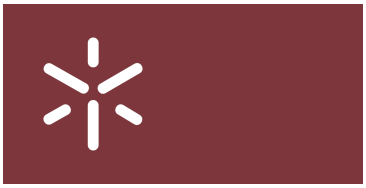

Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Unidade Curricular de Processamento de Linguagens

Ano Letivo de 2024/2025

Compilador de Pascal

Manuel Fernandes
A93213

Rúben Oliveira
A93625

June 01, 2025

PL

Índice

1. Introdução

1.1. Contextualização

Este projeto consiste na implementação de um compilador para uma língua de programação de nome *Pascal*, utilizando as ferramentas PLY (*Python Lex-Yacc*) do *Phyton* para análise léxica e sintática. Neste relatório está processado da seguinte maneira, uma pequena introdução com o analisador sintático e como está construido e que tipo de que algoritmo usa. De seguida é apresentado o analisador léxico, como está construido e como também as expressões regulares, por fim amostrar-mos, resultados da utilização do Compilador. O mesmo processa construções básicas de Pascal como declarações de variáveis, estruturas de controle, ciclos, expressões aritméticas.

1.2. Objetivos

Implementar analisador léxico com reconhecimento de *tokens* e um analisador sintático:

- Definir gramática livre de contexto para a sublinguagem
- Implementar analisador sintático baseado na gramática
- Realizar geração de código intermediário
- Testar com programas exemplo que cubram todas funcionalidades

2. Desenvolvimento do Trabalho Prático

2.1. Analizador Sintático

2.1.1. Algoritmo

O algoritmo utilizado, é o LR(*Left/Right*), este algoritmo, lê a entrada de texto da esquerda para a direita e produz uma derivação mais à direita, é ascendente (*bottom-up*) pois ele tenta deduzir as produções da gramática a partir dos nós folha da árvore, são muito eficientemente, o que é muito útil para compiladores.

2.1.2. Corpo do programa

Aqui temos a gramática da estrutura para o corpo, esta gramática que está implementada no ficheiro ***pascal_sin.py:***

```
def p_program(p):
    '''program : PROGRAM ID PONTO_VIRGULA var_declaration_part BEGIN body
END PONTO'''
    p[0] = ('program', p[2], p[4], p[6][1])

def p_var_declaration_part(p):
    '''var_declaration_part : VAR var_declaration_list
                            | empty'''
    p[0] = p[2] if len(p) > 2 else []

def p_var_declaration_list_single(p):
    'var_declaration_list : var_declaration'
    p[0] = [p[1]]

def p_var_declaration_list_no_semicolon(p):
    'var_declaration_list : var_declaration_list var_declaration'
    p[0] = p[1] + [p[2]]

def p_var_declaration_list_with_semicolon(p):
    'var_declaration_list : var_declaration_list PONTO_VIRGULA
var_declaration'
    p[0] = p[1] + [p[3]]

def p_var_declaration(p):
    '''var_declaration : id_list DOISPONTOS type PONTO_VIRGULA'''
    for var_id in p[1]:
        if var_id in symbol_table:
            raise SyntaxError(f"Variável '{var_id}' já declarada")
        symbol_table[var_id] = p[3]
    p[0] = ('var_declaration', p[1], p[3])
```

```
def p_id_list(p):
    '''id_list : ID
               | id_list VIRGULA ID'''
    p[0] = [p[1]] if len(p) == 2 else p[1] + [p[3]]

def p_body(p):
    'body : statements'
    p[0] = ('body', p[1])
```

Como vemos, a gramática começa com *PROGRAM* que é o começo sintático dos programas de *Pascal* de seguida temos o *TOKEN ID* que é do nome do programa,este *TOKEN* permite que um certo tipo de texto seja introduzido respeitando as regras das expressões regulares, este código tem um erro que foi visto um programa cíclico *WHILE.PAS*. Vimos que se dermos um nome ao programa *Pascal* de alguma palavra da gramática(*while*), não reconhece, como *ID* mas como uma palavra reservada, devido há falta de tempo não houve possibilidade de corrigir tal erro. em seguida temos o *TOKEN* PONTO_VIRGULA, e então temos a declaração das variáveis que vai para função *var_declaration_part*, que vê se necessita ou não, quando necessitar vê as várias variáveis do mesmo tipo separadas por virgulas ou só uma de um tipo, só depois é que começa corpo da função com os vários *Statements*(*BEGIN, IFS,* etc.) e no final verifica se tem o END e se acaba com o ponto e não ponto e virgula.

2.1.3. STATEMENTS

```
def p_statement(p):
    statement : assign
              | writeln
              | write
              | readln
              | read
              | if
              | case
              | while
              | for
              | function
              | procedure
              | func_call
    p[0] = p[1]
```

2.1.4. READ e WRITES

```
#WRITE
def p_writenln(p):
    'writeln : WRITELN "(" Args ")" PONTO_VIRGULA '
    p[0] = ('writeln', p[3])

def p_writen(p):
    'write : WRITE "(" Args ")" PONTO_VIRGULA '
    p[0] = ('write', p[3])

#READ
def p_readln(p):
    'readln : READLN "(" Args ")" PONTO_VIRGULA'
    p[0] = ('readln', p[3])

def p_read(p):
    'read : READ "(" Args ")" PONTO_VIRGULA'
    p[0] = ('read', p[3])

#Dados para read e write
def p_parametro(p):
    'Args : Args VIRGULA Arg'
    p[0] = p[1] + [p[3]]

def p_args_single(p):
    'Args : Arg'
    p[0] = [p[1]]

def p_arg(p):
    Arg : const
           | var
    p[0] = p[1]
```

2.1.5. IFS

```
def p_statement_if(p):
    if : IF expression THEN statements
       | IF expression THEN statements ELSE statements
    if len(p) == 5:
        p[0] = ('if', p[2], p[4])
    else:
        p[0] = ('else', p[2], p[4], p[6])
```

2.1.6. CASES

```
def p_statement_case(p):
    '''case : CASE expression OF case_branches END PONTO_VIRGULA
            | CASE expression OF case_branches ELSE statements END
PONTO_VIRGULA'''
    p[0] = ('case', p[2], p[4]) if len(p) == 7 else ('case', p[2], p[4],
p[6])

def p_case_branches(p):
    '''case_branches : case_branches case_branch
                     | case_branch'''
    p[0] = p[1] + [p[2]] if len(p) == 3 else [p[1]]

def p_case_branch(p):
    'case_branch : case_labels DOISPONTOS statement'
    p[0] = ('case_branch', p[1], p[3])

def p_case_labels(p):
    '''case_labels : case_labels VIRGULA const
                   | const'''
    p[0] = p[1] + [p[3]] if len(p) == 4 else [p[1]]
```

2.1.7. WHILES

```
  def p_ciclo_while(p):
      '''while : WHILE expression DO compound_statement PONTO_VIRGULA
               | WHILE "(" expression ")" DO compound_statement
  PONTO_VIRGULA'''
      p[0] = ('while', p[3], p[6]) if len(p) == 8 else ('while', p[2],
  p[4])
```

2.1.8. FOR

```
  def p_ciclo_for(p):
      for : FOR ID ASSIGN expression TO expression DO compound_statement
  PONTO_VIRGULA
              | FOR ID ASSIGN expression DOWNTO expression DO
  compound_statement PONTO_VIRGULA
      direction = p[5].lower()
      p[0] = ('for', direction, p[2], p[4], p[6], p[8])
```

2.1.9. FUNCTION

```
        def p_func_decl(p):
            'function : FUNCTION ID "(" params ")" DOISPONTOS type
    PONTO_VIRGULA var_declaration_part BEGIN body END PONTO_VIRGULA'
            p[0] = ('function', p[2], p[4], p[7], p[9], p[11])

        def p_func_call(p):
        func_call : ID "(" Args ")"
                    | BUILTIN_FUNC "(" Args ")"
        p[0] = ('func_call', p[1], p[3])
```

2.1.10. PROCEDURES

```
        def p_proc_decl(p):
            'procedure : PROCEDURE ID "(" params ")" PONTO_VIRGULA
        var_declaration_part BEGIN body END PONTO_VIRGULA'
            p[0] = ('procedure', p[2], p[4], p[7], p[9])
```

2.1.11. ARRAYS

```
        def p_array_type(p):
            array_type : ARRAY "[" NUM DOUBLEPOINTS NUM "]" OF type
                       | ARRAY "[" NUM "]" OF type
            if len(p) == 9:
                p[0] = ('array', p[3], p[5], p[8])
            else:
                p[0] = ('array', p[3], p[3], p[6])
```

2.1.12. Atribuições

```
        def p_statement_assign(p):
            'assign : var ASSIGN expression PONTO_VIRGULA'
            p[0] = ('assign', p[1], p[3])

            def p_type(p):
            '''type : INTEGER
                    | REAL
                    | STRING_TYPE
                    | BOOLEAN
                    | array_type'''
            p[0] = p[1].lower() if isinstance(p[1], str) else p[1]
```

2.1.13. Comparações e Expressões Aritméticas

```
def p_expression(p):
    'expression : expression OR and_expr'
    p[0] = ('or', p[1], p[3])

def p_expression_base(p):
    'expression : and_expr'
    p[0] = p[1]

def p_and_expr(p):
    'and_expr : and_expr AND not_expr'
    p[0] = ('and', p[1], p[3])

def p_and_expr_base(p):
    'and_expr : not_expr'
    p[0] = p[1]

def p_not_expr_not(p):
    'not_expr : NOT not_expr'
    p[0] = ('not', p[2])

def p_not_expr_base(p):
    'not_expr : rel_expr'
    p[0] = p[1]

def p_rel_expr(p):
    '''rel_expr : arith_expr rel_op arith_expr
                | arith_expr'''
    p[0] = ('rel_op', p[2], p[1], p[3]) if len(p) == 4 else p[1]

def p_rel_op(p):
    '''rel_op : IGUAL
              | DIFF
              | MAIOR
              | MENOR
              | MENOR_IGUAL
              | MAIOR_IGUAL'''
    p[0] = p[1]

def p_arith_expr(p):
    'arith_expr : arith_expr add_op term'
    p[0] = ('bin_op', p[2], p[1], p[3])
```

2.2. Análise Léxica

Implementada em ***pascal_lex.py:***

Tokens reconhecidos:

- Palavras reservadas (PROGRAM, VAR, BEGIN, END, etc.)
- Identificadores (ID)
- Números inteiros e reais (NUM)
- Strings (STRING)
- Operadores (+, -, *, /, =, <>, >, <, etc.)
- Símbolos (;, :, ,, ., (,), [,])

Características principais:

- Case-insensitive para palavras reservadas
- Suporte a comentários com { ... }
- Tratamento de números inteiros e reais
- Strings delimitadas por aspas simples
- Contagem de linhas para mensagens de erro

2.3. Gramática e Análise Sintática

Implementada em **pascal_sin.py:**

Estrutura principal:

- program : PROGRAM ID ';' var_declaration_part BEGIN body END '.'

Declarações:

- var_declaration_part : VAR var_declaration_list | empty
- var_declaration_list : var_declaration | var_declaration_list ',' var_declaration
- var_declaration : id_list ':' type ';'

Comandos:

- statement : assign | writeln | read | if | case | while | for | function | procedure
- body : statements
- statements : statements statement | statement

Expressões:

- expression : expression OR and_expr | and_expr
- and_expr : and_expr AND not_expr | not_expr
- not_expr : NOT not_expr | rel_expr
- rel_expr : arith_expr rel_op arith_expr | arith_expr
- arith_expr : arith_expr add_op term | term
- term : term mul_op factor | factor

3. Testes

3.1. Hello World (helloworld.pas)

Código:

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

Árvore Sintática gerada:

- program, HelloWorld, [],
  - body,
    - writeln, [‘Ola, Mundo!’]

```
Código Assembly Gerado:
// BEGIN PROGRAM HelloWorld
pushs "Ola, Mundo!"  // push string
writes
writeln
stop
// END PROGRAM HelloWorld
```

Resultado: Funcionamento correto de saída básica de strings.

3.2. Variáveis e Saída (adiciona.pas)

Código:

```
program HelloWorld;
var
    num1: integer;
begin
    num1 := 2;
    writeln('O numero é:' , num1);
end.
```

Árvore Sintática gerada:

- “program”, “Teste”,
  - “var_declaration”, [“x”, “y”], “integer”

,

  - “assign”, [“var”, “x”], [“const”, 5]
  - “assign”, [“var”, “y”], [“bin_op”, “+”, [“var”, “x”], [“const”, 3]]
  - “writeln”, [“var”, “x”], [“var”, “y”]

```
Código Assembly Gerado:
// BEGIN PROGRAM
// var num1 at offset 0
// var num2 at offset 1
pushi 2
storeg 0
pushs "O numero é:"
writes
pushg 0
writeln
stop
// END PROGRAM
```

Resultado: Funcionamento correto de atribuição de valores e concatenação em output.

3.3. Fatorial (fatorial.pas)

Código:

```
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.
```

```
Árvore Sintática gerada:
('program',
 'Fatorial',
 [('var_declaration', ['n', 'i', 'fat'], 'integer')],
 [
  ('writeln', [('const', 'Introduza um número inteiro positivo:')]),
  ('readln', [('var', 'n')]),
  ('assign', ('var', 'fat'), ('const', 1)),
  ('for',
   'to',
   'i',
   ('const', 1),
   ('var', 'n'),
   [
    ('assign',
     ('var', 'fat'),
     ('bin_op', '*', ('var', 'fat'), ('var', 'i')))
   ]),
  ('writeln',
   [('const', 'Fatorial de '),
    ('var', 'n'),
```

```
      ('const', ': '),
      ('var', 'fat')])
   ]
  )
```

```
Código Assembly gerado:
pushs "Introduza um número inteiro positivo:" // string "Introduza um
número inteiro positivo:"
writes
read
atoi
storeg 0            // store n
pushi 1             // const 1
storeg 2            // store fat
pushi 1             // const 1
storeg 1            // init i
L1:
pushg 1             // for i
pushg 0             // var n
supeq               // compare
jz L2               // exit loop
pushg 2             // var fat
pushg 1             // var i
mul                 // * operation
storeg 2            // store fat
pushg 1
pushi 1
add                 // step
storeg 1            // update i
jump L1
L2:
pushs "Fatorial de " // string "Fatorial de "
writes
pushg 0             // var n
writei
pushs ": "          // string ": "
writes
pushg 2             // var fat
writei
stop                // end of program
```

Resultado: Funcionamento correto de Loop for e operações aritméticas.

3.4. Número Primo (numPrimo.pas)

Código:

```
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.
```

Resultado: Funcionamento errado pois o analisador léxico não tem palavras reservadas para True e False,apesar de ter o tipo Boolean.

3.5. Arrays e Soma (soma.pas)

Código:

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

Resultado: Funcionamento errado devido a Acesso de Arrays

3.6. Estrutura CASE (testeCase.pas)

Código:

```
program testeCase;
var
  x: Integer;
begin
  x := 2;
  case x of
    1: WriteLn('Um');
    2,3: WriteLn('dois e três');
  else
    WriteLn('Outro número');
  end;
end.
```

Árvore Sintática gerada:

- “program”, “testeCase”,
  - “var_declaration”, [“x”], “integer”

  ,
  - “assign”, [“var”, “x”], [“const”, 2]
  - “case”, [“var”, “x”],
    - “case_branch”, [[“const”, 1]], [“writeln”, [[“const”, “Um”]]]
    - “case_branch”, [[“const”, 2], [“const”, 3]], [“writeln”, [[“const”, “dois e três”]]]

      ,
    - “writeln”, [[“const”, “Outro número”]]

Código Gerado Assembly:

```
pushi 2              // const 2
storeg 0             // store x
pushg 0              // var x
dup
pushi 1              // const 1
equal
jz next_case_0       // check case 1
jump L2
next_case_0:
dup
pushi 2              // const 2
equal
jz next_case_1       // check case 2
jump L3
next_case_1:
dup
pushi 3              // const 3
equal
jz next_case_1       // check case 3
```

```
jump L3
next_case_1:
pop                   // no case matched
pushs "Outro número" // string "Outro número"
writes
jump L1
L2:                   // case branch 0
pop
pushs "Um"            // string "Um"
writes
jump L1
L3:                   // case branch 1
pop
pushs "dois e três"  // string "dois e três"
writes
jump L1
L1:                   // end case
stop                  // end of program
```

Resultado: Funcionamento correto de Multiplos labels e cláusula else.

3.7. While Loop (while.pas)

Código:

```
program whileLoop;
var
   a: integer;
begin
   a := 10;
   writeln('value of a: ', a);
   a := a + 1;
   writeln('value of a: ', a);
end.
```

Árvore Sintática gerada:

- “program”, “Loop”,
  - “var_declaration”, [“a”], “integer”

  ,
  - “assign”, [“var”, “a”], [“const”, 0]
  - “while”, [“rel_op”, “<”, [“var”, “a”], [“const”, 0]],
    - “writeln”, [[“const”, “value of a: “], [“var”, “a”]]
    - “assign”, [“var”, “a”], [“bin_op”, “+”, [“var”, “a”], [“const”, 1]]

```
Código Assembly Gerado:
pushi 0               // const 0
storeg 0              // store a
L1:                   // while start
pushg 0               // var a
pushi 0               // const 0
inf                   // < comparison
```

```
jz L2                // exit loop
pushs "value of a: " // string "value of a: "
writes
pushg 0              // var a
writei
pushg 0              // var a
pushi 1              // const 1
add                  // + operation
storeg 0             // store a
jump L1
L2:                  // while end
stop                 // end of program
```

Resultado: Funcionamento correto de Atribuições e operações aritméticas básicas.

3.8. Conversão Binário (binDec.pas) + (binDecF.pas)

Devido à falta de suporte para funções de string como length no compilador implementado, este programa não pode ser executado. A funcionalidade de strings é limitada a operações básicas.

3.9. Maior de Três Números (maior3.pas)

Código:

```
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.
```

Árvore Sintática gerada:

- “program”, “Maior3”,
  - “var_declaration”, [“num1”, “num2”, “num3”, “maior”], “integer”

  ,
  - “writeln”, [[“const”, “Introduza o primeiro número: “]]
  - “readln”, [[“var”, “num1”]]
  - “writeln”, [[“const”, “Introduza o segundo número: “]]
  - “readln”, [[“var”, “num2”]]
  - “writeln”, [[“const”, “Introduza o terceiro número: “]]
  - “readln”, [[“var”, “num3”]]
  - “if”, [“rel_op”, “>”, [“var”, “num1”], [“var”, “num2”]], [“if”, [“rel_op”, “>”, [“var”, “num1”], [“var”, “num3”]], [“assign”, [“var”, “maior”], [“var”, “num1”]]]
  - “writeln”, [[“const”, “O maior é: “], [“var”, “maior”]]

```
Código Assembly Gerado:
pushs "Introduza o primeiro número: " // string "Introduza o primeiro
número: "
writes
read
atoi
storeg 0             // store num1
pushs "Introduza o segundo número: " // string "Introduza o segundo
número: "
writes
read
atoi
```

```
  storeg 1              // store num2
  pushs "Introduza o terceiro número: " // string "Introduza o terceiro
  número: "
  writes
  read
  atoi
  storeg 2              // store num3
  pushg 0               // var num1
  pushg 1               // var num2
  sup                   // > comparison
  jz L1                 // if condition false, jump to else
  pushg 0               // var num1
  pushg 2               // var num3
  sup                   // > comparison
  jz L2                 // if condition false, jump to else
  pushg 0               // var num1
  storeg 3              // store maior
  L2:                   // end if
  L1:                   // end if
  pushs "O maior é: "   // string "O maior é: "
  writes
  pushg 3               // var maior
  writei
  stop                  // end of program
```

Resultado: Funcionamento correto de Condicionais aninhadas.

3.10. Teste CASE Simples (testeCase2.pas)

Código:

```
program TesteCase;
var
  x: integer;
begin
  x := 2;
  case x of
    1: writeln('Um');
    2: writeln('Dois ou três');
  else
    writeln('Outro número');
  end;
end.
```

```
Código Assembly Gerado:
Código Assembly gerado:
start
pushs "Introduza o primeiro número: "
writes
read
atoi
storeg 0
```

```
pushs "Introduza o segundo número: "
writes
read
atoi
storeg 1
pushs "Introduza o terceiro número: "
writes
read
atoi
storeg 2
  // incicio de ite
  // inicio de get variavel "num1"
pushg 0
  // fim de get variavel "num1"
  // inicio de get variavel "num2"
pushg 1
  // fim de get variavel "num2"
sup
jz ELSE0
  // incicio de ite
  // inicio de get variavel "num1"
pushg 0
  // fim de get variavel "num1"
  // inicio de get variavel "num3"
pushg 2
  // fim de get variavel "num3"
sup
jz ELSE2
  // inicio de get variavel "num1"
pushg 0
  // fim de get variavel "num1"
storeg 3
jump ENDIF3
ELSE2:
  // inicio de get variavel "num3"
pushg 2
  // fim de get variavel "num3"
storeg 3
ENDIF3:
  // fim de ite
jump ENDIF1
ELSE0:
  // incicio de ite
  // inicio de get variavel "num2"
pushg 1
  // fim de get variavel "num2"
  // inicio de get variavel "num3"
pushg 2
  // fim de get variavel "num3"
sup
jz ELSE4
  // inicio de get variavel "num2"
```

```
pushg 1
  // fim de get variavel "num2"
storeg 3
jump ENDIF5
ELSE4:
  // inicio de get variavel "num3"
pushg 2
  // fim de get variavel "num3"
storeg 3
pushs "O maior é: "
writes
  // inicio de get variavel "maior"
pushg 3
  // fim de get variavel "maior"
writei
ENDIF5:
  // fim de ite
ENDIF1:
  // fim de ite
stop
```

Resultado: Funcionamento correto de Estrutura com um único label.

4. Conclusão

Os testes demonstram que o compilador implementado é capaz de lidar com:

- Estruturas básicas: variáveis, loops, condicionais
- Tipos de dados: inteiros, booleanos, arrays
- Operações: aritméticas, lógicas e comparação
- E/S: entrada de dados e output formatado

Limitações identificadas:

- Manipulação avançada de strings (indexação, funções como length)
- O programa binDec.pas não pode ser executado devido à falta de suporte para operações avançadas de strings.

Melhorias futuras:

- Implementar funções para manipulação de strings
- Adicionar suporte a registros (records)
- Aprimorar análise semântica com checagem de tipos
- Expandir conjunto de operadores (div, mod)

O sistema compilador cumpre seus objetivos principais para programas Pascal básicos, servindo como base sólida para expansões futuras.