



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

Projeto de Compilador

Etapa 1: Análise sintática

Alunos:
Gabriella Ponce
Samiry Sayed

Professora:
Lis Custódio

Novembro
2025

Sumário

1	Introdução	1
1.1	O analisador sintático	1
1.1.1	Tokens, padrões e lexemas	2
2	Descrição teórica	2
2.1	Gramática livre de contexto	2
2.2	Conjunto First	2
2.2.1	Definição	2
2.2.2	Observações	2
2.2.3	Algoritmo V.1	3
2.3	Conjunto Follow	3
2.3.1	Definição	3
2.3.2	Algoritmo V.2	3
2.4	Condições para que uma gramática seja LL(1)	3
2.4.1	Verificação para a GLC a ser analisada	4
2.4.2	Gramática válida para um analisador descendente recursivo	5
3	Estrutura e funcionamento do programa	6
3.1	parser.h	6
3.2	LE_TOKEN()	6
3.3	Tratamento de erros	7
3.4	Construção dos procedimentos	7
3.4.1	<i><programa></i>	7
3.4.2	<i><decls></i>	7
3.4.3	<i><decl></i>	8
3.4.4	<i><tipo></i>	8
3.4.5	<i><comandos></i>	8
3.4.6	<i><comando></i>	8
3.4.7	<i><identificador></i>	9
3.4.8	<i><acao></i>	9
3.4.9	<i><atribuicao></i>	9
3.4.10	<i><chamada></i>	10
3.4.11	<i><args></i>	10
3.4.12	<i><expr_{list}></i>	10
3.4.13	<i><entrada></i>	10
3.4.14	<i><saida></i>	11
3.4.15	<i><if_{stmt}></i>	11
3.4.16	<i><else_{opt}></i>	11
3.4.17	<i><while_{stmt}></i>	12

3.4.18	$\langle bloco \rangle$	12
3.4.19	$\langle expr \rangle$	12
3.4.20	$\langle expr' \rangle$	12
3.4.21	$\langle term \rangle$	13
3.4.22	$\langle term' \rangle$	13
3.4.23	$\langle factor \rangle$	13
3.5	main	13
4	Testes Realizados e Saídas Obtidas	15
4.1	Código válido	15
4.2	ERRO(1)	17
4.3	ERRO(2)	17
4.4	ERRO(3)	17
4.5	ERRO(4)	18
4.6	ERRO(5)	18
4.7	ERRO(6)	18
4.8	ERRO(7)	19
5	Bibliografia	19

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 sintática (em inglês, parsing), tema deste relatório, é o segundo de três estágios da etapa front-end.

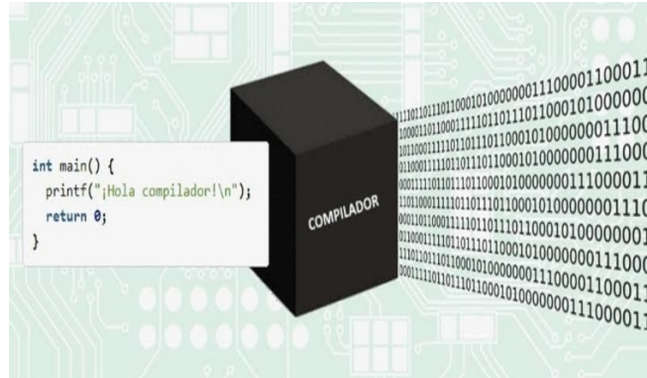


Figura 1: Um compilador traduz um programa escrito em C (linguagem fonte) para um programa escrito em código de máquina (linguagem alvo)

1.1 O analisador sintático

Um analisador sintático é responsável por verificar a ordem dos tokens, emitir mensagens para erros sintáticos e construir a árvore de derivação para a entrada

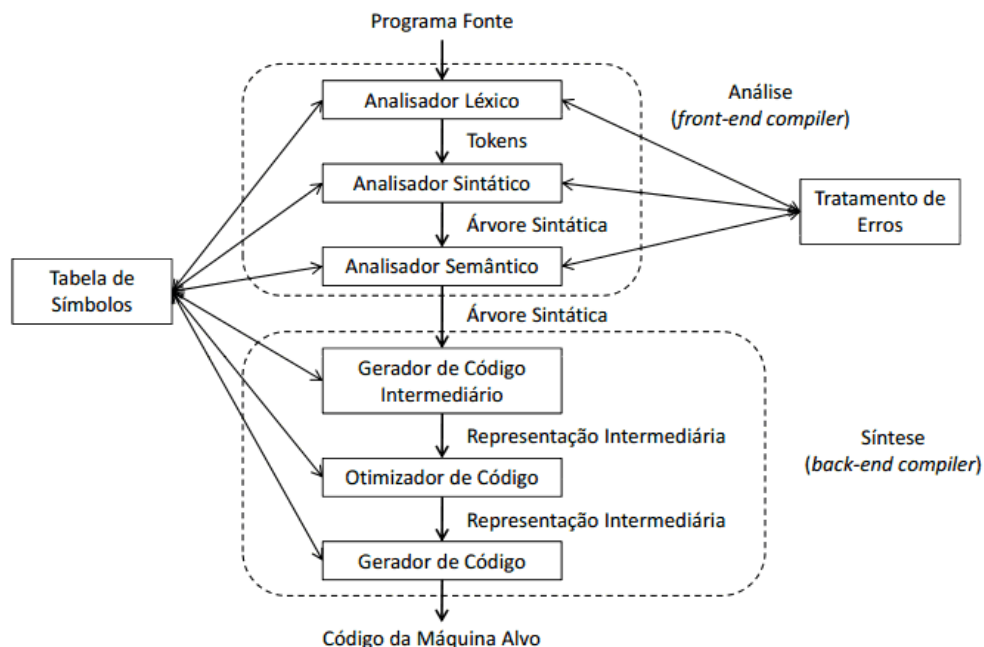


Figura 2: Estrutura de um compilador

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.

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 Conjunto First

2.2.1 Definição

Seja α uma sequência qualquer gerada por G . Definimos como sendo $\text{first}(\alpha)$ o conjunto de símbolos terminais que iniciam α ou sequências derivadas (direta ou indiretamente) de α .

2.2.2 Observações

Se $\alpha = \varepsilon$ ou $\alpha \Rightarrow^* \varepsilon$, então $\varepsilon \in \text{first}(\alpha)$.

2.2.3 Algoritmo V.1

Para calcular $\text{first}(X)$ para todo $X \in V_n \cup V_t$, aplicamos as seguintes regras:

- Se $X \in V_t$, então $\text{first}(X) = \{X\}$;
- Se $X \in V_n \wedge X \rightarrow a\alpha \in P$, então coloque a em $\text{first}(X)$; da mesma forma, se $X \rightarrow \varepsilon \in P$, coloque ε em $\text{first}(X)$;
- Se $X \rightarrow Y_1 Y_2 \dots Y_k \in P$, então, para todo $i \mid Y_1 Y_2 \dots Y_{i-1} \in V_n \wedge \text{first}(Y_j)$, para $j = i, i-1$, contenha ε , adicione $\text{first}(Y_i) - \{\varepsilon\}$ em $\text{first}(X)$.

Em outras palavras:

- Coloque $\text{first}(Y_1)$, exceto ε , em $\text{first}(X)$;
- Se $\varepsilon \in \text{first}(Y_1)$ então coloque $\text{first}(Y_2)$, exceto ε em $\text{first}(X)$;
- Se $\varepsilon \in \text{first}(Y_2)$ então \dots até Y_k ;
- Finalmente, se para todo i (de 1 a k) $\text{first}(Y_i)$ contém ε , então adicione ε em $\text{first}(X)$.

2.3 Conjunto Follow

2.3.1 Definição

Definimos $\text{Follow}(A)$, para todo $A \in V_n$, como sendo o conjunto de símbolos terminais que podem aparecer imediatamente após A em alguma forma sentencial de G .

2.3.2 Algoritmo V.2

Para todo $A \in V_n$, aplique as regras abaixo, até que $\text{Follow}(A)$ esteja completo (isto é, não sofra nenhuma alteração):

- Coloque $\$$ (a marca de final de sentença) em $\text{Follow}(S)$, onde S é o Símbolo Inicial da gramática em questão;
- Se $A \rightarrow \alpha B \beta \in P \wedge \beta \neq \varepsilon$, então adicione $\text{First}(\beta)$, exceto ε , em $\text{Follow}(B)$;
- Se $A \rightarrow \alpha B$ (ou $A \rightarrow \alpha B \beta$, onde $\varepsilon \in \text{First}(\beta)$) $\in P$, então adicione $\text{Follow}(A)$ em $\text{Follow}(B)$.

2.4 Condições para que uma gramática seja LL(1)

Para a implementação de um analisador sintático descendente recursivo, faz-se necessário que a gramática da linguagem pertença à classe das gramáticas LL(1), na qual:

- L (Left-to-right) significa que a cadeia de entrada é lida da esquerda para a direita;
- L (Leftmost) representa a aplicação de uma derivação mais à esquerda;
- 1 indica que a tomada de decisão precisa somente de um token na entrada.

Uma gramática G é LL(1) se, e somente se, para qualquer par de produções distintas

$$A \rightarrow \alpha \mid \beta,$$

as seguintes condições forem satisfeitas:

1. **FIRST**(α) \cap **FIRST**(β) = \emptyset . Essa condição garante que a gramática esteja **fatorada à esquerda**, evitando conflitos de escolha entre produções que começam com o mesmo terminal.

2. No máximo uma das produções deriva ε , isto é,

$$\varepsilon \in \mathbf{FIRST}(\alpha) \text{ e } \varepsilon \in \mathbf{FIRST}(\beta) \quad \text{não podem ocorrer simultaneamente.}$$

Isso evita ambiguidades na escolha da derivação vazia.

3. Se uma produção deriva ε , então o FIRST da outra deve ser disjuncto do FOLLOW de A :

$$\varepsilon \in \mathbf{FIRST}(\alpha) \Rightarrow \mathbf{FIRST}(\beta) \cap \mathbf{FOLLOW}(A) = \emptyset,$$

$$\varepsilon \in \mathbf{FIRST}(\beta) \Rightarrow \mathbf{FIRST}(\alpha) \cap \mathbf{FOLLOW}(A) = \emptyset.$$

Essa condição impede conflitos FIRST/FOLLOW e está relacionada à ausência de **recursão à esquerda**.

Em conjunto, essas três condições garantem que a gramática seja adequada para análise sintática preditiva, ou seja, seja LL(1).

2.4.1 Verificação para a GLC a ser analisada

A linguagem a ser analisada pelo compilador não é LL(1). Os principais problemas que impedem a análise preditiva recursiva descentente são:

- Há **recursão à esquerda** nas seguintes produções:

$$\begin{aligned} \langle expr \rangle &::= \langle expr \rangle + \langle term \rangle \mid \langle expr \rangle - \langle term \rangle \mid \langle term \rangle \\ \langle term \rangle &::= \langle term \rangle * \langle factor \rangle \mid \langle term \rangle / \langle factor \rangle \mid \langle factor \rangle \end{aligned}$$

- A gramática **não está fatorada à esquerda**. Há conflito FIRST nas seguintes produções:

$$\begin{aligned} \langle atribuicao \rangle &::= ID = \langle expr \rangle \\ \langle chamada \rangle &::= ID(\langle args \rangle) \end{aligned}$$

A partir dos problemas detectados, as seguintes transformações são necessárias para que a gramática pertença à classe L(1):

- Remover a recursão à esquerda:

$$\begin{aligned} \langle expr \rangle &::= \langle term \rangle \langle expr' \rangle \\ \langle expr' \rangle &::= + \langle term \rangle \langle expr' \rangle \mid - \langle term \rangle \langle expr' \rangle \mid \varepsilon \\ \langle term \rangle &::= \langle factor \rangle \langle term' \rangle \\ \langle term' \rangle &::= * \langle factor \rangle \langle term' \rangle \mid / \langle factor \rangle \langle term' \rangle \mid \varepsilon \end{aligned}$$

- Fatorar à esquerda:

$$\begin{aligned} \langle identificador \rangle &::= ID \langle acao \rangle \\ \langle acao \rangle &::= \langle atribuicao \rangle \mid \langle chamada \rangle \\ \langle atribuicao \rangle &::= \langle expr \rangle \\ \langle chamada \rangle &::= (\langle args \rangle) \\ \langle comando \rangle &::= \langle identificador \rangle ; \mid \langle entrada \rangle ; \mid \langle saida \rangle ; \\ &\mid \langle if_stmt \rangle \mid \langle while_stmt \rangle \mid \langle bloco \rangle \end{aligned}$$

2.4.2 Gramática válida para um analisador descendente recursivo

Logo, após as mudanças, a gramática a seguir é LL(1):

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

3 Estrutura e funcionamento do programa

Para a implementação do analisador sintático, foram construídos procedimentos em linguagem C para cada uma das variáveis da gramática LL(1). A estrutura do programa foi feita da seguinte maneira:

3.1 parser.h

```
1  #ifndef PARSE_H
2  #define PARSE_H
3
4  #include "lexer.h"
5
6  // Procedimentos construídos
7  void programa();
8  void decls();
9  void decl();
10 void tipo();
11 void comandos();
12 void comando();
13 void identificador();
14 void acao();
15 void atribuicao();
16 void chamada();
17 void args();
18 void expr_list();
19 void entrada();
20 void saida();
21 void if_stmt();
22 void else_opt();
23 void while_stmt();
24 void bloco();
25 void expr();
26 void expr_L();
27 void term();
28 void term_L();
29 void factor();
30
31 //Funcoes auxiliares
32 void LE_TOKEN();
33 void ERRO(int numero_erro);
34
35 // Variavel global para o token atual
36 extern Token token_atual;
37
38 #endif
```

3.2 LE_TOKEN()

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "../include/parser.h"
4  #include "../include/lexer.h"
5
6  Token token_atual;
7
8  void LE_TOKEN() {
9      do {
10         token_atual = proximo_token();
11     } while (token_atual.nome_token == SMALL_COMMENTARY ||
12            token_atual.nome_token == COMMENTARY);
13 }
```

3.3 Tratamento de erros

```
1 void ERRO(int numero_erro) {
2     printf("ERRO_SINT TICO_%d:", numero_erro);
3     switch(numero_erro) {
4         case 1:
5             printf("O programa deve come ar com a palavra 'in cio' e
6                 terminar com a palavra 'fim'. \n");
7             break;
8         case 2:
9             printf("Caractere ',' esperado. \n");
10            break;
11        case 3:
12            printf("Identificador era esperado. \n");
13            break;
14        case 4:
15            printf("Um comando v lido para a linguagem era esperado. \n");
16            break;
17        case 5:
18            printf("Caractere '(' ou '=' esperados. \n");
19            break;
20        case 6:
21            printf("Par nteses foram abertos ou fechados incorretamente.
22                \n");
23            break;
24        case 7:
25            printf("Chaves foram abertas ou fechadas incorretamente. \n");
26            ;
27            break;
28        default:
29            printf("Erro sint tico \n");
30            break;
31    }
32    exit(numero_erro);
33 }
```

3.4 Construção dos procedimentos

Para cada variável da gramática que descreve a linguagem, foi criado um procedimento que analisa se as expressões escritas no código de entrada seguem as regras de produção da gramática, ou seja, se o código está sintaticamente correto.

3.4.1 $\langle \textit{programa} \rangle$

```
1 void programa() {
2     if(token_atual.nome_token == BEGIN){
3         LE_TOKEN();
4         decls();
5         comandos();
6         if(token_atual.nome_token == END) {
7             LE_TOKEN();
8         } else {
9             ERRO(1);
10        }
11    } else {
12        ERRO(1);
13    }
14 }
```

3.4.2 $\langle \textit{decls} \rangle$

```

1 void decls() {
2     if (token_atual.nome_token == TYPE_INT ||
3         token_atual.nome_token == TYPE_FLOAT ||
4         token_atual.nome_token == TYPE_STRING) {
5         decl();
6         decls();
7     } else {
8         return;
9     }
10 }

```

3.4.3 $\langle decl \rangle$

```

1 void decl() {
2     tipo();
3     if(token_atual.nome_token == ID) {
4         LE_TOKEN();
5         if(token_atual.nome_token == SEMICOLON) {
6             LE_TOKEN();
7         } else {
8             ERRO(2);
9         }
10    } else {
11        ERRO(3);
12    }
13 }

```

3.4.4 $\langle tipo \rangle$

```

1 void tipo() {
2     if(token_atual.nome_token == TYPE_INT ||
3         token_atual.nome_token == TYPE_FLOAT ||
4         token_atual.nome_token == TYPE_STRING) {
5         LE_TOKEN();
6     } else {
7         ERRO(4);
8     }
9 }

```

3.4.5 $\langle comandos \rangle$

```

1 void comandos() {
2     if(token_atual.nome_token == ID || token_atual.nome_token == READ ||
3        token_atual.nome_token == PRINT || token_atual.nome_token == IF ||
4        token_atual.nome_token == WHILE || token_atual.nome_token == LEFT_BRACKET) {
5         comando();
6         comandos();
7     } else {
8         return;
9     }
10 }

```

3.4.6 $\langle comando \rangle$

```

1 void comando() {
2     if (token_atual.nome_token == ID) {
3         identificador();
4         if(token_atual.nome_token == SEMICOLON) {

```

```

5         LE_TOKEN();
6     } else {
7         ERRO(2);
8     }
9 } else if(token_atual.nome_token == READ) {
10     entrada();
11     if(token_atual.nome_token == SEMICOLON) {
12         LE_TOKEN();
13     } else {
14         ERRO(2);
15     }
16 } else if(token_atual.nome_token == PRINT) {
17     saida();
18     if(token_atual.nome_token == SEMICOLON) {
19         LE_TOKEN();
20     } else {
21         ERRO(2);
22     }
23 } else if(token_atual.nome_token == IF) {
24     if_stmt();
25 } else if(token_atual.nome_token == WHILE) {
26     while_stmt();
27 } else if(token_atual.nome_token == LEFT_BRACKET) {
28     bloco();
29 } else {
30     ERRO(4);
31 }
32 }

```

3.4.7 <identificador>

```

1 void identificador() {
2     if(token_atual.nome_token == ID) {
3         LE_TOKEN();
4         acao();
5     } else {
6         ERRO(3);
7     }
8 }

```

3.4.8 <acao>

```

1 void acao() {
2     if(token_atual.nome_token == ASSIGN) {
3         atribuicao();
4     } else if(token_atual.nome_token == LEFT_PARENTHESIS) {
5         chamada();
6     } else {
7         ERRO(5);
8     }
9 }

```

3.4.9 <atribuicao>

```

1 void atribuicao() {
2     if(token_atual.nome_token == ASSIGN) {
3         LE_TOKEN();
4         expr();
5     } else {
6         ERRO(5);
7     }

```

```
8 }
```

3.4.10 *⟨chamada⟩*

```
1 void chamada() {
2     if(token_atual.nome_token == LEFT_PARENTHESIS) {
3         LE_TOKEN();
4         args();
5         if(token_atual.nome_token == RIGHT_PARENTHESIS) {
6             LE_TOKEN();
7         } else {
8             ERRO(6);
9         }
10    } else {
11        ERRO(6);
12    }
13 }
```

3.4.11 *⟨args⟩*

```
1 void args() {
2     if(token_atual.nome_token == ID || token_atual.nome_token == NUMBER ||
3        token_atual.nome_token == STRING || token_atual.nome_token == LEFT_PARENTHESIS
4        ||
5        token_atual.nome_token == OP_SUB) {
6         expr_list();
7     } else {
8         return;
9     }
10 }
```

3.4.12 *⟨exprlist⟩*

```
1 void expr_list() {
2     expr();
3     if(token_atual.nome_token == COMMA) {
4         LE_TOKEN();
5         expr_list();
6     }
7 }
```

3.4.13 *⟨entrada⟩*

```
1 void entrada() {
2     if(token_atual.nome_token == READ) {
3         LE_TOKEN();
4         if(token_atual.nome_token == LEFT_PARENTHESIS) {
5             LE_TOKEN();
6             if(token_atual.nome_token == ID) {
7                 LE_TOKEN();
8                 if(token_atual.nome_token == RIGHT_PARENTHESIS) {
9                     LE_TOKEN();
10                } else {
11                    ERRO(6);
12                }
13            } else {
14                ERRO(3);
15            }
16        } else {
```

```

17         ERRO(6);
18     }
19 } else {
20     ERRO(4);
21 }
22 }

```

3.4.14 $\langle \textit{saida} \rangle$

```

1 void saida() {
2     if(token_atual.nome_token == PRINT) {
3         LE_TOKEN();
4         if(token_atual.nome_token == LEFT_PARENTHESIS) {
5             LE_TOKEN();
6             expr();
7             if(token_atual.nome_token == RIGHT_PARENTHESIS) {
8                 LE_TOKEN();
9             } else {
10                 ERRO(6);
11             }
12         } else {
13             ERRO(6);
14         }
15     } else {
16         ERRO(4);
17     }
18 }

```

3.4.15 $\langle \textit{if_stmt} \rangle$

```

1 void if_stmt() {
2     if(token_atual.nome_token == IF) {
3         LE_TOKEN();
4         if(token_atual.nome_token == LEFT_PARENTHESIS) {
5             LE_TOKEN();
6             expr();
7             if(token_atual.nome_token == RIGHT_PARENTHESIS) {
8                 LE_TOKEN();
9                 comando();
10                else_opt();
11            } else {
12                ERRO(6);
13            }
14        } else {
15            ERRO(6);
16        }
17    } else {
18        ERRO(4);
19    }
20 }

```

3.4.16 $\langle \textit{else_opt} \rangle$

```

1 void else_opt() {
2     if(token_atual.nome_token == ELSE) {
3         LE_TOKEN();
4         comando();
5     } else {
6         return;
7     }
8 }

```

3.4.17 $\langle while_stmt \rangle$

```
1 void while_stmt() {
2     if(token_atual.nome_token == WHILE) {
3         LE_TOKEN();
4         if(token_atual.nome_token == LEFT_PARENTHESIS) {
5             LE_TOKEN();
6             expr();
7             if(token_atual.nome_token == RIGHT_PARENTHESIS) {
8                 LE_TOKEN();
9                 comando();
10            } else {
11                ERRO(6);
12            }
13        } else {
14            ERRO(6);
15        }
16    } else {
17        ERRO(4);
18    }
19 }
```

3.4.18 $\langle bloco \rangle$

```
1 void bloco() {
2     if(token_atual.nome_token == LEFT_BRACKET) {
3         LE_TOKEN();
4         comandos();
5         if(token_atual.nome_token == RIGHT_BRACKET) {
6             LE_TOKEN();
7         } else {
8             ERRO(7);
9         }
10    } else {
11        ERRO(7);
12    }
13 }
```

3.4.19 $\langle expr \rangle$

```
1 void expr() {
2     term();
3     expr_L();
4 }
```

3.4.20 $\langle expr' \rangle$

```
1 void expr_L() {
2     if(token_atual.nome_token == OP_SUM || token_atual.nome_token == OP_SUB) {
3         LE_TOKEN();
4         term();
5         expr_L();
6     } else {
7         return;
8     }
9 }
```

3.4.21 $\langle term \rangle$

```
1 void term() {
2     factor();
3     term_L();
4 }
```

3.4.22 $\langle term' \rangle$

```
1 void term_L() {
2     if(token_atual.nome_token == OP_MUL || token_atual.nome_token == OP_DIV) {
3         LE_TOKEN();
4         factor();
5         term_L();
6     } else {
7         return;
8     }
9 }
```

3.4.23 $\langle factor \rangle$

```
1 void factor() {
2     if(token_atual.nome_token == ID) {
3         LE_TOKEN();
4     } else if(token_atual.nome_token == NUMBER) {
5         LE_TOKEN();
6     } else if(token_atual.nome_token == STRING) {
7         LE_TOKEN();
8     } else if(token_atual.nome_token == LEFT_PARENTHESIS) {
9         LE_TOKEN();
10        expr();
11        if(token_atual.nome_token == RIGHT_PARENTHESIS) {
12            LE_TOKEN();
13        } else {
14            ERRO(6);
15        }
16    } else if(token_atual.nome_token == OP_SUB) {
17        LE_TOKEN();
18        factor();
19    } else {
20        ERRO(4);
21    }
22 }
```

3.5 main

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "../include/lexer.h"
4 #include "../include/symbol_table.h"
5 #include "../include/parser.h"
6
7 int main() {
8     inicializar_lexer();
9     code = readFile("../tests/parser/caso_valido.txt");
10
11     LE_TOKEN();
12
13     programa();
14 }
```



```
15     printf("Analise sintatica concluida com SUCESSO!\n");
16
17     free(code);
18     liberar_tabela(&tabela_simbolos);
19     return 0;
20 }
```

4 Testes Realizados e Saídas Obtidas

A seguir, estão os testes realizados, cujas saídas obtidas consistem na impressão dos tokens identificados pelo analisador léxico, e de uma mensagem indicando ou sucesso, para o caso de sintaxe válida, ou o erro sintático correspondente, para o caso de sintaxe inválida.

4.1 Código válido

```
1  inicio
2      int variavel1;
3      float variavel2;
4      string variavel3;
5      int local;
6
7      variavel1 = 10;
8      variavel2 = -5.5;
9      variavel3 = "texto";
10     local = 5;
11
12     read(variavel1);
13     print(variavel1 + 20);
14
15     if (variavel1 + 5) {
16         variavel1 = variavel1 - 1;
17         print("Bloco if");
18     }
19
20     while (variavel2) {
21         variavel2 = variavel2 * 2;
22         print(variavel2);
23     }
24
25     funcao();
26     funcao2(10, "parametro", variavel1 * 2);
27
28     {
29         local = 10;
30         print(local);
31         variavel1 = local;
32     }
33 fim
```

Saída obtida:

```
1  <inicio, >
2  <int, >
3  <ID, 0>
4  <;, >
5  <float, >
6  <ID, 1>
7  <;, >
8  <string, >
9  <ID, 2>
10 <;, >
11 <int, >
12 <ID, 3>
13 <;, >
14 <ID, 0>
15 <=, >
16 <NUMBER, INT>
17 <;, >
18 <ID, 1>
19 <=, >
20 <NUMBER, FLOAT>
21 <;, >
```

```

22 <ID, 2>
23 <=, >
24 <STRING, >
25 <;, >
26 <ID, 3>
27 <=, >
28 <NUMBER, INT>
29 <;, >
30 <read, >
31 <(<, >
32 <ID, 0>
33 <), >
34 <;, >
35 <print, >
36 <(<, >
37 <ID, 0>
38 <+, >
39 <NUMBER, INT>
40 <), >
41 <;, >
42 <if, >
43 <(<, >
44 <ID, 0>
45 <+, >
46 <NUMBER, INT>
47 <), >
48 <{, >
49 <ID, 0>
50 <=, >
51 <ID, 0>
52 <- , >
53 <NUMBER, INT>
54 <;, >
55 <print, >
56 <(<, >
57 <STRING, >
58 <), >
59 <;, >
60 <}, >
61 <while, >
62 <(<, >
63 <ID, 1>
64 <), >
65 <{, >
66 <ID, 1>
67 <=, >
68 <ID, 1>
69 <*, >
70 <NUMBER, INT>
71 <;, >
72 <print, >
73 <(<, >
74 <ID, 1>
75 <), >
76 <;, >
77 <}, >
78 <ID, 4>
79 <(<, >
80 <), >
81 <;, >
82 <ID, 5>
83 <(<, >
84 <NUMBER, INT>
85 <, >
86 <STRING, >
87 <, >

```

```

88 <ID, 0>
89 <*, >
90 <NUMBER, INT>
91 <), >
92 <;, >
93 <{, >
94 <ID, 3>
95 <=, >
96 <NUMBER, INT>
97 <;, >
98 <print, >
99 <(, >
100 <ID, 3>
101 <), >
102 <;, >
103 <ID, 0>
104 <=, >
105 <ID, 3>
106 <;, >
107 <}, >
108 <fim, >
109 Analise sintatica concluida com SUCESSO!

```

4.2 ERRO(1)

```

1  int x;
2  x = 10;
3  fim

```

Saída obtida:

```

1 <int, >
2 ERRO SINTATICO 1: O programa deve começar com a palavra 'inicio' e terminar com a
   palavra 'fim'.

```

4.3 ERRO(2)

```

1 inicio
2     int x
3     x = 10
4 fim

```

Saída obtida:

```

1 <inicio, >
2 <int, >
3 <ID, 0>
4 <ID, 0>
5 ERRO SINTATICO 2: Caractere ';' esperado.

```

4.4 ERRO(3)

```

1 inicio
2     int ;
3     = 10;
4 fim

```

Saída obtida:

```

1 <inicio, >
2 <int, >
3 <;, >
4 ERRO SINTATICO 3: Identificador era esperado.

```

4.5 ERRO(4)

```

1 inicio
2     int x;
3     x = ;
4 fim

```

Saída obtida:

```

1 <inicio, >
2 <int, >
3 <ID, 0>
4 <;, >
5 <ID, 0>
6 <=, >
7 <;, >
8 ERRO SINTATICO 4: Um comando valido para a linguagem era esperado.

```

4.6 ERRO(5)

```

1 inicio
2     int x;
3     x + 10;
4 fim

```

Saída obtida:

```

1 <inicio, >
2 <int, >
3 <ID, 0>
4 <;, >
5 <ID, 0>
6 <+, >
7 ERRO SINTATICO 5: Caractere '(' ou '=' esperados.

```

4.7 ERRO(6)

```

1 inicio
2     int x;
3     read(x;
4     print(x + 10;
5 fim

```

Saída obtida:

```

1 <inicio, >
2 <int, >
3 <ID, 0>
4 <;, >
5 <read, >
6 <(<, >
7 <ID, 0>
8 <;, >
9 ERRO SINTATICO 6: Parenteses foram abertos ou fechados incorretamente.

```

4.8 ERRO(7)

```
1  inicio
2      int x;
3      {
4          x = 10;
5      ;
6  fim
```

Saída obtida:

```
1  <inicio, >
2  <int, >
3  <ID, 0>
4  <;, >
5  <{, >
6  <ID, 0>
7  <=, >
8  <NUMBER, INT>
9  <;, >
10 <;, >
11 ERRO SINTATICO 7: Chaves foram abertas ou fechadas incorretamente.
```

5 Bibliografia

Referências

- [1] MARIA, A.; TOSCANI, S. S.; VELOSO, P. A. S. **Implementação de Linguagens de Programação**. Porto Alegre: Instituto de Informática - UFRGS, [s.d.].
- [2] AHO, A. V.; LAM, M. S.; SETHI, R.; ULLMAN, J. D. **Compiladores: princípios, técnicas e ferramentas**. 2. ed. São Paulo: Pearson Addison Wesley, 2008.
- [3] COOPER, K. D.; TORCZON, L. **Engineering a Compiler**. 2. ed. [S.l.]: Morgan Kaufmann, 2012.
- [4] FERNANDES, H. M. **Código ASCII – Tabela ASCII Completa**. 2023. Disponível em: <https://dev.to/shadowlik/codigo-ascii-tabela-ascii-completa-397d>. Acesso em: set. 2025.
- [5] FURTADO, O. J. V. **Linguagens Formais e Compiladores**. Florianópolis: Departamento de Informática e Estatística, Universidade Federal de Santa Catarina, [s.d.]. Apostila da disciplina.