Universidade do Minho
Escola de Engenharia

Processamento de Linguagens 2024/25

COMPILADOR DE PASCAL STANDARD

Junho 2025

Gustavo Barros
A100656

Enzo Vieira
A98352

Pedro Ferreira
A97646

1.1. Deteção de Comentários

```
def t_BRACECOMMENT(t):
  r'\{[^}]*\}'
  t.lexer.lineno += t.value.count('\n')
  pass
def t_PARENCOMMENT(t):
  r'\(\*[.\s\S]*?\*\)'
  t.lexer.lineno += t.value.count('\n')
  pass
def t_SLASHCOMMENT(t):
  r'\/\/.*?\n'
  t.lexer.lineno += 1
  pass

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

Existem três sintaxes diferentes para comentários...
- **"//..."** finalizado por quebra de linha.
- **"(*...*)"** e **"{...}"** finalizados pelo delimitador de fecho mais próximo.

Há especificações da linguagem que permitem aninhar comentários, mas nesta implementação não se tem isso em conta. De forma a manter correta a contagem de linhas, as regras dos comentários multilinha incrementam no contador de newlines quantas linhas atravessam.

1.2. Palavras Reservadas

```
def t_FUNCTION(t):   r'[fF][uU][nN][cC][tT][iI][oO][nN]'; return t
def t_TYPEBOOL(t):   r'[bB][oO][oO][lL][eE][aA][nN]'; return t
...
def t_RETURN(t):     r'[rR][eE][tT][uU][rR][nN]'; return t
...
def t_WHILE(t):      r'[wW][hH][iI][lL][eE]'; return t
...
def t_TYPEREAL(t):   r'[rR][eE][aA][lL]'; return t
...
def t_VAR(t):        r'[vV][aA][rR]'; return t
...
def t_DO(t):         r'[dD][oO]'; return t
...
def t_DOTDOT(t):     r'\.\.'; return t
...
def t_DOT(t):        r'\.'; return t
...
```

As regras definidas para palavras reservadas são assim constituídas pois, fora as strings literais, Pascal é totalmente insensível a capitalização.

Note-se também que se dispuseram as regras em ordem decrescente de comprimento em caracteres. Isto garante que regras de palavras mais curtas não levem a deteções precoces quando partilham um prefixo com palavras mais longas.

1.3. Valores Literais e Identificadores

```
def t_REALVALUE(t):
    r'\d+\.\d+'
    t.value = float(t.value)
    return t

def t_INTVALUE(t):
    r'\d+'
    t.value = int(t.value)
    return t

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

def t_IDENTIFIER(t):
    r'[a-zA-Z_\$][a-zA-Z_0-9\$]*'
    t.value = t.value.lower()
    return t
```

O mesmo raciocínio de prioridade de regras foi tomado entre as de deteção de números reais e inteiros.

A regra dos identificadores segue o padrão do compilador da Sun Microsystems, que afirma a possibilidade de usar **"$"** e **"_"** em qualquer posição no nome dum identificador, por muito desencorajado que seja usá-los como inicial. isto ao contrário dos números, que nunca podem ser iniciais.

1.4. Output do Lexer

```
1  program Maior3;
2
3  var
4  num1, num2, num3, maior: Integer;
5
6  begin
7    { Ler 3 números
8
9  end.

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

(.venv) → PL2425 git:(main) ✗ python3 src/lexer.py inputs/if_condition.pas
[Ln 7] Lexical Error: '{'
(.venv) → PL2425 git:(main) ✗
```

Situação de Erro

```
(.venv) → PL2425 git:(main) ✗ python3 src/lexer.py inputs/helloworld.pas
   1: program helloworld ;
   2:
   3:
   4:
   5:
   6:
   7:
   8:
   9:
  10:
  11:
  12:
  13:
  14:
  15:
  16:
  17:
  18:
  19:
  20:
  21: begin
  22: writeln ( Ola, Mundo! ) ;
  23: end .

Line    Pos     Type            Value
-------------------------------------------------
1       0       PROGRAM         program
1       8       IDENTIFIER      helloworld
1       18      SEMICOLON       ;
21      171     BEGIN           begin
22      179     IDENTIFIER      writeln
22      186     LPAREN          (
22      187     STRINGVALUE     Ola, Mundo!
22      200     RPAREN          )
```

```
1  program HelloWorld;
2
3   {
4
5   Comentario chavetas
6
7   (*
8    Comentario Parenteses
9   *)
10
11  }
12  // Comentario em linha
13  (*
14
15  Comentario Parenteses
16
17  {Comentario chavetas}
18
19  *)
20
21  begin
22    writeln('Ola, Mundo!');
23  end.
```

Situação Correta

2.1. Expressões Primárias

```
def p_PrimaryExpression(p):
    """
    PrimaryExpression : LiteralValue
                      | LPAREN Expression RPAREN
                      | RoutineCall
                      | ArrayAccess
                      | DeclaredName
    """
    if len(p)==2: p[0] = p[1]
    else:         p[0] = Node("PrimaryExpression", p[2])

def p_LiteralValue(p):
    """
    LiteralValue : STRINGVALUE
                 | CHARVALUE
                 | INTVALUE
                 | REALVALUE
                 | TRUE
                 | FALSE
    """
    p[0] = Node("LiteralValue", p.slice[1].type, p[1])

def p_ArrayAccess(p):
    """
    ArrayAccess : DeclaredName LSPAREN RSPAREN
                | DeclaredName LSPAREN Expression RSPAREN
    """
    if len(p)==4 : p[0] = Node("ArrayAccess", p[1])
    else:          p[0] = Node("ArrayAccess", p[1], p[3])

def p_DeclaredName(p):
    """
    DeclaredName : IDENTIFIER
    """
    declared_type = "Unknown"
    if p[1] in declared_dict:
        declared_type = declared_dict[p[1]]

    p[0] = Node("DeclaredName", p[1], declared_type)
```

Uma expressão resume-se à representação de certo valor.

Foram designadas de "primárias" as expressões de maior precedência de operação, isto é, as as que são resolvidas primeiro quando aninhadas numa expressão composta. Aqui inclui-se as expressões com parênteses ou valores atómicos (literais, variáveis, acessos a array, chamadas de função).

Note-se que, como a chamada de função e de procedimento são de sintaxe igual, não fica ao encargo da análise sintática atirar erros por chamada errónea dum procedimento em lugar de uma função. Assim, ambos se representam pela regra *RoutineCall*.

2.2. Expressões Compostas

```
def p_Expression(p):
    """
    Expression : OrExpression
    """
    p[0] = Node("Expression", p[1])

def p_OrExpression(p):
    """
    OrExpression : AndExpression
                 | OrExpression OR AndExpression
    """
    if len(p)==2: p[0] = p[1]
    else:         p[0] = Node("OrExpression", p[1], p[3])

def p_AndExpression(p):
    """
    AndExpression : RelExpression
                  | AndExpression AND RelExpression
    """
    if len(p)==2: p[0] = p[1]
    else:         p[0] = Node("AndExpression", p[1], p[3])

def p_RelExpression(p):
    """
    RelExpression : AddExpression
                  | RelExpression RelOperator AddExpression
    """
    if len(p)==2: p[0] = p[1]
    else:         p[0] = Node("RelExpression", p[1], p[2], p[3])

def p_AddExpression(p):
    """
    AddExpression : MultExpression
                  | AddExpression AddOperator MultExpression
    """
    if len(p)==2: p[0] = p[1]
    else:         p[0] = Node("AddExpression", p[1], p[2], p[3])

def p_MultExpression(p):
    """
    MultExpression : UnaryExpression
                   | MultExpression MultOperator UnaryExpression
    """
    if len(p)==2: p[0] = p[1]
    else:         p[0] = Node("MultExpression", p[1], p[2], p[3])

def p_UnaryExpression(p):
    """
    UnaryExpression : UnaryOperator UnaryExpression
                    | PrimaryExpression
    """
    if len(p)==2: p[0] = p[1]
    else:         p[0] = Node("UnaryExpression", p[1], p[2])
```

Nas expressões que envolvem múltiplos operadores, há que ter em conta que cada um tem o seu grau característico de precedência. No Yacc, há duas maneiras de resolver isso:

a) Estipulando diretivas de associatividade.
b) Usando recursividade na gramática.

Tendo em conta o objetivo do projeto, é mais sensata a opção (b). Resumidamente, em certo grau de precedência existe um operador e subexpressões que incluem um operador do grau de precedência imediatamente acima, até chegar a uma expressão primária.

2.3. Statements Simples

```
def p_TerminalSemicolons(p):
    """
    TerminalSemicolons :
                       | SEMICOLON TerminalSemicolons
    """

def p_StatementBlock(p):
    """
    StatementBlock : BEGIN END
                   | BEGIN ManyStatements TerminalSemicolons END
    """
    if len(p) == 3:
        p[0] = Node("StatementBlock")
    else:
        p[0] = Node("StatementBlock", *p[2].args)

def p_ManyStatements(p):
    """
    ManyStatements : Statement
                   | ManyStatements SEMICOLON Statement
    """
    if len(p) == 2:
        p[0] = Node('ManyStatements', p[1])
    else:
        p[1].add_child(p[3])
        p[0] = p[1]

def p_Statement(p):
    """
    Statement : NoTailStatement
              | IfThen
              | IfThenElse
              | While
              | For
    """
    p[0] = Node("Statement", p[1])

# "closed" statement: all ifs within are ELSE'd
# to avoid ELSE ambiguity, THEN substatements must be closed
def p_ClosedStatement(p):
    """
    ClosedStatement : NoTailStatement
                    | ClosedIfThenElse
                    | ClosedWhile
                    | ClosedFor
    """
    p[0] = p[1]

def p_NoTailStatement(p):
    """
    NoTailStatement : StatementBlock
                    | Assignment
                    | RoutineCall
    """
    p[0] = Node("NoTailStatement", p[1])

def p_Assignment(p):
    """
    Assignment : DeclaredName ASSIGN Expression
    """
    p[0] = Node("Assignment", p[1], p[3])
```

Um statement nada mais é que uma ação.

As ações mais básicas que se podem ter é a atribuição de valor a uma variável, a chamada dum procedimento ou a execução dum bloco de statements separados por ponto-e-vírgula (este último só é possível quando aninhado numa construção que o domine).

São albergados na regra *NoTailStatement* pois, ao contrário dos statements de controlo de fluxo, carecem da omissibilidade de delimitadores de bloco em situações de um único substatement. Isto garante a sua impermeabilidade perante potenciais ambiguidades *begin-end*.

A regra *TerminalSemicolons* existe pois são tolerados ponto-e-vírgula após último statement dum bloco - são mero açúcar sintático, visto que é um token para separar e não terminar como em C.

2.4. Statements de Fluxo de Controlo

```
def p_IfThen(p):
    """
    IfThen : IF Expression THEN Statement
    """
    p[0] = Node("IfThen", p[2], p[4])

def p_IfThenElse(p):
    """
    IfThenElse : IF Expression THEN ClosedStatement ELSE Statement
    """
    p[0] = Node("IfThenElse", p[2], p[4], p[6])

def p_ClosedIfThenElse(p):
    """
    ClosedIfThenElse : IF Expression THEN ClosedStatement ELSE
    ClosedStatement
    """
    p[0] = Node("IfThenElse", p[2], p[4], p[6])

def p_While(p):
    """
    While : WHILE Expression DO Statement
    """
    p[0] = Node("While", p[2], p[4])

def p_ClosedWhile(p):
    """
    ClosedWhile : WHILE Expression DO ClosedStatement
    """
    p[0] = Node("ClosedWhile", p[1],p[2],p[3],p[4])

def p_For(p):
    """
    For : FOR Assignment TO Expression DO Statement
        | FOR Assignment DOWNTO Expression DO Statement
    """
    p[0] = Node("For", p[2], p[3], p[4], p[6])

def p_ClosedFor(p):
    """
    ClosedFor : FOR Assignment TO Expression DO ClosedStatement
              | FOR Assignment DOWNTO Expression DO ClosedStatement
    """
    p[0] = Node("ClosedFor", p[2], p[3], p[4])
```

Algo peculiar na sintaxe de controlo de fluxo é o problema do "else pendurado", que consiste na ambiguidade da pertença de certo token ELSE quando há IFs indentados. Isto acontece à conta do ELSE ser um campo opcional.

Para colmatar isto, obriga-se a que todos os substatements do campo THEN sejam "fechados", isto é, que tenham ELSE em todos os IFs que contenha. Foi uma estratégia inspirada pela documentação do gerador AnaGram da Parsifal Software.

2.5. Declarações de Variável

```
def p_ManyDeclarations(p):
    """
    ManyDeclarations : Declaration
                     | ManyDeclarations Declaration
    """
    if len(p) == 2: p[0] = Node("ManyDeclarations", p[1])
    else:           p[0] = Node("ManyDeclarations", p[1], p[2])

def p_Declaration(p):
    """
    Declaration : VAR VarDeclaration
                | PROCEDURE ProcedureDeclaration
                | FUNCTION FunctionDeclaration
    """
    p[0] = p[2]

def p_VarDeclaration(p):
    """
    VarDeclaration : ManyParameterTuples SEMICOLON
    """
    p[0] = Node("VarDeclaration", p[1])
...
def p_ParameterTuple(p):
    """
    ParameterTuple : ManyDeclaredNames ReturnType
    """
    declarations = p[1]
    return_type = p[2].args[0]

    for declaration in declarations.args:
        declaration_name = declaration.args[0]
        declared_dict[declaration_name] = return_type

    p[0] = Node("ParameterTuple", p[1], p[2])

    def p_ArrayType(p):
    """
    ArrayType : ARRAY LSPAREN ValueRange RSPAREN OF SimpleType
    """
    p[0] = Node("ArrayType", p[3], p[6])

def p_ValueRange(p):
    """
    ValueRange : ConstantValue DOTDOT ConstantValue
    """
    p[0] = Node("ValueRange", p[1], p[3])

def p_ConstantValue(p):
    """
    ConstantValue : LiteralValue
                  | DeclaredName
    """
    p[0] = Node("ConstantValue", p[1])
```

Na declaração de variáveis e arrays, a maior dificuldade encontrada foi em implementar declaração de várias variáveis de igual tipo na mesma linha. Levou à criação da regra *ParameterTuple* que alberga uma linha com uma lista de variáveis e o seu tipo após dois pontos. O nome deriva do facto de esta sintaxe ser reutilizada para parâmetros na declaração de rotinas, vista mais à frente.

A declaração de arrays só suporta tipos *SimpleType*, isto é, os predefinidos, o que não corresponde à realidade visto que é permitida a existência de arrays de tipo definido no próprio código-fonte. No entanto, isto traria complexidade acrescida na análise semântica.

2.6. Declarações de Rotina

```
def p_Scope(p):
    """
    Scope : ManyDeclarations StatementBlock
          | StatementBlock
    """
    if len(p)==2 : p[0] = Node("Scope", p[1])
    else:          p[0] = Node("Scope", p[1], p[2])

def p_ProcedureDeclaration(p):
    """
    ProcedureDeclaration : RoutineHeading SEMICOLON Scope SEMICOLON
    """
    p[0] = Node("ProcedureDeclaration", p[1], p[2], p[3], p[4])

def p_FunctionDeclaration(p):
    """
    FunctionDeclaration : RoutineHeading ReturnType SEMICOLON Scope
    SEMICOLON
    """
    func_name = p[1].args[0].args[0]
    return_type = p[2]

    declared_dict[func_name] = return_type

    p[0] = Node("FunctionDeclaration", p[1], p[2], p[4])

def p_ProgramDeclaration(p): # start here
    """
    ProgramDeclaration : DeclaredName SEMICOLON Scope DOT
    """
    p[0] = Node("ProgramDeclaration", p[1], p[3])

def p_RoutineHeading(p):
    """
    RoutineHeading : DeclaredName RoutineParameters
    """
    p[0] = Node("RoutineHeading", p[1], p[2])

def p_RoutineParameters(p):
    """
    RoutineParameters :
                      | LPAREN RPAREN
                      | LPAREN ManyParameterTuples RPAREN
    """
    if len(p) == 1:   p[0] = Node("RoutineParameters")
    elif len(p) == 3: p[0] = Node("RoutineParameters")
    else:             p[0] = Node("RoutineParameters", p[2])
```

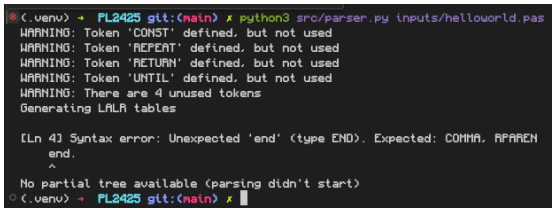
A cada *scope* ou *frame* existe uma zona de declarações seguida duma zona de statements, e isso é algo uniforme a qualquer rotina, seja no programa na sua totalidade ou em qualquer subrotina funcional/procedimental. Isto torna relativamente direta a definição da sintaxe para estes três conceitos.

Há a diferença da função para com os outros no facto de, como é uma expressão, retorna um valor.

Note-se que, numa implementação mais minuciosa, dever-se-ia ter em conta que é válido ter programas com parâmetros, o que neste estado não é possível.

2.7. Output do Parser

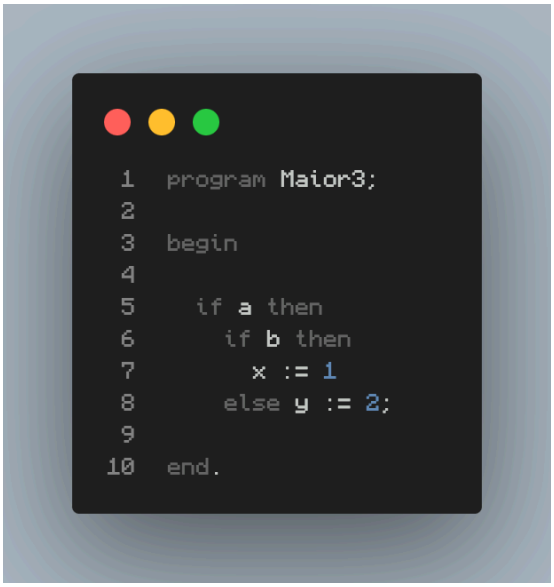
```
1  program HelloWorld;
2  begin
3    writeln('Ola, Mundo!'
4  end.
```



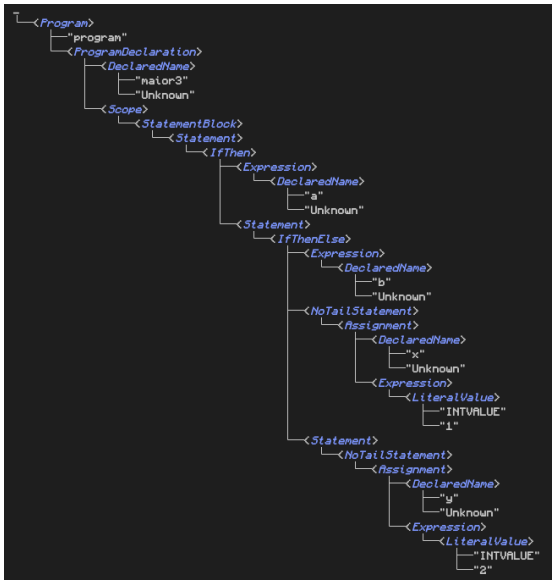
```
(.venv) ➜  PL2425 git:(main) ✗ python3 src/parser.py inputs/helloworld.pas
WARNING: Token 'CONST' defined, but not used
WARNING: Token 'REPEAT' defined, but not used
WARNING: Token 'RETURN' defined, but not used
WARNING: Token 'UNTIL' defined, but not used
WARNING: There are 4 unused tokens
Generating LALR tables

[Ln 4] Syntax error: Unexpected 'end' (type END). Expected: COMMA, RPAREN
    end.
    ^
No partial tree available (parsing didn't start)
(.venv) ➜  PL2425 git:(main) ✗
```

Situação de Erro



```
1  program Maior3;
2
3  begin
4
5    if a then
6      if b then
7        x := 1
8      else y := 2;
9
10 end.
```



```
└─<Program>
    ├─"program"
    └─<ProgramDeclaration>
        ├─<DeclaredName>
        │   ├─"maior3"
        │   └─"Unknown"
        └─<Scope>
            └─<StatementBlock>
                └─<Statement>
                    └─<IfThen>
                        ├─<Expression>
                        │   └─<DeclaredName>
                        │       ├─"a"
                        │       └─"Unknown"
                        └─<Statement>
                            └─<IfThenElse>
                                ├─<Expression>
                                │   └─<DeclaredName>
                                │       ├─"b"
                                │       └─"Unknown"
                                ├─<NoTailStatement>
                                │   └─<Assignment>
                                │       ├─<DeclaredName>
                                │       │   ├─"x"
                                │       │   └─"Unknown"
                                │       └─<Expression>
                                │           └─<LiteralValue>
                                │               ├─"INTVALUE"
                                │               └─"1"
                                └─<Statement>
                                    └─<NoTailStatement>
                                        └─<Assignment>
                                            ├─<DeclaredName>
                                            │   ├─"y"
                                            │   └─"Unknown"
                                            └─<Expression>
                                                └─<LiteralValue>
                                                    ├─"INTVALUE"
                                                    └─"2"
```

Situação Correta

3. Análise Semântica

A análise semântica garante que a AST gerada pela fase sintática obedeça às regras de significado da linguagem Pascal ao mesmo tempo que é gerado o seu código EWVM.

3.1. Chamada de Funções Builtin

Algumas funções builtin foram implementadas em linguagem EWVM, como `readln`, `writeln`, `length`, etc. A função `writeln`, por sua vez, pode receber diversos parâmetros na sua chamada, que empilha na stack todos eles por ordem inversa. Após isso, de acordo com cada parâmetro que foi chamado, é verificado o seu tipo e gerada a respectiva instrução para impressão daquele tipo (funções `WRITES`, `WRITEI` e `WRITECHR`). Caso um parâmetro não permitido é passado como argumento, uma exceção é lançada e a compilação é interrompida. De forma similar, as funções declaradas no programa também fazem uma verificação de tipos quando chamadas.

3.2. Arrays

Em Pascal a definição de arrays estáticos se dá pela declaração do seu `lower_bound` e do `upper_bound`, como exemplo:

```
numeros: array[5..9] of integer
```

Isso demonstra como a declaração dos índices não segue o padrão *0-based*, onde o primeiro elemento pode ser acessado com `numeros[5]`. Sendo assim, cabe ao `Generator` calcular o seu índice correspondente ao *0-based*.

De outra forma, as strings começam pelo índice 1. Portanto, toda vez que encontramos um *Node* `ArrayAccess` precisamos verificar o seu tipo de retorno. Caso corresponda a um array estático - de inteiros por exemplo - o seu índice é transformado em runtime para o padrão *0-based*. Caso corresponda a uma string, apenas substrai-se 1 do índice de acesso.

3.3. Operações Boolean

Em operações Boolean (`if`, `while`, `for`, `and`, `or`, etc.) também são verificados os seus tipos.

Por exemplo: no código `a = b` só é possível caso o `a` e `b` possuam os mesmos tipos. Caso contrário, o programa deverá lançar um erro de semântica.

4.1. Compilar

A fase de geração de código é responsável por traduzir a árvore sintática abstrata (AST) gerada no Cap. 2 e validada semanticamente, em instruções para a Máquina Virtual. Nesta etapa convertemos recursivamente cada instrução em Pascal em uma sequência de instruções EWVM que, quando executadas, reproduzem o comportamento do programa original.

Os principais objetivos dessa fase são:
1. Mapear cada *Node* da AST para instruções EWVM equivalentes
2. Controlar fluxo de execução: emitir saltos condicionais, criar labels únicas para cada fluxo *if*, *while* ou *for*
3. Lidar com escopos.
4. Suportar funções e procedures.

A estrutura do código que lida com todos esses aspetos é representada pelo `generator.py`, `emitter.py` e `scope.py`.

4.2. Generator

Seu propósito é percorrer recursivamente a AST e, para cada nó, chamar um método específico `gen_<label>` que emite instruções EWVM.

```
def generate(self, node: Node):
    method = getattr(self, f"gen_{node.label}", self.not_implemented)
    return method(node)

def gen_Program(self, node):
    self.generate(node.args[1])

def gen_StatementBlock(self, node):
    for stmt in node.args:
        self.generate(stmt)

# exemplo de geração de código da operação de soma
def gen_AddExpression(self, node: Node):
    left_node = node.args[0]
    right_node = node.args[2]

    self.generate(left_node)
    self.generate(right_node)
    self.em.emit("ADD")
```

A classe `CodeGenerator` utilza uma outra classe auxiliar `CodeEmitter`, que abstrai o processo de escrita do código de output.

Ao final do processo, toda a árvore foi percorrida e o compilador cria um ficheiro de output `<nome_do_programa>.out` com o código em linguagem EWVM completo e alguns comentários para auxliar o processo de debug.

Programa `helloworld.pas`:

Pascal:

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

Output:

```
JUMP main
main:
START
// literal value Ola, Mundo!
PUSHS "Ola, Mundo!"
// builtin function writeln
WRITES
WRITELN
STOP
```

4.3. Emitter

A classe `CodeEmitter` é responsável por abstrair a escrita final das instruções EWVM para o ficheiro de saída. Em vez de espalhar chamadas diretas a print ou a manipular buffers de strings em todo o gerador, todo o código EWVM é acumulado dentro de `CodeEmitter`

A API básica do `CodeEmitter` inclui algumas funções de emissão de código, criação de labels únicas e agrupamento do código.

```
class CodeEmitter:
    def __init__(self):
        self.code = []
        self.label_count = 0

    def new_label(self, base="L"):
        new_label = f"{base}{self.label_count}"
        self.label_count += 1
        return new_label

    def emit(self, instruction: str):
        self.code.append(instruction)

    def dump(self):
        return "\n".join(self.code)
```

4.4. Scope

Para gerir corretamente variáveis e funções em diferentes níveis (global e local), implementámos uma classe `Scope` em `scope.py`. Ela mantém uma pilha de dicionários de símbolos, permitindo – ao entrar em funções ou blocos aninhados – "empilhar" um novo escopo, e ao sair, "desempilhar" esse escopo

Sua API também bastante simples contém metódos para *push* e *pop* de um scope e funções de declaração e procura de variáveis:

```
class Scope:
  def __init__(self):
      self.scopes = list()
      self.scopes.append({'label': 'global', 'next_index': 0, 'vars': dict()})

  def push(self):
      self.scopes.append({'label': 'local', 'next_index': 0, 'vars': dict()})

  def pop(self):
    if len(self.scopes) == 1:
      raise Exception("Já no escopo global; não há escopo para sair.")
    self.scopes.pop()

  def declare(self, name):
    current_scope = self.scopes[-1]
    if name in current_scope['vars']:
      raise Exception(f"Variable '{name}' already declared in current scope.")
    idx = current_scope['next_index']
    current_scope['vars'][name] = idx
    current_scope['next_index'] += 1
    return idx

  def lookup(self, name) -> int | None:
    for scope in reversed(self.scopes):
      if name in scope['vars']:
        return scope['vars'][name]
    return None
```

5.1. Function Calls

Ao chamar funções em Pascal, é alocado na stack todos os seus parâmetros por ordem inversa e também um slot para o retorno da função. Em seguida, ao efetuar a instrução `CALL` na linguagem EWVM, para cada parâmetro passado é realizado um `LOAD <index>` para empilhar os parâmetro passados no novos stack frame. Porém, quando a função é terminada e o seu retorno é efetuado, esses `LOAD`s todos são deixados na stack, deixando "lixo" empilhado. É onde a função de clean-up é chamada.

Antes de efetivamente realizar o `RETURN`, a função é responsável por "limpar" para retirar todos os parâmetros empilhados, realizando um `POP` para cada parâmetro