Processamento de Linguagens

Construção de um Compilador para Pascal Standard

Artur Luís A95414
Henrique Pereira A97205
Carlos Pina A95349

Universidade do Minho

1 de Junho de 2025

Índice

1. Introdução

Este trabalho consiste no desenvolvimento de um compilador para a linguagem Pascal Standard, no âmbito da disciplina de Processamento de Linguagens. O objetivo é criar um compilador funcional capaz de processar código Pascal e convertê-lo para código executável numa máquina virtual.

O desenvolvimento do compilador segue as seguintes etapas: análise léxica, análise sintática, análise semântica e geração de código. O compilador final processa código Pascal e gera código executável para uma máquina virtual.

Este relatório apresenta a implementação realizada e os testes efetuados.

2. Etapas do Projeto

2.1. Análise Léxica

O analisador léxico é a primeira etapa do processo de compilação.

O *lexer* desenvolvido neste projeto foi implementado em Python, recorrendo à biblioteca PLY (Python Lex-Yacc).

A sua principal função é ler o código escrito em Pascal e convertê-lo numa sequência de tokens, fundamentais para a análise sintática posterior. São identificados elementos como palavras reservadas, identificadores, operadores, literais e símbolos especiais, descartando comentários e espaços em branco.

2.1.1. Tokens

Começámos por definir todos os tokens relevantes, incluindo palavras reservadas, identificadores, números, strings, operadores, símbolos de pontuação e comentários.

```
tokens = (
    # Palavras reservadas com base em er a pedido do prof. prh
    'PROGRAM', 'VAR', 'INTEGER', 'BOOLEAN', 'BEGIN', 'END', 'IF', 'THEN', 'ELSE',
    'WHILE', 'FOR', 'TO', 'DOWNTO', 'DO', 'READLN', 'WRITELN', 'WRITE', 'TRUE', 'FALSE',
    'AND', 'OR', 'NOT', 'DIV', 'MOD', 'ID', 'NUMBER', 'STRING', 'ASSIGN', 'EQUAL',
    'NOTEQUAL', 'LT', 'GT', 'LE', 'GE', 'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'MODOP',
    'COMMA', 'COLON', 'SEMI', 'DOT', 'LPAREN', 'RPAREN', 'COMMENT'
)
```

As **palavras reservadas** são reconhecidas através de expressões regulares que aceitam letras maiúsculas e minúsculas, garantindo compatibilidade com diferentes estilos de escrita.

Os **identificadores** são reconhecidos por uma expressão regular que aceita letras, dígitos e underscores.

Os **números** são reconhecidos como sequências de dígitos e convertidos para inteiros.

O reconhecimento de **strings** permite tanto aspas simples como duplas.

Definimos tokens nomeados para todos os **operadores** e **símbolos**.

Os **comentários** são ignorados pelo lexer e podem ser escritos nos três formatos aceites em Pascal: **{** ... **}**, **(*** ... ***)** e //... .

Espaços, **tabs** e outras **identações compatíveis com o sistema Windows** são ignorados.

Sempre que um **carácter ilegal** é encontrado, o analisador emite uma mensagem de erro indicando a linha do problema.

```
def t_FOR(t):       r'[Ff][Oo][Rr]'                ; return t
def t_TO(t):        r'[Tt][Oo]'                    ; return t
def t_DOWNTO(t):    r'[Dd][Oo][Ww][Nn][Tt][Oo]'    ; return t
def t_PROGRAM(t):   r'[Pp][Rr][Oo][Gg][Rr][Aa][Mm]' ; return t
def t_VAR(t):       r'[Vv][Aa][Rr]'                ; return t
def t_INTEGER(t):   r'[Ii][Nn][Tt][Ee][Gg][Ee][Rr]' ; return t
def t_BOOLEAN(t):   r'[Bb][Oo][Oo][Ll][Ee][Aa][Nn]' ; return t
def t_BEGIN(t):     r'[Bb][Ee][Gg][Ii][Nn]'        ; return t
def t_END(t):       r'[Ee][Nn][Dd]'                ; return t
def t_IF(t):        r'[Ii][Ff]'                    ; return t
def t_THEN(t):      r'[Tt][Hh][Ee][Nn]'            ; return t
def t_ELSE(t):      r'[Ee][Ll][Ss][Ee]'            ; return t
def t_WHILE(t):     r'[Ww][Hh][Ii][Ll][Ee]'        ; return t
def t_DO(t):        r'[Dd][Oo]'                    ; return t
def t_READLN(t):    r'[Rr][Ee][Aa][Dd][Ll][Nn]'    ; return t
def t_WRITELN(t):   r'[Ww][Rr][Ii][Tt][Ee][Ll][Nn]' ; return t
def t_WRITE(t):     r'[Ww][Rr][Ii][Tt][Ee]'        ; return t
def t_TRUE(t):      r'[Tt][Rr][Uu][Ee]'            ; t.value = 'true' ; return t
def t_FALSE(t):     r'[Ff][Aa][Ll][Ss][Ee]'        ; t.value = 'false'; return t
def t_AND(t):       r'[Aa][Nn][Dd]'                ; return t
def t_OR(t):        r'[Oo][Rr]'                    ; return t
def t_NOT(t):       r'[Nn][Oo][Tt]'                ; return t
def t_DIV(t):       r'[Dd][Ii][Vv]'                ; return t
def t_MOD(t):       r'[Mm][Oo][Dd]'                ; return t
def t_ID(t):        r'[a-zA-Z_][a-zA-Z0-9_]*'      ; return t
def t_NUMBER(t):    r'\d+'                         ; t.value = int(t.value); return t
def t_newline(t):   r'\n+'                         ; t.lexer.lineno += len(t.value)
def t_error(t):     print(f"Illegal character '{t.value[0]}' at line {t.lineno}"); t.lexer.skip(1)
```

```
# Operadores
t_ASSIGN = r':='
t_EQUAL = r'='
t_NOTEQUAL = r'<>'
t_LT = r'<'
t_GT = r'>'
t_LE = r'<='
t_GE = r'>='
t_PLUS = r'\+'
t_MINUS = r'-'
t_TIMES = r'\*'
t_DIVIDE = r'/'
t_MODOP = r'mod'
t_COMMA = r','
t_COLON = r':'
t_SEMI = r';'
t_DOT = r'\.'
t_LPAREN = r'\('
t_RPAREN = r'\)'
t_STRING = r'\"[^\"\n]*\"|\'[^\'\n]*\''
```

```
t_ignore = ' \t\r'

def t_COMMENT(t):
    r'\(\*[\s\S]*?\*\)|\{[\s\S]*?\}|//.*'
    t.lexer.lineno += t.value.count('\n')
    pass
```

2.2. Análise Sintática

Após a análise léxica, o passo seguinte no processo de compilação é a análise sintática.

Implementámos a análise sintática utilizando a biblioteca PLY (Python Lex-Yacc). Elaboramos todas as regras sintáticas relevantes para reconhecer as principais construções em Pascal.

O objetivo desta fase é verificar se a sequência de tokens produzida pelo analisador léxico segue as regras gramaticais da linguagem Pascal, construindo uma estrutura que representa a organização do programa.

2.2.1. Gramática

A gramática do nosso compilador Pascal foi desenhada para ser adequada ao funcionamento do gerador de parsers PLY (Python Lex-Yacc), que utiliza a abordagem **LALR(1)**. A nossa gramática é também independente de contexto e recursiva à esquerda.

A precedência e associatividade dos operadores são definidas para garantir que as operações são avaliadas na ordem correta:

```
precedence = (
    ('left', 'OR'),
    ('left', 'AND'),
    ('left', 'NOT'),
    ('nonassoc', 'LT', 'LE', 'GT', 'GE', 'EQUAL', 'NOTEQUAL'),
    ('left', 'PLUS', 'MINUS'),
    ('left', 'TIMES', 'DIVIDE', 'DIV', 'MOD'),
    ('right', 'UMINUS'),
)
```

- **Estrutura do Programa**

A função *p_program* reconhece a estrutura global do programa.

```
def p_program(p):
    '''program : PROGRAM ID SEMI declarations compound_stmt DOT
               | PROGRAM ID SEMI compound_stmt DOT'''
    init_code = [f"pushi 0 // alocar espaço para {i}" for i in range(current_address)]
    compound_code = p[5] if len(p) == 7 else p[4]
    if compound_code is None:
        compound_code = ""
    output_lines = init_code + ["start"] + [compound_code] + ["stop"]
    p[0] = "\n".join(line for line in output_lines if line)
```

Esta regra engloba programas com ou sem declarações de variáveis.

- **Declarações de Variáveis**

As variáveis são declaradas e registadas na tabela de símbolos.

```
def p_declarations(p):
    '''declarations : VAR var_decl_list
                    | empty'''
    p[0] = ""

def p_var_decl_list(p):
    '''var_decl_list : var_decl SEMI
                     | var_decl SEMI var_decl_list'''
    p[0] = ""

def p_var_decl(p):
    'var_decl : id_list COLON type'
    global current_address
    for var_name in p[1]:
        symbol_table[var_name] = current_address
        current_address += 1
    p[0] = ""

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

- **Blocos de Comandos e Listas**

O corpo principal do programa e blocos de comandos são reconhecidos.

```
def p_compound_stmt(p):
    'compound_stmt : BEGIN stmt_list END'
    p[0] = p[2] if p[2] else ""

def p_stmt_list(p):
    '''stmt_list : stmt
                 | stmt SEMI stmt_list'''
    if len(p) == 2:
        p[0] = p[1]
    else:
        p[0] = f"{p[1]}\n{p[3]}" if p[1] and p[3] else p[1] or p[3]
```

- **Comandos Principais**

Os comandos suportados incluem atribuições, condicionais, ciclos for e while, leitura, escrita e blocos:

```
def p_stmt(p):
    '''stmt : assignment
            | if_stmt
            | while_stmt
            | for_stmt
            | readln_stmt
            | writeln_stmt
            | compound_stmt
            | empty'''
    p[0] = p[1] if len(p) > 1 else ""
```

– **Atribuições**

Calcula a expressão e armazenar o resultado na variável.

```
def p_assignment(p):
    'assignment : ID ASSIGN expr'
    var_name = p[1]
    if var_name in symbol_table:
        addr = symbol_table[var_name]
        p[0] = f"{p[3]}\nstoreg {addr} // ir à stack para atribuir o valor a {var_name}"
    else:
        print(f"!!Erro: variável - {var_name} - não declarada!!")
        p[0] = ""
```

– **If**

Gera labels para saltos condicionais e código para os blocos then e else.

```
def p_if_stmt(p):
    '''if_stmt : IF expr THEN stmt
               | IF expr THEN stmt ELSE stmt'''
    if len(p) == 5:
        end_label = new_label("ifend")
        p[0] = f"{p[2]}\njz {end_label} // If condição falsa\n{p[4]}\n{end_label}:"
    else:
        else_label = new_label("else")
        end_label = new_label("ifend")
        p[0] = f"{p[2]}\njz {else_label} // If condição falsa\n{p[4]}\njump {end_label}\n{else_label}:\n{p[6]}\n{end_label}:"
```

– **While**

Gera labels para o início e fim do ciclo, e código para repetir enquanto a condição for verdadeira.

```
def p_while_stmt(p):
    'while_stmt : WHILE expr DO stmt'
    start_label = new_label("whileloop")
    end_label = new_label("whileend")
    p[0] = f"{start_label}:\n{p[2]}\njz {end_label} // While condição falsa\n{p[4]}\njump {start_label}\n{end_label}:"
```

– **For**

Gera código para inicializar, comparar, incrementar e repetir o corpo do ciclo.

```
def p_for_stmt(p):
    'for_stmt : FOR ID ASSIGN expr TO expr DO stmt'
    loop_var = p[2]
    global current_address
    if loop_var not in symbol_table:
        symbol_table[loop_var] = current_address
        current_address += 1
    addr_loop = symbol_table[loop_var]
    addr_bound = current_address
    current_address += 1
    loop_label = new_label("forloop")
    end_label = new_label("forend")
    init_code = f"{p[4]}\nstoreg {addr_loop} // inicializar {loop_var}\n"
    bound_code = f"{p[6]}\nstoreg {addr_bound} // limite do ciclo\n"
    start_code = f"{loop_label}:\n"
    cond_code = (
        f"pushg {addr_loop} // ir à stack para atribuir o valor a {loop_var}\n"
        f"pushg {addr_bound} // carrega o limite\n"
        "infeq // verifica a condição do ciclo\n"
        f"jz {end_label} // sai se: loop_var > bound\n"
    )
    body_code = f"{p[8]}\n"
    inc_code = (
        f"pushg {addr_loop} // carrega {loop_var}\n"
        "pushi 1 // Push 1\n"
        "add\n"
        f"storeg {addr_loop} // acaba logica da var do ciclo\n"
    )
    jump_code = f"jump {loop_label}\n"
    end_code = f"{end_label}:\n"
    p[0] = init_code + bound_code + start_code + cond_code + body_code + inc_code + jump_code + end_code
```

– **Leitura e Escrita**
Permitem ler valores para variáveis e imprimir valores ou strings.

```
def p_readln_stmt(p):
    'readln_stmt : READLN LPAREN ID RPAREN'
    var_name = p[3]
    if var_name in symbol_table:
        addr = symbol_table[var_name]
        p[0] = f"read\natoi\nstoreg {addr} // carregar {var_name}"
    else:
        print(f"!!Erro: variável - {var_name} - não declarada!!")
        p[0] = ""
```

```
def p_writeln_stmt(p):
    '''writeln_stmt : WRITELN LPAREN writeln_args RPAREN
                    | WRITE LPAREN writeln_args RPAREN
                    | WRITELN LPAREN RPAREN
                    | WRITE LPAREN RPAREN'''
    if len(p) == 5:
        if p[1].lower() == 'writeln':
            p[0] = f"{p[3]}\nWRITELN"
        else:
            p[0] = p[3]
    else:
        if p[1].lower() == 'writeln':
            p[0] = "WRITELN"
        else:
            p[0] = ""

def p_writeln_args(p):
    '''writeln_args : arg
                    | arg COMMA writeln_args'''
    if len(p) == 2:
        p[0] = p[1]
    else:
        p[0] = f"{p[1]}\n{p[3]}"

def p_arg(p):
    '''arg : expr
           | STRING'''
    if isinstance(p[1], str) and (p[1].startswith('"') or p[1].startswith("'")):
        s = p[1][1:-1]
        p[0] = f'pushs "{s}"\nwrites // escreve string'
    else:
        p[0] = f"{p[1]}\nwritei // escreve inteiro"
```

– **Expressões**
As expressões permitem operações aritméticas, relacionais e lógicas.

```
def p_expr(p):
    '''expr : simple_expr
            | simple_expr relop simple_expr'''
    if len(p) == 2:
        p[0] = p[1]
    else:
        op_map = {'=': 'equal', '<>': 'equal\nnot', '<': 'inf', '>': 'sup', '<=': 'infeq', '>=': 'supeq'}
        p[0] = f"{p[1]}\n{p[3]}\n{op_map[p[2]]} // lógica"

def p_relop(p):
    '''relop : EQUAL
             | NOTEQUAL
             | LT
             | GT
             | LE
             | GE'''
    p[0] = p[1]
```

```
def p_simple_expr(p):
    '''simple_expr : term
                   | simple_expr addop term'''
    if len(p) == 2:
        p[0] = p[1]
    else:
        op_map = {'+': 'add', '-': 'sub', 'or': 'or'}
        p[0] = f"{p[1]}\n{p[3]}\n{op_map[p[2]]} // maths"

def p_addop(p):
    '''addop : PLUS
             | MINUS
             | OR'''
    p[0] = p[1]
```

```
def p_term(p):
    '''term : factor
            | term mulop factor'''
    if len(p) == 2:
        p[0] = p[1]
    else:
        op_map = {'*': 'mul', '/': 'div', 'div': 'div', 'mod': 'mod', 'and': 'and'}
        p[0] = f"{p[1]}\n{p[3]}\n{op_map[p[2]]} // operação"

def p_mulop(p):
    '''mulop : TIMES
             | DIVIDE
             | DIV
             | MOD
             | AND'''
    p[0] = p[1]
```

```
def p_factor(p):
    '''factor : ID
              | NUMBER
              | LPAREN expr RPAREN
              | NOT factor
              | TRUE
              | FALSE
              | MINUS factor %prec UMINUS'''
    if len(p) == 2:
        if isinstance(p[1], int):
            p[0] = f"pushi {p[1]} // coloca numero na stack"
        elif p[1] == 'true':
            p[0] = f"pushi 1 // coloca booleano como true"
        elif p[1] == 'false':
            p[0] = f"pushi 0 // coloca booleano como false"
        else:  # ID
            var_name = p[1]
            if var_name in symbol_table:
                addr = symbol_table[var_name]
                p[0] = f"pushg {addr} // carrega da stack {var_name}"
            else:
                print(f"!!Erro: variável - {var_name} - não declarada!!")
                p[0] = ""
    elif len(p) == 3:
        if p[1] == 'not':
            p[0] = f"{p[2]}\nnot"
        elif p[1] == '-':
            p[0] = f"pushi 0\n{p[2]}\nsub"
    else:
        p[0] = p[2]
```

Estas funções geram o código intermédio correspondente a cada operação.

3. Testes

3.1. "Olá Mundo!"

Input:

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

Output:

```
start
pushs "Ola, Mundo!"
writes // escreve string
WRITELN
stop
```

3.2. "Maior de 3"

Input:

```
program Maior3;
var
   num1, num2, num3, maior: Integer;
begin
   { Ler 3 números }
   writeln('Introduza o primeiro número: ');
   readln(num1);
   writeln('Introduza o segundo número: ');
   readln(num2);
   write('Introduza o terceiro número: ');
   readln(num3);
   { Calcular o maior }
   if num1 > num2 then
      if num1 > num3 then maior := num1
      else maior := num3
      else
         if num2 > num3 then maior := num2
         else maior := num3;
   { Escrever o resultado }
   writeLn('O maior é: ', maior)
end.
```

Output:

```
pushi 0 // alocar espaço para 0
pushi 0 // alocar espaço para 1
pushi 0 // alocar espaço para 2
pushi 0 // alocar espaço para 3
start
pushs "Introduza o primeiro número: "
```

```
writes // escreve string
WRITELN
read
atoi
storeg 0 // carregar num1
pushs "Introduza o segundo número: "
writes // escreve string
WRITELN
read
atoi
storeg 1 // carregar num2
pushs "Introduza o terceiro número: "
writes // escreve string
WRITELN
read
atoi
storeg 2 // carregar num3
pushg 0 // carrega da stack num1
pushg 1 // carrega da stack num2
sup // lógica
jz else5 // If condição falsa
pushg 0 // carrega da stack num1
pushg 2 // carrega da stack num3
sup // lógica
jz else1 // If condição falsa
pushg 0 // carrega da stack num1
storeg 3 // ir à stack para atribuir o valor a maior
jump ifend2
else1:
pushg 2 // carrega da stack num3
storeg 3 // ir à stack para atribuir o valor a maior
ifend2:
jump ifend6
else5:
pushg 1 // carrega da stack num2
pushg 2 // carrega da stack num3
sup // lógica
jz else3 // If condição falsa
pushg 1 // carrega da stack num2
storeg 3 // ir à stack para atribuir o valor a maior
jump ifend4
else3:
pushg 2 // carrega da stack num3
storeg 3 // ir à stack para atribuir o valor a maior
ifend4:
ifend6:
pushs "O maior é: "
writes // escreve string
pushg 3 // carrega da stack maior
writei // escreve inteiro
WRITELN
stop
```

3.3. "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

```
pushi 0 // alocar espaço para 0
pushi 0 // alocar espaço para 1
pushi 0 // alocar espaço para 2
pushi 0 // alocar espaço para 3
start
pushs "Introduza um número inteiro positivo:"
writes // escreve string
WRITELN
read
atoi
storeg 0 // carregar n
pushi 1 // coloca numero na stack
storeg 2 // ir à stack para atribuir o valor a fat
pushi 1 // coloca numero na stack
storeg 1 // ir à stack para atribuir o valor a i
pushi 1 // coloca numero na stack
storeg 1 // inicializar i
pushg 0 // carrega da stack n
storeg 3 // limite do ciclo
forloop1:
pushg 1 // ir à stack para atribuir o valor a i
pushg 3 // carrega o limite
infeq // verifica a condição do ciclo
jz forend2 // sai se: loop_var > bound
pushg 2 // carrega da stack fat
pushg 1 // carrega da stack i
mul // operação
storeg 2 // ir à stack para atribuir o valor a fat
pushg 1 // carrega i
pushi 1 // Push 1
add
storeg 1 // acaba logica da var do ciclo
jump forloop1
forend2:

pushs "Fatorial de "
writes // escreve string
```

```
pushg 0 // carrega da stack n
writei // escreve inteiro
pushs ": "
writes // escreve string
pushg 2 // carrega da stack fat
writei // escreve inteiro
WRITELN
stop
```

3.4. "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:

```
pushi 0 // alocar espaço para 0
pushi 0 // alocar espaço para 1
pushi 0 // alocar espaço para 2
start
pushs "Introduza um número inteiro positivo:"
writes // escreve string
WRITELN
read
atoi
storeg 0 // carregar num
pushi 1 // coloca booleano como true
storeg 2 // ir à stack para atribuir o valor a primo
pushi 2 // coloca numero na stack
storeg 1 // ir à stack para atribuir o valor a i
whileloop2:
pushg 1 // carrega da stack i
pushg 0 // carrega da stack num
```

```
pushi 2 // coloca numero na stack
div // operação
infeq // lógica
pushg 2 // carrega da stack primo
and // operação
jz whileend3 // While condição falsa
pushg 0 // carrega da stack num
pushg 1 // carrega da stack i
mod // operação
pushi 0 // coloca numero na stack
equal // lógica
jz ifend1 // If condição falsa
pushi 0 // coloca booleano como false
storeg 2 // ir à stack para atribuir o valor a primo
ifend1:
pushg 1 // carrega da stack i
pushi 1 // coloca numero na stack
add // maths
storeg 1 // ir à stack para atribuir o valor a i
jump whileloop2
whileend3:
pushg 2 // carrega da stack primo
jz else4 // If condição falsa
pushg 0 // carrega da stack num
writei // escreve inteiro
pushs " é um número primo"
writes // escreve string
WRITELN
jump ifend5
else4:
pushg 0 // carrega da stack num
writei // escreve inteiro
pushs " não é um número primo"
writes // escreve string
WRITELN
ifend5:
stop
```

4. Conclusão

Este trabalho teve como objetivo principal o desenvolvimento de um compilador para a linguagem Pascal.

Durante o desenvolvimento, foram enfrentados alguns desafios, nomeadamente na análise semântica, onde não foi possível implementar uma verificação exaustiva de todos os tipos de erro, ficando alguma responsabilidade a cargo do utilizador. Além disso, surgiram dúvidas quanto à profundidade com que certas funcionalidades da linguagem deveriam ser implementadas, o que levou a uma abordagem mais superficial em alguns casos.

Apesar das limitações, consideramos que o compilador desenvolvido cumpre com os objetivos essenciais do projeto, apresentando uma performance satisfatória e uma estrutura coerente e bem justificada. As soluções adotadas foram fundamentadas, e o trabalho demonstrou um entendimento sólido das etapas de construção de um compilador, desde a análise léxica até à geração de código.

Em suma, o grupo está satisfeito com os resultados obtidos e com o conhecimento adquirido ao longo da realização do projeto.