Projeto e Implementação de uma Ferramenta de Análise Léxica para a linguagem C Minus

Natanael Aparecido Tagliaferro Galafassi[1]
[1]Departamento Acadêmico de Computação (DACOM)
Universidade Tecnológica Federal do Paraná (UTFPR)

Abstract

This article demonstrates the steps of the construction of a *Lexical Analyzer* using a *Mealy Machine*, presenting its formal definition, to analyze the *C-Minus* programming language, receiving a *C Minus* code as input and returning it as tokens in the output, making the process of syntactic analysis easier in the future.

Resumo

Este artigo demonstra os passos para a construção de um *Analisador Léxico* baseado em uma Máquina de Mealy, detalhando sua definição formal. O analisador foi desenvolvido para ler e processar código na linguagem de programação *C-Minus*, convertendo-o em *tokens* (marcas), facilitando posteriormente a etapa de análise sintática.

Corresponding author(s): Natanael Aparecido Tagliaferro Galafassi, Universidade Tecnológica Federal do Paraná
natanaelgalafassi@alunos.utfpr.edu.br

1 Introdução

A análise léxica é uma etapa fundamental na construção de compiladores, pois é nesta etapa que uma entrada de código, isto é, uma sequência de caracteres, é compilada em *tokens* (marcas), que são utilizados posteriormente nas próximas etapas, como na análise sintática e semântica. Neste contexto, este trabalho propõe-se a demonstrar o funcionamento e os fundamentos teóricos da implementação de um analisador léxico para a linguagem de programação *C-Minus*, assim como demonstrar a utilização do autômato escolhido e sua definição formal. Os objetivos deste trabalho são:

1. Apresentar a linguagem de programação *C-Minus*
2. Explicar os fundamentos teóricos por trás da *Análise Léxica*
3. Apresentar o diagrama da Máquina de Mealy e descrever sua fundamentação e formalização.
4. Demonstrar exemplos de entrada e saída gerados pelo Analisador Léxico.

Este artigo abordará estes tópicos nesta respectiva ordem, além de demonstrar a razão da escolha da linguagem *C Minus* como base deste projeto.

2 A Linguagem *C-Minus*

Trata-se de um subconjunto reduzido da linguagem C, sendo então denominado *C-Minus*. Ele possui inteiros, *arrays* de inteiros, e funções (também inclusas funções *void*). Contém declarações de variáveis locais e globais e funções recursivas simples. Além disto, o *C-Minus* possui *if-statements* juntamente com *else* e *while-statements*, sem a presença do *for*, com o *C-minus* deixando de apresentar todo o resto que se espera na linguagem C. Assim como no C, uma função **main** deve ser declarada e a execução do programa começa por ela. LOUDEN (2004).

A razão de sua escolha para o projeto é pelos seguintes motivos: (I) como já abordado, ela é uma versão reduzida e simplificada do C, facilitando o desenvolvimento prático do trabalho; (II) escolheu-se o *C-Minus* justamente por sua similaridade com o C, que é geralmente utilizado e ensinado nos períodos introdutórios de cursos de computação. É possível observar as regras sintáticas de sua sintaxe no Código 1.

Código 1: Regras sintáticas do C-Minus

```
1  program ::= declaration-list
2  declaration-list ::= declaration-list declaration | declaration
3  declaration ::= var-declaration | fun-declaration
4  var-declaration ::= type-specifier ID ; | type-specifier ID [ NUM ] ;
5  type-specifier ::= int | float | void
6  fun-declaration ::= type-specifier ID ( params ) compound-stmt
7  params ::= param-list | void
8  param-list ::= param-list , param | param
9  param ::= type-specifier ID | type-specifier ID [ ]
10 compound-stmt ::= { local-declarations statement-list }
11 local-declarations ::= local-declarations var-declaration | empty
12 statement-list ::= statement-list statement | empty
13 statement ::= expression-stmt | compound-stmt | selection-stmt | iteration-stmt
       |
14 return-stmt
15 expression-stmt ::= expression ; | ;
16 selection-stmt ::= if ( expression ) statement | if ( expression ) statement
       else
17 statement
18 iteration-stmt ::= while ( expression ) statement
19 return-stmt ::= return ; | return expression ;
20 expression ::= var = expression | simple-expression
21 var ::= ID | ID [ expression ]
22 simple-expression ::= additive-expression relop additive-expression |
23 additive-expression
24 relop ::= <= | < | > | >= | == | !=
25 additive-expression ::= additive-expression addop term | term
26 addop ::= + | -
27 term ::= term mulop factor | factor
28 mulop ::= * | /
29 factor ::= ( expression ) | var | call | NUMBER
30 call ::= ID ( args )
31 args ::= arg-list | empty
32 arg-list ::= arg-list , expression | expression
```

Para um exemplo de programa em *C-Minus* temos o Código 2 LOUDEN (2004), que calcula o fatorial de um número por meio de uma função recursiva. A entrada/saída neste programa são fornecidas pelas funções **read** e **write** respectivamente, equivalentes às funções **scanf** e **printf** na linguagem C.

Código 2: Função fatorial recursiva em C-Minus

```
1  int fact( int x)
2  /* recursive factorial function */
3  {
4      if (x > 1)
5          return x * fact(x-1);
6      else
7          return 1;
8  }
9
10 void main( void )
11 {
12     int x;
13     x = read;
14     if (x > 0) write( fact(x) );
15
16 }
```

3 Análise Léxica

A fase de análise léxica do compilador tem a tarefa de ler o programa fonte como um arquivo de caracteres e dividi-lo em *tokens* (marcas). *Tokens* são como palavras em uma linguagem natural, cada *token* é uma sequência de caracteres que representa uma unidade de informação no programa fonte LOUDEN (2004). Para este projeto, as informações foram divididas da seguinte forma:

- **Palavras Reservadas**:
  Conjunto de caracteres fixos que são utilizados para representar tipos ou funcionalidades. No *C-Minus* temos as seguintes palavras na Tabela 1.

Palavras Reservadas	*Tokens*
if	IF
else	ELSE
int	INT
float	FLOAT
return	RETURN
void	VOID
while	WHILE

Tabela 1: *Tokens* das palavras reservadas

- **Operadores**:
  Os operadores são geralmente caracteres utilizados para operações lógicas ou aritméticas; estes são apresentados na Tabela 2.

Operadores	*Tokens*
+	PLUS
-	MINUS
*	TIMES
/	DIVIDE
<	LESS
<=	LESS_EQUAL
>	GREATER
>=	GREATER_EQUAL
==	EQUALS
!=	DIFFERENT
=	ATTRIBUTION

Tabela 2: *Tokens* dos operadores

- **Separadores**:
  Os separadores representam caracteres que delimitam o escopo do código ou possuem alguma representação adicional. Temos então sua demonstração na Tabela 3

Separadores	*Tokens*
(	LPAREN
)	RPAREN
[	LBRACKETS
]	RBRACKETS
{	LBRACES
}	RBRACES
;	SEMICOLON
,	COMMA

Tabela 3: *Tokens* dos separadores

- **Identificadores** (ID) Um caractere ou conjunto de caracteres que geralmente são utilizados como identificadores de variáveis e funções, isto é, o que dá nome a elas. As condições para a formação de *IDs* são:
  - O primeiro caractere deve ser uma letra de A-Z, maiúsculo ou minúsculo, podendo também ser o caractere '_', não podendo começar com um número.
  - Após o primeiro caractere, é aceito qualquer caractere válido, incluindo números, por exemplo "_variavelteste123".
  - Caracteres inválidos incluem os separadores e operadores, caracteres de quebra de linha, e símbolos especiais como "?", "~", "!", "$", "#", "%", dentre outros.

- **Números** (NUMBER) Representam dígitos numéricos de 0-9, podendo conter diversos algarismos. A condição de identificação de *NUMBERS* é bem simples, bastando apenas que toda a leitura do conjunto de caracteres se trate de dígitos numéricos, por exemplo, o número 12345.

Por meio da compreensão da definição dos *tokens* em um analisador léxico, pode-se observar os tokens esperados para o Código 2 no Código 3.

Código 3: Tokens para Função fatorial recursiva em C-Minus

```
1  INT
2  ID
3  LPAREN
4  INT
5  ID
6  RPAREN
7  LBRACES
8  IF
9  LPAREN
10 ID
11 GREATER
12 NUMBER
13 RPAREN
14 RETURN
15 ID
16 TIMES
17 ID
18 LPAREN
19 ID
20 MINUS
21 NUMBER
22 RPAREN
23 SEMICOLON
24 ELSE
25 RETURN
26 NUMBER
27 SEMICOLON
28 RBRACES
29 VOID
30 ID
31 LPAREN
32 VOID
33 RPAREN
34 LBRACES
35 INT
36 ID
37 SEMICOLON
38 ID
39 ATTRIBUTION
40 ID
41 SEMICOLON
42 IF
43 LPAREN
44 ID
45 GREATER
46 NUM
47 RPAREN
48 ID
49 LPAREN
50 ID
51 LPAREN
52 ID
53 RPAREN
54 RPAREN
55 SEMICOLON
```

Como a tarefa do analisador léxico é fazer a correspondência de padrões, é necessário estudar métodos que envolvam a especificação e reconhecimento de padrões. Esses métodos a princípio são por meio de **expressões regulares** e **autômato finito** LOUDEN (2004). Para este projeto foi escolhido um autômato finito para implementação deste analisador léxico, especificamente a *Máquina de Mealy*.

4 A Máquina de *Mealy*

A *Máquina de Mealy* é um autômato com saída, onde este pode gerar uma palavra de saída para cada transição. MENEZES (2008)

4.1 Definição

Uma *Máquina de Mealy* M é um Autômato Finito Determinístico onde suas saídas estão associadas às suas transições. Esta é representada por uma 6-upla:

$$\mathrm{M} = (\Sigma, Q, \delta, q_0, F, \Delta)$$

onde:

- Σ: Representa o alfabeto de símbolos de entrada;
- **Q**: Conjunto de estados possíveis do autômato, o qual é finito;
- δ: Representa a função de transição:

$$\delta : Q\mathrm{x}\Sigma \rightarrow Q\mathrm{x}\Delta^*$$

  a qual é uma função parcial;
- q_0: Representa o estado inicial do autômato. q_0 deve estar contido em **Q**;
- **F**: Conjunto dos estados finais, tal que **F** deve estar contido em **Q**;
- Δ: Alfabeto de símbolos da saída.

4.2 Exemplo de *Máquina de Mealy*

A fim de compreender melhor o funcionamento da *Máquina de Mealy*, pode-se observar um exemplo visual na Figura 1, que se trata de uma *Máquina de Mealy* simples, onde temos sua definição formal:

Σ = {'i', 'n', 't', 'f'},
Q = {'q_0', 'q_i', 'q_{in}', q_{int}', 'q_{if}'},
δ = {
(q_0, i) $\rightarrow$ (q_i, ' '),
(q_i, n) $\rightarrow$ ($q_i n$, ' '),
($q_i n$, t) $\rightarrow$ ($q_i nt$, ' '),
($q_i nt$, '\n') $\rightarrow$ (q_0, 'INT'),
(q_i, f) $\rightarrow$ ($q_i f$, ' '),
($q_i f$, '\n') $\rightarrow$ (q_0, 'IF')
}

F: O *JFLAP* não suporta estado de aceitação para a *Máquina de Mealy*, contudo, podemos imaginar o próprio estado q_0 como o estado de aceitação.
Δ = ['INT', 'IF']

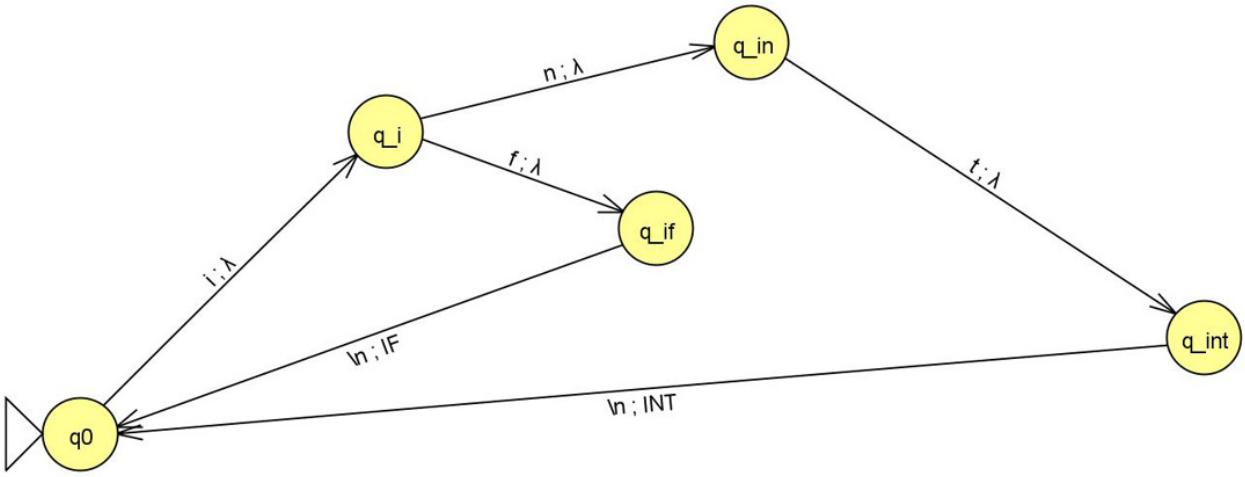

Figura 1: Exemplo de *Máquina de Mealy*

Por exemplo, se a entrada for a sequencia de caracteres 'int\n', as transições se darão por, de q_0 para q_i, não escreve nada na saída; de q_i para q_{in}, não escreve nada na saída; de q_{in} para q_{int}, não escreve nada na saída; de q_{int} para q_0, escreve 'INT' na saída.

4.3 *Máquina de Mealy* como Analisador Léxico

Com a compreensão destes conceitos da *Máquina de Mealy*, podemos observar o diagrama que representa a construção de uma como um analisador léxico na Figura 2. Por fins de visualização, este diagrama não foi construído com todos os estados, contudo, compreende-se que estados que representam operadores, separadores, palavras reservadas, *IDs* e *NUMBERs* tenham transições entre si, e caso a transição seja de um estado de completude (Por exemplo, q_{int} é o estado de completude da palavra reservada INT) para um outro estado qualquer, este faz a transição para este novo estado, escrevendo na saída o *token* que o estado de completude representava.

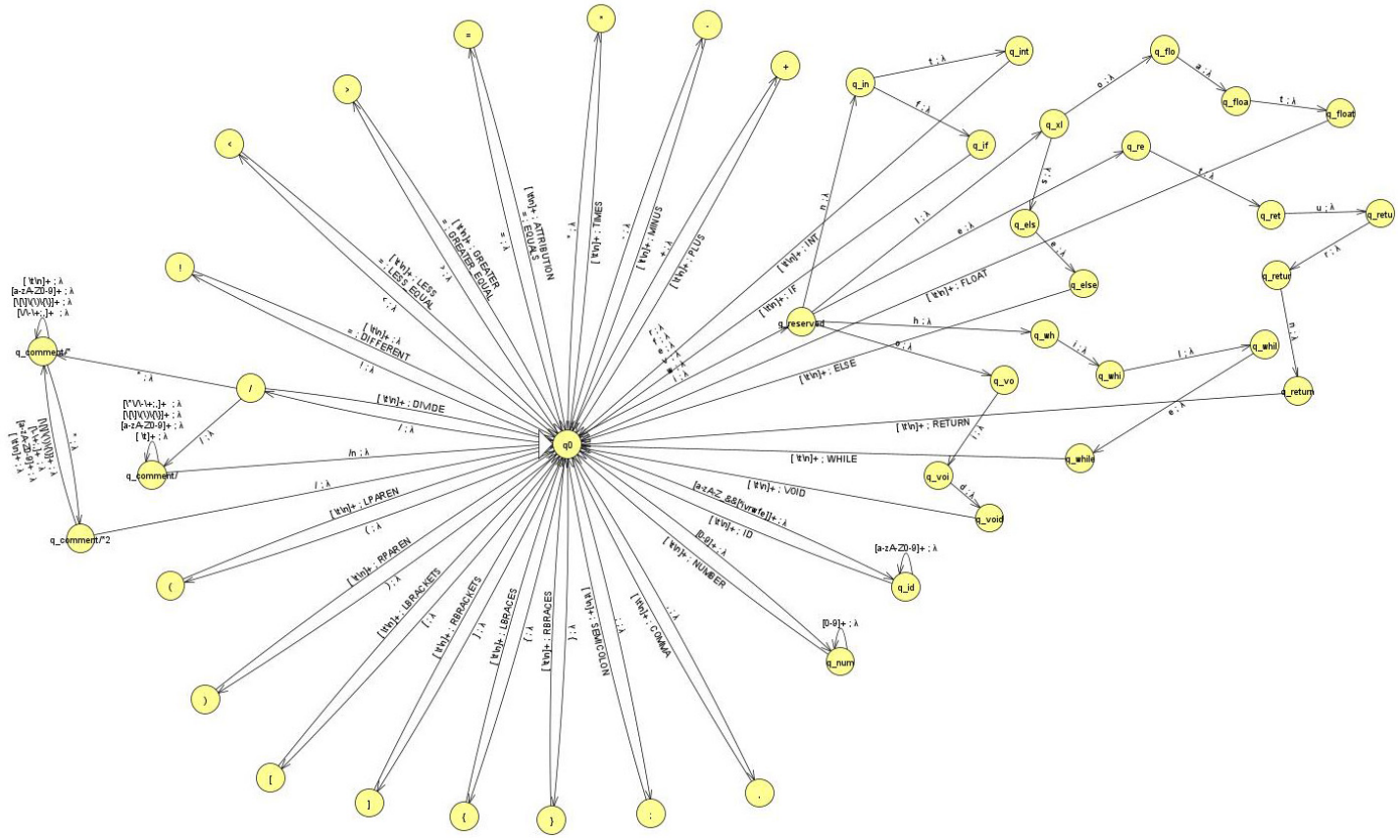

Figura 2: *Mealy* como Analisador Léxico

O estado inicial é o q_0 deste, caso ele receba algum caractere que represente a inicial de uma palavra reservada, este faz a transição para o estado $q_{reserved}$, e por meio do segundo caractere lido, ele decide se vai em direção à um estado de completude, ou se faz a transição para o estado q_{ID}, por exemplo, os estados até atingir o estado de completude q_{while} a partir de $q_{reserved}$ são q_{wh}, q_{whi} e q_{whil}, este último, ao fazer a leitura do caractere 'e', faz a transição para o estado de completude q_{while}. É importante notar que os *tokens* só são escritos na saída ao se realizar a transição de um estado de completude para um estado qualquer. Temos então que a **definição formal** desta *Máquina de Mealy* se dá por:

Σ = {*operadores.keys()*, *separadores.keys()*, [0-9], [a-zA-Z_]},
Q = {
'q_0', '$q_{reserved}$', 'q_{xl}', q_{digit}, 'q_{if}', 'q_{id}',
operators.keys(), *separators.keys()*, int_states,
comment_states, void_states, float_states,
return_states, else_states, while_states
},
δ = O número de transições é demasiadamente grande para representar, as transições são explicadas na Seção 4.3 e observadas na Figura 2.
F: A *Máquina de Mealy* tanto no JFLAP quanto na biblioteca python *automata-lib* não apresentam estados finais.
Δ = [palavras_reservadas, *operators.values()*, *separators.values()*, 'NUMBER', 'ID']

Compreenda *separators* e *operators* como dicionários representando separadores e operadores, onde *keys()* representa o símbolo e *values()* representa o *token* equivalente.

4.4 Exemplos de entrada e saída

Por meio da definição formal da *Máquina de Mealy* construída neste trabalho, temos exemplos de *tokens* gerados para entradas específicas. Por meio dos testes, teve-se que o Código 4 gerou os *tokens* do Código 5, o Código 6 gerou os *tokens* do Código 7 e o Código 8 gerou os *tokens* do Código 9.

Código 4: Declaração da main

```
1 int main(void){
2     return 0;
3 }
```

Código 5: Tokens obtidos

```
1  INT
2  ID
3  LPAREN
4  VOID
5  RPAREN
6  LBRACES
7  RETURN
8  NUMBER
9  SEMICOLON
10 RBRACES
```

Código 6: Declarações de variáveis

```
1  int a;
2  float b;
3
4  int main(void){
5    int x;
6    float y;
7
8    return(0);
9  }
```

Código 7: Tokens obtidos

```
1  INT
2  ID
3  SEMICOLON
4  FLOAT
5  ID
6  SEMICOLON
7  INT
8  ID
9  LPAREN
10 VOID
11 RPAREN
12 LBRACES
13 INT
14 ID
15 SEMICOLON
16 FLOAT
17 ID
18 SEMICOLON
19 RETURN
20 LPAREN
21 NUMBER
22 RPAREN
23 SEMICOLON
24 RBRACES
```

Código 8: Código com comentários

```
1  int gcd (int u, int v){
2    if (v == 0) return u;
3    else return gcd(v,u-u/v*v);
4    /* u-u/v*v == u mod v */
5  }
6
7  void main(void){
8    int x; int y;
9    x = input();
10   y = input();
11   output(gcd(x,y));
12 }
```

Código 9: Tokens Obtidos

```
1  INT
2  ID
3  LPAREN
4  INT
5  ID
6  COMMA
7  INT
8  ID
9  RPAREN
10 LBRACES
11  IF
12 LPAREN
13 ID
14 EQUALS
15 NUMBER
16 RPAREN
17 RETURN
18 ID
19 SEMICOLON
20 ELSE
21 RETURN
22 ID
23 LPAREN
24 ID
25 COMMA
26 ID
27 MINUS
28 ID
29 DIVIDE
30 ID
31 TIMES
32 ID
33 RPAREN
34 SEMICOLON
35 RBRACES
36 VOID
37 ID
38 LPAREN
39 VOID
40 RPAREN
41 LBRACES
42 INT
43 ID
44 SEMICOLON
45 INT
46 ID
47 SEMICOLON
48 ID
49 ATTRIBUTION
50 ID
51 LPAREN
52 RPAREN
53 SEMICOLON
54 ID
55 ATTRIBUTION
56 ID
57 LPAREN
58 RPAREN
59 SEMICOLON
60 ID
61 LPAREN
62 ID
63 LPAREN
64 ID
65 COMMA
66 ID
67 RPAREN
68 RPAREN
69 SEMICOLON
70 RBRACES
```

Referências

LOUDEN, Kenneth C. 2004. *Compiladores: Princípios e práticas*. São Paulo, SP: Thomson 1st edn.

MENEZES, Paulo B. 2008. *Linguagens formais e autômatos*. São Paulo, SP: Bookman 5th edn.