



Universidade do Estado do Rio de Janeiro
Instituto de Matemática e Estatística
Disciplina de Compiladores

Projeto de Compilador

Etapa 1: Análise léxica

Alunos:
Gabriella Ponce
Samiry Sayed

Professora:
Lis Custódio

Setembro
2025

Sumário

1	Introdução	1
1.1	O analisador léxico	1
1.1.1	Tokens, padrões e lexemas	2
2	Descrição teórica	2
2.1	Gramática livre de contexto	2
2.2	Expressões e definições regulares	2
2.2.1	Definições iniciais do alfabeto da linguagem	3
2.2.2	Identificador	3
2.2.3	Palavras reservadas	3
2.2.4	Tipos	3
2.2.5	Operadores	3
2.2.6	Delimitadores	3
2.2.7	Comentários	4
2.3	Autômatos finitos	4
2.3.1	Identificador	4
2.3.2	Palavras reservadas	4
2.3.3	Tipos	5
2.3.4	Operadores e Delimitadores	6
2.3.5	Comentários	6
3	Estrutura e funcionamento do programa	7
3.1	Estados	7
3.2	Estado inicial	7
3.3	Leitura do programa de entrada	8
3.4	Tokens de saída	8
3.5	Tratamento de erros	9
3.6	Função de transição	10
3.7	Conjunto de estados de aceitação	12
3.8	Função main	13
4	Testes Realizados e Saídas Obtidas	14
4.1	Código correto com todos os lexemas da linguagem	14
4.2	Código com erro em números de ponto flutuante	16
4.3	Código com erro de string não fechada	17
4.4	Código com caracter não reconhecido pela gramática	17
4.5	Código com comentário longo não fechado	17
5	Bibliografia	18

1 Introdução

Compiladores são programas de computador que traduzem um software escrito em uma linguagem fonte para um software escrito em uma linguagem alvo. O processo de tradução é composto por duas etapas básicas: a análise front-end, na qual o código de entrada é examinado e compreendido; e a síntese back-end, na qual o código de saída traduzido é gerado. A análise léxica (em inglês, scanning), tema deste relatório, é o primeiro de três estágios da etapa front-end.

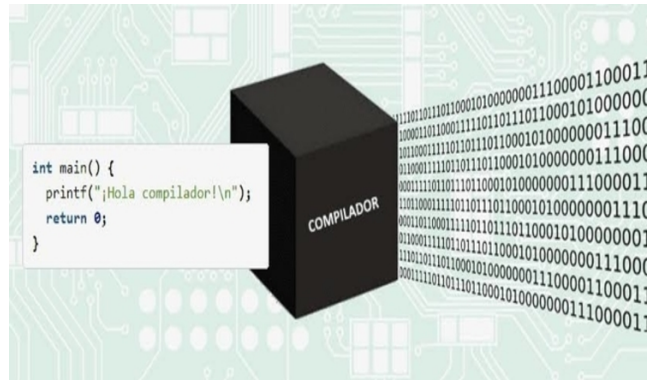


Figura 1: Um compilador traduz um programa escrito em C (linguagem fonte) para um programa escrito em código de máquina (linguagem alvo)

1.1 O analisador léxico

Um analisador léxico é responsável por ler as cadeias do programa de entrada, caracter à caracter, e verificar se as palavras contidas no arquivo são válidas ou não para a linguagem fonte, de modo que palavras válidas são categorizadas sintaticamente. Ele também verifica se foram digitados símbolos que não pertencem à linguagem.

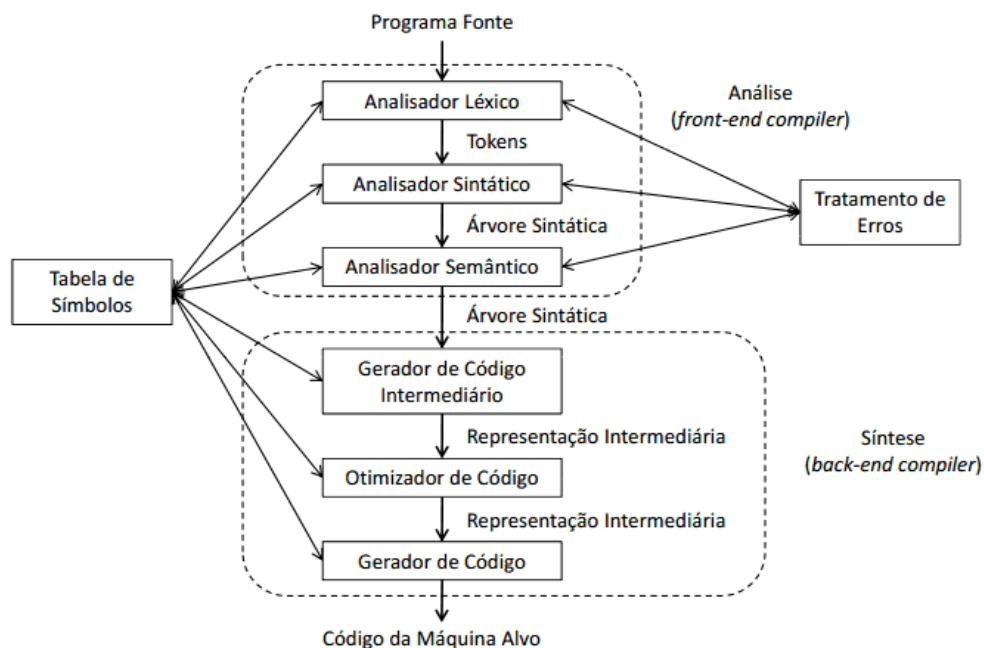


Figura 2: Estrutura de um compilador

1.1.1 Tokens, padrões e lexemas

- **Token:** um par composto de $\langle nome_token, valor_atributo \rangle$, no qual:
 - nome_token: representa qual o tipo (padrão) de unidade léxica
 - valor_atributo: é o valor ou referência na tabela de símbolos
- **Padrão:** regra que descreve a forma assumida por um lexema
- **Lexema:** sequência de caracteres reconhecida pelos padrões da linguagem, ou seja, uma palavra válida.

Sendo assim, se ao ler uma cadeia de caracteres, o analisador léxico foi capaz de reconhecê-la a partir de um dos padrões contido na linguagem, essa cadeia trata-se de um lexema. O scanner, então, converte o lexema para um token e o armazena na tabela de símbolos.

2 Descrição teórica

2.1 Gramática livre de contexto

A linguagem a ser analisada, cujo alfabeto Σ contém os símbolos da tabela ASCII, é descrita pela gramática a seguir:

$$\begin{aligned}\langle programa \rangle &::= \text{inicio } \langle decls \rangle \langle comandos \rangle \text{ fim} \\ \langle decls \rangle &::= \langle decl \rangle \langle decls \rangle \mid \varepsilon \\ \langle decl \rangle &::= \langle tipo \rangle \text{ ID}; \\ \langle tipo \rangle &::= \text{int} \mid \text{float} \mid \text{string} \\ \langle comandos \rangle &::= \langle comando \rangle \langle comandos \rangle \mid \varepsilon \\ \langle comando \rangle &::= \langle atribuicao \rangle; \mid \langle chamada \rangle; \mid \langle entrada \rangle; \mid \langle saida \rangle; \\ &\quad \mid \langle if_stmt \rangle \mid \langle while_stmt \rangle \mid \langle bloco \rangle \\ \langle atribuicao \rangle &::= \text{ID} = \langle expr \rangle \\ \langle chamada \rangle &::= \text{ID}(\langle args \rangle) \\ \langle args \rangle &::= \langle expr_list \rangle \mid \varepsilon \\ \langle expr_list \rangle &::= \langle expr \rangle, \langle expr_list \rangle \mid \langle expr \rangle \\ \langle entrada \rangle &::= \text{read}(\text{ID}) \\ \langle saida \rangle &::= \text{print}(\langle expr \rangle) \\ \langle if_stmt \rangle &::= \text{if}(\langle expr \rangle) \langle comando \rangle \langle else_opt \rangle \\ \langle else_opt \rangle &::= \text{else} \langle comando \rangle \mid \varepsilon \\ \langle while_stmt \rangle &::= \text{while}(\langle expr \rangle) \langle comando \rangle \\ \langle bloco \rangle &::= \{ \langle comandos \rangle \} \\ \langle expr \rangle &::= \langle expr \rangle + \langle term \rangle \mid \langle expr \rangle - \langle term \rangle \mid \langle term \rangle \\ \langle term \rangle &::= \langle term \rangle * \langle factor \rangle \mid \langle term \rangle / \langle factor \rangle \mid \langle factor \rangle \\ \langle factor \rangle &::= \text{ID} \mid \text{NUMBER} \mid \text{STRING} \mid (\langle expr \rangle) \mid -\langle factor \rangle\end{aligned}$$

2.2 Expressões e definições regulares

A implementação do reconhecimento de padrões da linguagem foi feita a partir de um autômato finito cujos estados de aceitação disparam a emissão de um token ou procedimento específico. Uma vez que autômatos finitos e expressões regulares são equivalentes, é possível

utilizar definições regulares para descrever a sintaxe dos lexemas, espaçadores e comentários da linguagem em questão.

Com base na gramática livre de contexto fornecida, o analisador léxico deve reconhecer os tokens e classificá-los como um dos símbolos terminais fornecidos. Dessa maneira, foram definidas formalmente expressões e definições regulares para cada um dos lexemas da gramática.

2.2.1 Definições iniciais do alfabeto da linguagem

$$\begin{aligned} \textit{letra} &\rightarrow A|B|\dots|Z|a|b|\dots|z \\ \textit{digito} &\rightarrow 0|1|2|3|4|5|6|7|8|9 \\ \textit{espaçamento} &\rightarrow \backslash (a|b|f|n|r|t|v|\backslash |") \end{aligned}$$

2.2.2 Identificador

$$ID \rightarrow \textit{letra} | (\textit{letra} | \textit{digito} | _)^*$$

2.2.3 Palavras reservadas

$$\begin{aligned} \textit{BEGIN} &\rightarrow \textit{inicio} \\ \textit{END} &\rightarrow \textit{fim} \\ \textit{IF} &\rightarrow \textit{if} \\ \textit{ELSE} &\rightarrow \textit{else} \\ \textit{WHILE} &\rightarrow \textit{while} \\ \textit{READ} &\rightarrow \textit{read} \\ \textit{PRINT} &\rightarrow \textit{print} \\ \textit{TYPE_INT} &\rightarrow \textit{int} \\ \textit{TYPE_FLOAT} &\rightarrow \textit{float} \\ \textit{TYPE_STRING} &\rightarrow \textit{string} \\ \textit{palavra.reservada} &\rightarrow \textit{BEGIN} | \textit{END} | \textit{IF} | \textit{ELSE} | \textit{WHILE} | \textit{READ} | \textit{PRINT} | \textit{TYPE_INT} | \\ &\textit{TYPE_FLOAT} | \textit{TYPE_STRING} \end{aligned}$$

2.2.4 Tipos

$$\begin{aligned} \textit{NUMBER_INT} &\rightarrow (-\textit{digito} | \textit{digito})(\textit{digito}^*) \\ \textit{NUMBER_FLOAT} &\rightarrow (\textit{NUMBER_INT}).(\textit{digito}^*) \\ \textit{STRING} &\rightarrow " \Sigma^* " \end{aligned}$$

2.2.5 Operadores

$$\begin{aligned} \textit{ASSIGN} &\rightarrow = \\ \textit{OP_ADD} &\rightarrow + \\ \textit{OP_SUB} &\rightarrow - \\ \textit{OP_MULT} &\rightarrow * \\ \textit{OP_DIV} &\rightarrow / \end{aligned}$$

Obs.: Em `OP_MULT`, `*` trata-se do caractere asterisco, não da operação estrela

2.2.6 Delimitadores

$$\begin{aligned} \textit{SEMICOLON} &\rightarrow ; \\ \textit{LEFT_PARENTHESIS} &\rightarrow (\\ \textit{RIGHT_PARENTHESIS} &\rightarrow) \\ \textit{LEFT_BRACKET} &\rightarrow \{ \\ \textit{RIGHT_BRACKET} &\rightarrow \} \\ \textit{COMMA} &\rightarrow , \end{aligned}$$

$DELIM \rightarrow SEMICOLON|LEFT_PARENTHESIS|RIGHT_PARENTHESIS|$
 $LEFT_BRACKET|RIGHT_BRACKET|COMMA$

2.2.7 Comentários

$SMALL_COMMENTARY \rightarrow (\Sigma^*)$

$BIG_COMMENTARY \rightarrow [[(\Sigma^*)]]$

$COMMENTARY \rightarrow OP_DIV - (SMALL_COMMENTARY|BIG_COMMENTARY)$

2.3 Autômatos finitos

Um autômato finito é uma 5-upla $(Q, \Sigma, \delta, q_0, F)$, onde:

- Q é um conjunto finito chamado de estados
- Σ é um conjunto finito chamado de alfabeto
- $\delta : Q \rightarrow \Sigma$ é a função de transição
- q_0 é o estado inicial
- $F \subseteq Q$ é o conjunto de estados de aceitação

Uma vez que autômatos finitos podem ser expressos como diagramas de estado, seguem abaixo os diagramas correspondentes às expressões regulares da seção anterior:

2.3.1 Identificador



Figura 3: Autômato finito para identificadores

2.3.2 Palavras reservadas

Foi observado que os padrões que reconhecem palavras reservadas da linguagem são sub-padrões de ID. Dessa maneira, optou-se por verificar se uma cadeia é reconhecida pelo padrão de ID e, em seguida, verificar se ela é uma das palavras reservadas da linguagem.

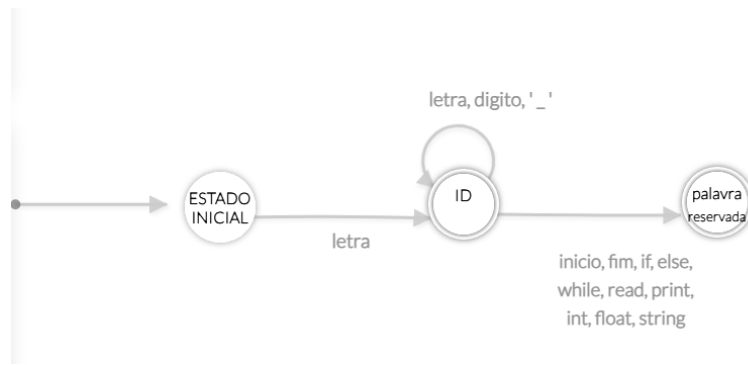


Figura 4: Autômato finito para palavras reservadas

2.3.3 Tipos

Notou-se que números do tipo float possuem uma parte inteira antes do caracter '.', de modo que a máquina de estados primeiro reconhece um número inteiro e, caso haja um ponto decimal, passa para o estado de reconhecimento de um float.

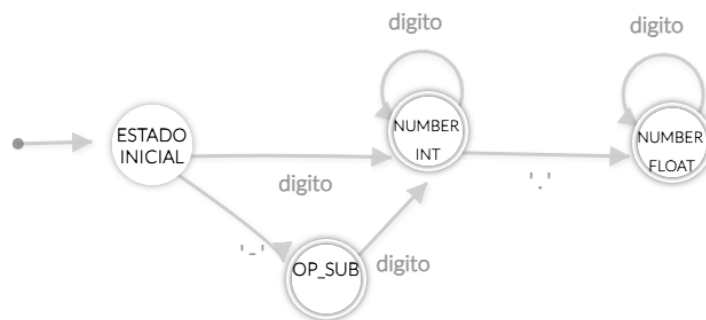


Figura 5: Autômato finito para int e float

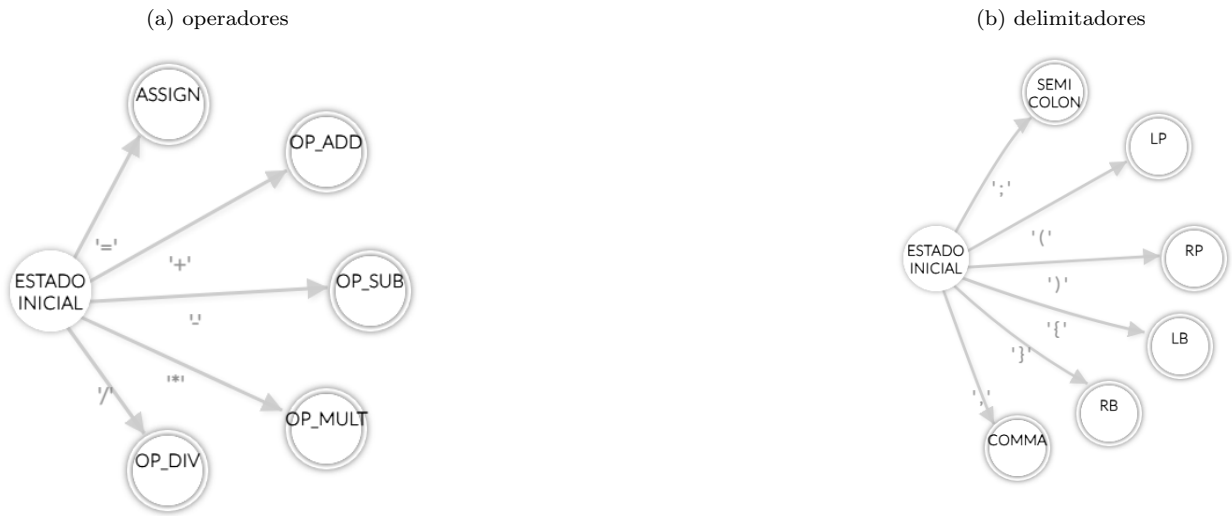


Figura 6: Autômato finito para string

2.3.4 Operadores e Delimitadores

Tanto operadores quanto delimitadores são reconhecidos por um padrão formado por um único caractere. Portanto, a estrutura dos autômatos para esses lexemas é semelhante.

Figura 7: Autômatos finitos para:



2.3.5 Comentários

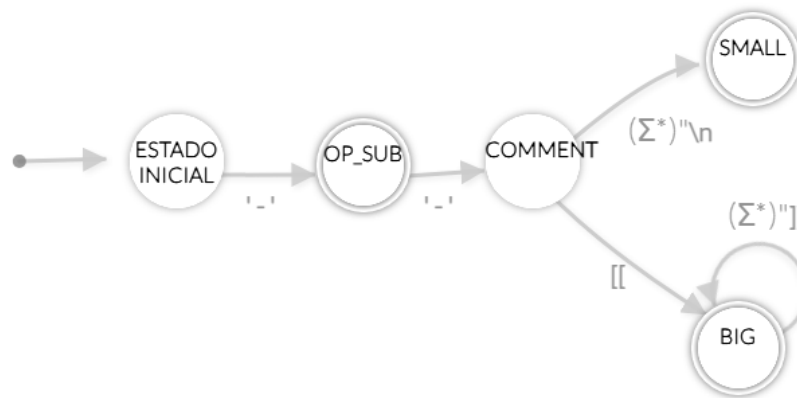


Figura 8: Autômato finito para comentários

3 Estrutura e funcionamento do programa

Para a implementação do analisador léxico, os autômatos supracitados foram agrupados e convertidos para a linguagem C. A estrutura do programa foi feita da seguinte maneira:

3.1 Estados

O conjunto de estados Q do autômato foi representado por meio da estrutura de dados enum denominada `tipo_token`, que define um conjunto fixo de constantes. Cada constante relaciona-se com um estado do autômato.

```
1 enum tipo_token {
2     ESTADO_INICIAL = 256,
3     BEGIN, //inicio
4     ID, //id
5     NUMBER,
6     NUMBER_FLOAT,
7     STRING,
8     SMALL_COMMENTARY,
9     COMMENTARY,
10    TYPE_INT, //int
11    TYPE_FLOAT, //float
12    TYPE_STRING, //string
13    IF, //if
14    ELSE, //else
15    WHILE, //while
16    READ, //read
17    PRINT, //print
18    SEMICOLON, //;
19    ASSIGN, // =
20    LEFT_PARENTHESIS, // (
21    RIGHT_PARENTHESIS, // )
22    LEFT_BRACKET, // {
23    RIGHT_BRACKET, // }
24    COMMA, // ,
25    OP_SUM, // +
26    OP_SUB, // -
27    OP_MUL, // *
28    OP_DIV, // /
29    END // fim
30 };
```

```
1 enum atributos {
2     INT = 282,
3     FLOAT,
4 };
```

3.2 Estado inicial

- A variável estado armazena o estado atual do autômato, e é inicializada com a constante `ESTADO_INICIAL`, ou seja, o q_0 da máquina de estados.
- A variável `cont_sim_lido`, inicializada com zero, armazena quantos caracteres do arquivo já foram lidos, ou seja, indica a posição atual durante a leitura do código fonte.
- A variável `code` é um apontador para char utilizada para apontar para uma string contendo todo o código do arquivo.

```
1 int estado = ESTADO_INICIAL;
2 int cont_sim_lido = 0;
3 char *code;
```

3.3 Leitura do programa de entrada

A função `readFile()` é responsável por ler caractere à caractere do arquivo `.txt` que contém o programa de entrada.

```
1 unsigned char *readFile(char *fileName) {
2     FILE *file = fopen(fileName, "r");
3     char *code;
4     int n = 0;
5     int c;
6
7     if (!file) {
8         fprintf(stderr, "Erro ao ler o arquivo: %s.\n", fileName);
9         exit(3);
10    }
11
12    fseek(file, 0, SEEK_END);
13    long f_size = ftell(file);
14    fseek(file, 0, SEEK_SET);
15
16    code = (unsigned char *)malloc((f_size + 1) * sizeof(char));
17
18    while ((c = fgetc(file)) != EOF) {
19        code[n++] = (unsigned char)c;
20    }
21    code[n] = '\0';
22    return code;
23 }
```

3.4 Tokens de saída

A estrutura de dados **Token** armazena, como mencionado em 1.1.1, um par composto pelo nome do token e o valor associado a ele na tabela de símbolos.

```
1 typedef struct Token {
2     enum tipo_token nome_token;
3     int atributo;
4 } Token;
```

A função `proximo_token()`, que será detalhada em 3.6, é responsável por fazer a conversão citada em 1.1.1 e retornar um **Token** que será armazenado na tabela de símbolos. Optou-se por implementar essa tabela em uma estrutura de dados de lista encadeada, na qual cada nó da lista armazena a posição do token na tabela, `lex[30]`, o tipo do token e o próximo item da lista. A tabela é inicializada vazia e a função `inserir_na_tabela()` será utilizada para que os tokens identificados pelo scanner sejam adicionados à tabela de símbolos.

```
1
2 //tabela de simbolos
3
4 typedef struct no{
5     int pos;
6     char lex[30];
7     enum tipo_token tipo;
8     struct no *proximo;
9 }No;
10
11 No *tabela_simbolos = NULL;
12 int posicao_na_tabela;
13
14 int inserir_na_tabela(No **lista, char *lexema, enum tipo_token tipo){
15     No *aux, *novo = malloc(sizeof(No));
16     No *ultimo = NULL;
17     int pos = 0;
18 }
```

```

19     if(novo){
20         strcpy(novo -> lex, lexema);
21         novo -> tipo = tipo;
22         novo -> proximo = NULL;
23
24         if(*lista == NULL){
25             novo -> pos = pos;
26             *lista = novo;
27         }
28         else{
29             aux = *lista;
30             while(aux != NULL){
31                 if(strcmp(aux -> lex, lexema) == 0){
32                     int pos_atual = aux -> pos;
33                     free(novo);
34                     return pos_atual;
35                 }
36                 ultimo = aux;
37                 pos++;
38                 aux = aux -> proximo;
39             }
40             novo -> pos = pos;
41             ultimo -> proximo = novo;
42         }
43
44         return novo -> pos;
45     }
46     else{
47         printf("Erro ao alocar memoria!\n");
48         return -1;
49     }
50 }

```

3.5 Tratamento de erros

Optou-se por associar tratamento de erros individuais aos estados dos autômatos, de modo que cada falha possui um código correspondente na função falhar.

- A falha 1 é chamada em ESTADO_INICIAL
- A falha 2 é chamada em NUMBER_FLOAT
- A falha 3 é chamada em STRING
- A falha 4 é chamada em COMMENTARY

```

1 void falhar(int erro) { //recebe o numero do erro como parametro, encerra o programa
    quando chamada
2
3     switch (erro) {
4
5         case 1:
6             printf("ERRO: caracter nao pertencente a linguagem!\n");
7             break;
8
9         case 2:
10            printf("ERRO: numero invalido!\n");
11            break;
12
13        case 3:
14            printf("ERRO: string nao foi fechada!\n");
15            break;
16
17        case 4:

```

```

18     printf("ERRO: \ucomentario nao foi fechado!\n");
19     break;
20
21 }
22 exit(-1);
23 }

```

3.6 Função de transição

As transições entre os estados do autômato, representadas na seção 2.3, ocorrem dentro da função `proximo_token()` através de estruturas condicionais.

```

1 Token proximo_token() {
2     Token token;
3     char c;
4     while (cont_sim_lido < strlen(code) && code[cont_sim_lido] != '\0') {
5         switch (estado) {
6
7             case ESTADO_INICIAL:
8
9                 c = code[cont_sim_lido];
10                if ((c == '\u') || (c == '\n')) {
11                    estado = ESTADO_INICIAL;
12                    cont_sim_lido++;
13                }
14
15                //ids e palavras reservadas(verificadas dentro de ID)
16
17                else if (isalpha(c)){
18                    estado = ID;
19                }
20
21                //strings
22
23                else if (c == '"'){
24                    estado = STRING;
25                }
26
27                //numeros
28
29                else if (isdigit(c)){
30                    estado = NUMBER;
31                }
32
33                //caracteres unicos
34
35                else if (c == ';') {
36                    estado = SEMICOLON;
37                }
38                else if (c == '=') {
39                    estado = ASSIGN;
40                }
41                else if (c == '(') {
42                    estado = LEFT_PARENTHESIS;
43                }
44                else if (c == ')') {
45                    estado = RIGHT_PARENTHESIS;
46                }
47                else if (c == '{') {
48                    estado = LEFT_BRACKET;
49                }
50                else if (c == '}') {
51                    estado = RIGHT_BRACKET;
52                }
53                else if (c == ',') {

```

```

54         estado = COMMA;
55     }
56     else if (c == '+') {
57         estado = OP_SUM;
58     }
59     else if (c == '-') {
60         estado = OP_SUB;
61     }
62     else if (c == '*') {
63         estado = OP_MUL;
64     }
65     else if (c == '/') {
66         estado = OP_DIV;
67     }
68     else
69         falhar(1);
70     break;
71
72     case SMALL_COMMENTARY:
73         ...
74     case COMMENTARY:
75         ...
76
77     case NUMBER:
78         ...
79
80     case NUMBER_FLOAT:
81         ...
82
83     case STRING:
84         ...
85
86     case ID:
87         ...
88
89     case BEGIN:
90         ...
91
92     case END:
93         ...
94
95     case IF:
96         ...
97
98     case TYPE_INT:
99         ...
100
101     case ELSE:
102         ...
103
104     case WHILE:
105         ...
106
107     case READ:
108         ...
109
110     case PRINT:
111         ...
112
113     case TYPE_FLOAT:
114         ...
115
116     case TYPE_STRING:
117         ...
118
119     case SEMICOLON:

```

```

120         ...
121
122     case ASSIGN:
123         ...
124
125     case LEFT_PARENTHESIS:
126         ...
127
128     case RIGHT_PARENTHESIS:
129         ...
130
131     case LEFT_BRACKET:
132         ...
133
134     case RIGHT_BRACKET:
135         ...
136
137     case COMMA:
138         ...
139
140     case OP_SUM:
141         ...
142
143     case OP_SUB:
144         ...
145
146     case OP_MUL:
147         ...
148
149     case OP_DIV:
150         ...
151
152     }
153
154     }
155 }

```

3.7 Conjunto de estados de aceitação

Se a função `proximo_token()` reconhece que determinada cadeia de caracteres é aceita pela linguagem, então o token é impresso na tela, o autômato volta ao seu estado inicial e o token é retornado pela função.

```

1  ...
2  printf("<NUMBER, □FLOAT>\n");
3  token.nome_token = NUMBER;
4  token.atributo = FLOAT;
5  estado = ESTADO_INICIAL;
6  return(token);
7  ...

```

```

1  ...
2  cont_sim_lido++;
3  printf("<inicio, □>\n");
4  token.nome_token = BEGIN;
5  token.atributo = -1;
6  estado = ESTADO_INICIAL;
7  return(token);
8  ...

```

3.8 Função main

Na função principal, a variável token é criada para armazenar os tokens lidos. A variável code carrega o programa fonte na memória, de modo que o scanner lê o arquivo "programa.txt" caractere por caractere em busca de lexemas, que são reconhecidos pela função proximo_token(), até que se chegue ao fim do arquivo ou então aconteça uma falha. Por fim, libera-se toda a memória que foi alocada dinamicamente.

```
1 int main() {  
2     Token token;  
3     code = readFile("programa.txt");  
4     do {  
5         token = proximo_token();  
6     } while (token.nome_token != EOF);  
7     free(code);  
8     free(tabela_simbolos);  
9 }
```

4 Testes Realizados e Saídas Obtidas

A seguir, estão os testes realizados, cujas saídas obtidas consistem na impressão dos tokens identificados pelo analisador léxico para o caso de palavras válidas, e na impressão das mensagens de erro correspondentes para o caso de palavras inválidas,.

4.1 Código correto com todos os lexemas da linguagem

```
1  inicio
2      string s; int a, float b; int i; --[[Comentario]]
3      i = 1;
4      read(a); read(b); --le os valores de a e b
5      while(i){
6          if (a + b = 4){
7              a = a + 1;
8          }
9          else if (a / b = 5){
10             b = b + 1;
11         }
12         else if (a * b = 10){
13             a = b + 1;
14         }
15         else if (a - b = 0){
16             a = b + b;
17         }
18         else
19             i = 0;
20     }
21     s = "finalizado";
22     print(s);
23 fim
```

Saída obtida:

```
1  <inicio, >
2  <string, >
3  <ID, 0>
4  <;, >
5  <int, >
6  <ID, 1>
7  <,, >
8  <float, >
9  <ID, 2>
10 <;, >
11 <int, >
12 <ID, 3>
13 <;, >
14 <COMMENTARY, >
15 <ID, 3>
16 <=, >
17 <NUMBER, INT>
18 <;, >
19 <read, >
20 <(<, >
21 <ID, 1>
22 <), >
23 <;, >
24 <read, >
25 <(<, >
26 <ID, 2>
27 <), >
28 <;, >
29 <SMALL_COMMENTARY, >
30 <while, >
31 <(<, >
```



```

32 <ID, 3>
33 <), >
34 <{, >
35 <if, >
36 <(<, >
37 <ID, 1>
38 <+, >
39 <ID, 2>
40 <=, >
41 <NUMBER, INT>
42 <), >
43 <{, >
44 <ID, 1>
45 <=, >
46 <ID, 1>
47 <+, >
48 <NUMBER, INT>
49 <;, >
50 <}, >
51 <else, >
52 <if, >
53 <(<, >
54 <ID, 1>
55 </, >
56 <ID, 2>
57 <=, >
58 <NUMBER, INT>
59 <), >
60 <{, >
61 <ID, 2>
62 <=, >
63 <ID, 2>
64 <+, >
65 <NUMBER, INT>
66 <;, >
67 <}, >
68 <else, >
69 <if, >
70 <(<, >
71 <ID, 1>
72 <*, >
73 <ID, 2>
74 <=, >
75 <NUMBER, INT>
76 <), >
77 <{, >
78 <ID, 1>
79 <=, >
80 <ID, 2>
81 <+, >
82 <NUMBER, INT>
83 <;, >
84 <}, >
85 <else, >
86 <if, >
87 <(<, >
88 <ID, 1>
89 <- , >
90 <ID, 2>
91 <=, >
92 <NUMBER, INT>
93 <), >
94 <{, >
95 <ID, 1>
96 <=, >
97 <ID, 2>

```

```

98 <+, >
99 <ID, 2>
100 <;, >
101 <}, >
102 <else, >
103 <ID, 3>
104 <=, >
105 <NUMBER, INT>
106 <;, >
107 <}, >
108 <ID, 0>
109 <=, >
110 <STRING, >
111 <;, >
112 <print, >
113 <(<, >
114 <ID, 0>
115 <), >
116 <;, >
117 <fim, >

```

4.2 Código com erro em números de ponto flutuante

```

1 inicio
2     int a; int b;
3     read(a);
4     read(b);
5     b = (a + 2..)
6 fim

```

Saída obtida:

```

1 <SMALL_COMMENTARY, >
2 <inicio, >
3 <int, >
4 <ID, 0>
5 <;, >
6 <int, >
7 <ID, 1>
8 <;, >
9 <read, >
10 <(<, >
11 <ID, 0>
12 <), >
13 <;, >
14 <read, >
15 <(<, >
16 <ID, 2>
17 <), >
18 <;, >
19 <ID, 1>
20 <=, >
21 <(<, >
22 <ID, 0>
23 <+, >
24 ERRO: numero invalido!

```

4.3 Código com erro de string não fechada

```
1  inicio
2      string s;
3      s = "analizador_lexico;
4  fim;
```

Saída obtida:

```
1  <inicio, >
2  <string, >
3  <ID, 0>
4  <;, >
5  <ID, 1>
6  <=, >
7  ERRO: string nao foi fechada!
```

4.4 Código com caracter não reconhecido pela gramática

```
1  inicio
2      int i;
3      i = i + $;
4  fim
```

Saída obtida:

```
1  <inicio, >
2  <int, >
3  <ID, 0>
4  <;, >
5  <ID, 1>
6  <=, >
7  <ID, 0>
8  <+, >
9  ERRO: caracter nao pertence a linguagem!
```

4.5 Código com comentário longo não fechado

```
1  inicio
2      string s; int a, float b; int i; --[[analizador
3      i = 1;
4      read(a); read(b); --le os valores de a e b
5      s = "finalizado";
6      print(s);
7  fim
```

Saída obtida:

```
1  <inicio, >
2  <string, >
3  <ID, 0>
4  <;, >
5  <int, >
6  <ID, 1>
7  <,, >
8  <float, >
9  <ID, 2>
10 <;, >
11 <int, >
12 <ID, 3>
13 <;, >
14 ERRO: comentario nao foi fechado!
```

5 Bibliografia

MARIA, A.; TOSCANI, S. S.; DE, I. Implementação de linguagens de programação. [s.l: s.n.].

SETHI, R.; ULLMAN, J. D.; LAM, M. S. Compiladores : principios, tecnicas e ferramentas. Sao Paulo: Pearson Addison Wesley, 2008.

KEITH DANIEL COOPER; TORCZON, L. Construindo Compiladores. [s.l: s.n.].

FERNANDES, H. M. Código ASCII – Tabela ASCII Completa. Disponível em: <<https://dev.to/shadowlik/codigo-ascii-tabela-ascii-completa-397d>>. Acesso em: set. 2025.