 2 apresenta os componentes da linguagem. O Capítulo 3 descreve com detalhes o módulo do Analisador Léxico. Da mesma forma, o Capitulo 4 apresenta o módulo da Tabela de Símbolos. O Capítulo 5 apresenta o módulo de gerenciamento de erros. O Capítulo 6 apresenta o resultado de algumas execuções com código de entrada e, por fim, o Capítulo 7 apresenta a conclusão do trabalho apresentado.

2 TOKENS E EXPRESSÕES REGULARES

Neste capítulo são apresentados os *tokens* utilizados pelo analisador léxico, bem como aquelas referentes a palavras reservadas nas tabelas 1 e 2, respectivamente.

Token		**Lexema**	**Expressões Regulares**
Numero	**Alias**		
1	COLON	;	;
2	MOD	%	%
3	PLUS	+	+
4	MULT	*	*
5	EOF	\0	\0
6	LBRACE	{	{
7	RBRACE	}	}
8	LBRACKET	[	[
9	RBRACKET	]	]
10	LPARENTHESES	(	(
11	RPARENTHESES	)	)
13	NEQ	!=	!=
14	NOT	!	!
16	GEQ	>=	>=
17	GREAT	>	>
19	LESS	<	<
20	LEQ	<=	<=
22	ASSIGN	=	=
23	EQ	==	==
27	DIV	/	/
35	LITERAL	-	"(DIGIT|LETTER)$^+$" ’(DIGIT|LETTER)$^+$′
37	OR	||	||
38	PIPE	|	|
40	AND	&&	&&
41	AMBERSAND	&	&
43	POINTER	->	->
44	MINUS	-	-
46	NUMINT	DIGIT(DIGIT)*	
53	NUMFLOAT	-	(DIGIT)*.(DIGIT)$^+$ (DIGIT)$^+E + (DIGIT)^+$ (DIGIT)$^+E - (DIGIT)^+$
54	POINT	.	.
56	ID	-	(LETTER)+(LETTER|DIGIT)+

Tabela 1 – Tabela de Tokens, Lexemas e Expressões Regulares

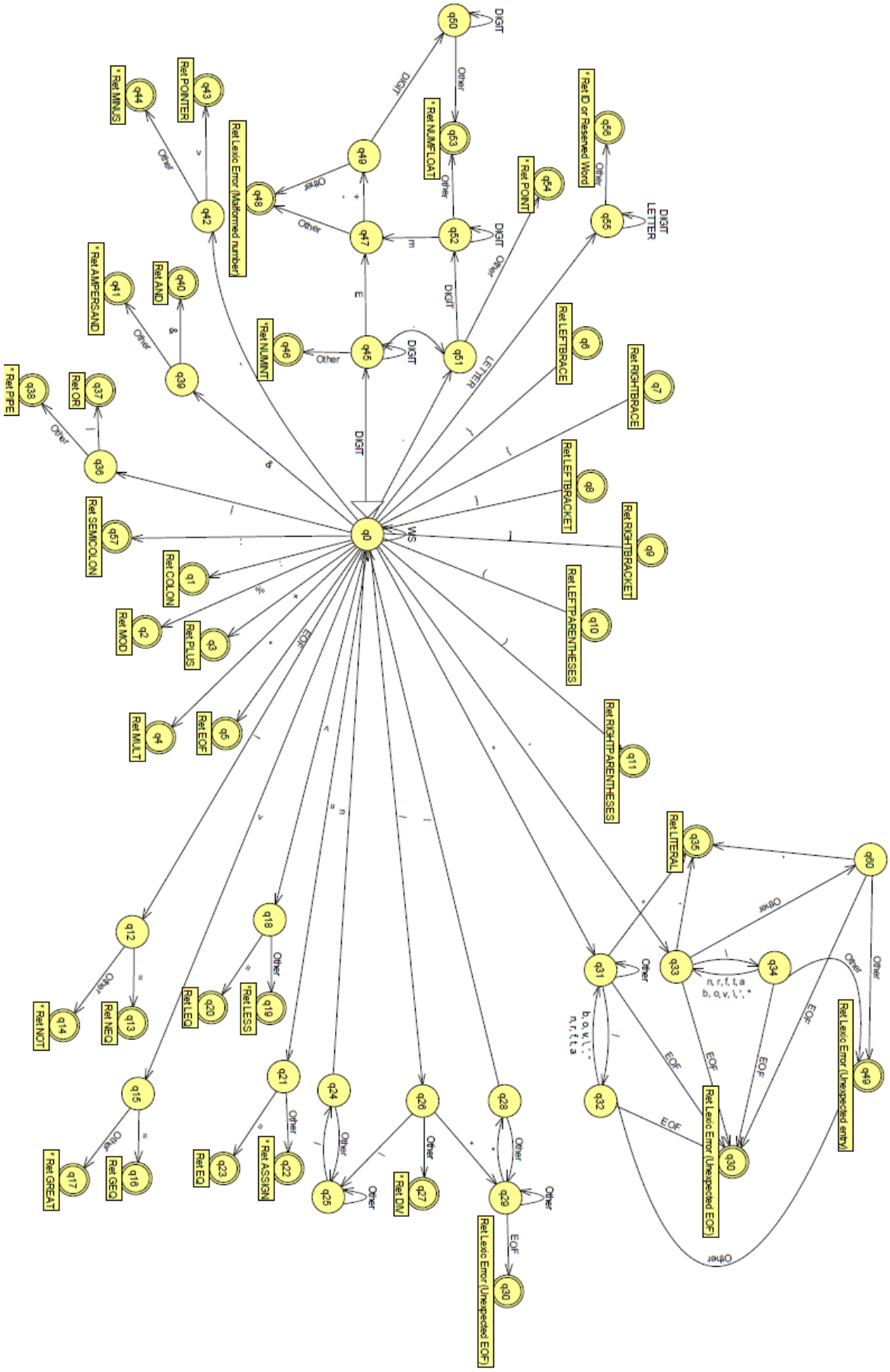

Figura 3 – Autômato Finito Determinístico da linguagem

Token		**Lexema**	**Expressões Regulares**
Numero	**Alias**		
561	TYPEDEF	typedef	typedef
562	STRUCT	struct	struct
563	LONG	long	long
564	INT	int	int
565	FLOAT	float	float
566	BOOL	bool	bool
567	CHAR	char	char
568	DOUBLE	double	double
569	IF	if	if
570	WHILE	while	while
571	SWITCH	switch	switch
572	BREAK	break	break
573	PRINT	print	print
574	READLN	readln	readln
575	RETURN	return	return
576	THROW	throw	
577	TRY		
578	CATCH		
579	CASE		
580	FOR		

Tabela 2 – Tabela de Tokens, Lexemas e Expressões Regulares para palavras reservadas

3 ANALISADOR LÉXICO

O Módulo do Analisador Léxico implementa duas estruturas fundamentais:

- **Token**: Armazena o número do respectivo *token*, contendo o index do lexema no *buffer* de lexemas, quando este for necessário, bem como o seu tamanho;
- **LexicalAnalyser**: Mantém o arquivo do código fonte, o *buffer* de *streams*, um índice para demarcar a posição atual a ser observada no buffer, a quantidade de caracteres recebidas pelo *buffer* em sua última atualização e um contador de linhas.

Três funções realizam testes no caracteres para categorizá-lo. A função *isLetter* verifica se um determinado caractere representa uma letra maiúscula ou minúscula do alfabeto. A função *isNumber* verifica se o mesmo faz parte do conjunto de caracteres de um a nove. Essas verificações são realizadas com base no valor da tabela ASCII. Uma terceira função, *WhiteSpace*, verifica se o caractere representa um espaço em branco. São eles: o próprio espaçamento, os caracteres $'\backslash t', '\backslash n' e '\backslash r'$.

Essas funções de verificação visam facilitar a delegação do autômato durante sua execução, bem com deixá-lo mais conciso.

A função *verifyFileConsistence* verifica a corretude da extensão do arquivo de entrada, retornando erro de sistema caso este não seja compatível.

loadStream realiza atualização do *buffer* de *streams* do *LexicalAnalyser* quando todos os caracteres do *stream* atual já foram lidos.

A função *nextToken* realiza a construção do *token* com base na análise caractere a caractere do *stream* do *buffer* em um autômato finito determinístico. Ao final, é retornado um *token* com o respectivo índice do *lexema* no *buffer* de *lexemas*.

Cada caractere do *buffer* de *streams* é obtido através da função *getBufferCharacter*, possuindo o atributo *advance* para fim de indicar se o caractere atual será consumido ou não.

O Analisador Léxico é inicializado com a função *buildLexicalAnalyzer*, a qual verifica a integridade do arquivo, com *verifyFileConsistence*, abre o referido arquivo e realiza a primeira carca de *stream* com o *loadStream*.

A mesma estrutura é desalocada com a função *deleteLexicalAnalyzer*, que realiza a deleção das estruturas dinamicamente alocadas ao longo da execução do algoritmo.

A função *buildSymbolTable* popula a tabela de símbolos com os elementos da linguagem. São instanciadas três tabelas de símbolo. Uma para palavras reservadas, uma para literais e outra para identificadores. O módulo de tabela de símbolos é explicado adiante no Capítulo 4.

4 TABELA DE SÍMBOLOS

Neste capítulo será descrito o módulo que implementa a tabela de símbolos. A principal estrutura que é gerencia como os dados são dispostos na tabela de símbolos é uma *Hash table with buckets*, ou seja, ela é constituída de um vetor de listas encadeadas de tamanho fixo. Deste jeito, mesmo a tabela possuindo tamanho fixo em seu vetor as listas crescem de forma dinâmica, tendo seu desempenho de busca e inserção pautado pelo número de percorrimentos que deve ser feito na lista encadeada.

Precisamos, então, definir três parâmetros principais: A função *hash* a ser utilizada, o tamanho do vetor que alocará a tabela e a estratégia de inserção caso haja uma colisão.

As chaves das entradas das tabelas serão todas *Strings*.

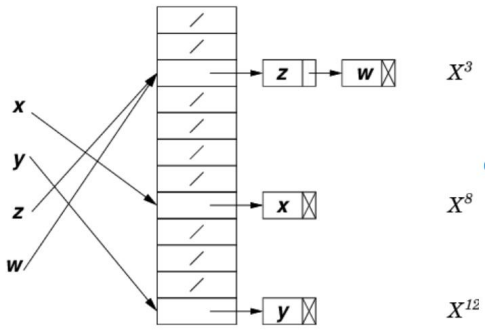

Figura 4 – Ilustração da *hash bucket*

- **Função *hash***: *Polynomial rolling hash function* (String hashing using polynomial rolling hash f
  Esta função mapeia um conjunto de caracteres para um número inteiro usando apenas multiplicações e adições;
- **Tamanho do vetor**: O valor escolhido para o tamanho das tabelas foi 499 já que este é um número primo razoável para alocar todas as palavras chaves da linguagem C– pelo menos;
- **Estratégia de inserção**: Caso haja uma colisão, a chave é inserida no início da lista encadeada *bucket* para evitar o percorrimento desta. Se caso esta chave for a mesma da que já está na tabela, ela não é inserida, já que a tabela de símbolos será usada apenas para verificarmos a existência de uma palavra, literal ou identificador, e.g não faz sentido termos 10 identificadores 'x' que foram lidos do código armazenados na tabela.

A classe que define a tabela de símbolos é a *SymbolTable* enquanto a que especifica a entrada da tabela é a *TabEntry*. No módulo da tabela de símbolos ainda tem-se o gerenciamento do *buffer de lexemas* que é um vetor de caracteres para armazenar os lexemas dos elementos léxicos.

- **Principais métodos para o gerenciamento do buffer de lexema**:
  - *buildLexemBuffer: Constrói o buffer*;
  - *getLexem: Retornar o lexema do buffer de lexemas*;
  - *pushLexem: Coloca um lexema no buffer de lexemas*;
  - *reallocLexemBuffer: Realoca o tamanho do buffer de lexemas caso necessário*;
- **Principais métodos da classe da tabela de símbolos**:
  - *hashFunction: Calcula o valor hash da chave*;
  - *insertKey: Insere a chave na tabela*;
  - *searchKey: Procura a chave na tabela*;
  - *identifierOrReservedWord: Retorna se a palavra é um identificador ou palavra reservada caso seja a tabela de símbolos de palavras reservas*;

5 GERENCIADOR DE ERROS LÉXICOS

O módulo de gerenciamento de erros é responsável pela classificação e impressão de erros encontrado ao longo da execução do algoritmo. Os erros foram classificados em dois tipos:

- **Erros de Sistema**: São classificados como aqueles de arquivo de entrada não encontrado (*File not found*), arquivo não suportado (*File not supported*) e índice fora dos limites)*Index out of bounds*);
- **Erros Léxicos**: São erros contidos no código do arquivo de entrada, referentes a erros padrão de caracteres. São eles: Entrada não esperada (*Unexpected Entry*), fim de arquivo não esperado (*Unspected end of file*) e numeros mal formado (*Malformd number*). Para estes, é recebido e impresso também o número da linha na qual ocorreu o erro.

6 RESULTADOS EXPERIMENTAIS

Nesta seção serão apresentandos os testes feitos e os resultados obtidos que comprovam o funcionamento do analisador léxico.

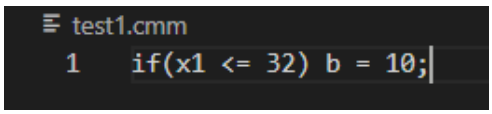

Figura 5 – Código exemplo - Teste 1

```
IF
LEFTPARENTHESES
ID.x1
LEQ
NUMINT.32
RIGHTPARENTHESES
ID.b
ASSIGN
NUMINT.10
SEMICOLON
CMMEOF

SYMBOL TABLE: RESERVED WORDS
----------------------------------------
LEXEM                    NUMERIC TOKEN

 while                     570
 char                      567
 typedef                   561
 case                      579
 print                     573
 float                     565
 bool                      566
 try                       577
 if                        569
 switch                    571
 for                       580
 break                     572
 catch                     578
 return                    575
 readln                    574
 throw                     576
 long                      563
 struct                    562
 double                    568

SYMBOL TABLE: IDENTIFIER
----------------------------------------
LEXEM

 b
 x1

SYMBOL TABLE: LITERALS
----------------------------------------
LEXEM
```

Figura 6 – Resultados Teste 1

Os demais testes foram feitos em cima de códigos maiores, portanto, nas próximas visualizações de teste não tem como listar todos os tokens gerados na saída, o que pode

ser visualizado através do projeto no GitHub.

test2.cmm

```
int main(){
    int arr[50], num, x, y, temp;
    printf("Please Enter the Number of Elements you want in the array: ");
    scanf("%d", &num);
    printf("Please Enter the Value of Elements: ");
    for(x = 0; x < num; x++)
        scanf("%d", &arr[x]);
    for(x = 0; x < num - 1; x++){
        for(y = 0; y < num - x - 1; y++){
            if(arr[y] > arr[y + 1]){
                temp = arr[y];
                arr[y] = arr[y + 1];
                arr[y + 1] = temp;
            }
        }
    }
    printf("Array after implementing bubble sort: ");
    for(x = 0; x < num; x++){
        printf("%d  ", arr[x]);
    }
    return 0;
}
```

Figura 7 – Código BubbleSort - Teste 2

Com esses resultados obtidos, podemos perceber que o autômato, as tabelas de símbolos e o analisador léxico obtiveram um resultado correto conforme a especificação do trabalho.

```
 if                     569
 int                    564
 switch                 571
 for                    580
 break                  572
 catch                  578
 return                 575
 readln                 574
 throw                  576
 long                   563
 struct                 562
 double                 568

SYMBOL TABLE: IDENTIFIER
---------------------------------------
LEXEM

 x
 y
 main
 num
 scanf
 printf
 temp
 arr

SYMBOL TABLE: LITERALS
---------------------------------------
LEXEM

 "Please Enter the Number of Elements you want in the array: "
 "Aíraáy after implementing bubble sort: "
 "Please Enter the Value of Elements: "
 "%d  "
 "%d"
```

Figura 8 – Resultados Teste 2

test3.cmm

```
1   int main()
2   {
3       printf("'Nome','Idade','Sexo',CPF'");
4       printf("'Joao', 18, 'M', '111111111-11");
5   }
```

Figura 9 – Código print literal - Teste 3

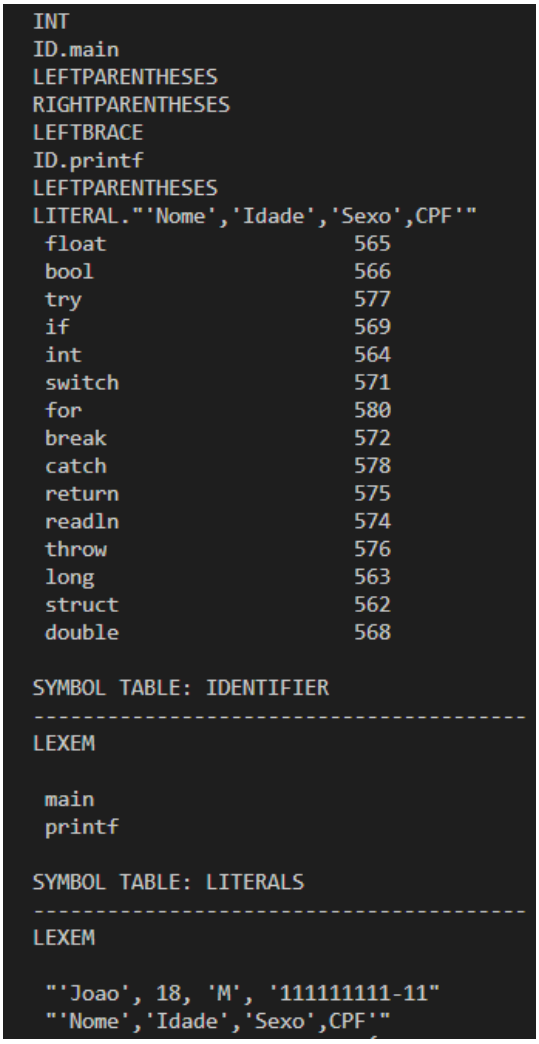

Figura 10 – Resultados Teste 3

test4.cmm

```
1   //protótipo da função fatorial
2   double fatorial(int n);
3   int main(void)
4   {
5     int numero;
6     double f;
7
8     printf("Digite o numero que deseja calcular o fatorial: ");
9     scanf("%d",&numero);
10
11    //chamada da função fatorial
12    f = fatorial(numero);
13
14    printf("Fatorial de %d = %.0lf",numero,f);
15
16    getch();
17    return 0;
18  }
19  //Função recursiva que calcula o fatorial
20  //de um numero inteiro n
21  double fatorial(int n)
22  {
23    double vfat;
24
25    if ( n <= 1 )
26      //Caso base: fatorial de n <= 1 retorna 1
27      return (1);
28    else
29    {
30      //Chamada recursiva
31      vfat = n * fatorial(n - 1);
32      return (vfat);
33    }
34  }
```

Figura 11 – Código Fatorial - Teste 4

```
NUMINT.1
RIGHTPARENTHESES
SEMICOLON
ID.else
LEFTBRACE
ID.vfat
ASSIGN
ID.n
MULT
ID.fatorial
LEFTPARENTHESES
ID.n
MINUS
NUMINT.1
RIGHTPARENTHESES
SEMICOLON
RETURN
LEFTPARENTHESES
ID.vfat
RIGHTPARENTHESES
SEMICOLON
RIGHTBRACE
RIGHTBRACE
CMMEOF

SYMBOL TABLE: RESERVED WORDS
----------------------------------------
LEXEM                    NUMERIC TOKEN

 while                     570
 char                      567
 typedef                   561
 case                      579
 print                     573
 float                     565
 bool                      566
 try                       577
 if                        569
SYMBOL TABLE: LITERALS
----------------------------------------
LEXEM

 "Digite o numero que deseja calcular o fatorial: "
 "Fatorial de %d = %.0lf"
 "%d"
```

Figura 12 – Resultados Teste 4

7 CONCLUSÃO

De acordo com os resultados apresentados no capítulo anterior, pode se concluir que o analisador léxico implementado atendeu às expectativas, tanto na sua funcionalidade, através *tokens* retornados, quanto na sua modularização, que tornou o código facilmente legível e otimizado.

REFERÊNCIAS

Aho et al. 2007AHO, A. V. et al. *Compilers: principles, techniques, & tools*. [S.l.]: Pearson Education India, 2007.

Appel 1997APPEL, A. W. Modern compiler implementation in c: Basic techniques. *Computers and Mathematics with Applications*, v. 7, n. 33, p. 139, 1997.

String hashing using polynomial rolling hash function 2022STRING hashing using polynomial rolling hash function. 2022. Disponível em: <https://www.geeksforgeeks.org/string-hashing-using-polynomial-rolling-hash-function/>.

Stroustrup 1986STROUSTRUP, B. *The C++ Programming Language, First Edition*. [S.l.]: Addison-Wesley, 1986. ISBN 0-201-12078-X.