Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Unidade Curricular de Processamento de Linguagens

Ano Letivo de 2024/2025

Compilador de Pascal Grupo 9 - Paz, Kcal e Saúde

Diogo Barros(a100600) Pedro Silva(a100745) Diogo Costa(a100751)

26 de maio de 2025

PL

Índice

1 Introdução **4**

2 Objetivos **5**

3 Análise Léxica **6**
- 3.1 Símbolos . . . 6
- 3.2 Operadores Aritméticos . . . 7
- 3.3 Operadores de Comparação . . . 7
- 3.4 Tokens de Estrutura . . . 7
- 3.5 Tokens de Controlo de Fluxos . . . 8
- 3.6 Tokens de Operadores Lógicos . . . 9
- 3.7 Tokens de Operadores Aritméticos Inteiros . . . 9
- 3.8 Tokens para Funções, Declarações e Arrays . . . 9
- 3.9 Tokens para Tipos de Dados . . . 10
- 3.10 Tokens Numéricos, Textuais e Simbólicos . . . 11
- 3.11 Tokens para Comentários e Quebras de Linha . . . 11

4 Análise Sintática **12**
- 4.1 Símbolos presentes na Gramática . . . 12
- 4.2 Análise de Regras da Gramática . . . 13

5 Análise Semântica **21**
- 5.1 Variáveis . . . 22
- 5.2 Identifiers . . . 23
- 5.3 Body . . . 23
- 5.4 Statements . . . 24
- 5.5 Assignments . . . 25
- 5.6 Expressions . . . 26
  - 5.6.1 Multiplicação e Divisão (maior prioridade) . . . 27
  - 5.6.2 Soma, Subtração, Div, Mod, Intervalos . . . 27
  - 5.6.3 Expressões Booleanas (última prioridade) . . . 27
- 5.7 Operadores Lógicos . . . 28
- 5.8 READLN . . . 28
- 5.9 WRITELN . . . 29
- 5.10 Condições (*if*) . . . 31
- 5.11 Ciclos . . . 32
  - 5.11.1 Ciclo For . . . 33

5.11.2 Ciclo While . . . 34
5.11.3 Ciclo de Repeat ... Until . . . 35
5.12 Functions e Procedures . . . 35
5.13 Acesso a Arrays e Strings . . . 36
5.13.1 Acesso a Arrays . . . 37
5.13.2 Acesso a Strings . . . 37

6 Testes presentes no enunciado **38**
6.1 Olá, Mundo! . . . 38
6.2 Maior de 3 . . . 38
6.3 Fatorial . . . 40
6.4 Verificação de Número Primo . . . 41
6.5 Soma de uma lista de inteiros . . . 43
6.6 Conversão binário-decimal . . . 44
6.7 Conversão binário-decimal (c/ uma função) . . . 46

7 Testes Criados pelo Grupo **49**
7.1 Cálculo de Mínimo, Máximo e Média . . . 49
7.2 Teste para Condições . . . 53
7.3 Calculadora Avançada . . . 55

8 Conclusão **58**

1 Introdução

O Pascal é uma linguagem de programação de alto nível, estruturada, desenvolvida por **Niklaus Wirth**, no início da década de **1970**. Foi criada com o objetivo de promover boas práticas de programação e facilitar o ensino de conceitos fundamentais de algoritmia e estruturas de dados. Pascal teve um papel significativo na formação de gerações de programadores e ainda é utilizado em contextos académicos e em algumas aplicações industriais.

A linguagem Pascal tem como principais características:

- **Tipagem forte e estática:** Pascal é uma linguagem fortemente tipada, o que significa que todas as variáveis devem ter o seu tipo declarado explicitamente. Esta característica ajuda a evitar erros comuns durante a compilação e a garantir maior robustez no código;
- **Sintaxe clara e estruturada:** A sintaxe do Pascal foi projetada para ser fácil de ler e escrever, com uma estrutura semelhante à da linguagem natural. Esta clareza torna o Pascal ideal para o ensino e compreensão de algoritmos;
- **Programação modular:** Pascal suporta a definição de procedimentos e funções, permitindo a criação de programas organizados, reutilizáveis e fáceis de manter;
- **Controlo rigoroso de fluxo e dados:** A linguagem oferece estruturas de controlo como `if`, `for`, `while`, e `repeat`, além de permitir uma gestão eficiente de variáveis, arrays e tipos definidos pelo utilizador.

2 Objetivos

Neste projeto, o objetivo principal foi desenvolver um compilador para a linguagem **Pascal**, com a capacidade de análise, interpretação e tradução do código-fonte diretamente para o código máquina a ser executado por uma máquina virtual disponibilizada aos alunos.

Com a realização deste projeto, os principais objetivos foram:

- consolidar os conhecimentos sobre engenharia de linguagens e programação de compiladores, em particular o domínio da linguagem Pascal e das suas construções sintáticas e semânticas;
- praticar a escrita de gramáticas livres de contexto (GIC) e gramáticas tradutoras (GT), capazes de descrever formalmente a sintaxe e a semântica da linguagem Pascal;
- gerar código para uma máquina virtual (VM), respeitando as instruções e arquitetura definidas para o ambiente de execução fornecido;
- utilizar ferramentas baseadas em gramáticas formais, nomeadamente os geradores de analisadores léxicos e sintáticos da biblioteca **PLY** em Python (Lex + Yacc).

3 Análise Léxica

Para dar início ao desenvolvimento do compilador da linguagem **Pascal**, foi implementado um analisador léxico utilizando a biblioteca PLY (Python Lex-Yacc), especificamente o módulo Lex. Esta componente tem como principal objetivo realizar a análise léxica, ou seja, a conversão do código-fonte em uma sequência estruturada de tokens, que representam as menores unidades significativas da linguagem. Estes tokens servem como entrada para o analisador sintático, permitindo uma interpretação mais precisa e eficiente da estrutura do programa.

3.1 Símbolos

Token	Símbolo	Significado
ASSIGNMENT	:=	Símbolo para a atribuição de valores
SEMICOLON	;	Símbolo que representa o fim de uma instrução
COLON	:	Separador ou o início da declaração do tipo de uma variável
COMMA	,	Separador de elementos
DOT	.	Fim do programa ou acesso
LPAREN	(	Início de uma expressão
RPAREN	)	Fim de uma expressão
LBRACKET	[	Início de um índice ou de um array
RBRACKET	]	Fim de um índice ou de um array
RANGE	..	Intervalo de valores (ex: 1..5)

Tabela 3.1: Tabela dos tokens dos símbolos presentes na linguagem

3.2 Operadores Aritméticos

Token	Símbolos/Palavras Reservadas	Significado
PLUS	+	Adição
MINUS	-	Subtração
TIMES	*	Multiplicação
DIVISION	/	Divisão

Tabela 3.2: Tabela dos tokens dos operadores aritméticos

3.3 Operadores de Comparação

Token	Símbolos	Significado
EQ	=	Igualdade
NEQ	<>	Diferente
LT	<	Menor que
GT	>	Maior que
LTE	<=	Menor ou igual
GTE	>=	Maior ou igual

Tabela 3.3: Tabela dos tokens dos operadores de comparação

3.4 Tokens de Estrutura

PROGRAM: indica o início do programa.

```
def t_PROGRAM(t): r"[Pp][Rr][Oo][Gg][Rr][Aa][Mm]"; return t
```

VAR: inicia a secção de declaração de variáveis

```
def t_VAR(t): r"var"; return t
```

BEGIN: marca o início do bloco principal de comandos.

```
def t_BEGIN(t): r"[Bb][Ee][Gg][Ii][Nn]"; return t
```

END: marca o fim de um bloco de comandos iniciado por begin.

```
def t_END(t): r"[Ee][Nn][Dd]"; return t
```

3.5 Tokens de Controlo de Fluxos

IF: avalia uma condição lógica.

```
def t_IF(t): r"[Ii][Ff]"; return t
```

THEN: especifica o que fazer se a condição if for verdadeira.

```
def t_THEN(t): r"[Tt][Hh][Ee][Nn]"; return t
```

ELSE: define o bloco a ser executado caso a condição do if seja falsa.

```
def t_ELSE(t): r"[Ee][Ll][Ss][Ee]"; return t
```

FOR: inicia um bloco com um ciclo.

```
def t_FOR(t): r"[Ff][Oo][Rr]"; return t
```

TO e **DOWNTO:** definem o sentido da contagem num ciclo *for*.

```
// Contagem Crescente
def t_TO(t): r"[Tt][Oo]"; return t

// Contagem Decrescente
def t_DOWNTO(t): r"[Dd][Oo][Ww][Nn][Tt][Oo]"; return t
```

DO: indica o início do bloco que será executado após o *for*, *while* ou *repeat*.

```
def t_DO(t): r"[Dd][Oo]"; return t
```

WHILE: executa um ciclo enquanto uma condição for verdadeira.

```
def t\_WHILE(t): r"[Ww][Hh][Ii][Ll][Ee]"; return t
```

REPEAT e **UNTIL:** criam um ciclo com um teste no final, para que este encerre ou continue.

```
def t\_REPEAT(t): r"[Rr][Ee][Pp][Ee][Aa][Tt]"; return t

def t\_UNTIL(t): r"[Uu][Nn][Tt][Ii][Ll]"; return t
```

3.6 Tokens de Operadores Lógicos

AND: retorna true somente se ambas as expressões forem verdadeiras.

```
def t_AND(t): r"[Aa][Nn][Dd]"; return t
```

OR: retorna true se pelo menos uma das expressões for verdadeira.

```
def t_OR(t): r"[Oo][Rr]"; return t
```

NOT: inverte o valor lógico de uma expressão.

```
def t_NOT(t): r"[Nn][Oo][Tt]"; return t
```

3.7 Tokens de Operadores Aritméticos Inteiros

DIV: realiza a divisão inteira, ou seja, retorna apenas a parte inteira do quociente, descartando o resto.

```
def t_DIV(t): r"[Dd][Ii][Vv]"; return t
```

MOD: retorna o resto da divisão entre dois números inteiros.

```
def t_MOD(t): r"[Mm][Oo][Dd]"; return t
```

3.8 Tokens para Funções, Declarações e Arrays

PROCEDURE: define um subprograma que executa ações, mas não retorna um valor.

```
def t_PROCEDURE(t): r"[Pp][Rr][Oo][Cc][Ee][Dd][Uu][Rr][Ee]"; return t
```

FUNCTION: define um subprograma que retorna um valor.

```
def t_FUNCTION(t): r"[Ff][Uu][Nn][Cc][Tt][Ii][Oo][Nn]"; return t
```

ARRAY: declara um array que armazena vários elementos do mesmo tipo.

```
def t_ARRAY(t): r"[Aa][Rr][Rr][Aa][Yy]"; return t
```

OF: parte da declaração de arrays e tipos personalizados.

```
def t_OF(t): r"[Oo][Ff]"; return t
```

WRITELN: realiza a escrita de dados, seguida de uma quebra de linha.

```
def t_WRITELN(t): r"[Ww][Rr][Ii][Tt][Ee][Ll][Nn]"; return t
```

READLN: lê dados escritos pelo utilizador e armazena-os em variáveis.

```
def t_READLN(t): r"[Rr][Ee][Aa][Dd][Ll][Nn]"; return t
```

LENGTH: retorna o comprimento de uma string ou tamanho de um array.

```
def t_LENGTH(t): r"[Ll][Ee][Nn][Gg][Tt][Hh]"; return t
```

3.9 Tokens para Tipos de Dados

NREAL: representa o tipo *real* (número com casas decimais).

```
def t_NREAL(t): r"[Rr][Ee][Aa][Ll]"; return t
```

NINTEGER: representa o tipo *inteiro*.

```
def t_NINTEGER(t): r"[Ii][Nn][Tt][Ee][Gg][Ee][Rr]"; return t
```

NSTRING: representa o tipo *string* (conjunto de caracteres).

```
def t_NSTRING(t): r"[Ss][Tt][Rr][Ii][Nn][Gg]"; return t
```

NCHAR: representa o tipo *char*.

```
def t_NCHAR(t): r"[Cc][Hh][Aa][Rr]"; return t
```

NBOOLEAN: representa o tipo *boolean*.

```
def t_NBOOLEAN(t): r"[Bb][Oo][Oo][Ll][Ee][Aa][Nn]"; return t
```

BOOLEAN: token para os valores booleanos. Pode ser usado diretamente em atribuições ou expressões condicionais.

```
def t_BOOLEAN(t): r"[Tt][Rr][Uu][Ee]|[Ff][Aa][Ll][Ss][Ee]"; return t
```

3.10 Tokens Numéricos, Textuais e Simbólicos

REAL: reconhece números reais (com parte decimal), incluindo negativos. Converte o valor de string para float.

```
def t_REAL(t): r"(\-)?\d+\.\d+"; t.value = float(t.value) ; return t
```

INTEGER: reconhece números inteiros, incluindo negativos. Converte o valor de string para int.

```
def t_INTEGER(t): r"(\-)?\d+"; t.value = int(t.value); return t
```

STRING: reconhece strings entre aspas. Remove as aspas para obter o conteúdo da string.

```
def t_STRING(t): r"'([^']*)'"; t.value = t.value[1:-1]; return t
```

IDENTIFIER: reconhece nomes de variáveis, funções, procedimentos, etc.

```
def t_IDENTIFIER(t): r"[a-zA-Z]([a-zA-Z0-9])*"; return t
```

CHAR: reconhece um único caracter entre aspas simples. Extrai apenas o caracter sem as aspas.

```
def t_CHAR(t): r"'\w'"; t.value = t.value[1]; return t
```

3.11 Tokens para Comentários e Quebras de Linha

COMMENT: reconhece comentários no código e ignora-os com recurso ao ***pass.***

```
def t_COMMENT(t): r'\{.*?\}|\(\*.*?\*\)|\/\/.*'; pass
```

NEWLINE: atualiza a contagem de linhas do código, somando o número de quebras de linha encontradas.

```
def t_newline(t): r'\n+'; t.lexer.lineno += len(t.value)
```

4 Análise Sintática

A análise sintática teve como objetivo validar a estrutura do código-fonte em Pascal, verificando se a sequência de tokens gerada pelo **analisador léxico**, seguia as regras da gramática da linguagem. Através do uso do módulo **Yacc** da biblioteca **PLY**, foi definida uma gramática que cobre os principais elementos de Pascal, como declarações, expressões, estruturas de controlo e subprogramas. Esta etapa permitiu construir uma representação estrutural do programa e garantir a sua correção sintática antes da geração de código.

4.1 Símbolos presentes na Gramática

Símbolos Terminais:

```
 # Assignment
'IDENTIFIER', 'ASSIGNMENT', 'SEMICOLON', 'COLON', 'COMMA',
# Main
 'COMMENT', 'PROGRAM', 'DOT',
# Blocks
'VAR', 'BEGIN', 'END',
# Control Flow
'IF', 'THEN', 'ELSE', 'FOR', 'WHILE',
 'REPEAT', 'UNTIL', 'DO', 'TO', 'DOWNTO',
# Logic
'AND', 'OR', 'NOT',
# Operations
'PLUS', 'MINUS', 'TIMES', 'DIVISION', 'DIV', 'MOD', 'RANGE',
# Comparations
'EQ', 'NEQ', 'LT', 'GT', 'LTE', 'GTE',
# Functions
'LPAREN', 'RPAREN', 'LBRACKET', 'RBRACKET', 'PROCEDURE',
 'FUNCTION', 'ARRAY', 'OF', 'WRITELN', 'READLN', 'LENGTH',
# Types Names
'NREAL', 'NINTEGER', 'NSTRING', 'NCHAR', 'NBOOLEAN',
# Types
'REAL', 'INTEGER', 'STRING', 'CHAR', 'BOOLEAN'
```

Símbolos Não Terminais:

```
'and', 'arg_list', 'array_access', 'array_type', 'assignment', 'block',
'body', 'boolean', 'char', 'comparator', 'cond_if', 'condition',
'div', 'division', 'downto', 'eq', 'expression', 'expressionGeneric',
'expression_paren', 'fator', 'for_loop', 'func_arglist', 'func_args',
'func_body', 'func_call', 'func_header', 'func_variable_declaration',
'function', 'function_with_no_vars', 'function_with_vars', 'gt', 'gte',
'header', 'identifier', 'identifier_list', 'if_body', 'integer',
'length', 'lt', 'lte', 'minus', 'mod', 'negation', 'neq', 'not',
'operationAdd', 'operationMul', 'or', 'plus', 'prepare_func_call',
'procedure', 'procedure_arg_list', 'procedure_args_no_vars',
'procedure_args_vars', 'procedure_body', 'procedure_call',
'procedure_no_args_no_vars', 'procedure_no_args_vars',
'procedure_variable_declaration', 'program', 'readln', 'real',
'repeat_loop', 'statement', 'statements', 'string', 'term',
'times', 'to', 'type', 'type_name', 'variable_declaration',
'while_loop', 'writeln', 'writeln_args'
```

4.2 Análise de Regras da Gramática

Com o objetivo de reconhecer a estrutura e os elementos essenciais da linguagem **Pascal**, foi definida uma gramática que reflete as principais construções sintáticas da linguagem. Abaixo, serão analisadas algumas regras representativas dessa gramática, organizadas por categorias, como estrutura geral do programa, declarações, expressões e controlo de fluxo. Essas regras permitem validar a organização do código-fonte e garantir a sua conformidade com a sintaxe da linguagem.

Regras 1 e 2:

```
Rule 1    program -> header block DOT
Rule 2    header -> PROGRAM IDENTIFIER SEMICOLON
```

- **Regra 1:** define a forma completa de um programa em Pascal. Um programa começa com um **cabeçalho** (header), contém um **corpo principal** (block) e termina com um **ponto**. Essa construção é importante porque delimita explicitamente onde o programa inicia e termina.
- **Regra 2:** o **cabeçalho** especifica o **nome do programa** e exige que ele seja encerrado com **ponto e vírgula**. Esta sintaxe ajuda a identificar o **início do programa** e atribuir-lhe um **identificador**, o que facilita a organização do código, especialmente em projetos maiores. O uso obrigatório do ponto e vírgula também mantém a consistência da linguagem e permite que o parser diferencie claramente o cabeçalho das instruções que se seguem.

Regras 3 a 6:

```
Rule 3    block -> VAR variable_declaration body
Rule 4    block -> body
Rule 5    block -> function block
Rule 6    block -> procedure block
```

- **Regra 3:** esta forma define um bloco que contém **declarações de variáveis** (após VAR) e, em seguida, o corpo principal (BEGIN ... END). É a forma mais comum e simples usada em muitos programas básicos Pascal.
- **Regra 4:** permite a existência de programas sem **declaração de variáveis** explícitas. Isso é útil para programas muito simples que apenas executam instruções diretamente.
- **Regra 5:** permite que **um ou mais subprogramas** (neste caso, funções) sejam definidos antes do corpo principal do programa. Esta recursividade permite que múltiplas funções se posicionem antes do begin
- **Regra 6:** permite que **procedimentos** (procedures) sejam definidos antes do corpo principal. Também é recursiva e permite que o bloco contenha vários procedimentos.

Regras 7 a 10:

```
Rule 7    variable_declaration -> identifier_list COLON type_name
          SEMICOLON variable_declaration
Rule 8    variable_declaration -> identifier_list COLON type_name
          SEMICOLON
Rule 9    identifier_list -> IDENTIFIER COMMA identifier_list
Rule 10   identifier_list -> IDENTIFIER
```

As **regras 7 a 10** definem a forma como várias variáveis podem ser declaradas em linha, associadas a um determinado tipo. Por exemplo, a instrução ***var a, b, c: integer;*** é processada como uma lista de identificadores (a, b, c) associados ao tipo ***integer.*** A gramática permite ainda que várias declarações sejam feitas em sequência, separadas por ponto e vírgula, o que é típico em blocos var no início de programas ou subprogramas.

Regra 11:

```
Rule 11   array_type -> ARRAY LBRACKET type RANGE type RBRACKET OF
          type_name
```

Esta regra trata especificamente da **declaração de arrays.** Esta permite definir estruturas indexadas, especificando os limites inferior e superior e o tipo de cada elemento. Por exemplo, valores: ***array[1..5] of integer;*** é reconhecido como um array de cinco inteiros. Esta funcionalidade amplia a capacidade de armazenamento e manipulação de dados estruturados.

Regra 12:

```
Rule 12   array_access -> IDENTIFIER LBRACKET expressionGeneric RBRACKET
```

Esta regra garante que seja possível aceder a elementos específicos de arrays através de índices, como em ***valores[i].*** Este tipo de acesso é fundamental para **leitura e escrita** em estruturas compostas, sendo essencial para expressões de atribuição, entrada e saída.

Regra 13:

```
Rule 13   body -> BEGIN statements END
```

Esta regra é responsável por **identificar e estruturar** o bloco principal de execução de um programa Pascal. Este bloco começa obrigatoriamente com a palavra reservada ***BEGIN*** e termina com ***END***. Entre essas palavras, está contido o conjunto de instruções que representam a lógica a ser executada.

Regras 14 e 15:

```
Rule 14   statements -> statement SEMICOLON statements
Rule 15   statements -> statement SEMICOLON
```

A definição de **instruções/statements** é um dos elementos centrais da gramática, pois representa as ações que o programa Pascal irá executar.

- **Regra 14:** permite definir uma única instrução terminada por ponto e vírgula.
- **Regra 15:** permite definir várias instruções encadeadas, cada uma finalizada com um ponto e vírgula.

Regras 16 e 23:

```
Rule 16   statement -> writeln
Rule 17   statement -> assignment
Rule 18   statement -> procedure_call
Rule 19   statement -> cond_if
Rule 20   statement -> while_loop
Rule 21   statement -> for_loop
Rule 22   statement -> repeat_loop
Rule 23   statement -> readln
```

- **Regra 16:** instrução para escrever dados de output. Aceita múltiplos argumentos, inclusive texto e variáveis.
- **Regra 17:** atribuição de valores a variáveis, com suporte a expressões.
- **Regra 18:** permite a chamada de procedimentos definidos pelo utilizador, com ou sem argumentos.
- **Regra 19:** instruções condicionais, incluindo suporte a blocos compostos.
- **Regra 20:** estrutura de repetição com condição no início.
- **Regra 21:** loop com incremento ou decremento automático.
- **Regra 22:** estrutura de repetição com condição no fim, garantindo pelo menos uma execução.
- **Regra 23:** leitura de dados do utilizador.

Regras 24 a 26:

```
Rule 24   assignment -> type ASSIGNMENT expressionGeneric
Rule 25   assignment -> type ASSIGNMENT length
Rule 26   assignment -> type ASSIGNMENT negation
```

A gramática foi cuidadosamente desenvolvida para reconhecer uma ampla gama de construções válidas na linguagem Pascal — desde atribuições simples até expressões complexas que envolvem operadores aritméticos, relacionais e lógicos — garantindo uma interpretação correta da semântica do código.

Regras 27 a 34:

```
Rule 27   expressionGeneric -> expression
Rule 28   expressionGeneric -> expressionGeneric comparator expression
Rule 29   expression -> term
Rule 30   expression -> expression operationAdd term
Rule 31   term -> fator
Rule 32   term -> term operationMul fator
Rule 33   fator -> type
Rule 34   expression_paren -> LPAREN expressionGeneric RPAREN
```

Estas regras são responsáveis por reconhecer e estruturar as expressões genéricas utilizadas em Pascal. A gramática foi desenhada com base na **precedência de operadores**, organizando as expressões por níveis hierárquicos. A regra ***expressionGeneric*** permite composições que envolvem comparações relacionais e operadores lógicos. A regra ***expression*** agrupa operadores de **adição e subtração**, enquanto ***term*** trata das operações de **multiplicação e divisão.** Por fim, o nível mais básico, ***fator***, reconhece elementos simples, como literais, variáveis, chamadas de função ou expressões entre parênteses.

Regras 42 a 47:

```
Rule 42   type_name -> NINTEGER
Rule 43   type_name -> NREAL
Rule 44   type_name -> NSTRING
Rule 45   type_name -> NCHAR
Rule 46   type_name -> NBOOLEAN
Rule 47   type_name -> array_type
```

A gramática reconhece os principais tipos primitivos de Pascal, como ***integer, real, string, char, boolean*** e também suporta ***arrays*** como tipo válido. Estas definições são essenciais para a **declaração de variáveis**, tipagem de parâmetros e validação semântica durante atribuições e chamadas de funções.

Regras 48 a 61:

```
Rule 48   type -> integer
Rule 49   type -> real
Rule 50   type -> string
Rule 51   type -> char
Rule 52   type -> boolean
Rule 53   type -> identifier
Rule 54   type -> func_call
Rule 55   type -> array_access
Rule 56   type -> expression_paren
Rule 57   integer -> INTEGER
Rule 58   real -> REAL
Rule 59   string -> STRING
Rule 60   char -> CHAR
Rule 61   boolean -> BOOLEAN
```

Estas regras permitem reconhecer literais como **inteiros, reais, strings, caracteres e booleanos**, além de identificadores usados para **variáveis ou funções.** Também suportam chamadas de função, acesso a arrays e expressões entre parênteses, tornando a análise sintática mais completa e flexível.

Regras 69 a 86:

```
Rule 69   comparator -> eq
Rule 70   comparator -> neq
Rule 71   comparator -> lt
Rule 72   comparator -> gt
Rule 73   comparator -> lte
Rule 74   comparator -> gte
Rule 75   comparator -> and
Rule 76   comparator -> or
Rule 77   comparator -> not
Rule 78   eq -> EQ
Rule 79   neq -> NEQ
Rule 80   lt -> LT
Rule 81   gt -> GT
Rule 82   lte -> LTE
Rule 83   gte -> GTE
Rule 84   and -> AND
Rule 85   or -> OR
Rule 86   not -> NOT
```

Estas regras são responsáveis por reconhecer os operadores **relacionais e lógicos** utilizados na construção de expressões condicionais, fundamentais para o controlo de fluxo em Pascal, como nas instruções ***if, while e repeat.*** No que diz respeito aos operadores **relacionais**, as regras permitem comparar dois valores, verificando condições de igualdade, diferença ou ordem.

Regras 89 a 96:

```
Rule 89   procedure -> procedure_no_args_no_vars
Rule 90   procedure -> procedure_args_no_vars
Rule 91   procedure -> procedure_no_args_vars
Rule 92   procedure -> procedure_args_vars
Rule 93   procedure_no_args_no_vars -> PROCEDURE IDENTIFIER SEMICOLON
          procedure_body SEMICOLON
Rule 94   procedure_args_no_vars -> PROCEDURE IDENTIFIER LPAREN
          func_args RPAREN SEMICOLON procedure_body SEMICOLON
Rule 95   procedure_no_args_vars -> PROCEDURE IDENTIFIER SEMICOLON VAR
          func_variable_declaration procedure_body SEMICOLON
Rule 96   procedure_args_vars -> PROCEDURE IDENTIFIER LPAREN func_args
          RPAREN SEMICOLON VAR func_variable_declaration procedure_body
          SEMICOLON
```

Os **procedures** são subprogramas que **não retornam valores**, mas executam um conjunto de **instruções.** Estas regras permitem definir procedures **com ou sem parâmetros**, e **com ou sem variáveis locais.**

Regras 100 a 103:

```
Rule 100  procedure_call -> prepare_func_call
Rule 101  procedure_call -> prepare_func_call LPAREN
          procedure_arg_list RPAREN
Rule 102  procedure_arg_list -> expressionGeneric COMMA
          procedure_arg_list
Rule 103  procedure_arg_list -> expressionGeneric
```

Os **procedures** podem ser chamados **com ou sem argumentos**, dependendo de sua definição. A chamada de procedures sem argumentos é simplesmente o **nome do procedimento.** A chamada de procedures com argumentos inclui o nome do procedimento seguido de uma **lista de argumentos entre parênteses.**

Regras 105 a 110:

```
Rule 105  function -> function_with_vars
Rule 106  function -> function_with_no_vars
Rule 107  function_with_vars -> func_header SEMICOLON VAR
          func_variable_declaration func_body SEMICOLON
Rule 108  function_with_no_vars -> func_header SEMICOLON
          func_body SEMICOLON
Rule 109  func_header -> FUNCTION IDENTIFIER LPAREN func_args
          RPAREN COLON type_name
Rule 110  func_header -> FUNCTION IDENTIFIER LPAREN RPAREN COLON
          type_name
```

As **functions** são subprogramas que **retornam um valor**, e a sua definição segue uma estrutura semelhante à dos **procedures**, mas com a adição de um tipo de return.

Regras 117 a 121:

```
Rule 117  func_call -> prepare_func_call LPAREN arg_list RPAREN
Rule 118  prepare_func_call -> IDENTIFIER
Rule 119  arg_list -> expressionGeneric COMMA arg_list
Rule 120  arg_list -> expressionGeneric
Rule 121  arg_list -> <empty>
```

As **functions** também podem ser chamadas dentro de expressões, e o seu valor retornado pode ser usado diretamente. A chamada de uma **function** inclui o nome da função seguido por uma **lista de argumentos** entre parênteses. Se a function não tiver argumentos, os **parênteses ficam vazios.**

Regras 122 a 127:

```
Rule 122  cond_if -> IF condition THEN statement
Rule 123  cond_if -> IF condition THEN statement ELSE statement
Rule 124  cond_if -> IF condition THEN statement ELSE if_body
Rule 125  cond_if -> IF condition THEN if_body
Rule 126  cond_if -> IF condition THEN if_body ELSE if_body
Rule 127  cond_if -> IF condition THEN if_body ELSE statement
```

Estas regras implementam a estrutura condicional ***if ... then***, com ou sem cláusula ***else.***

Regras 132 a 135:

```
Rule 132  for_loop -> FOR assignment to type DO statement
Rule 133  for_loop -> FOR assignment to type DO if_body
Rule 134  for_loop -> FOR assignment downto type DO statement
Rule 135  for_loop -> FOR assignment downto type DO if_body
```

Estas regras implementam a estrutura de repetição ***for***, usada para iterar um número fixo de vezes.

- ***for ... to:*** executa um bloco de instruções enquanto o valor de controlo cresce até um **limite superior.**
- ***for ... downto:*** executa o loop em **ordem decrescente.**

Regras 136 a 137:

```
Rule 136  while_loop -> WHILE condition DO statement
Rule 137  while_loop -> WHILE condition DO if_body
```

Estas regras implementam o ciclo ***while***, que repete um conjunto de instruções enquanto uma condição for **verdadeira.** Avalia a **condição antes da execução** do corpo do ciclo.

Regra 138:

```
Rule 138  repeat_loop -> REPEAT statements UNTIL condition
```

Esta regra implementa o ciclo ***repeat ... until***, uma estrutura de repetição que garante pelo menos uma execução do corpo do ciclo. A condição é avaliada no **final**, o que difere do ***while.***

5 Análise Semântica

Na fase de **análise semântica**, percorremos as diversas produções definidas na gramática e começámos a associar a cada uma delas o seu correspondente **código para a Máquina Virtual.** O objetivo principal desta etapa é transformar a estrutura sintaticamente válida do programa Pascal num formato executável pela **máquina virtual fornecida.**

O programa percorre as diferentes **produções e gera o código da máquina linha a linha**, à medida que interpreta as estruturas da gramática. Para isso, são utilizadas estruturas auxiliares, como a **tabela de símbolos**, para manter o controlo dos **identificadores, tipos e posições de memória das variáveis e funções.**

Normalmente, um programa **Pascal** está dividido em **quatro partes**, sendo que algumas podem ser opcionais:

- um **header**, que representa o nome do programa;
- uma secção onde **funções e procedures** são definidas;
- uma zona de **declaração de variáveis;**
- e o **corpo principal do programa.**

As **partes opcionais** são aglomeradas numa única regra chamada ***block***, que contempla todas as possíveis combinações de estruturas de um programa Pascal. Isso garante que é possível compilar programas com diferentes **graus de complexidade**, desde os mais simples — com apenas um corpo ***BEGIN ... END*** — até aos mais complexos, com **subprogramas e múltiplas declarações.**

Independentemente da sua **complexidade**, todos os programas gerados incluem obrigatoriamente as instruções ***START*** e ***STOP***, correspondendo à **inicialização e finalização** da execução na Máquina Virtual.

A seguir, serão detalhadas as produções mais relevantes da gramática e a forma como foram traduzidas em **instruções da VM.**

5.1 Variáveis

```
def p_variable_declaration(p):
    """variable_declaration : identifier_list COLON type_name SEMICOLON
        variable_declaration
                        | identifier_list COLON type_name SEMICOLON"""
    # Permite declarações de múltiplas variáveis do mesmo tipo.
    global variables
    if isinstance(p[3], str):
        for var in p[1]: # Para cada variável declarada
            variables[var] = p[3] # Associa à tabela de símbolos com seu
                tipo
        if len(p) == 6:
            p[0] = [(var, p[3]) for var in p[1]] + p[5]
        else:
            p[0] = [(var, p[3]) for var in p[1]]
    else: # Caso em que a variável é um array
        variables[f'{p[1][0]}'] = p[3][0]
        for current in p[3][1]:
            variables[f'{p[1][0]}{current}'] = p[3][0]

def p_identifier_list(p):
    '''identifier_list : IDENTIFIER COMMA identifier_list
                    | IDENTIFIER'''
    # Permite listar múltiplos identificadores separados por vírgula.
    if len(p) == 4:
        p[0] = [p[1]] + p[3] # Lista de identificadores
    else:
        p[0] = [p[1]] # Apenas um identificador
```

O tratamento das declarações de variáveis é feito a partir da regra **variable_declaration**. O compilador reconhece os **nomes das variáveis**, associa-os aos seus respetivos **tipos** e armazena essa informação num dicionário chamado ***variables***, que guarda o nome de cada variável como **chave** e o seu tipo como **valor.**

Como a ordem das declarações é preservada, conseguimos determinar a **posição de cada variável** na memória da máquina virtual, o que facilita o uso da instrução ***STOREG x*** para armazenar valores na stack na posição correspondente.

5.2 Identifiers

Todos os **nomes de variáveis e funções** são tratados pelo compilador como identificadores ***(IDENTIFIER).*** O acesso a estes elementos é realizado através das instruções da máquina virtual, ***PUSHG*** e ***STOREG***, utilizando a posição associada a cada identificador na memória global.

5.3 Body

```
def p_body(p):
    'body : BEGIN statements END'
    global vm_code
    vm_code += "START\n"
    vm_code += "\n".join(p[2]) + "\n"
    vm_code += "STOP\n"
    p[0] = ["START"] + p[2] + ["STOP"]
```

O ***body*** representa um bloco principal de código, delimitado pelas palavras-chave ***BEGIN*** e ***END.*** Pode corresponder tanto ao corpo principal do programa como ao corpo de uma função ou procedure. Este bloco é constituído por uma **sequência de instruções (statements)** que serão executadas pela máquina virtual.

Na geração de código, a entrada num bloco ***body*** marca o início da execução com a instrução ***START***, seguida pelo conjunto de instruções geradas a partir das ***statements.*** A execução é finalizada com a instrução ***STOP.*** Esta estrutura garante que o bloco principal do programa ou subprograma seja corretamente delimitado e interpretado pela **máquina virtual.**

5.4 Statements

```
def p_statements(p):
    """statements : statement SEMICOLON statements
                  | statement SEMICOLON"""
    if len(p) == 4:
        p[0] = p[1] + p[3] # Junta as instruções das statements
    else:
        p[0] = p[1] # Apenas um statement

def p_statement(p):
    """statement : writeln
                 | assignment
                 | procedure_call
                 | cond_if
                 | while_loop
                 | for_loop
                 | repeat_loop
                 | readln"""
    p[0] = p[1]
```

As ***statements*** representam as instruções principais que compõem o **corpo de um programa em Pascal.** Cada ***statement*** corresponde a uma ação concreta que será convertida em instruções da máquina virtual. A gramática suporta os seguintes tipos de instruções:

- ***writeln***: escrita de valores no output;
- ***assignment***: atribuição de valores a variáveis;
- ***procedure_call***: chamada de procedimentos;
- ***cond_if***: estruturas condicionais (if/else);
- ***while_loop***: ciclos do tipo while;
- ***for_loop***: ciclos for (com to ou downto);
- ***repeat_loop***: ciclos repeat ... until;
- ***readln***: leitura de valores introduzidos pelo utilizador.

5.5 Assignments

```
def p_assignment(p):
    """assignment : type ASSIGNMENT expressionGeneric
                  | type ASSIGNMENT length
                  | type ASSIGNMENT negation"""
    global variables, functions
    if not isinstance(p[1], list) and (p[1] in variables.keys() or p[1]
        in functions.keys()):
        index_destiny = list(variables.keys()).index(p[1])
        if isinstance(p[3], list):
            p[0] = p[3]
        else: # Caso em que é um identifier
            index_source = list(variables.keys()).index(p[3])
            p[0] = [f'PUSHG {index_source}']

        p[0] += [f'STOREG {index_destiny}']
    elif isinstance(p[1], list):
        if isinstance(p[3], list):
            p[0] = p[1] + p[3]
        else: # Caso em que é um identifier
            index_source = list(variables.keys()).index(p[3])
            p[0] = p[1] + [f'PUSHG {index_source}']

        p[0] += p[0] + [f'STORE 0']
    else:
        raise Exception(f"Erro: Variável '{p[1]}' não declarada.")
```

Os ***assignments*** representam instruções de atribuição, típicas da linguagem Pascal. Este tipo de statement permite atribuir:

- **Valores literais** (a := 5;);
- **Valores de outras variáveis** (a := b;);
- **Valores resultantes de expressões algébricas** (a := b + c;);
- **Resultados de chamadas a funções** (a := Soma(x, y););
- **Posições de arrays** (a := vetor[3];) e entre outros.

A tradução para código VM segue uma lógica simples: primeiro é com ***PUSHG*** o valor a ser atribuído, e em seguida é armazenado com ***STOREG*** na posição da variável de destino.

5.6 Expressions

```
def p_expressionGeneric(p):
    """expressionGeneric : expression
                         | expressionGeneric comparator expression"""
    if len(p) == 2:
        p[0] = p[1]
    else:
        if isinstance(p[1], list) and isinstance(p[3], list):
            p[0] = p[1] + utils.add_ascii_conversion(p[1]) + p[3] +
                utils.add_ascii_conversion(p[3]) + p[2]
        else:
            p[0] = p[1] + p[3] + p[2]

def p_expression(p):
    """expression : term
                  | expression operationAdd term"""
    p[0] = p[1] + (p[3] + p[2] if len(p) == 4 else [])

def p_termo(p):
    """term : fator
            | term operationMul fator"""
    p[0] = p[1] + (p[3] + p[2] if len(p) == 4 else [])

def p_fator(p):
    """fator : type"""
    if not isinstance(p[1], str):
        p[0] = p[1]
    else:
        index_source1 = list(variables.keys()).index(p[1])
        p[0] = [f'PUSHG {index_source1}']

def p_expression_paren(p):
    """expression_paren : LPAREN expressionGeneric RPAREN"""
    p[0] = p[2]

def p_operationAdd(p):
    """operationAdd : plus
                    | minus
                    | div
                    | mod
                    | RANGE"""
    p[0] = p[1]

def p_operationMul(p): """operationMul : times | division"""; p[0] = p[1]
```

As **expressions** representam um dos pilares fundamentais da linguagem **Pascal**, e por isso são também uma componente crítica na análise semântica do compilador. A implementação foi cuidadosamente construída com base numa hierarquia de prioridades, respeitando as **regras matemáticas e lógicas da linguagem.**

A tradução das expressões para código da VM segue uma estrutura padronizada:

- primeiro valor → segundo valor → operação.

5.6.1 Multiplicação e Divisão (maior prioridade)

São as primeiras a ser avaliadas.

Exemplo: *a + b * c*

Na geração de código, será processado primeiro ***b * c*** e só depois ***a + (resultado)***.

5.6.2 Soma, Subtração, Div, Mod, Intervalos

Estes operadores estão imediatamente abaixo das **multiplicações** na gramática, sendo avaliados a seguir.

Exemplo: *a + b = 0*

Primeiro é avaliada a adição, e só depois a comparação.

5.6.3 Expressões Booleanas (última prioridade)

As **expressões booleanas** são avaliadas por **último**, o que permite escrever condições compostas complexas, onde cada lado pode ser uma **expressão matemática completa.**

Exemplo: *(a + b) * c + d = a - 5 * 4*

Neste caso, cada lado da comparação é calculado antes de aplicar o **operador =**.

5.7 Operadores Lógicos

O compilador suporta os principais operadores lógicos do Pascal.

```
def p_comparators(p):
    #  =, <>, <, >, <=, >=
    """comparator : eq
                  | neq
                  | lt
                  | gt
                  | lte
                  | gte
                  | and
                  | or
                  | not"""
    p[0] = p[1]
```

Cada um deles é convertido para a sua **instrução correspondente na máquina virtual**, assegurando a correta avaliação das condições.

5.8 READLN

```
def p_readln(p):
    """readln : READLN LPAREN type RPAREN"""
    global variables
    if not isinstance(p[3], list):
        var_index = list(variables.keys()).index(p[3])
        var_type = variables[p[3]]

        p[0] = ['READ']
        if var_type == "integer":
            p[0] += ["ATOI"]
        elif var_type == "float":
            p[0] += ["ATOF"]

        p[0] += [f'STOREG {var_index}']
    else: # Caso em que o elemento de destino está num array
        p[0] = p[3][:-1] + ['READ'] + ['ATOI'] # O -1 remove a instrução
            LOAD
        p[0] += [f'STORE 0']
```

A instrução ***readln*** tem dois casos distintos:

- **Leitura simples:** quando o destino é uma variável comum, o valor lido é convertido de string para o tipo correspondente — utilizando ***ATOI*** para inteiros ou ***ATOF*** para reais, quando aplicável — e armazenado na posição da variável com ***STOREG***. Para tipos como ***string***, ***char*** ou ***boolean***, o valor é armazenado diretamente sem necessidade de conversão adicional.
- **Leitura para arrays:** o acesso à posição do array já gera o endereço correto, pelo que basta empilhar o valor lido com ***READ*** seguido da conversão, e utilizar ***STOREG*** para guardar o valor na posição.

5.9 WRITELN

```
def writeln_for_function(caller):
    writer = []
    var_type = variables[current_called_func]
    if var_type == 'integer':
        writer = caller + ["WRITEI"]
    elif var_type == 'real':
        writer = caller + ["WRITEF"]
    elif var_type == 'string':
        writer = caller + ["WRITES"]
    elif var_type == 'char':
        writer = caller + ["WRITECHR"]
    elif var_type == 'boolean':
        writer = caller + ["WRITES"]
    else:
        raise Exception(f"Erro: Tipo inválido.")

    return writer

def p_writeln(p):
    """writeln : WRITELN LPAREN writeln_args RPAREN"""
    p[0] = p[3] + ["WRITELN"]
```

```
def p_writeln_args(p):
    """writeln_args : type COMMA writeln_args
                    | type"""
    global variables
    if isinstance(p[1], list):
        if "PADD" in p[1]: # Caso em que é um acesso a um array
            p[0] = p[1] + ["WRITEI"]
        elif "PUSHS" in p[1][0]: # Caso em que é um acesso a uma string
            p[0] = p[1] + ["WRITES"]
        elif "PUSHI" in p[1][0]: # Caso em que é um acesso a um inteiro
            p[0] = p[1] + ["WRITEI"]
        elif "PUSHF" in p[1][0]: # Caso em que é um acesso a um float
            p[0] = p[1] + ["WRITEF"]
        elif "CALL" in p[1]: # Caso em que é uma função, vai dar write à
            variável onde o return foi colocado
            p[0] = writeln_for_function(p[1])
        elif "CHARAT" in p[1]: # Caso em que é um acesso a um carater numa
            string
            p[0] = p[1] + ["WRITEI"]
    # Caso em que é um identifier
    elif p[1] not in variables
        raise Exception(f"Erro: Variável '{p[1]}' não declarada.")
    else:
        var_type = variables[p[1]]
        index = list(variables.keys()).index(p[1])

        push_instruction = [f'PUSHG {index}']
        if var_type == 'integer':
            p[0] = push_instruction + ["WRITEI"]
        elif var_type == 'real':
            p[0] = push_instruction + ["WRITEF"]
        elif var_type == 'string':
            p[0] = push_instruction + ["WRITES"]
        elif var_type == 'char':
            p[0] = push_instruction + ["WRITECHR"]
        elif var_type == 'boolean':
            p[0] = push_instruction + ["WRITES"]
        else:
            raise Exception(f"Erro: Tipo inválido para a variável
                '{p[1]}'.")

    if len(p) == 4:
        p[0] += p[3]
```

A instrução ***writeln*** é responsável por **imprimir dados na saída.** O compilador suporta diversos tipos de argumentos nesta operação:

- **Literals** (**ex.:** 'texto', 5, 3.14, 'c');
- **Variáveis;**
- **Expressões;**
- **Chamadas de funções.**

Cada argumento passado na ***writeln*** é analisado individualmente para identificar o seu **tipo.** Com base nisso, é gerada a **instrução adequada** para a Máquina Virtual.

Se o argumento for uma **função**, o compilador consulta o **tipo de retorno** dessa função e determina qual a **instrução de escrita** apropriada. Isto garante que o **valor de retorno** é corretamente **empilhado e impresso.** No caso de **arrays ou funções**, que retornem valores, o código é também adaptado para manipular **corretamente essas estruturas.**

No fim de cada chamada a ***writeln***, é adicionada a instrução ***WRITELN*** à saída, que imprime uma nova **quebra de linha.**

5.10 Condições (*if*)

```
def p_if(p):
    """cond_if : IF condition THEN statement
               | IF condition THEN statement ELSE statement
               | IF condition THEN statement ELSE if_body
               | IF condition THEN if_body
               | IF condition THEN if_body ELSE if_body
               | IF condition THEN if_body ELSE statement"""
    global if_counter
    else_label = f'ELSE{if_counter}'
    p[0] = [f'IF{if_counter}:'] + p[2] + [f'JZ {else_label}'] + p[4] +
        [f'JUMP ENDIF{if_counter}']
    p[0] += [f'{else_label}:']
    if len(p) == 7:
        p[0] += p[6]
    p[0] += [f'ENDIF{if_counter}:']
    if_counter += 1

def p_condition(p):
    """condition : expressionGeneric"""
    p[0] = p[1]

def p_if_body(p): """if_body : BEGIN statements END"""; p[0] = p[2]
```

A **estrutura *if*** permite ao programa tomar decisões com base em condições booleanas. O compilador suporta todas as variantes de blocos condicionais da linguagem Pascal.

Cada **bloco *if*** é identificado por um **número único (if_counter)** que é incrementado automaticamente sempre que um novo ***if*** é encontrado. Isso garante que todas as **labels (IFx, ELSEy, ENDIFz)** sejam únicas, evitando **conflitos ou sobreposição** no código gerado.

A geração do código para condições segue a seguinte lógica:

```
1. Label: IF(ID);
2. Condição: a condição vai sempre ter valor 0 ou 1, que será
   interpretado pela instrução JZ;
3. Corpo: instruções que correm caso a condição seja verdade. No final
   encontra-se um salto para fora do IF, visto que o ELSE não corre se a
   condição for verdadeira;
4. Label: ELSE(ID), o identificador do ELSE é o mesmo do seu respetivo IF;
5. Corpo do ELSE: instruções que correm caso a condição seja falsa;
6. Label de fim da condição: ENDIF(ID).
```

5.11 Ciclos

Os **ciclos** seguem todos uma lógica semelhante: a **verificação de uma condição** e, quando essa condição não é mais atendida, um **salto de saída é executado.**

Cada ciclo recebe um **identificador único**, que é incrementado a cada novo ciclo criado, garantindo sua unicidade.

Embora cada ciclo possa ter detalhes específicos que os **diferenciam**, a sua estrutura é em grande parte semelhante, assim como as condições.

5.11.1 Ciclo For

```
def p_to(p):
    """to : TO"""
    p[0] = ["FINFEQ"]

def p_downTo(p):
    """downto : DOWNTO"""
    p[0] = ["FSUPEQ"]

def p_for(p):
    """for_loop : FOR assignment to type DO statement
                | FOR assignment to type DO if_body
                | FOR assignment downto type DO statement
                | FOR assignment downto type DO if_body"""
    global loop_counter
    index = utils.get_index_from_storeg(p[2])
    p[0] = p[2]
    p[0] += [f'FOR{loop_counter}:']

    # Condição
    if not isinstance(p[4], str):
        p[0] += [f'PUSHG {index}'] + p[4] + p[3] + [f'JZ
            ENDFOR{loop_counter}']
    else:
        p[0] += [f'PUSHG {index}'] + [f'PUSHG
            {list(variables.keys()).index(p[4])}'] + p[3] + [f'JZ
            ENDFOR{loop_counter}']

    # Conteúdo do loop
    p[0] += p[6]

    # Se respeitar a condição então incrementa/decrementa o valor
    p[0] += [f'PUSHG {index}'] + [f'PUSHI 1']
    if p[3][0] == "FSUPEQ":
        p[0] += ["SUB"]
    else:
        p[0] += ["ADD"]
    p[0] += [f'STOREG {index}']

    p[0] += [f'JUMP FOR{loop_counter}']
    p[0] += [f'ENDFOR{loop_counter}:']
    loop_counter += 1
```

O ciclo ***for*** segue a lógica de verificar uma **condição e realizar um cálculo**. A estrutura do ciclo ***for*** pode ser descrita da seguinte maneira:

```
1. Inicialização da variável de controlo;
2. Label: FOR(ID), Um identificador único para o loop;
3. Condição: o ciclo continua enquanto a condição for verdadeira. Caso
   contrário, o controlo salta para o fim do ciclo usando a instrução JZ;
   3.1. As palavras-chave TO e DOWNTO indicam a direção da comparação,
      ou seja, se a variável de controlo deve ser menor ou maior que o
      objetivo;
4. Corpo do ciclo: as instruções que serão executadas a cada iteração do
   loop;
5. Incremento ou Decremento da variável de controlo;
6. Salto para nova iteração: JUMP;
7. Label de fim de ciclo: ENDFOR, quando a condição não é mais atendida,
   o loop é finalizado e o controlo vai para o ponto de fim do ciclo.
```

5.11.2 Ciclo While

```
def p_while(p):
    """while_loop : WHILE condition DO statement
                  | WHILE condition DO if_body"""
    global loop_counter
    p[0] = [f'WHILE{loop_counter}:'] + p[2] + [f'JZ
       ENDWHILE{loop_counter}'] + p[4]
    p[0] += [f'JUMP WHILE{loop_counter}']
    p[0] += [f'ENDWHILE{loop_counter}:']
    loop_counter += 1
```

O ciclo ***while*** segue uma lógica de verificação contínua de uma condição sem realizar **cálculos explícitos**, como ocorre no ciclo ***for***. O fluxo do ciclo ***while*** é executado enquanto a condição for **verdadeira** e, quando a condição se torna **falsa**, o controlo sai do loop.

Estrutura do ciclo ***while***:

```
1. Label: WHILE(ID), um identificador único para o início do loop;
2. Condição: o ciclo verifica se a condição é verdadeira. Caso contrário,
   a instrução JZ é ativada;
3. Corpo do ciclo: as instruções a serem executadas enquanto a condição
   for verdadeira;
4. Salto para nova iteração: JUMP, após o corpo do loop ser executado, a
   instrução JUMP faz voltar à parte de verificação da condição;
5. Label de fim de ciclo: ENDWHILE, quando a condição se torna falsa,
   sai-se do loop e vai para o ponto de fim de ciclo.
```

5.11.3 Ciclo de Repeat ... Until

```
def p_repeat(p):
    """repeat_loop : REPEAT statements UNTIL condition"""
    global loop_counter
    p[0] = [f'REPEAT{loop_counter}:'] + p[2]
    p[0] += p[4] + ['NOT'] + [f'JZ ENDREPEAT{loop_counter}'] # É um not
       porque o ciclo apenas corre se a condição não se verificar
    p[0] += [f'JUMP REPEAT{loop_counter}']
    p[0] += [f'ENDREPEAT{loop_counter}:']
    loop_counter += 1
```

O ciclo ***repeat ... until*** funciona de maneira semelhante ao ciclo ***while***, mas a principal diferença é que o corpo do ciclo é executado **antes da verificação da condição.** Isso garante que o ciclo será executado pelo menos uma vez, independentemente da **condição inicial.**

Estrutura do ciclo ***repeat ... until***:

```
1. Label: REPEAT(ID), um identificador único para o início do loop;
2. Corpo do ciclo: as instruções que serão executadas na primeira
   iteração e em cada iteração subsequente, até que a condição seja
   satisfeita;
3. Condição: após o corpo do ciclo, a condição é verificada. Como o ciclo
   só continua enquanto a condição for falsa, é utilizado um NOT para
   inverter a condição;
4. Salto para nova iteração: JUMP, caso a condição ainda não seja
   verdadeira, o ciclo retorna ao início com a instrução JUMP.
5. Label de fim de ciclo: ENDREPEAT, quando a condição se torna
   verdadeira, o ciclo termina e vai para o ponto de fim de ciclo.
```

5.12 Functions e Procedures

Na nossa máquina virtual (VM), o código associado a ***functions*** e/ou ***procedures*** é gerado antecipadamente, mas **guardado temporariamente** até ser necessário. Isto acontece porque, embora em Pascal as ***functions*** e/ou ***procedures*** sejam definidos antes do **código principal**, a **VM** só pode executá-los corretamente se estiverem no **final do código**, depois do corpo principal do programa.

Para resolver esse problema, usamos dicionários ***(functions e procedures)*** para armazenar, durante a análise sintática, o **nome**, os **argumentos** e o **código VM** de cada ***function*** e/ou ***procedure***. Assim, quando ocorre uma **chamada**, o código gerado pode ser corretamente injetado no **final da sequência principal** e os valores dos argumentos passados são atribuídos às variáveis locais da ***function*** e/ou ***procedure***

5.13 Acesso a Arrays e Strings

```
def p_array_type(p):
    'array_type : ARRAY LBRACKET type RANGE type RBRACKET OF type_name'
    # Representa arrays, incluindo limites inferiores e superiores
    #p[0] = ("array", p[3], p[5], p[8]) # Exemplo: ('array', 1, 5,
        'NINTEGER')
    start_value = int(p[3][0].split(" ")[1])
    p[0] = []
    current = start_value
    while current <= int(p[5][0].split(" ")[1]):
        p[0] += [current]
        current += 1
    p[0] = (p[8], p[0])

def p_array_access(p): # Trata também de acessos a carateres em strings
    """array_access : IDENTIFIER LBRACKET expressionGeneric RBRACKET"""
    head_index = list(variables.keys()).index(p[1])
    if variables[p[1]] == "string": # Caso em que o elemento é uma string
        p[0] = [f'PUSHG {list(variables.keys()).index(p[1])}']
        if not isinstance(p[3], list):
            p[0] += [f'PUSHG {list(variables.keys()).index(p[3])}']
        else:
            p[0] += p[3]
        p[0] += [f'PUSHI 1', 'SUB'] + ['CHARAT'] # Tem de ser assim porque
            a stack começa a contar do zero mas o pascal conta a partir de
            1
    else:
        if not isinstance(p[3], list):
            index = [f'PUSHG {list(variables.keys()).index(p[3])}']
        else:
            index = p[3]

        p[0] = ['PUSHFP'] + [f'PUSHI {head_index}'] + ['PADD'] + index +
            ["PADD"]
        p[0] += utils.add_array_load(p[0])
```

5.13.1 Acesso a Arrays

O **acesso a arrays na VM** é realizado obtendo o **índice da cabeça do array** e somando o **índice do elemento desejado.** Por exemplo, para acessar ***'arr[4]'***, o processo funciona da seguinte forma:

O primeiro passo é fazer um ***PUSHFP***, que coloca o frame pointer no **topo da stack**. Isso permite que possamos manipular diretamente os endereços, já que a VM não suporta a instrução de ***PUSH*** para endereços específicos.

Se, por exemplo, o array estiver na posição **3 da stack**, começamos por colocar o índice da cabeça do array com um ***PUSHI***, seguido de um ***PADD*** para somar o endereço base com o índice. Depois, colocamos o índice do elemento desejado, e novamente utilizamos ***PADD*** para calcular o **endereço final do elemento dentro do array.**

O resultado final destas operações é o **endereço exato na stack** onde o elemento ***'arr[4]'*** se encontra, permitindo o acesso ao seu valor.

5.13.2 Acesso a Strings

O **acesso a strings** na VM é feito empurrando a string desejada para a stack juntamente com o **índice do caracter** que se deseja aceder. Em seguida, utiliza-se a instrução ***CHARAT***, que coloca na stack o **código ASCII** do caracter na posição indicada.

Por exemplo, para aceder ***'str[i]'***, o processo é o seguinte:

Primeiramente, é feito um ***PUSHS*** para colocar a string ***'str'*** na stack e um ***PUSHG*** para obter o índice ***'i'***. É importante lembrar que, na VM, a **indexação começa do zero**, enquanto no Pascal começa de **1**. Portanto, antes de aceder à posição desejada, subtrai-se uma unidade do índice de ***'i'***, já que ***'str[3]'*** em Pascal corresponde a ***'str[2]'*** na VM.

Depois, a instrução ***CHARAT*** é utilizada para obter o **código ASCII do caracter** na posição calculada e colocá-lo na stack.

Este conjunto de operações permite aceder corretamente a um caracter de uma string na VM.

6 Testes presentes no enunciado

6.1 Olá, Mundo!

Input:

```
program HelloWorld;
begin
  writeln('Ola, Mundo!');
end.
```

Output:

```
START
PUSHS "Ola, Mundo!"
WRITES
WRITELN
STOP
```

6.2 Maior de 3

Input:

```
program Maior3;
var
  num1, num2, num3, maior: integer;
begin
  WriteLn('Introduza o primeiro número: ');
  ReadLn(num1);
  WriteLn('Introduza o segundo número: ');
  ReadLn(num2);
  WriteLn('Introduza o terceiro número: ');
  ReadLn(num3);
  if num1 > num2 then
    if num1 > num3 then maior := num1
    else maior := num3
  else
    if num2 > num3 then maior := num2
    else maior := num3;
  WriteLn('O maior é: ', maior);
end.
```

Output:

```
START
PUSHS "Introduza o primeiro número: "
WRITES
WRITELN
READ
ATOI
STOREG 0
PUSHS "Introduza o segundo número: "
WRITES
WRITELN
READ
ATOI
STOREG 1
PUSHS "Introduza o terceiro número: "
WRITES
WRITELN
READ
ATOI
STOREG 2
IF2:
PUSHG 0
PUSHG 1
FSUP
JZ ELSE2
IF0:
PUSHG 0
PUSHG 2
FSUP
JZ ELSE0
PUSHG 0
STOREG 3
JUMP ENDIF0
ELSE0:
PUSHG 2
STOREG 3
ENDIF0:
JUMP ENDIF2
ELSE2:
IF1:
PUSHG 1
PUSHG 2
FSUP
JZ ELSE1
PUSHG 1
STOREG 3
```

```
JUMP ENDIF1
ELSE1:
PUSHG 2
STOREG 3
ENDIF1:
ENDIF2:
PUSHS "O maior é: "
WRITES
PUSHG 3
WRITEI
WRITELN
STOP
```

6.3 Fatorial

Input:

```
program Fatorial;
var
  n, i, fat: integer;
begin
  writeln('Introduza um número inteiro positivo:');
  readln(n);
  fat := 1;
  for i := 1 to n do
    fat := fat * i;
  writeln('Fatorial de ', n, ': ', fat);
end.
```

Output:

```
START
PUSHS "Introduza um número inteiro positivo:"
WRITES
WRITELN
READ
ATOI
STOREG 0
PUSHI 1
STOREG 2
PUSHI 1
STOREG 1
FOR0:
PUSHG 1
PUSHG 0
FINFEQ
JZ ENDFOR0
```

```
PUSHG 2
PUSHG 1
MUL
STOREG 2
PUSHG 1
PUSHI 1
ADD
STOREG 1
JUMP FOR0
ENDFOR0:
PUSHS "Fatorial de "
WRITES
PUSHG 0
WRITEI
PUSHS ": "
WRITES
PUSHG 2
WRITEI
WRITELN
STOP
```

6.4 Verificação de Número Primo

Input:

```
program NumeroPrimo;
var
  num, i: integer;
  primo: boolean;
begin
  writeln('Introduza um número inteiro positivo:');
  readln(num);
  primo := true;
  i := 2;
  while (i <= (num div 2)) and primo do
    begin
        if (num mod i) = 0 then
            primo := false;
        i := i + 1;
    end;
  if primo then
    writeln(num, ' é um número primo')
  else
    writeln(num, ' não é um número primo');
end.
```

Output:

```
START
PUSHS "Introduza um número inteiro positivo:"
WRITES
WRITELN
READ
ATOI
STOREG 1
PUSHI 1
STOREG 0
PUSHI 2
STOREG 2
WHILE0:
PUSHG 2
PUSHG 1
PUSHI 2
DIV
FTOI
FINFEQ
PUSHG 0
AND
JZ ENDWHILE0
IF0:
PUSHG 1
PUSHG 2
MOD
PUSHI 0
EQUAL
JZ ELSE0
PUSHI 0
STOREG 0
JUMP ENDIF0
ELSE0:
ENDIF0:
PUSHG 2
PUSHI 1
ADD
STOREG 2
JUMP WHILE0
ENDWHILE0:
IF1:
PUSHG 0
JZ ELSE1
PUSHG 1
WRITEI
PUSHS " é um número primo"
```

```
WRITES
WRITELN
JUMP ENDIF1
ELSE1:
PUSHG 1
WRITEI
PUSHS " não é um número primo"
WRITES
WRITELN
ENDIF1:
STOP
```

6.5 Soma de uma lista de inteiros

Input:

```
program SomaArray;
var
  numeros: array[1..5] of integer;
  i, soma: integer;
begin
  soma := 0;
  writeln('Introduza 5 números inteiros:');
  for i := 1 to 5 do
  begin
    readln(numeros[i]);
    soma := soma + numeros[i];
  end;
  writeln('A soma dos números é: ', soma);
end.
```

Output:

```
START
PUSHI 0
STOREG 1
PUSHS "Introduza 5 números inteiros:"
WRITES
WRITELN
PUSHI 1
STOREG 0
FOR0:
PUSHG 0
PUSHI 5
FINFEQ
JZ ENDFOR0
PUSHFP
```

```
PUSHI 2
PADD
PUSHG 0
PADD
READ
ATOI
STORE 0
PUSHG 1
PUSHFP
PUSHI 2
PADD
PUSHG 0
PADD
LOAD 0
ADD
STOREG 1
PUSHG 0
PUSHI 1
ADD
STOREG 0
JUMP FOR0
ENDFOR0:
PUSHS "A soma dos números é: "
WRITES
PUSHG 1
WRITEI
WRITELN
STOP
```

6.6 Conversão binário-decimal

Input:

```
program BinarioParaInteiro;
var
  bin: string;
  i, valor, potencia: integer;
begin
  writeln('Introduza uma string binária:');
  readln(bin);
  valor := 0;
  potencia := 1;
  for i := length(bin) downto 1 do
  begin
    if bin[i] = '1' then
      valor := valor + potencia;
    potencia := potencia * 2;
```

```
  end;
  writeln('O valor inteiro correspondente é: ', valor);
end.
```

Output:

```
START
START
PUSHS "Introduza uma string binária:"
WRITES
WRITELN
READ
STOREG 3
PUSHI 0
STOREG 1
PUSHI 1
STOREG 2
PUSHG 3
STRLEN
STOREG 0
FOR0:
PUSHG 0
PUSHI 1
FSUPEQ
JZ ENDFOR0
IF0:
PUSHG 3
PUSHG 0
PUSHI 1
SUB
CHARAT
PUSHS "1"
CHRCODE
EQUAL
JZ ELSE0
PUSHG 1
PUSHG 2
ADD
STOREG 1
JUMP ENDIF0
ELSE0:
ENDIF0:
PUSHG 2
PUSHI 2
MUL
STOREG 2
PUSHG 0
```

```
PUSHI 1
SUB
STOREG 0
JUMP FOR0
ENDFOR0:
PUSHS "O valor inteiro correspondente é: "
WRITES
PUSHG 1
WRITEI
WRITELN
STOP
```

6.7 Conversão binário-decimal (c/ uma função)

Input:

```
program BinarioParaInteiro;
function BinToInt(bin: string): integer;
var
  i, valor, potencia: integer;
begin
  valor := 0;
  potencia := 1;
  for i := length(bin) downto 1 do
  begin
    if bin[i] = '1' then
      valor := valor + potencia;
    potencia := potencia * 2;
  end;
  BinToInt := valor;
end;

var
  bin: string;
  valor: integer;

begin
  writeln('Introduza uma string binária:');
  readln(bin);
  valor := BinToInt(bin);
  writeln('O valor inteiro correspondente é: ', valor);
end.
```

Output:

```
START
PUSHS "Introduza uma string binária:"
WRITES
WRITELN
READ
STOREG 0
PUSHG 0
STOREG 0
PUSHA BinToInt
CALL
PUSHG 1
STOREG 3
PUSHS "O valor inteiro correspondente é: "
WRITES
PUSHG 3
WRITEI
WRITELN
STOP

BinToInt:
PUSHI 0
STOREG 3
PUSHI 1
STOREG 4
PUSHG 0
STRLEN
STOREG 2
FOR0:
PUSHG 2
PUSHI 1
FSUPEQ
JZ ENDFOR0
IF0:
PUSHG 0
PUSHG 2
PUSHI 1
SUB
CHARAT
PUSHS "1"
CHRCODE
EQUAL
JZ ELSE0
PUSHG 3
PUSHG 4
ADD
```

```
STOREG 3
JUMP ENDIF0
ELSE0:
ENDIF0:
PUSHG 4
PUSHI 2
MUL
STOREG 4
PUSHG 2
PUSHI 1
SUB
STOREG 2
JUMP FOR0
ENDFOR0:
PUSHG 3
STOREG 1
RETURN
```

---

7 Testes Criados pelo Grupo

7.1 Cálculo de Mínimo, Máximo e Média

Input:

```
program Estatisticas;
function MediaTotal(soma: integer): real;
begin
  MediaTotal := soma / 5;
end;
var
  valores: array[1..5] of integer;
  i, min, max, soma: integer;
  media: real;
begin
  writeln('Introduza 5 números:');
  for i := 1 to 5 do
    readln(valores[i]);

  min := valores[1];
  max := valores[1];
  soma := 0;

  for i := 1 to 5 do
  begin
    if valores[i] < min then
      min := valores[i];
    if valores[i] > max then
      max := valores[i];
    soma := soma + valores[i];
  end;

  media := MediaTotal(soma);
  writeln('Mínimo: ', min);
  writeln('Máximo: ', max);
  writeln('Média: ', media);
end.
```

Output:

```
START
PUSHS "Introduza 5 números:"
WRITES
WRITELN
PUSHI 1
STOREG 3
FOR0:
PUSHG 3
PUSHI 5
FINFEQ
JZ ENDFOR0
PUSHFP
PUSHI 6
PADD
PUSHG 3
PADD
READ
ATOI
STORE 0
PUSHG 3
PUSHI 1
ADD
STOREG 3
JUMP FOR0
ENDFOR0:
PUSHFP
PUSHI 6
PADD
PUSHI 1
PADD
LOAD 0
STOREG 4
PUSHFP
PUSHI 6
PADD
PUSHI 1
PADD
LOAD 0
STOREG 5
PUSHI 0
STOREG 0
PUSHI 1
STOREG 3
FOR1:
PUSHG 3
```

```
PUSHI 5
FINFEQ
JZ ENDFOR1
IF0:
PUSHFP
PUSHI 6
PADD
PUSHG 3
PADD
LOAD 0
PUSHG 4
FINF
JZ ELSE0
PUSHFP
PUSHI 6
PADD
PUSHG 3
PADD
LOAD 0
STOREG 4
JUMP ENDIF0
ELSE0:
ENDIF0:
IF1:
PUSHFP
PUSHI 6
PADD
PUSHG 3
PADD
LOAD 0
PUSHG 5
FSUP
JZ ELSE1
PUSHFP
PUSHI 6
PADD
PUSHG 3
PADD
LOAD 0
STOREG 5
JUMP ENDIF1
ELSE1:
ENDIF1:
PUSHG 0
PUSHFP
PUSHI 6
PADD
```

```
PUSHG 3
PADD
LOAD 0
ADD
STOREG 0
PUSHG 3
PUSHI 1
ADD
STOREG 3
JUMP FOR1
ENDFOR1:
PUSHG 0
STOREG 0
PUSHA MediaTotal
CALL
PUSHG 1
STOREG 2
PUSHS "Mínimo: "
WRITES
PUSHG 4
WRITEI
WRITELN
PUSHS "Máximo: "
WRITES
PUSHG 5
WRITEI
WRITELN
PUSHS "Média: "
WRITES
PUSHG 2
WRITEF
WRITELN
STOP

MediaTotal:
PUSHG 0
PUSHI 5
DIV
STOREG 1
RETURN
```

7.2 Teste para Condições

Input:

```
program Conds;
function Conta(e, f: integer): integer;
begin
  Soma := e - f;
end;
var
  a, b, c, d: integer;
begin
  a := 2;
  b := 3;
  c := 11;
  d := 6;
  if a + b = Conta(c, d) then
    writeln('Verdade')
  else
    writeln('Falso');
end.
```

Output:

```
START
PUSHI 2
STOREG 3
PUSHI 3
STOREG 4
PUSHI 11
STOREG 5
PUSHI 6
STOREG 6
IF0:
PUSHG 3
PUSHG 4
ADD
PUSHG 5
STOREG 1
PUSHG 6
STOREG 0
PUSHA Soma
CALL
PUSHG 2
EQUAL
JZ ELSE0
PUSHS "Verdade"
```

```
WRITES
WRITELN
JUMP ENDIF0
ELSE0:
PUSHS "Falso"
WRITES
WRITELN
ENDIF0:
STOP

Soma:
PUSHG 1
PUSHG 0
SUB
STOREG 2
RETURN
```

7.3 Calculadora Avançada

Input:

```
program CalculadoraAvancada;
function Max(a, b: integer): integer;
begin
  if a > b then
    Max := a
  else
    Max := b;
end;
function Min(c, d: integer): integer;
begin
  if c < d then
    Min := c
  else
    Min := d;
end;
var
  x, y: integer;
begin
  writeln('Introduza dois números:');
  readln(x);
  readln(y);
  writeln('O maior é: ', Max(x, y));
  writeln('O menor é: ', Min(x, y));
  if ((x + y) mod 2) = 0 then
    writeln('A soma é par')
  else
    writeln('A soma é ímpar');
end.
```

Output:

```
START
PUSHS "Introduza dois números:"
WRITES
WRITELN
READ
ATOI
STOREG 6
READ
ATOI
STOREG 7
PUSHS "O maior é: "
WRITES
PUSHG 6
STOREG 1
PUSHG 7
STOREG 0
PUSHA Max
CALL
PUSHG 2
WRITEI
WRITELN
PUSHS "O menor é: "
WRITES
PUSHG 6
STOREG 4
PUSHG 7
STOREG 3
PUSHA Min
CALL
PUSHG 5
WRITEI
WRITELN
IF2:
PUSHG 6
PUSHG 7
ADD
PUSHI 2
MOD
PUSHI 0
EQUAL
JZ ELSE2
PUSHS "A soma é par"
WRITES
WRITELN
JUMP ENDIF2
```

```
ELSE2:
PUSHS "A soma é ímpar"
WRITES
WRITELN
ENDIF2:
STOP

Max:
IF0:
PUSHG 1
PUSHG 0
FSUP
JZ ELSE0
PUSHG 1
STOREG 2
JUMP ENDIF0
ELSE0:
PUSHG 0
STOREG 2
ENDIF0:
RETURN


Min:
IF1:
PUSHG 4
PUSHG 3
FINF
JZ ELSE1
PUSHG 4
STOREG 5
JUMP ENDIF1
ELSE1:
PUSHG 3
STOREG 5
ENDIF1:
RETURN
```

8 Conclusão

O desenvolvimento deste projeto de analisador léxico e sintático para a linguagem Pascal constituiu em uma oportunidade para consolidarmos conhecimentos fundamentais sobre a construção de compiladores e o funcionamento interno de linguagens de programação.

Ao longo do trabalho, foi possível compreender a importância da definição rigorosa de uma gramática formal, essencial para garantir a análise correta de programas escritos em Pascal. A estruturação adequada das regras de parsing, bem como a identificação e gestão precisa dos tokens, revelou-se crucial para a interpretação de construções típicas da linguagem, como declarações de variáveis, estruturas de controlo de fluxo, procedimentos e expressões aritméticas.

Além disso, o projeto exigiu um planeamento meticuloso na elaboração de testes, permitindo validar o funcionamento do compilador em diferentes cenários da linguagem Pascal, desde programas simples até exemplos com maior complexidade. Esta fase foi decisiva para identificar inconsistências, ajustar a gramática e garantir a estabilidade da solução.

De um ponto de vista mais global, este projeto não só reforçou competências técnicas específicas na área dos compiladores, como também promoveu a nossa capacidade de análise, abstração e resolução de problemas. Enriquecemos a nossa visão crítica sobre a construção de linguagens de programação, consolidando um conjunto de ferramentas essenciais para futuros desenvolvimentos nesta área ou em domínios relacionados com engenharia de software e linguagens formais.