

Matheus Floriano Saito Da Silva

Analizador Léxico para a Linguagem C-: Projeto de Implementação utilizando Máquina de Moore

Relatório técnico de atividade prática solicitado pelo professor Rogerio Aparecido Gonçalves na disciplina de Teoria da computação do Bacharelado em Ciência da Computação da Universidade Tecnológica Federal do Paraná.

Universidade Tecnológica Federal do Paraná – UTFPR

Departamento Acadêmico de Computação – DACOM

Bacharelado em Ciência da Computação – BCC

Campo Mourão

Fevereiro / 2025

Resumo

Este relatório apresenta os desenvolvimentos e resultados da atividade prática realizada, cujo objetivo foi projetar e implementar um analisador léxico para a linguagem C- por meio de uma máquina de Moore e linguagem de programação (Python).

Palavras-chave: Moore. Python. analisador léxico. implementar.

Sumário

1	Introdução e objetivos	4
2	Fundamentação	4
3	Materiais	4
4	Projeto do Autômato	4
5	Descrição do funcionamento do Código	5
5.1	Máquina de Moore	5
5.2	Pré-processamento	6
5.3	Processamento	6
6	Testes e Resultados	6
6.1	Correção dos testes	6
6.2	Testes Finais	9
7	Conclusão	11
8	Referências	11

1 Introdução e objetivos

Neste relatório será descrito o trabalho 1 da disciplina de Teoria da computação, o qual visa construir um analisador léxico para a linguagem C- criado por meio de uma máquina de Moore e a linguagem Python. Este analisador léxico recebe por entrada um arquivo contendo um código C- e processa este código devolvendo tokens das operações.

2 Fundamentação

Para a realização deste trabalho foi utilizado como base as instruções, o autômato inicial e o código base inicial fornecido pelo professor, além do capítulo 2 do livro do ([LOUDEN, 2004](#)).

3 Materiais

Foi utilizado um Notebook Dell G15 com as especificações:

- Intel Core i5-10500H (@2.50GHz e 12 *thread*)
- 16GB RAM
- SSD NVMe ADATA 512GB
- Pop!_OS 22.04 LTS

E ferramentas :

- JFLAP Versão 7.1
- Python

4 Projeto do Autômato

Foi construído e utilizado como base o seguinte autômato:

5 Descrição do funcionamento do Código

A máquina de Moore do autômato apresentado (Figura 1) foi adaptado ao código por meio da automata-python. Esta é responsável por toda a análise léxica. Seu comportamento base é:

- Todos os símbolos simples lidos retornam a respectiva saída
- Quando um caractere letra da tabela ascii não representa a inicial de uma palavra reservada ele é imediatamente considerado ID
- Quando o caractere letra faz parte de uma palavra reservada ele só se torna um ID se não completar a palavra reservada
- Dígitos são reconhecidos e quando detectam um símbolo retornam a q0, se aparecer um caractere no meio dos dígitos, o dígito termina retornando NUMBER e começa um ID
- Comentários são reconhecidos após ter um caractere "/" seguido de "*" e param de ser reconhecidos quando é detectado "*" seguido de "/"
- Quando uma palavra ou símbolo termina de processar e gera uma saída, a máquina volta para q0

5.2 Pré-processamento

```
def preprocess_input(input_string)
```

Esta função é responsável por tratar o código diretamente fornecido pelo arquivo, a ideia aqui é simplificar os dados passados para o autômato, neste caso o principal seria passar cada seção do código original em formato de: operação <quebra de linha>. Por exemplo:

"return(0);" seria transformado em :

```
return  
(  
0  
)  
;
```

Vale ressaltar que entre as palavras podem ser gerados vários espaços ou quebras de linha e o autômato está preparado para ignorá-los, portanto, esta função também foi utilizada para tratar um problema com operadores compostos, onde ao separar != por exemplo, causaria um problema de identificação.

5.3 Processamento

```
def process_input(input_string)
```

Esta função é responsável por lidar com os tokens sem si, obtendo as saídas da máquina de Moore e as colocando em uma lista de strings, onde cada string representa um token.

6 Testes e Resultados

6.1 Correção dos testes

Vale a pena ressaltar que alguns testes estavam errados, e um apenas foi ajustado para funcionar de acordo com a minha implementação como demonstrado a seguir:

5	int main(void){	15	SEMICOLON
6	int x; int y; int z;	16	RBRACES
7	x = input();	17	INT
8	y = input();	18	ID
9	z = soma(x,y);	19	LPAREN
10	output(z);	20	VOID
11	return 0;	21	RPAREN
12	}	22	LBRACES
13		23	INT
14		24	ID
		25	SEMICOLON
		26	INT
		27	ID
		28	SEMICOLON
		29	INT
		30	ID
		31	SEMICOLON
		32	INT <i>github</i>
		33	ATtribution
		34	ID
		35	LPAREN
		36	RPAREN

Figura 2 – Teste 004

Na figura 2 podemos ver que a linha 32 descreve como INT quando na verdade é um ID.

5	}	51	LPAREN
6		52	RPAREN
7	void main(void){	53	SEMICOLON
8	int x; int y;	54	ID
9	x = input();	55	ATtribution
10	y = input();	56	ID
11	output(gcd(x,y));	57	LPAREN
12	}	58	RPAREN <i>git</i>
13		59	SEMICOLON
14		60	ID
		61	LPAREN
		62	ID
		63	LPAREN
		64	ID
		65	COMMA
		66	ID
		67	RPAREN
		68	RPAREN
		69	SEMICOLON
		70	RBRACES
		71	

Figura 3 – Teste 005

Na figura 3 vemos que estava faltando um RPAREN.

8	void output(int x){	47	SEMICOLON
9	}	48	IF
10		49	LPAREN
11		50	ID
12	int func(int x, int y){	51	DIFFERENT
13	int res;	52	NUMBER
14	res = x + y - 2; <i>github-classroom[bot] [8 weeks ago] • Initial c</i>	53	RPAREN
15	if(res != 0){	54	LBRACES
16	res = -1;	55	ID
17	}	56	ATTRIBUTION
18	return(res);	57	MINUS <i>Not</i>
19	}	58	NUMBER
20		59	SEMICOLON
21	int main(void){	60	RBRACES
22	a = input();	61	RETURN
23	b = input();	62	LPAREN
24		63	ID
25	b[0] = a;	64	RPAREN
26	b[1] = func(a,b);	65	SEMICOLON
27		66	RBRACES
28	output(a);	67	INT
29	return(0);	68	ID

Figura 4 – Teste 006

Na figura 4 é possível ver que ocorre uma diferença entre implementações, no meu caso está sendo considerado MINUS, NUMBER, enquanto o teste original considera um número negativo como apenas NUMBER.

2	int b[10];	2	ID
3	int c[3][5];	3	SEMICOLON
4		4	INT
5	b = 10	5	ID
6		6	LBRACKETS
7	int func(int x, int y){	7	NUMBER
8	int res;	8	RBRACKETS
9	res = x + y;	9	SEMICOLON
10	return(res);	10	INT
11	}	11	ID
12		12	LBRACKETS
13	int main(){	13	NUMBER
14	a = input();	14	RBRACKETS
15	b = input();	15	LBRACKETS
16		16	NUMBER
17	b[0] = a;	17	RBRACKETS
18	b[1] = b;	18	SEMICOLON
19	c[0][1] = func(a,b);	19	ID
20		20	ATTRIBUTION
21	output(c[3]);	21	NUMBER
22	return(0);	22	INT
23	}	23	ID

Figura 5 – Teste 007

Na figura 5 após o NUMBER tínhamos um SEMICOLON adicional após b = 10.

6.2 Testes Finais

Após todas as mudanças nos testes como descrito na seção 6.1 o resultado do teste final:

```

(.venv) saito@pop-os:~/Documents/automaton/analex-MattSaito$ pytest -v
===== 11 passed in 0.32s =====
test session starts
platform linux -- Python 3.10.12, pytest-8.3.4, pluggy-1.5.0 -- /home/saito/Documents/automaton/.venv/bin/python3
cachedir: .pytest_cache
rootdir: /home/saito/Documents/automaton/analex-MattSaito
configfile: pytest.ini
collected 11 items

analex_test.py::test_execute[--k] PASSED [ 9%]
analex_test.py::test_execute[test.e--k] PASSED [ 18%]
analex_test.py::test_execute[notexists.cm--k] PASSED [ 27%]
analex_test.py::test_execute[prog-003.cm--k] PASSED [ 36%]
analex_test.py::test_execute[prog-004.cm--k] PASSED [ 45%]
analex_test.py::test_execute[prog-005.cm--k] PASSED [ 54%]
analex_test.py::test_execute[prog-007.cm--k] PASSED [ 63%]
analex_test.py::test_execute[prog-000.cm--k] PASSED [ 72%]
analex_test.py::test_execute[prog-001.cm--k] PASSED [ 81%]
analex_test.py::test_execute[prog-006.cm--k] PASSED [ 90%]
analex_test.py::test_execute[prog-002.cm--k] PASSED [100%]

===== 11 passed in 0.33s =====

```

Figura 6 – Testes

Também fiz um arquivo de teste que demonstra todas as funções do lexer, e a comparação entre código e saída de tokens pode ser vista na Figura 7:

1	INT	45	LBRACES	88	ID	131	VOID	1	int main()
2	ID	46	RETURN	89	EQUALS	132	ID	2	int x1, x2,x3;
3	LPAREN	47	SEMICOLON	90	ID	133	SEMICOLON	3	int y1, y2;
4	RPAREN	48	RBRACES	91	RPAREN	134	ID	4	void funcaoTeste();
5	LBRACES	49	ELSE	92	LBRACES	135	ATtribution	5	
6	INT	50	LBRACES	93	ID	136	NUMBER	6	x1 = 10 + 20 - 5 * 2 / 1;
7	ID	51	ID	94	ATtribution	137	SEMICOLON	7	y1 = x2;
8	COMMA	52	ATtribution	95	ID	138	RETURN	8	if (x1 < x2) {
9	ID	53	ID	96	RBRACES	139	ID	9	return;
10	COMMA	54	PLUS	97	SEMICOLON	140	ATtribution	10	} else {
11	ID	55	NUMBER	98	IF	141	NUMBER	11	x1 = x1 + 1;
12	SEMICOLON	56	SEMICOLON	99	LPAREN	142	SEMICOLON	12	}
13	INT	57	RBRACES	100	ID	143	FLOAT	13	
14	ID	58	WHILE	101	LESS_EQUAL	144	ID	14	while (x1 != 100) {
15	COMMA	59	LPAREN	102	ID	145	ATtribution	15	x1 = x1 + 1;
16	ID	60	ID	103	RPAREN	146	NUMBER	16	}
17	SEMICOLON	61	DIFFERENT	104	LBRACES	147	SEMICOLON	17	
18	VOID	62	NUMBER	105	ID	148	VOID	18	x2 = 100;
19	ID	63	RPAREN	106	ATtribution	149	ID	19	y2 = 42;
20	LPAREN	64	LBRACES	107	ID	150	ATtribution	20	x1 = x2 > 50;
21	RPAREN	65	ID	108	RBRACES	151	NUMBER	21	if(x3 == x1) {x3 = x1};
22	SEMICOLON	66	ATtribution	109	SEMICOLON	152	SEMICOLON	22	if(x2 <= x3) {x2 = x3};
23	ID	67	ID	110	IF	153	FLOAT	23	if (x1 >= x2) {x1 = x2};
24	ATtribution	68	PLUS	111	LPAREN	154	ID	24	
25	NUMBER	69	NUMBER	112	ID	155	COMMA	25	int ifelse, returnValue, floatNumber, voidPointer;
26	PLUS	70	SEMICOLON	113	GREATER_EQUAL	156	ID	26	ifelse = 42;
27	NUMBER	71	RBRACES	114	ID	157	COMMA	27	returnValue = 99;
28	MINUS	72	ID	115	RPAREN	158	ID	28	floatNumber = 314;
29	NUMBER	73	ATtribution	116	LBRACES	159	COMMA	29	voidPointer = 0;
30	TIMES	74	NUMBER	117	ID	160	ID	30	
31	NUMBER	75	SEMICOLON	118	ATtribution	161	SEMICOLON	31	/* Este e um
32	DIVIDE	76	ID	119	ID	162	RETURN	32	comentario de multiplas linhas */
33	NUMBER	77	ATtribution	120	RBRACES	163	NUMBER	33	Not committed yet
34	SEMICOLON	78	NUMBER	121	SEMICOLON	164	SEMICOLON	34	float a,b,c,e;
35	ID	79	SEMICOLON	122	INT	165	RBRACES	35	
36	ATtribution	80	ID	123	ID	166		36	return 0;
37	ID	81	ATtribution	124	COMMA			37	
38	SEMICOLON	82	ID	125	RETURN			38	
39	IF	83	GREATER	126	ID				
40	LPAREN	84	NUMBER	127	COMMA				
41	ID	85	SEMICOLON	128	FLOAT				
42	LESS	86	IF	129	ID				
43	ID	87	LPAREN	130	COMMA				
44	RPAREN	88	ID	131	VOID				

Figura 7 – MyTest

Saída do Teste 007 (Resultados começam na esquerda e terminam na direita):

```

1  int a;
2  int b[10];
3  int c[3][5];
4
5  b = 10
6
7  int func(int x, int y){
8      int res;
9      res = x + y;
10     return(res);
11 }
12
13 int main(){
14     a = input();
15     b = input();
16
17     b[0] = a;
18     b[1] = b;
19     c[0][1] = func(a,b);
20
21     output(c[3]);
22     return(0);
23 }
24

```

Tokenization results (tokens are listed in two columns):

INT	INT
ID	ID
SEMICOLON	LPAREN
INT	RPAREN
ID	LBRACES
LBRACKETS	ID
NUMBER	ATTRIBUTION
RBRACKETS	ID
SEMICOLON	LPAREN
INT	RPAREN
ID	SEMICOLON
LBRACKETS	ID
NUMBER	ATTRIBUTION
RBRACKETS	ID
LBRACKETS	LPAREN
NUMBER	RPAREN
RBRACKETS	SEMICOLON
SEMICOLON	ID
ID	LBRACKETS
ATTRIBUTION	NUMBER
NUMBER	RBRACKETS
INT	ATTRIBUTION
ID	ID
LPAREN	SEMICOLON
INT	ID
ID	LBRACKETS
SEMICOLON	NUMBER
ID	RBRACKETS
ATTRIBUTION	LBRACKETS
ID	NUMBER
PLUS	RBRACKETS
ID	ATTRIBUTION
SEMICOLON	ID
RETURN	LPAREN
LPAREN	ID
ID	COMMA
RPAREN	ID
SEMICOLON	RPAREN
RBRACES	SEMICOLON

Figura 8 – Test-007

Saída do Teste 006 (Resultados começam na esquerda e terminam na direita):

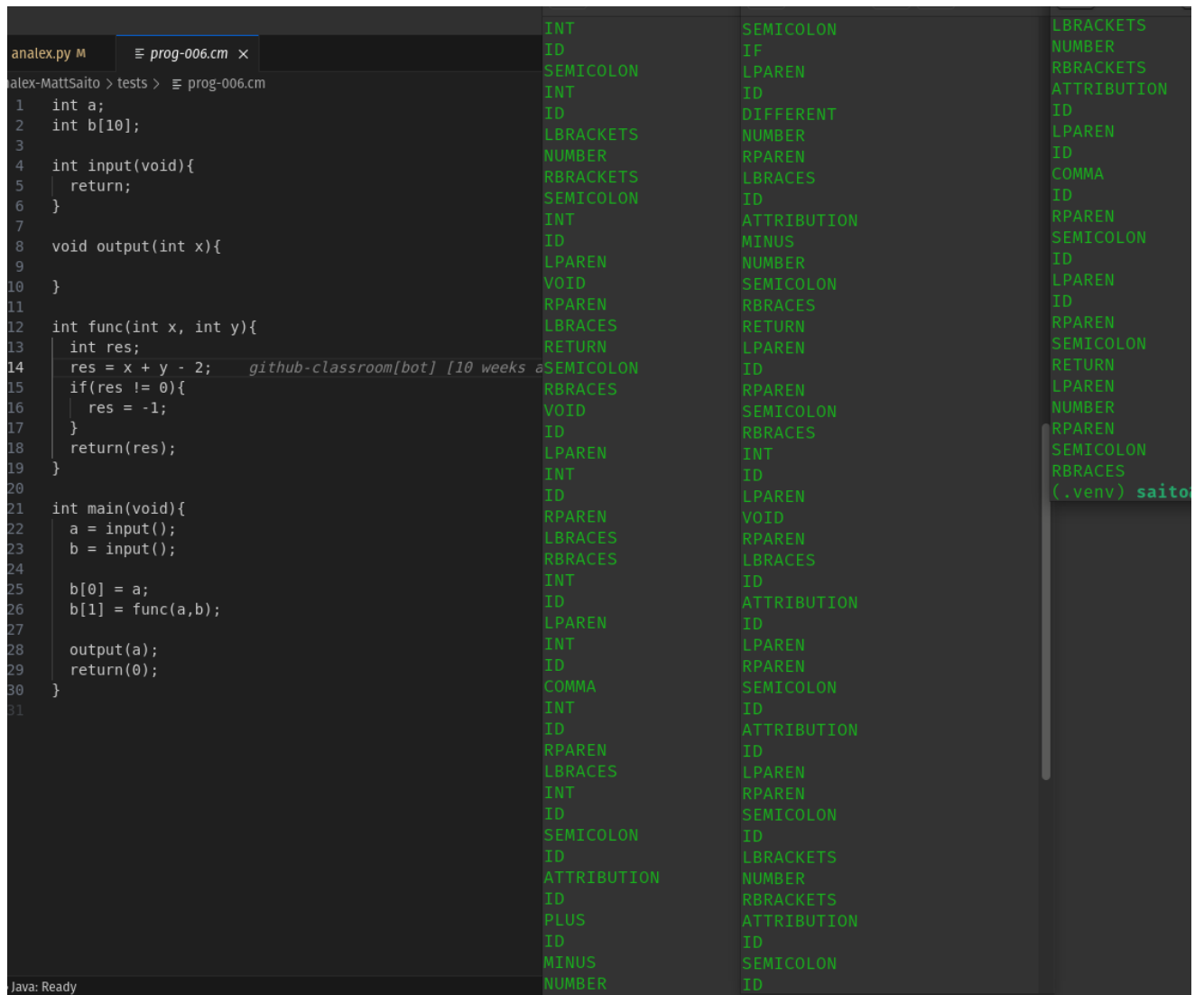


Figura 9 – Test-006

7 Conclusão

O projeto em si foi realizado com sucesso e os resultados finais foram favoráveis, durante a implementação do projeto foi possível obter mais conhecimento sobre um analisador léxico e sobre como lidar com autômatos em Python.

8 Referências

LOUDEN, K. C. *Compiladores: princípios e práticas*. 2004. [Accessed 27-01-2025]. Citado na página 4.