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

Compiladores são programas de computador que traduzem um software escrito em uma liguagem 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.

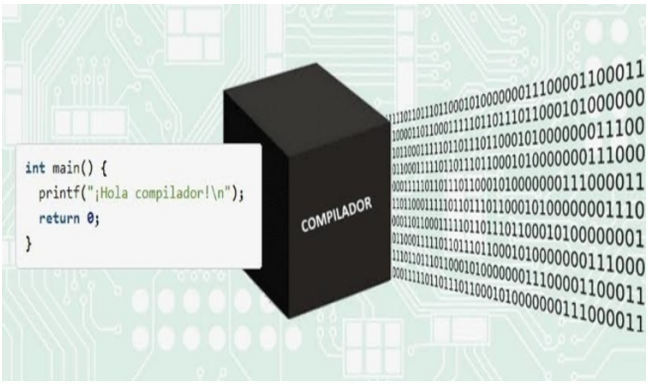


Figura 1: Um compilador traduz um programa escrito em C (linguagem fonte) para um programa escrito em código de máquina (linguagem alvo)

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.

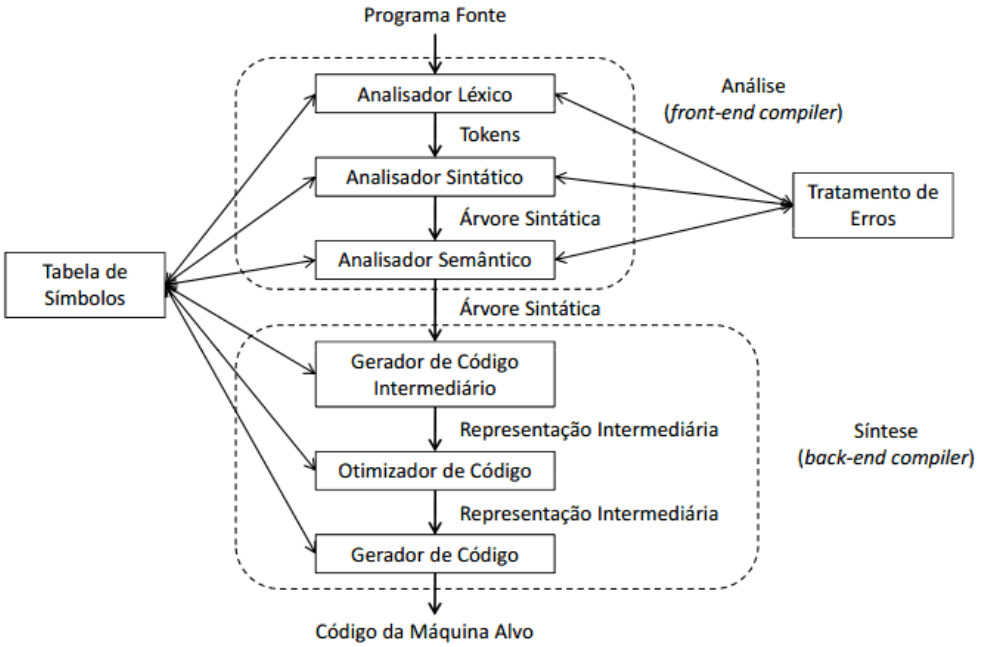


Figura 2: Estrutura de um compilador

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:

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

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 grámatica 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

$letra \rightarrow A|B|...|Z|a|b|...|z$
$digito \rightarrow 0|1|2|3|4|5|6|7|8|9$
$espacamento \rightarrow \backslash\ (a|b|f|n|r|t|v|\backslash\ |")$

2.2.2 Identificador

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

2.2.3 Palavras reservadas

$BEGIN \rightarrow inicio$
$END \rightarrow fim$
$IF \rightarrow if$
$ELSE \rightarrow else$
$WHILE \rightarrow while$
$READ \rightarrow read$
$PRINT \rightarrow print$
$TYPE\_INT \rightarrow int$
$TYPE\_FLOAT \rightarrow float$
$TYPE\_STRING \rightarrow string$
$palavra\_reservada \rightarrow BEGIN|END|IF|ELSE|WHILE|READ|PRINT|TYPE\_INT|TYPE\_FLOAT|TYPE\_STRING$

2.2.4 Tipos

$NUMBER\_INT \rightarrow (-digito|digito)(digito^*)$
$NUMBER\_FLOAT \rightarrow (NUMBER\_INT).(digito^*)$
$STRING \rightarrow "\Sigma^*"$

2.2.5 Operadores

$ASSIGN \rightarrow =$
$OP\_ADD \rightarrow +$
$OP\_SUB \rightarrow -$
$OP\_MULT \rightarrow *$
$OP\_DIV \rightarrow /$
Obs.: Em OP_MULT, '*' trata-se do caractere asterisco, não da operação estrela

2.2.6 Delimitadores

$SEMICOLON \rightarrow ;$
$LEFT\_PARENTHESIS \rightarrow ($
$RIGHT\_PARENTHESIS \rightarrow)$
$LEFT\_BRACKET \rightarrow \{$
$RIGHT\_BRACKET \rightarrow \}$
$COMMA \rightarrow ,$

$DELIM \rightarrow SEMICOLON|LEFT\_PARENTHESIS|RIGHT\_PARENTHESIS|$
$LEFT\_BRACKET|RIGHT\_BRACKET|COMMA$

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, Σ, δ, 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.

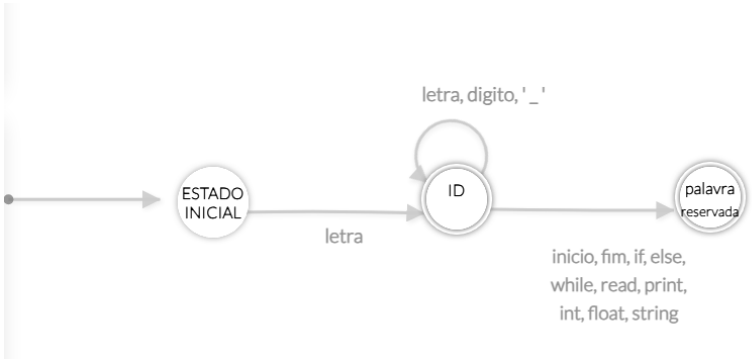


Figura 4: Autômato finito para palavras reservadas

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.

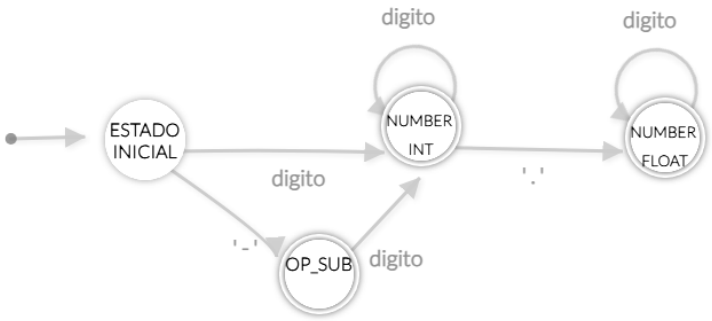


Figura 5: Autômato finito para int e float

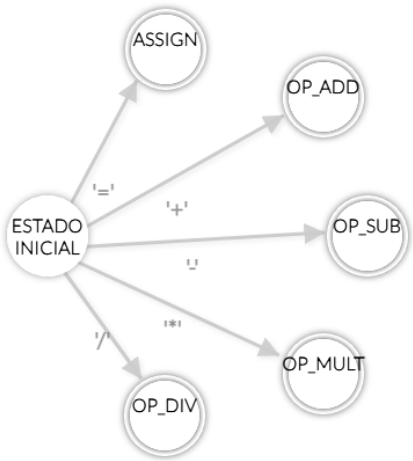
Figura 6: Autômato finito para string

2.3.4 Operadores e Delimitadores

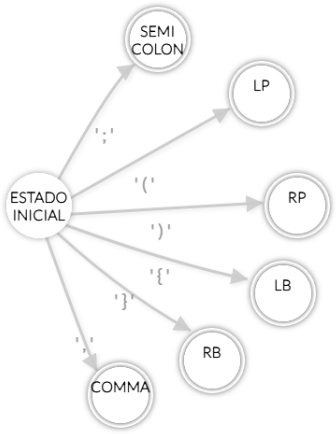
Tanto operadores quanto delimitadores são reconhecidos por um padrão formado por um único caractere. Portanto, a estrutura dos autômatos para esses lexemas é semelhante.

Figura 7: Autômatos finitos para:

(a) operadores



(b) delimitadores



2.3.5 Comentários

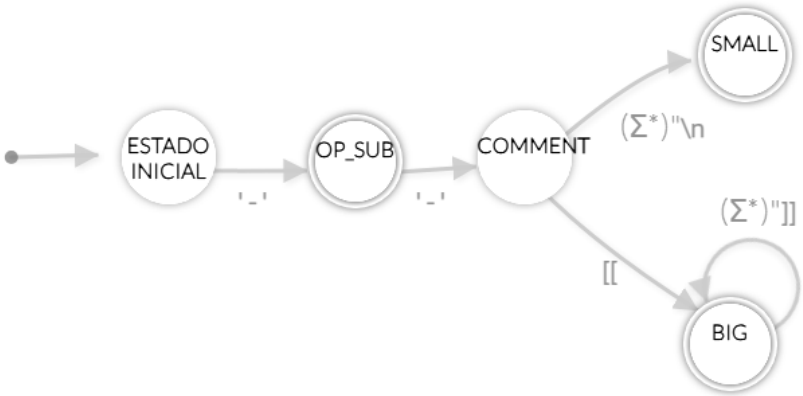


Figura 8: Autômato finito para comentários

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.

```
enum tipo_token {
    ESTADO_INICIAL = 256,
    BEGIN, //inicio
    ID, //id
    NUMBER,
    NUMBER_FLOAT,
    STRING,
    SMALL_COMMENTARY,
    COMMENTARY,
    TYPE_INT, //int
    TYPE_FLOAT, //float
    TYPE_STRING, //string
    IF, //if
    ELSE, //else
    WHILE, //while
    READ, //read
    PRINT, //print
    SEMICOLON,//;
    ASSIGN, // =
    LEFT_PARENTHESIS, // (
    RIGHT_PARENTHESIS,// )
    LEFT_BRACKET, // {
    RIGHT_BRACKET, // }
    COMMA, // ,
    OP_SUM, // +
    OP_SUB, // -
    OP_MUL, // *
    OP_DIV, // /
    END // fim
};
```

```
  enum atributos {
    INT = 282,
    FLOAT,
};
```

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 variavel 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.

```
int estado = ESTADO_INICIAL;
int cont_sim_lido = 0;
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.

```
unsigned char *readFile(char *fileName) {
  FILE *file = fopen(fileName, "r");
  char *code;
  int n = 0;
  int c;

  if (!file) {
    fprintf(stderr, "Erro␣ao␣ler␣o␣arquivo:␣%s.\n", fileName);
    exit(3);
  }

  fseek(file, 0, SEEK_END);
  long f_size = ftell(file);
  fseek(file, 0, SEEK_SET);

  code = (unsigned char *)malloc((f_size + 1) * sizeof(char));

  while ((c = fgetc(file)) != EOF) {
    code[n++] = (unsigned char)c;
  }
  code[n] = '\0';
  return code;
}
```

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.

```
typedef struct Token {
  enum tipo_token nome_token;
  int atributo;
} 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.

```

//tabela de simbolos

typedef struct no{
  int pos;
  char lex[30];
  enum tipo_token tipo;
  struct no *proximo;
}No;

No *tabela_simbolos = NULL;
int posicao_na_tabela;

int inserir_na_tabela(No **lista, char *lexema, enum tipo_token tipo){
    No *aux, *novo = malloc(sizeof(No));
    No *ultimo = NULL;
    int pos = 0;

```

```
    if(novo){
        strcpy(novo -> lex, lexema);
        novo -> tipo = tipo;
        novo -> proximo = NULL;

        if(*lista == NULL){
            novo -> pos = pos;
            *lista = novo;
        }
        else{
            aux = *lista;
            while(aux != NULL){
              if(strcmp(aux -> lex, lexema) == 0){
                int pos_atual = aux -> pos;
                free(novo);
                return pos_atual;
              }
              ultimo = aux;
              pos++;
              aux = aux -> proximo;
            }
            novo -> pos = pos;
            ultimo -> proximo = novo;
        }

        return novo -> pos;
    }
    else{
        printf("Erro ao alocar memoria!\n");
        return -1;
    }
}
```

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

```
void falhar(int erro) { //recebe o numero do erro como parametro, encerra o programa
    quando chamada

  switch (erro) {

    case 1:
      printf("ERRO: caracter nao pertencente a linguagem!\n");
      break;

    case 2:
      printf("ERRO: numero invalido!\n");
      break;

    case 3:
      printf("ERRO: string nao foi fechada!\n");
      break;

    case 4:
```

```
      printf("ERRO: comentario nao foi fechado!\n");
      break;

  }
  exit(-1);
}
```

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.

```
Token proximo_token() {
  Token token;
  char c;
  while (cont_sim_lido < strlen(code) && code[cont_sim_lido] != '\0') {
    switch (estado) {

      case ESTADO_INICIAL:

            c = code[cont_sim_lido];
            if ((c == ' ') || (c == '\n')) {
              estado = ESTADO_INICIAL;
              cont_sim_lido++;
            }

            //ids e palavras reservadas(verificadas dentro de ID)

            else if (isalpha(c)){
                estado = ID;
            }

            //strings

            else if (c == '"'){
              estado = STRING;
            }

            //numeros

            else if (isdigit(c)){
              estado = NUMBER;
            }

            //caracteres unicos

            else if (c == ';') {
              estado = SEMICOLON;
            }
            else if (c == '=') {
              estado = ASSIGN;
            }
            else if (c == '(') {
              estado = LEFT_PARENTHESIS;
            }
            else if (c == ')') {
              estado = RIGHT_PARENTHESIS;
            }
            else if (c == '{') {
              estado = LEFT_BRACKET;
            }
            else if (c == '}') {
              estado = RIGHT_BRACKET;
            }
            else if (c == ',') {
```

```
            estado = COMMA;
          }
          else if (c == '+') {
            estado = OP_SUM;
          }
          else if (c == '-') {
            estado = OP_SUB;
          }
          else if (c == '*') {
            estado = OP_MUL;
          }
          else if (c == '/') {
            estado = OP_DIV;
          }
          else
            falhar(1);
          break;

      case SMALL_COMMENTARY:
          ...
      case COMMENTARY:
          ...

      case NUMBER:
          ...

      case NUMBER_FLOAT:
          ...

      case STRING:
          ...

      case ID:
          ...

        case BEGIN:
          ...

        case END:
          ...

        case IF:
          ...

        case TYPE_INT:
          ...

        case ELSE:
          ...

        case WHILE:
          ...

        case READ:
          ...

        case PRINT:
          ...

        case TYPE_FLOAT:
          ...

        case TYPE_STRING:
          ...

        case SEMICOLON:
```

```
            ...

        case ASSIGN:
            ...

        case LEFT_PARENTHESIS:
            ...

        case RIGHT_PARENTHESIS:
            ...

        case LEFT_BRACKET:
            ...

        case RIGHT_BRACKET:
            ...

        case COMMA:
            ...

        case OP_SUM:
            ...

        case OP_SUB:
            ...

        case OP_MUL:
            ...

        case OP_DIV:
            ...

      }

    }
}
```

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.

```
...
    printf("<NUMBER, FLOAT>\n");
    token.nome_token = NUMBER;
    token.atributo = FLOAT;
    estado = ESTADO_INICIAL;
    return(token);
...
```

```
...
    cont_sim_lido++;
    printf("<inicio, >\n");
    token.nome_token = BEGIN;
    token.atributo = -1;
    estado = ESTADO_INICIAL;
    return(token);
...
```

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

```
inicio
    string s; int a, float b; int i; --[[Comentario]]
    i = 1;
    read(a); read(b); --le os valores de a e b
    while(i){
        if (a + b = 4){
            a = a + 1;
        }
        else if (a / b = 5){
            b = b + 1;
        }
        else if (a * b = 10){
            a = b + 1;
        }
        else if (a - b = 0){
            a = b + b;
        }
        else
            i = 0;
    }
    s = "finalizado";
    print(s);
fim
```

Saída obtida:

```
<inicio, >
<string, >
<ID, 0>
<;, >
<int, >
<ID, 1>
<,, >
<float, >
<ID, 2>
<;, >
<int, >
<ID, 3>
<;, >
<COMMENTARY, >
<ID, 3>
<=, >
<NUMBER, INT>
<;, >
<read, >
<(, >
<ID, 1>
<), >
<;, >
<read, >
<(, >
<ID, 2>
<), >
<;, >
<SMALL_COMMENTARY, >
<while, >
<(, >
```

```
<ID, 3>
<), >
<{, >
<if, >
<(, >
<ID, 1>
<+, >
<ID, 2>
<=, >
<NUMBER, INT>
<), >
<{, >
<ID, 1>
<=, >
<ID, 1>
<+, >
<NUMBER, INT>
<;, >
<}, >
<else, >
<if, >
<(, >
<ID, 1>
</, >
<ID, 2>
<=, >
<NUMBER, INT>
<), >
<{, >
<ID, 2>
<=, >
<ID, 2>
<+, >
<NUMBER, INT>
<;, >
<}, >
<else, >
<if, >
<(, >
<ID, 1>
<*, >
<ID, 2>
<=, >
<NUMBER, INT>
<), >
<{, >
<ID, 1>
<=, >
<ID, 2>
<+, >
<NUMBER, INT>
<;, >
<}, >
<else, >
<if, >
<(, >
<ID, 1>
<-, >
<ID, 2>
<=, >
<NUMBER, INT>
<), >
<{, >
<ID, 1>
<=, >
<ID, 2>
```

```
<+, >
<ID, 2>
<;, >
<}, >
<else, >
<ID, 3>
<=, >
<NUMBER, INT>
<;, >
<}, >
<ID, 0>
<=, >
<STRING, >
<;, >
<print, >
<(, >
<ID, 0>
<), >
<;, >
<fim, >
```

4.2 Código com erro em números de ponto flutuante

```
inicio
    int a; int b;
    read(a);
    read(b);
    b = (a + 2..)
fim
```

Saída obtida:

```
<SMALL_COMMENTARY, >
<inicio, >
<int, >
<ID, 0>
<;, >
<int, >
<ID, 1>
<;, >
<read, >
<(, >
<ID, 0>
<), >
<;, >
<read, >
<(, >
<ID, 2>
<), >
<;, >
<ID, 1>
<=, >
<(, >
<ID, 0>
<+, >
ERRO: numero invalido!
```

4.3 Código com erro de string não fechada

```
inicio
    string s;
    s = "analisador␣lexico;
fim;
```

Saída obtida:

```
<inicio, >
<string, >
<ID, 0>
<;, >
<ID, 1>
<=, >
ERRO: string nao foi fechada!
```

4.4 Código com caracter não reconhecido pela gramática

```
inicio
    int i;
    i = i + $;
fim
```

Saída obtida:

```
<inicio, >
<int, >
<ID, 0>
<;, >
<ID, 1>
<=, >
<ID, 0>
<+, >
ERRO: caracter nao pertercente a linguagem!
```

4.5 Código com comentário longo não fechado

```
inicio
    string s; int a, float b; int i; --[[analisador
    i = 1;
    read(a); read(b); --le os valores de a e b
    s = "finalizado";
    print(s);
fim
```

Saída obtida:

```
<inicio, >
<string, >
<ID, 0>
<;, >
<int, >
<ID, 1>
<,, >
<float, >
<ID, 2>
<;, >
<int, >
<ID, 3>
<;, >
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/c ascii-tabela-ascii-completa-397d >. Acesso em: set. 2025.