1. INTRODUÇÃO

1.1. OBJETIVO

Este documento descreve o funcionamento e a implementação de um compilador para a linguagem MicroPascal (μ-Pascal) e suas etapas de compilação. O objetivo do compilador é transformar o código-fonte (arquivo de entrada) em um código objeto e executável.

1.2. ESCOPO

O analisador léxico cobre todos os aspectos da linguagem MicroPascal, incluindo palavras-chave, identificadores, literais, operadores e delimitadores presentes na gramática da linguagem.

O analisador sintático verifica todos os possíveis erros de sintaxe presente no código de acordo com as regras gramaticais da linguagem, analisando se a sequência de tokens dado pelo analisador léxico é uma frase válida de acordo com as regras gramaticais.

2. ESTRUTURA DO ANALISADOR LÉXICO

2.1. DEFINIÇÃO

Os tokens são as unidades básicas da linguagem. Abaixo estão as definições dos principais tokens reconhecidos pelo analisador léxico:

- Palavras reservadas: **`program`**, **`begin`**, **`end`**, **`if`**, **`then`**, **`else`**, **`while`**, **`do`**, **`var`**, etc.
- Identificadores: sequências de letras e dígitos, começando com uma letra;
- Literais: números inteiros e reais;
- Operadores: `+`, `-`, `*`, `/`, `:=`, `<`, `>`, etc.
- Símbolos: `;`, `,`, `(`, `)`, `[`, `]`, `{`, `}`, etc.

2.2. ESTRUTURAS

O analisador léxico utiliza uma tabela de símbolos para armazenar informações sobre identificadores e palavras-chave. A tabela de símbolos é implementada como uma tabela hash.

```
typedef enum TokenType
{
    RESERVED_WORD,
    RESERVED_TYPE,
    RESERVED_OPERATOR,
    IDENTIFIER,
    OPERATOR,
    SYMBOL,
    NUMBER,
    STRING,
    END_OF_FILE,
    ERROR
} TokenType;
```

— Esta enumeração representa os diferentes tipos de tokens que podem ser identificados durante a fase de análise lexical. Cada tipo de token corresponde a uma categoria específica de elementos lexicais.

```
typedef struct Token
{
    char *name;
    char *word;
    int row;
    int column;
    TokenType type;
    struct Token *next;
} Token;
```

— Esta estrutura é usada para armazenar informações sobre um token identificado durante a análise lexical.

```
typedef struct Entry
{
    char *key;
    Token *token;
    struct Entry *next;
} Entry;
```

— Esta estrutura é usada para armazenar um par chave-valor onde a chave é uma string e o valor é um ponteiro para um Token. Cada entrada também contém um ponteiro para a próxima entrada, permitindo a criação de uma lista encadeada.

```
typedef struct
{
    Entry **entries;
    int entryCount;
} Table;
```

— Esta estrutura é usada para armazenar uma coleção de entradas, onde cada entrada é um ponteiro. As entradas são armazenadas em um array alocado dinamicamente.

3. ESTRUTURA DO ANALISADOR SINTÁTICO

3.1. DEFINIÇÃO

O analisador sintático utilizará da lista de tokens fornecidos pelo analisador léxico para realizar a análise sintática de acordo com as regras gramaticais da linguagem. A lista de tokens estará em uma lista encadeada encontrada na `struct Entry` onde cada elemento pode ser acessado utilizando o membro `struct Entry* next;`

3.2. ESTRUTURAS

```
typedef struct ASTNode
{
    int type;
    char *value;
    struct ASTNode *left;
```

```
    struct ASTNode *right;
} ASTNode;
```

— Esta estrutura é utilizada para representar nós em uma AST (Abstract Syntax Tree), que é uma representação em árvore da estrutura sintática abstrata do código-fonte.

4. IMPLEMENTAÇÃO

4.1. FLUXO DE EXECUÇÃO

1. **Leitura do código-fonte**: O código-fonte é lido;
2. **Reconhecimento de tokens**: Durante a leitura, é feita uma análise de cada caractere, e é feito um reconhecimento de caracteres conhecidos e desconhecidos, palavras reservadas, identificados, números, operadores, símbolos, etc.
3. **Armazenamento de tokens**: Os tokens são armazenados em uma lista para posterior análise sintática e é gerado um output de extensão **`.lex`** com os tokens reconhecidos;
4. **Análise gramatical**: É realizada uma análise sintática com base na lista de tokens fornecidos pelo analisador léxico;

4.2. FUNÇÕES

```
int main(int argc, char** argv)
```

`@brief:` Ponto de entrada principal para o programa de análise lexical;
`@returns:` Retorna 0 em caso de sucesso, ou 1 em caso de erro;

Lê um arquivo Pascal e realiza análise lexical nele.

- Parâmetros:
  `@param argc:` Inteiro com o número de argumentos da linha de comando;
  `@param argv:` O array de argumentos da linha de comando;
- Argumentos:
  `--help:` Exibe os possíveis argumentos que o usuário pode acessar;
  `--file <file>:` Argumento em que é necessário passar o caminho do arquivo de entrada, este mesmo deverá ser de extensão .pas;

```
Token* lexerAnalysis(Table* table)
```

`@brief:` Realiza análise lexical na entrada e gera tokens;
`@returs:` Um ponteiro para o token gerado;

Lê caracteres da entrada e identifica diferentes tipos de tokens, como espaços, valores numéricos, valores alfanuméricos, símbolos, operadores, palavras reservadas, identificadores, valores inteiros, valores reais, operadores relacionais, operadores de atribuição e strings. Também lida com erros lexicais e condições de fim de arquivo.

- Parâmetros:

@param table: Um ponteiro para a tabela de símbolos onde os tokens serão inseridos.

```
static void addWord(char** word, int* size, const char ch)
```

@brief: Adiciona um caractere a um array de palavras alocado dinamicamente.

Adiciona um caractere ao final de um array de palavras alocado dinamicamente, redimensionando o array se necessário.

- Parâmetros:

  @param word: Um ponteiro para o array de palavras alocado dinamicamente.
  @param size: Um ponteiro para o tamanho atual do array de palavras.
  @param ch: O caractere a ser adicionado ao array de palavras.

```
Table* initTable()
```

@brief: Inicializa uma nova estrutura Table;
@returns: Um ponteiro para a estrutura Table recém-inicializada;

Esta função aloca memória para uma estrutura Table e suas entradas. Ela inicializa cada entrada como **NULL**.

```
static void insertTable(Table* table, char* key, Token* token)
```

@brief: Insere uma nova entrada na tabela hash;

Cria uma entrada com a chave e o token fornecidos, calcula o índice hash para a chave, e insere a entrada na tabela hash no índice calculado.

- Parâmetros:
  @param table: Um ponteiro para a tabela hash onde a entrada será inserida.
  @param key: Uma string representando a chave para a nova entrada.
  @param token: Um ponteiro para o token associado à chave.

```
Token* searchTable(Table* table, char* key)
```

@brief: Procura um token na tabela hash usando a chave fornecida;
@returns: Um ponteiro para o token associado à chave, ou **NULL** se a chave não for encontrada;

Procura na tabela hash uma entrada com a chave especificada. Se a chave for encontrada, o token associado é retornado. Se a chave não for encontrada, a função retorna NULL.

- Parâmetros:
  @param table: Um ponteiro para a tabela hash a ser pesquisada.
  @param key: A chave a ser pesquisada na tabela hash.

```
static Token* createToken(TokenType type, char* name, char* word, int row, int column)
```

@brief: Cria um novo **Token** com os atributos especificados;
@returns: Um ponteiro para o **Token** recém-criado;

- Parâmetros:
  @param type: O tipo do token.
  @param name: O nome do token.
  @param word: A palavra associada ao token.
  @param word: O número da linha onde o token é encontrado.
  @param column: O número da coluna onde o token é encontrado.

```
static unsigned int hash(char* key, int tableSize)
```

@brief: Calcula um valor hash para uma chave fornecida;
@returns: O valor hash calculado modulado pelo tamanho da tabela;

Esta função recebe uma chave de string e calcula seu valor hash usando uma função hash simples. O valor hash é então modulado pelo tamanho da tabela para garantir que ele se encaixe dentro dos limites da tabela hash.

- Parâmetros:
  @param key: A chave de string a ser hash;
  @param tableSize: O tamanho da tabela hash;

```
static void saveFile(Token* token)
```

@brief: Salva as informações do token no arquivo de saída;

Esta função escreve os detalhes de um token fornecido no arquivo de saída em um formato específico.

- Parâmetros:
  @param token: Um ponteiro para a estrutura Token contendo as informações do token.

5. TESTES

5.1. LÉXICO

Arquivos sem erros léxicos:

1. T001: Atribuição Simples;

```
program T_001;

var i: integer;

begin
    i := 5;
end.
```

2. T002: Condicional

```
program T_002;

var x, y: integer;

begin
    if x > y then
        x := y;
end.
```

3. T003: Valores reais e outras palavras reservadas:

```
program T_003;

var x, y, total: real;

begin
    x := 1.0;
    y := 5.0;
    total := 0.0;

    while x <= y do
    begin
        total := total + x;
        x := x + 1.0;
```

```
    end
end.
```

Arquivos com erros léxicos:

1. T004: Caractere desconhecido:

```
program T_004;

var x: integer;

begin
    x := 12@; // LexicalError: Unknown character:
'@' at 6:12
end.
```

2. T005: String não fechada:

```
program T_005;

var
    s: string;
    i: integer;

begin
    i := 10;
    s := "teste; // LexicalError: String not closed
at 9:17
end.
```

3. T006: Caractere inválido:

```
program T_006;

var
    $i: integer; // LexicalError: Unknown character: '$' at
4:5

begin
    $i := 10 * 5;
end.
```

4. T007: Identificador inválido:

```
program T_007;

var 123x: integer; // Lexical error: invalid identifier
'123x' at 3:8

begin
    123x := 12;
```

```
end.
```

5.2. SINTÁTICO

Arquivos sem erros sintáticos:

4. T011: Calcular média;

```
program T_011;

var a, b, c, average: integer;

begin
    a := 10;
    b := 7;
    c := 6;
    average := (a + b + c) / 3;

    if average > 7 then
        // do something
    else
        // do something
end.
```

5. T012: Condicionais, atribuição e laço de repetição:

```
program T_012;

var x, y: integer;
var z: real;

begin
   x := 10;
   y := 20;
   z := x + y * 2.5;

   if x > y then
      x := x - 1;
   else
      y := y + 1;

   while z <= 100 do
        begin
            z := z * 1.5;
            x := x + 2;
        end
end.
```

6. T013: Laço de repetição:

```
program T_013;

var
    a: integer;

begin
    while (a < 10) do
        begin
            a := a + 1;
        end
end.
```

Arquivos com erros sintáticos:

1. T008: Token esperado:

```
program T_008;

var x: integer;

begin
    x := 10;

    if (x > 5 then // Syntax error: expected ')'
at 8:18
        x := 5;
    else
        x := 0;
    end;
end.
```

2. T009: Token não esperado:

```
program T_009;

var a, b, c: integer;

begin
    c := (a + b)); // Syntax error: Unexpected
token ')' at 6:17
end.
```

3. T010:

```
program T_010;

var a, b, c: integer;
```

```
begin
    a := 10;
    b := 20;
    c := (a + b) * (a - b) / (a + 1) + (b * 2 -
a / 2) * (a + 3)); // Syntax error: Unexpected
token ')' at 8:63
end.
```

6. OUTPUTS

1. T001:

```
<0, Reserved-word, 'program'> : <1, 7>
<3, Identifier, 'T_001'> : <1, 12>
<5, Symbol, ';'> : <1, 13>
<0, Reserved-word, 'var'> : <3, 3>
<3, Identifier, 'a'> : <4, 5>
<5, Symbol, ':'> : <4, 6>
<1, Reserved-type, 'integer'> : <4, 14>
<5, Symbol, ';'> : <4, 15>
<0, Reserved-word, 'begin'> : <5, 5>
<3, Identifier, 'a'> : <6, 5>
<4, Assignment Operator, ':='> : <6, 8>
<1, Integer value, '5'> : <6, 10>
<5, Symbol, ';'> : <6, 11>
<0, Reserved-word, 'end'> : <7, 3>
<5, Symbol, '.'> : <7, 4>
```

2. T002:

```
<0, Reserved-word, 'program'> : <1, 7>
<3, Identifier, 'T_002'> : <1, 12>
<5, Symbol, ';'> : <1, 13>
<0, Reserved-word, 'var'> : <3, 3>
<3, Identifier, 'x'> : <3, 5>
<5, Symbol, ','> : <3, 6>
<3, Identifier, 'y'> : <3, 8>
<5, Symbol, ':'> : <3, 9>
<1, Reserved-type, 'integer'> : <3, 17>
<5, Symbol, ';'> : <3, 18>
<0, Reserved-word, 'begin'> : <5, 5>
<0, Reserved-word, 'if'> : <6, 6>
<3, Identifier, 'x'> : <6, 8>
<4, Relational Operator, '>'> : <6, 10>
<3, Identifier, 'y'> : <6, 12>
<0, Reserved-word, 'then'> : <6, 17>
<3, Identifier, 'x'> : <7, 9>
<4, Assignment Operator, ':='> : <7, 12>
<3, Identifier, 'y'> : <7, 14>
<5, Symbol, ';'> : <7, 15>
<0, Reserved-word, 'end'> : <8, 3>
<5, Symbol, '.'> : <8, 4>
```

3. T003:

```
<0, Reserved-word, 'program'> : <1, 7>
<3, Identifier, 'T_003'> : <1, 13>
<5, Symbol, ';'> : <1, 14>
<0, Reserved-word, 'var'> : <3, 3>
<3, Identifier, 'x'> : <4, 5>
<5, Symbol, ','> : <4, 6>
<3, Identifier, 'y'> : <4, 8>
<5, Symbol, ','> : <4, 9>
<3, Identifier, 'total'> : <4, 15>
<5, Symbol, ':'> : <4, 16>
<1, Reserved-type, 'real'> : <4, 21>
<5, Symbol, ';'> : <4, 22>
<0, Reserved-word, 'begin'> : <6, 5>
<3, Identifier, 'x'> : <7, 5>
<4, Assignment Operator, ':='> : <7, 8>
<1, Real value, '1.0'> : <7, 12>
<5, Symbol, ';'> : <7, 13>
<3, Identifier, 'y'> : <8, 5>
<4, Assignment Operator, ':='> : <8, 8>
<1, Real value, '5.0'> : <8, 12>
<5, Symbol, ';'> : <8, 13>
<3, Identifier, 'total'> : <9, 9>
<4, Assignment Operator, ':='> : <9, 12>
<1, Real value, '0.0'> : <9, 16>
<5, Symbol, ';'> : <9, 17>
<0, Reserved-word, 'while'> : <11, 9>
<3, Identifier, 'x'> : <11, 11>
<4, Relational Operator, '<='> : <11, 14>
<3, Identifier, 'y'> : <11, 16>
<0, Reserved-word, 'do'> : <11, 19>
<0, Reserved-word, 'begin'> : <12, 9>
<3, Identifier, 'total'> : <13, 13>
<4, Assignment Operator, ':='> : <13, 16>
<3, Identifier, 'total'> : <13, 22>
<4, Binary Arithmetic Operator, '+'> : <13, 24>
<3, Identifier, 'x'> : <13, 26>
<5, Symbol, ';'> : <13, 27>
<3, Identifier, 'x'> : <14, 9>
<4, Assignment Operator, ':='> : <14, 12>
<3, Identifier, 'x'> : <14, 14>
<4, Binary Arithmetic Operator, '+'> : <14, 16>
<1, Real value, '1.0'> : <14, 20>
<5, Symbol, ';'> : <14, 21>
<0, Reserved-word, 'end'> : <15, 7>
<0, Reserved-word, 'end'> : <16, 3>
<5, Symbol, '.'> : <16, 4>
```

4. T004:

```
<0, Reserved-word, 'program'> : <1, 7>
<3, Identifier, 'T_004'> : <1, 13>
<5, Symbol, ';'> : <1, 14>
```

```
<0, Reserved-word, 'var'> : <3, 3>
<3, Identifier, 'x'> : <3, 5>
<5, Symbol, ':'> : <3, 6>
<1, Reserved-type, 'integer'> : <3, 14>
<5, Symbol, ';'> : <3, 15>
<0, Reserved-word, 'begin'> : <5, 5>
<3, Identifier, 'x'> : <6, 5>
<4, Assignment Operator, ':='> : <6, 8>
<1, Integer value, '12'> : <6, 11>
LexicalError: Unknown character: '@' at 6:12
```

5. T005:

```
<0, Reserved-word, 'program'> : <1, 7>
<3, Identifier, 'T_005'> : <1, 13>
<5, Symbol, ';'> : <1, 14>
<0, Reserved-word, 'var'> : <3, 3>
<3, Identifier, 's'> : <4, 5>
<5, Symbol, ':'> : <4, 6>
<1, Reserved-type, 'string'> : <4, 13>
<5, Symbol, ';'> : <4, 14>
<3, Identifier, 'i'> : <5, 5>
<5, Symbol, ':'> : <5, 6>
<1, Reserved-type, 'integer'> : <5, 14>
<5, Symbol, ';'> : <5, 15>
<0, Reserved-word, 'begin'> : <7, 5>
<3, Identifier, 'i'> : <8, 5>
<4, Assignment Operator, ':='> : <8, 8>
<1, Integer value, '10'> : <8, 11>
<5, Symbol, ';'> : <8, 12>
<3, Identifier, 's'> : <9, 5>
<4, Assignment Operator, ':='> : <9, 8>
LexicalError: String not closed at 9:17
```

6. T006:

```
<0, Reserved-word, 'program'> : <1, 7>
<3, Identifier, 'T_006'> : <1, 13>
<5, Symbol, ';'> : <1, 14>
<0, Reserved-word, 'var'> : <3, 3>
LexicalError: Unknown character: '$' at 4:5
```

7. T007:

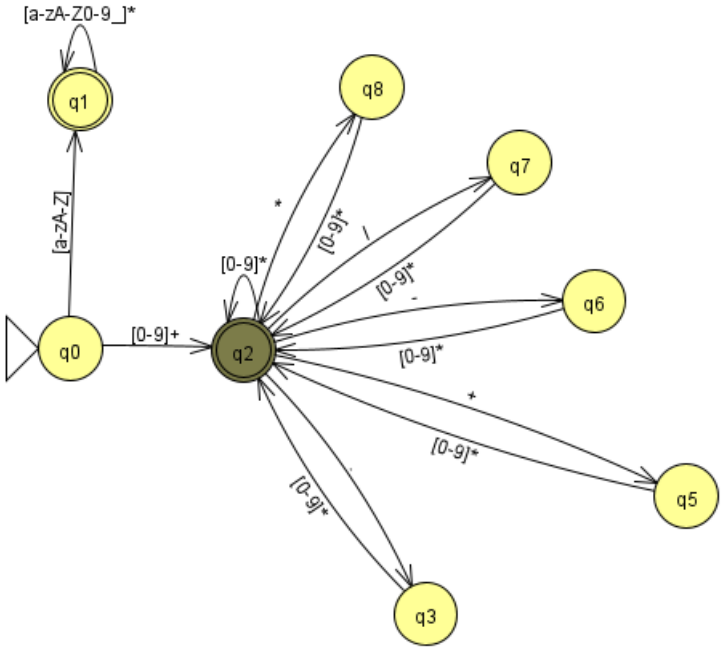
```
<0, Reserved-word, 'program'> : <1, 7>
<3, Identifier, 'T_007'> : <1, 13>
<5, Symbol, ';'> : <1, 14>
<0, Reserved-word, 'var'> : <3, 3>
Lexical error: invalid identifier '123x' at 3:8
```

7. DIAGRAMA AFD

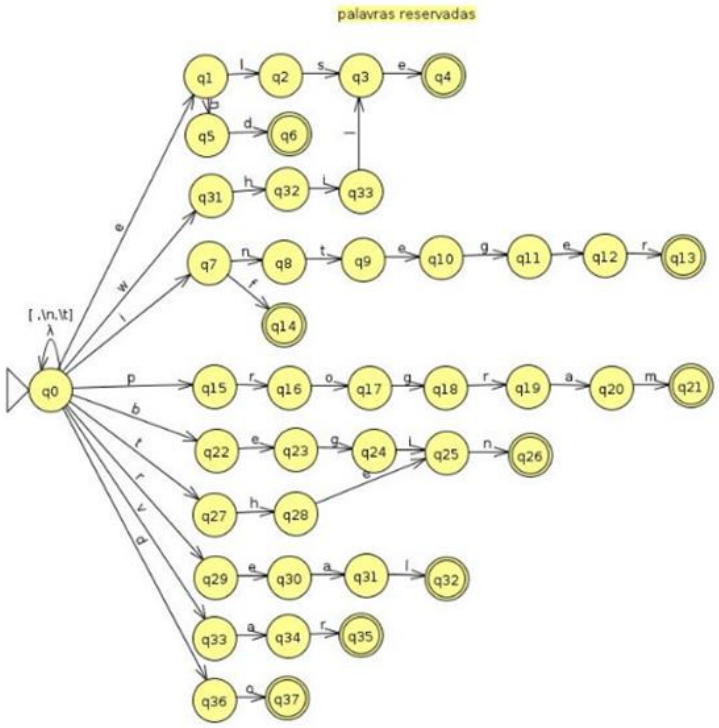
O diagrama apresentado a seguir representa o Autômato Finito Determinístico (AFD) utilizado para a identificação de tokens na linguagem MicroPascal (μ-Pascal). Este diagrama detalha o processo pelo qual o analisador léxico reconhece e categoriza diferentes tokens, baseando-se nos estados e transições definidos. Cada estado do AFD corresponde a uma etapa específica do reconhecimento lexical, enquanto as transições entre estados são determinadas pelos caracteres de entrada. Este mecanismo é fundamental para a correta análise e interpretação do código fonte, garantindo que cada token seja identificado de acordo com as regras sintáticas da linguagem.

Para a criação deste diagrama, foi utilizada a ferramenta JFLAP, que é amplamente reconhecida no meio acadêmico por sua eficácia na modelagem e visualização de autômatos finitos e outras estruturas formais. A utilização do JFLAP permitiu uma representação clara e precisa do AFD, facilitando a compreensão e a validação do processo de análise léxica.

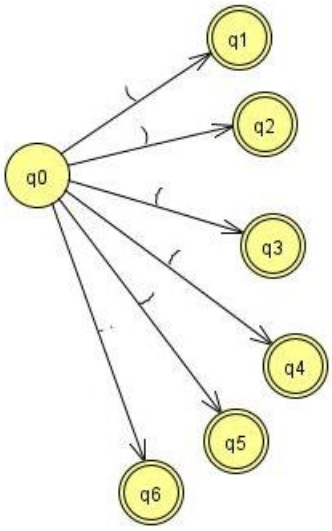
7.1. IDENTIFICADORES E OPERAÇÕES COM NÚMEROS



7.2. PALAVRAS RESERVADAS



7.3. SÍMBOLOS



8. GRAMÁTICA

$$[Program] \rightarrow program\ [Identifier]\ ;\ [Block]\ .$$

$$[Block] \rightarrow [OptionalVarDeclPart]\ [CompoundCommand]$$

$$[OptionalVarDeclPart] \rightarrow \epsilon \mid [VarDeclPart]$$

$$[VarDeclPart] \rightarrow var\ [VarDecl]\ \{\ ;\ [VarDecl]\}\ ;$$

$$[VarDecl] \rightarrow [IdentifierList]\ :\ [Type]$$

$$[IdentifierList] \rightarrow [Identifier]\ \{\ ,\ [Identifier]\}$$

$$[Type] \rightarrow integer \mid real \mid sfd$$

$$[CompoundCommand] \rightarrow begin\ [OptionalVarDeclPart]\ [Command]\ ;\ \{\ [Command]\ ;\}\ end$$

$$[Command] \rightarrow \begin{cases} [Assignment] \\ [CompoundCommand] \\ [ConditionalCommand] \\ [RepetitiveCommand] \end{cases}$$

$$[Assignment] \rightarrow [Variable]\ :=\ [Expression]$$

$$[ConditionalCommand] \rightarrow if\ [Expression]\ then\ [Command]\ [OptionalElse]$$

$$[OptionalElse] \rightarrow \begin{cases} else\ [Command] \\ \epsilon \end{cases}$$

$$[RepetitiveCommand] \rightarrow while\ [Expression]\ do\ [Command]$$

$$[Expression] \rightarrow [SimpleExpression]\ [OptionalRelation]$$

$$[OptionalRelation] \rightarrow \begin{cases} [Relation]\ [SimpleExpression] \\ \epsilon \end{cases}$$

$$[Relation] \rightarrow =\ \mid <>\mid\ <\ \mid <=\mid\ >=\ \mid\ >$$

$$[SimpleExpression] \rightarrow [OptionalSign]\ [Term]\ \{\ [AddOperator]\ [Term]\}$$

$$[OptionalSign] \rightarrow +\mid -\mid \epsilon$$

$$[AddOperator] \rightarrow +\mid\ -$$

$$[Term] \rightarrow [Factor]\ \{\ [MultiplicationOperator]\ [Factor]\}$$

$$[MultiplicationOperator] \rightarrow *\ \mid /$$

$$[Factor] \rightarrow \begin{cases} [Variable] \\ [Number] \\ (\ [Expression]\ ) \end{cases}$$

$$[Variable] \rightarrow [Identifier]$$

$$[Identifier] \rightarrow ident$$

$$[Number] \rightarrow int\_lit \mid real\_lit$$

9. REQUISITOS

É de requisito para este programa o compilador GCC (GNU Compiler Collection) ou equivalente para compilar os arquivos C.

Utilizando o GCC, a compilação é feita utilizando a seguinte linha de comando:

```
gcc ./src/lexer/lexer.c ./src/parser/parser.c ./src/main.c -o main.exe
```

10. EXECUÇÃO

A execução do programa pode ser feita executando o arquivo de extensão **.exe** gerado pelo compilador. Como dito anteriormente o programa possui alguns argumentos, com o argumento **--help** você poderá ter uma lista de argumentos aceitos:

```
./main.exe --help
```

- Argumentos:
  --help ou -h
  --file <arquivo> ou -f <arquivo>

No argumento tipo **--file** ou **-f**, o usuário deverá passar o caminho relativo ou absoluto para um arquivo de extensão .pas, qualquer outra extensão o programa retornará informando que o argumento passado está sendo utilizado de maneira indevida. Um exemplo do uso correto desde argumento seria:

- **./main.exe --file ./file.pas** (Passando um caminho relativo, supondo que haja um arquivo chamado 'file.pas' no diretório de onde o programa esteja)
- **./main.exe -f C:\Users\Public\Documents\file.pas** (Passando um caminho absoluto)