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

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

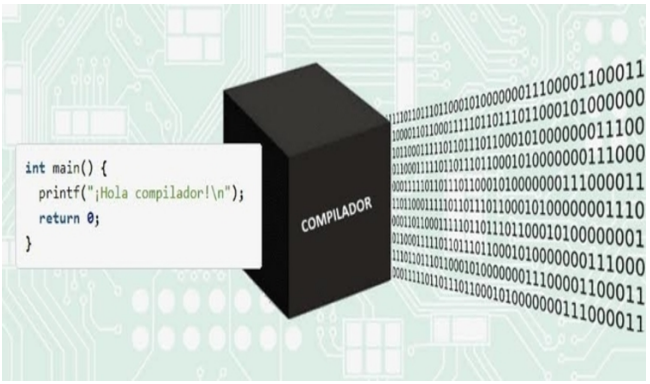

Figura 1: Um compilador traduz um programa escrito em C (linguagem fonte) para um programa escrito em código de máquina (linguagem alvo)

1.1 O analisador sintático

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

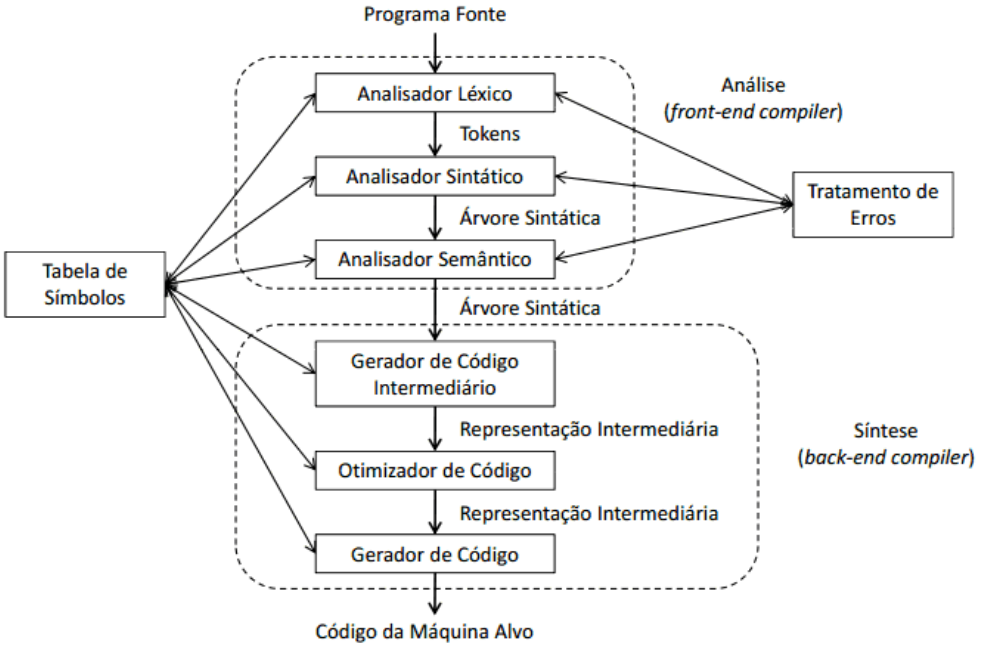

Figura 2: Estrutura de um compilador

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:

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

2.2 Conjunto First

2.2.1 Definição

Seja α uma sequência qualquer gerada por G. Definimos como sendo first(α) 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 first(X) para todo $X \in V_n \cup V_t$, aplicamos as seguintes regras:

a) Se $X \in V_t$, então first(X) = $\{X\}$;

b) Se $X \in V_n \wedge X \to a\alpha \in P$, então coloque a em first(X); da mesma forma, se $X \to \varepsilon \in P$, coloque ε em first(X);

c) Se $X \to Y_1 Y_2 \ldots Y_k \in P$, então, para todo $i \mid Y_1 Y_2 \ldots Y_{i-1} \in V_n \wedge \text{first}(Y_j)$, para $j = i, i-1$, contenha ε, adicione first(Y_i) $- \{\varepsilon\}$ em first(X).

Em outras palavras:

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

2.3 Conjunto Follow

2.3.1 Definição

Definimos 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 Follow(A) esteja completo (isto é, não sofra nenhuma alteração):

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

$$A \to \alpha \mid \beta,$$

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 disjunto 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:

$$\langle expr \rangle ::= \langle expr \rangle + \langle term \rangle \mid \langle expr \rangle - \langle term \rangle \mid \langle term \rangle$$
$$\langle term \rangle ::= \langle term \rangle * \langle factor \rangle \mid \langle term \rangle / \langle factor \rangle \mid \langle factor \rangle$$

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

$$\langle atribuicao \rangle ::= \text{ID} = \langle expr \rangle$$
$$\langle chamada \rangle ::= \text{ID}(\langle args \rangle)$$

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

- Remover a recursão à esquerda:

$$\langle expr \rangle ::= \langle term \rangle \langle expr' \rangle$$
$$\langle expr' \rangle ::= +\langle term \rangle \langle expr' \rangle \mid -\langle term \rangle \langle expr' \rangle \mid \varepsilon$$
$$\langle term \rangle ::= \langle factor \rangle \langle term' \rangle$$
$$\langle term' \rangle ::= *\langle factor \rangle \langle term' \rangle \mid /\langle factor \rangle \langle term' \rangle \mid \varepsilon$$

- Fatorar à esquerda:

$$\langle identificador \rangle ::= \text{ID} \langle acao \rangle$$
$$\langle acao \rangle ::= \langle atribuicao \rangle \mid \langle chamada \rangle$$
$$\langle atribuicao \rangle ::== \langle expr \rangle$$
$$\langle chamada \rangle ::= (\langle args \rangle)$$
$$\langle comando \rangle ::= \langle identificador \rangle; \mid \langle entrada \rangle; \mid \langle saida \rangle;$$
$$\mid \langle if\_stmt \rangle \mid \langle while\_stmt \rangle \mid \langle bloco \rangle$$

2.4.2 Gramática válida para um analisador descendente recursivo

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

$$
\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 identificador \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 identificador \rangle &::= \text{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 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 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 \\
\langle factor \rangle &::= \text{ID} \mid \text{NUMBER} \mid \text{STRING} \mid (\langle expr \rangle) \mid - \langle 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

```c
#ifndef PARSER_H
#define PARSER_H

#include "lexer.h"

// Procedimentos constru dos
void programa();
void decls();
void decl();
void tipo();
void comandos();
void comando();
void identificador();
void acao();
void atribuicao();
void chamada();
void args();
void expr_list();
void entrada();
void saida();
void if_stmt();
void else_opt();
void while_stmt();
void bloco();
void expr();
void expr_L();
void term();
void term_L();
void factor();

//Funcoes auxiliares
void LE_TOKEN();
void ERRO(int numero_erro);

// Variavel global para o token atual
extern Token token_atual;

#endif
```

3.2 LE_TOKEN()

```c
#include <stdio.h>
#include <stdlib.h>
#include "../include/parser.h"
#include "../include/lexer.h"

Token token_atual;

void LE_TOKEN() {
    do {
        token_atual = proximo_token();
    } while (token_atual.nome_token == SMALL_COMMENTARY ||
             token_atual.nome_token == COMMENTARY);
}
```

3.3 Tratamento de erros

```
void ERRO(int numero_erro) {
    printf("ERRO␣SINT TICO␣%d:␣", numero_erro);
        switch(numero_erro) {
                case 1:
                        printf("O␣programa␣deve␣come ar␣com␣a␣palavra␣'in cio'␣e␣
                            terminar␣com␣a␣palavra␣'fim'.␣\n");
                        break;
                case 2:
                        printf("Caractere␣';'␣esperado.␣\n");
                        break;
                case 3:
                        printf("Identificador␣era␣esperado.␣\n");
                        break;
                case 4:
                        printf("Um␣comando␣v lido␣para␣a␣liguagem␣era␣esperado.\n");
                        break;
                case 5:
                        printf("Caractere␣'('␣ou␣'='␣esperados.␣\n");
                        break;
                case 6:
                        printf("Par nteses␣foram␣abertos␣ou␣fechados␣incorretamente.
                            ␣\n");
                        break;
                case 7:
                        printf("Chaves␣foram␣abertas␣ou␣fechadas␣incorretamente.␣\n")
                            ;
                        break;
                default:
                        printf("Erro␣sint tico\n");
                        break;
        }
        exit(numero_erro);
}
```

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 codigo de entrada seguem as regras de produção da gramática, ou seja, se o código está sintaticamente correto.

3.4.1 ⟨*programa*⟩

```
void programa() {
    if(token_atual.nome_token == BEGIN){
        LE_TOKEN();
        decls();
        comandos();
        if(token_atual.nome_token == END) {
            LE_TOKEN();
        } else {
            ERRO(1);
        }
    } else {
        ERRO(1);
    }
}
```

3.4.2 ⟨*decls*⟩

```
void decls() {
    if (token_atual.nome_token == TYPE_INT ||
        token_atual.nome_token == TYPE_FLOAT ||
        token_atual.nome_token == TYPE_STRING) {
        decl();
        decls();
    } else {
        return;
    }
}
```

3.4.3 ⟨*decl*⟩

```
void decl() {
    tipo();
    if(token_atual.nome_token == ID) {
        LE_TOKEN();
        if(token_atual.nome_token == SEMICOLON) {
            LE_TOKEN();
        } else {
            ERRO(2);
        }
    } else {
        ERRO(3);
    }
}
```

3.4.4 ⟨*tipo*⟩

```
void tipo() {
    if(token_atual.nome_token == TYPE_INT ||
       token_atual.nome_token == TYPE_FLOAT ||
       token_atual.nome_token == TYPE_STRING) {
        LE_TOKEN();
    } else {
        ERRO(4);
    }
}
```

3.4.5 ⟨*comandos*⟩

```
void comandos() {
    if(token_atual.nome_token == ID || token_atual.nome_token == READ ||
       token_atual.nome_token == PRINT || token_atual.nome_token == IF ||
       token_atual.nome_token == WHILE || token_atual.nome_token == LEFT_BRACKET) {
        comando();
        comandos();
    } else {
        return;
    }
}
```

3.4.6 ⟨*comando*⟩

```
void comando() {
    if (token_atual.nome_token == ID)  {
        identificador();
        if(token_atual.nome_token == SEMICOLON) {
```

```
            LE_TOKEN();
        } else {
            ERRO(2);
        }
    } else if(token_atual.nome_token == READ) {
        entrada();
        if(token_atual.nome_token == SEMICOLON) {
            LE_TOKEN();
        } else {
            ERRO(2);
        }
    } else if(token_atual.nome_token == PRINT) {
        saida();
        if(token_atual.nome_token == SEMICOLON) {
            LE_TOKEN();
        } else {
            ERRO(2);
        }
    } else if(token_atual.nome_token == IF) {
        if_stmt();
    } else if(token_atual.nome_token == WHILE) {
        while_stmt();
    } else if(token_atual.nome_token == LEFT_BRACKET) {
        bloco();
    } else {
        ERRO(4);
    }
}
```

3.4.7 ⟨*identificador*⟩

```
void identificador() {
    if(token_atual.nome_token == ID) {
        LE_TOKEN();
        acao();
    } else {
        ERRO(3);
    }
}
```

3.4.8 ⟨*acao*⟩

```
void acao() {
    if(token_atual.nome_token == ASSIGN) {
        atribuicao();
    } else if(token_atual.nome_token == LEFT_PARENTHESIS) {
        chamada();
    } else {
        ERRO(5);
    }
}
```

3.4.9 ⟨*atribuicao*⟩

```
void atribuicao() {
    if(token_atual.nome_token == ASSIGN) {
        LE_TOKEN();
        expr();
    } else {
        ERRO(5);
    }
```

```
}
```

3.4.10 ⟨*chamada*⟩

```
void chamada() {
    if(token_atual.nome_token == LEFT_PARENTHESIS) {
        LE_TOKEN();
        args();
        if(token_atual.nome_token == RIGHT_PARENTHESIS) {
            LE_TOKEN();
        } else {
            ERRO(6);
        }
    } else {
        ERRO(6);
    }
}
```

3.4.11 ⟨*args*⟩

```
void args() {
    if(token_atual.nome_token == ID || token_atual.nome_token == NUMBER ||
       token_atual.nome_token == STRING || token_atual.nome_token == LEFT_PARENTHESIS
           ||
       token_atual.nome_token == OP_SUB) {
        expr_list();
    } else {
        return;
    }
}
```

3.4.12 ⟨*expr_list*⟩

```
void expr_list() {
    expr();
    if(token_atual.nome_token == COMMA) {
        LE_TOKEN();
        expr_list();
    }
}
```

3.4.13 ⟨*entrada*⟩

```
void entrada() {
    if(token_atual.nome_token == READ) {
        LE_TOKEN();
        if(token_atual.nome_token == LEFT_PARENTHESIS) {
            LE_TOKEN();
            if(token_atual.nome_token == ID) {
                LE_TOKEN();
                if(token_atual.nome_token == RIGHT_PARENTHESIS) {
                    LE_TOKEN();
                } else {
                    ERRO(6);
                }
            } else {
                ERRO(3);
            }
        } else {
```

```
            ERRO(6);
        }
    } else {
        ERRO(4);
    }
}
```

3.4.14 ⟨*saida*⟩

```
void saida() {
    if(token_atual.nome_token == PRINT) {
        LE_TOKEN();
        if(token_atual.nome_token == LEFT_PARENTHESIS) {
            LE_TOKEN();
            expr();
            if(token_atual.nome_token == RIGHT_PARENTHESIS) {
                LE_TOKEN();
            } else {
                ERRO(6);
            }
        } else {
            ERRO(6);
        }
    } else {
        ERRO(4);
    }
}
```

3.4.15 ⟨*if_stmt*⟩

```
void if_stmt() {
    if(token_atual.nome_token == IF) {
        LE_TOKEN();
        if(token_atual.nome_token == LEFT_PARENTHESIS) {
            LE_TOKEN();
            expr();
            if(token_atual.nome_token == RIGHT_PARENTHESIS) {
                LE_TOKEN();
                comando();
                else_opt();
            } else {
                ERRO(6);
            }
        } else {
            ERRO(6);
        }
    } else {
        ERRO(4);
    }
}
```

3.4.16 ⟨*else_opt*⟩

```
void else_opt() {
    if(token_atual.nome_token == ELSE) {
        LE_TOKEN();
        comando();
    } else {
        return;
    }
}
```

3.4.17 ⟨*while_stmt*⟩

```
void while_stmt() {
    if(token_atual.nome_token == WHILE) {
        LE_TOKEN();
        if(token_atual.nome_token == LEFT_PARENTHESIS) {
            LE_TOKEN();
            expr();
            if(token_atual.nome_token == RIGHT_PARENTHESIS) {
                LE_TOKEN();
                comando();
            } else {
                ERRO(6);
            }
        } else {
            ERRO(6);
        }
    } else {
        ERRO(4);
    }
}
```

3.4.18 ⟨*bloco*⟩

```
void bloco() {
    if(token_atual.nome_token == LEFT_BRACKET) {
        LE_TOKEN();
        comandos();
        if(token_atual.nome_token == RIGHT_BRACKET) {
            LE_TOKEN();
        } else {
            ERRO(7);
        }
    } else {
        ERRO(7);
    }
}
```

3.4.19 ⟨*expr*⟩

```
void expr() {
    term();
    expr_L();
}
```

3.4.20 ⟨*expr'*⟩

```
void expr_L() {
    if(token_atual.nome_token == OP_SUM || token_atual.nome_token == OP_SUB) {
        LE_TOKEN();
        term();
        expr_L();
    } else {
        return;
    }
}
```

3.4.21 $\langle term \rangle$

```
void term() {
    factor();
    term_L();
}
```

3.4.22 $\langle term' \rangle$

```
void term_L() {
    if(token_atual.nome_token == OP_MUL || token_atual.nome_token == OP_DIV) {
        LE_TOKEN();
        factor();
        term_L();
    } else {
        return;
    }
}
```

3.4.23 $\langle factor \rangle$

```
void factor() {
    if(token_atual.nome_token == ID) {
        LE_TOKEN();
    } else if(token_atual.nome_token == NUMBER) {
        LE_TOKEN();
    } else if(token_atual.nome_token == STRING) {
        LE_TOKEN();
    } else if(token_atual.nome_token == LEFT_PARENTHESIS) {
        LE_TOKEN();
        expr();
        if(token_atual.nome_token == RIGHT_PARENTHESIS) {
            LE_TOKEN();
        } else {
            ERRO(6);
        }
    } else if(token_atual.nome_token == OP_SUB) {
        LE_TOKEN();
        factor();
    } else {
        ERRO(4);
    }
}
```

3.5 main

```
#include <stdio.h>
#include <stdlib.h>
#include "../include/lexer.h"
#include "../include/symbol_table.h"
#include "../include/parser.h"

int main() {
    inicializar_lexer();
    code = readFile("./tests/parser/caso_valido.txt");

    LE_TOKEN();

    programa();

```

```
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

```
inicio
    int variavel1;
    float variavel2;
    string variavel3;
    int local;

    variavel1 = 10;
    variavel2 = -5.5;
    variavel3 = "texto";
    local = 5;

    read(variavel1);
    print(variavel1 + 20);

    if (variavel1 + 5) {
        variavel1 = variavel1 - 1;
        print("Bloco if");
    }

    while (variavel2) {
        variavel2 = variavel2 * 2;
        print(variavel2);
    }

    funcao();
    funcao2(10, "parametro", variavel1 * 2);

    {
        local = 10;
        print(local);
        variavel1 = local;
    }
fim
```

Saída obtida:

```
<inicio, >
<int, >
<ID, 0>
<;, >
<float, >
<ID, 1>
<;, >
<string, >
<ID, 2>
<;, >
<int, >
<ID, 3>
<;, >
<ID, 0>
<=, >
<NUMBER, INT>
<;, >
<ID, 1>
<=, >
<NUMBER, FLOAT>
<;, >
```

```
<ID, 2>
<=, >
<STRING, >
<;, >
<ID, 3>
<=, >
<NUMBER, INT>
<;, >
<read, >
<(, >
<ID, 0>
<), >
<;, >
<print, >
<(, >
<ID, 0>
<+, >
<NUMBER, INT>
<), >
<;, >
<if, >
<(, >
<ID, 0>
<+, >
<NUMBER, INT>
<), >
<{, >
<ID, 0>
<=, >
<ID, 0>
<-, >
<NUMBER, INT>
<;, >
<print, >
<(, >
<STRING, >
<), >
<;, >
<}, >
<while, >
<(, >
<ID, 1>
<), >
<{, >
<ID, 1>
<=, >
<ID, 1>
<*, >
<NUMBER, INT>
<;, >
<print, >
<(, >
<ID, 1>
<), >
<;, >
<}, >
<ID, 4>
<(, >
<), >
<;, >
<ID, 5>
<(, >
<NUMBER, INT>
<,, >
<STRING, >
<,, >
```

```
<ID, 0>
<*, >
<NUMBER, INT>
<), >
<;, >
<{, >
<ID, 3>
<=, >
<NUMBER, INT>
<;, >
<print, >
<(, >
<ID, 3>
<), >
<;, >
<ID, 0>
<=, >
<ID, 3>
<;, >
<}, >
<fim, >
Analise sintatica concluida com SUCESSO!
```

4.2 ERRO(1)

```
int x;
x = 10;
fim
```

Saída obtida:

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

4.3 ERRO(2)

```
inicio
    int x
    x = 10
fim
```

Saída obtida:

```
<inicio, >
<int, >
<ID, 0>
<ID, 0>
ERRO SINTATICO 2: Caractere ';' esperado.
```

4.4 ERRO(3)

```
inicio
    int ;
    = 10;
fim
```

Saída obtida:

```
<inicio, >
<int, >
<;, >
ERRO SINTATICO 3: Identificador era esperado.
```

4.5 ERRO(4)

```
inicio
    int x;
    x = ;
fim
```

Saída obtida:

```
<inicio, >
<int, >
<ID, 0>
<;, >
<ID, 0>
<=, >
<;, >
ERRO SINTATICO 4: Um comando valido para a liguagem era esperado.
```

4.6 ERRO(5)

```
inicio
    int x;
    x + 10;
fim
```

Saída obtida:

```
<inicio, >
<int, >
<ID, 0>
<;, >
<ID, 0>
<+, >
ERRO SINTATICO 5: Caractere '(' ou '=' esperados.
```

4.7 ERRO(6)

```
inicio
    int x;
    read(x;
    print(x + 10;
fim
```

Saída obtida:

```
<inicio, >
<int, >
<ID, 0>
<;, >
<read, >
<(, >
<ID, 0>
<;, >
ERRO SINTATICO 6: Parenteses foram abertos ou fechados incorretamente.
```

4.8 ERRO(7)

```
inicio
    int x;
    {
        x = 10;
    ;
fim
```

Saída obtida:

```
<inicio, >
<int, >
<ID, 0>
<;, >
<{, >
<ID, 0>
<=, >
<NUMBER, INT>
<;, >
<;, >
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.