



Universidade do Minho
Escola de Engenharia

Processamento de Linguagens

Trabalho Prático

LEI - 3º Ano - 2º Semestre

Trabalho realizado por:

pg57579 - Lara Beatriz Pinto Ferreira
pg57582 - Luís Miguel Moreira Ferreira

Braga, 1 de junho de 2025

Conteúdo

| | |
|---|-----------|
| 1. Introdução | 3 |
| 2. Desenho da Resolução | 4 |
| 2.1. Analisador Léxico | 4 |
| 2.1.1. Tokens | 4 |
| 2.2. Analisador Sintático | 7 |
| 2.2.1. Gramática | 7 |
| 3. Exemplos de utilização | 13 |
| 3.1. Olá, Mundo! | 13 |
| 3.2. Maior de 3 | 13 |
| 3.3. Fatorial | 15 |
| 3.4. Verificação de Número Primo | 16 |
| 3.5. Soma de uma lista de inteiros | 17 |
| 3.6. Conversão binário-decimal | 19 |
| 4. Conclusão e Trabalho Futuro | 21 |

1. Introdução

No âmbito da UC de Processamento de Linguagens foi-nos proposto o desenvolvimento de um compilador para processar código na linguagem **Pascal** e traduzi-lo em código ***Assembly*** a ser lido pela *Virtual Machine* fornecida pela equipa docente. Este projeto teve como principais objetivos:

- Consolidar os conhecimentos em engenharia de linguagens e reforçar a capacidade de escrever gramáticas, sejam elas independentes de contexto (**GIC**) ou tradutoras (**GT**).
- Desenvolver processadores de linguagem de acordo com o método da tradução dirigida pela sintaxe, partindo de uma gramática tradutora.
- Implementar um compilador capaz de gerar código para um determinado objetivo (neste caso, a VM), respeitando as regras sintáticas e semânticas da linguagem **Pascal**.
- Explorar ferramentas de geração automática de compiladores, nomeadamente o uso das bibliotecas *Lex* e *Yacc* do *Python* (*PLY*).

2. Desenho da Resolução

2.1. Analisador Léxico

A primeira etapa do desenvolvimento deste trabalho prático baseou-se na elaboração do analisador Léxico utilizando a ferramenta geradora de analisadores léxicos *Lex*, versão *PLY* do *Python*. O analisador léxico é responsável por identificar os tokens, individualmente, onde cada token terá um significado específico no contexto do programa.

Dentro do ficheiro *pascal_lex.py* estão presentes os componentes para o nosso analisador léxico:

- **tokens**: tuplo que define todos os **tokens** que o analisador pode reconhecer. Cada **token** é uma *string* que representa um tipo na *string* de entrada;
- **Funções $t_$** : Cada função cujo nome começa por $t_$ é responsável por identificar um token. Estas funções utilizam expressões regulares para corresponder partes da string de entrada. Como o analisador é criado com a opção *re.IGNORECASE*, a distinção entre maiúsculas e minúsculas é ignorada. Quando é encontrada uma correspondência, a função retorna o token correspondente;
- t_ignore : *string* de caracteres que o analisador deve ignorar;
- t_error : função chamada quando o analisador encontra um caractere que não pode ser combinado com nenhum dos **tokens**

2.1.1. Tokens

Para representar todas as funcionalidades do nosso programa, definimos os seguintes **tokens**:

- **PLUS**: representar o símbolo +
- **MINUS**: representar o símbolo -
- **TIMES**: representar o símbolo *
- **DIVIDE**: representar o símbolo /
- **DIV**: representar a palavra reservada “div”
- **MOD**: representar a palavra reservada “mod”
- **EQ**: representar o conjunto de símbolos =
- **NEQ**: representar o conjunto de símbolos <>
- **LT**: representar o símbolo <
- **LE**: representar conjunto de símbolos <=
- **GT**: representar o símbolo >
- **GE**: representar conjunto de símbolos >=
- **ASSIGN**: representar o símbolo de atribuição :=
- **LPAREN**: representar o símbolo (
- **RPAREN**: representar o símbolo)
- **LBRACKET**: representar o símbolo [
- **RBRACKET**: representar o símbolo]
- **DOT**: representar o símbolo .

- **DOTDOT**: representar o conjunto de símbolos ..
- **COLON**: representar o símbolo :
- **COMMA**: representar o símbolo ,
- **SEMI**: representar o símbolo ;
- **NUMBER**: representar um número inteiro
- **REAL**: representar um número real
- **STRING_LITERAL**: representar um literal de string
- **ID**: representar identificadores
- **AND**: representar a palavra reservada “and”
- **OR**: representar a palavra reservada “or”
- **NOT**: representar a palavra reservada “not”
- **TRUE**: representar a constante booleana “true”
- **FALSE**: representar a constante booleana “false”
- **NIL**: representar a constante “nil”
- **IF**: representar a palavra reservada “if”
- **THEN**: representar a palavra reservada “then”
- **ELSE**: representar a palavra reservada “else”
- **WHILE**: representar a palavra reservada “while”
- **DO**: representar a palavra reservada “do”
- **REPEAT**: representar a palavra reservada “repeat”
- **UNTIL**: representar a palavra reservada “until”
- **FOR**: representar a palavra reservada “for”
- **TO**: representar a palavra reservada “to”
- **DOWNTO**: representar a palavra reservada “downto”
- **GOTO**: representar a palavra reservada “goto”
- **BEGIN**: representar a palavra reservada “begin”
- **END**: representar a palavra reservada “end”
- **PROGRAM**: representar a palavra reservada “program”
- **PROCEDURE**: representar a palavra reservada “procedure”
- **FUNCTION**: representar a palavra reservada “function”
- **VAR**: representar a palavra reservada “var”
- **CONST**: representar a palavra reservada “const”
- **TYPE**: representar a palavra reservada “type”
- **ARRAY**: representar a palavra reservada “array”

- **OF**: representar a palavra reservada “of”
- **RECORD**: representar a palavra reservada “record”
- **SET**: representar a palavra reservada “set”
- **PACKED**: representar a palavra reservada “packed”
- **CASE**: representar a palavra reservada “case”
- **WITH**: representar a palavra reservada “with”
- **READ**: representar a palavra reservada “read”
- **READLN**: representar a palavra reservada “readln”
- **WRITE**: representar a palavra reservada “write”
- **WRITELN**: representar a palavra reservada “writeln”
- **LABEL**: representar a palavra reservada “label”
- **FILE**: representar a palavra reservada “file”
- **STRING**: representar a palavra reservada “string”
- **BOOLEAN**: representar a palavra reservada “boolean”
- **INTEGER**: representar a palavra reservada “integer”
- **IN**: representar a palavra reservada “in”

2.2. Analisador Sintático

Após a implementação do analisador léxico, procedemos a implementar o analisador sintático. O nosso analisador sintático foi implementado em *Ply* e estruturado com base numa gramática tradutora. Cada regra não só valida a estrutura do programa *Pascal*, mas também traduz o código para uma forma intermédia compatível com a máquina virtual fornecida pela equipa docente.

2.2.1. Gramática

Depois da definição dos *tokens*, definimos a nossa gramática. De modo a usufruirmos ao máximo das funcionalidades do *Ply*, decidimos usar uma gramática *Bottom-Up*.

Abaixo apresentamos as regras da nossa gramática, que possibilitam a leitura de expressões.

```
1 def p_program(p):
2     """program : PROGRAM ID SEMI declarations functions BEGIN statements END
   DOT"""
3     functions_code_str = "\n".join(f_code for f_code in
   parser_functions.values() if f_code)
4     main_statements_code = p[7]
5
6     final_code = []
7     if functions_code_str:
8         final_code.append(functions_code_str)
9         final_code.append("start")
10    if main_statements_code:
11        final_code.append(main_statements_code)
12        final_code.append("stop")
13
14    p[0] = "\n".join(final_code) + "\n"
15
```

A função *p_program(p)* define o símbolo inicial da gramática, correspondendo à estrutura global de um programa *Pascal*. Esta regra verifica a presença de todos os elementos formais do programa e gera o código intermédio correspondente.

```
1 def p_var_declaration(p):
2     """var_declaration : id_list COLON type SEMI"""
3     global parser_var, parser_var_count, parser_success, parser_var_types
4     type_representation = p[3]
5
6     for var_name in p[1]:
7         if var_name in parser_var:
8             print(f"Erro: variável duplicada {var_name}")
9             parser_success = False
10        else:
11            parser_var[var_name] = parser_var_count
12            parser_var_types[var_name] = type_representation
13
14            if isinstance(type_representation, str) and
   type_representation.startswith("array["):
15                try:
16                    range_part = type_representation.split('[')[1].split(']')
17                    [0]
18
19                    low_bound_str, high_bound_str = range_part.split('..')
```

```

18         low = int(low_bound_str)
19         high = int(high_bound_str)
20         size = high - low + 1
21         parser_var_count += size
22     except:
23         print(f"Aviso: Não foi possível determinar o tamanho para o
array {var_name}. A contagem de vars pode estar incorreta.")
24         parser_var_count += 1
25     else:
26         parser_var_count += 1
27
28     p[0] = ""
29

```

A função *p_var_declaration* deteta e guarda tanto variáveis simples como arrays, atualizando as tabelas globais de variáveis com o seu tipo e a sua posição na stack. Para os arrays, infere-se o tamanho com base nos limites fornecidos. Além disto, se uma variável já tiver sido declarada ou o tipo de um array estiver errado, é gerado um erro.

```

1 def p_assignment_statement(p):
2     """assignment_statement : variable ASSIGN expression"""
3     global parser_success, parser_var, parser_var_types
4     lhs_var_info = p[1]
5     rhs_expr_code, rhs_expr_type = p[3]
6     if lhs_var_info.get('type') == 'error':
7         parser_success = False
8         p[0] = ""
9         return
10    final_rhs_code = rhs_expr_code
11    target_basetype = lhs_var_info.get('basetype', 'integer')
12    if target_basetype == "real" and rhs_expr_type == "integer":
13        final_rhs_code += "itof\n"
14    elif target_basetype == "integer" and rhs_expr_type == "real":
15        final_rhs_code += "ftoi\n"
16    if lhs_var_info['type'] == 'simple':
17        var_name = lhs_var_info['name']
18        p[0] = final_rhs_code + f"storeg {parser_var[var_name]}\n"
19    elif lhs_var_info['type'] == 'indexed':
20        array_name = lhs_var_info['name']
21        index_code = lhs_var_info['index_code']
22        array_slot = parser_var[array_name]
23        low_bound = lhs_var_info.get('low_bound', 1)
24        addr_calc_code = "pushgp\n"
25        addr_calc_code += f"pushi {array_slot}\n"
26        addr_calc_code += "padd\n"
27        addr_calc_code += index_code
28        addr_calc_code += f"pushi {low_bound}\n"
29        addr_calc_code += "sub\n"
30        p[0] = addr_calc_code + final_rhs_code + "storen\n"
31    else:
32        p[0] = ""

```


A regra *p_assignment_statement* é uma das regras principais da gramática pois é ela que traduz instruções em *Pascal* para instruções da máquina virtual, gerando erros caso existam variáveis inválidas ou expressões que não possam ser avaliadas.

```
1 def p_write_statement(p):
2     """write_statement : WRITE LPAREN writelist RPAREN"""
3     p[0] = p[3]
4
5 def p_writeln_statement(p):
6     """writeln_statement : WRITELN LPAREN writelist RPAREN"""
7     p[0] = p[3] + "writeln\n"
```

As funções *p_write_statement* e *p_writeln_statement* geram código para a máquina virtual escrever no ecrã. A única diferença entre as duas funções é a adição de uma quebra de linha no *p_writeln_statement*.

```
1 def p_readln_statement(p):
2     """readln_statement : READLN LPAREN variable RPAREN"""
3     global parser_success, parser_var, parser_var_types
4     var_info = p[3]
5
6     if var_info.get('type') == 'error':
7         parser_success = False
8         p[0] = ""
9         return
10
11     addr_calc_code = ""
12     is_indexed = False
13     if var_info['type'] == 'indexed_array':
14         is_indexed = True
15         array_name = var_info['name']
16         index_code_for_addr = var_info.get('index_code', "")
17         array_slot = parser_var[array_name]
18         low_bound = var_info.get('low_bound', 1)
19
20         addr_calc_code += "pushgp\n"
21         addr_calc_code += f"pushi {array_slot}\n"
22         addr_calc_code += "padd\n"
23         addr_calc_code += index_code_for_addr
24         addr_calc_code += f"pushi {low_bound}\n"
25         addr_calc_code += "sub\n"
26
27     base_read_code = "read\n"
28     conversion_code = ""
29     actual_target_type = var_info.get('basetype')
30
31     if actual_target_type == 'real':
32         conversion_code = "atof\n"
33     elif actual_target_type == 'integer' or actual_target_type == 'boolean':
34         conversion_code = "atoi\n"
35     elif actual_target_type == 'string':
36         conversion_code = ""
37     else:
```

```

38     print(f"Erro: Tipo de variável desconhecido ou não suportado
    '{actual_target_type}' para READLN.")
39     parser_success = False
40     p[0] = ""
41     return
42
43     full_read_and_convert_code = base_read_code + conversion_code
44
45     if not is_indexed and var_info['type'] == 'simple':
46         var_name = var_info['name']
47         if var_name not in parser_var:
48             print(f"Erro: Variável '{var_name}' não declarada para READLN.")
49             parser_success = False
50             p[0] = ""
51             return
52         p[0] = full_read_and_convert_code + f"storeg {parser_var[var_name]}\n"
53     elif is_indexed:
54         p[0] = addr_calc_code + full_read_and_convert_code + "storen\n"
55     else:
56         print(f"Erro: Tipo de variável não suportado '{var_info['type']}' para
    atribuição em READLN.")
57         parser_success = False
58         p[0] = ""

```

A função *p_readln_statement* é responsável por ler dados do utilizador, convertê-los para os tipos corretos e armazená-los corretamente na stack. Utiliza “READ” seguido de “ATOI” ou “ATOF” para conversão, e “STOREG” ou “STOREN” para guardar um valor.

```

1  def p_if_statement(p):
2      """if_statement : IF expression THEN statement %prec IFX
3                          | IF expression THEN statement ELSE
    statement"""
4      cond_code, _ = p[2]
5      then_statement_code = p[4]
6
7      label_num = generate_unique_label_num()
8
9      if len(p) == 5:
10         label_end = f"ifend{label_num}"
11         p[0] = (
12             cond_code +
13             f"jz {label_end}\n" +
14             then_statement_code +
15             f"{label_end}:\n"
16         )
17     else:
18         else_statement_code = p[6]
19         label_else = f"ifelse{label_num}"
20         label_end = f"ifend{label_num}"
21         p[0] = (
22             cond_code +
23             f"jz {label_else}\n" +
24             then_statement_code +
25             f"jump {label_end}\n" +

```

```

26         f"{label_else}:\n" +
27         else_statement_code +
28         f"{label_end}:\n"
29     )

```

A função `p_if_statement` implementa a estrutura condicional, seja uma estrutura “IF THEN” ou uma estrutura “IF THEN ELSE”. São gerados labels únicos para isolar cada bloco condicional.

```

1  def p_for_statement(p):
2      """for_statement : FOR ID ASSIGN expression TO expression DO statement
3                          | FOR ID ASSIGN expression DOWNT0
4                          expression DO statement"""
5
6      global parser_success, parser_var_count, parser_var
7
8      loop_var_name = p[2]
9      if loop_var_name not in parser_var:
10         print(f"Erro: variável de ciclo '{loop_var_name}' não declarada.")
11         parser_success = False
12         p[0] = ""
13         return
14
15     loop_var_slot = parser_var[loop_var_name]
16
17     limit_storage_slot = parser_var_count
18     parser_var_count += 1
19
20     init_expr_code, init_expr_type = p[4]
21     limit_expr_code, limit_expr_type = p[6]
22     body_code = p[8]
23
24     label_num = generate_unique_label_num()
25     loop_label = f"forloop{label_num}"
26     end_label = f"forend{label_num}"
27
28     direction_token_type = p.slice[5].type
29
30     comparison_instruction = ""
31     step_instruction = ""
32
33     if direction_token_type == 'TO':
34         comparison_instruction = "ineq"
35         step_instruction = "add"
36     elif direction_token_type == 'DOWNT0':
37         comparison_instruction = "supeq"
38         step_instruction = "sub"
39     else:
40         print(f"Erro interno: token de direção desconhecido
41               '{direction_token_type}' no loop FOR.")
42         parser_success = False
43         p[0] = ""
44         return

```

```

44     p[0] = (
45         init_expr_code +
46         f"storeg {loop_var_slot}\n" +
47         limit_expr_code +
48         f"storeg {limit_storage_slot}\n" +
49         f"{loop_label}:\n" +
50         f"pushg {loop_var_slot}\n" +
51         f"pushg {limit_storage_slot}\n" +
52         f"{comparison_instruction}\n" +
53         f"jz {end_label}\n" +
54         body_code +
55         f"pushg {loop_var_slot}\n" +
56         "pushi 1\n" +
57         f"{step_instruction}\n" +
58         f"storeg {loop_var_slot}\n" +
59         f"jump {loop_label}\n" +
60         f"{end_label}:\n"
61     )

```

A função *p_for_statement* trata dos ciclos *for*, sejam eles “FOR ... TO ... DO” ou “FOR ... DOWNT0 ... DO” gerando instruções de salto condicional.

```

1  def p_while_statement(p):
2      """while_statement : WHILE expression DO statement"""
3      cond_code, _ = p[2]
4      body_code = p[4]
5
6      label_num = generate_unique_label_num()
7      start_label = f"whilestart{label_num}"
8      end_label = f"whileend{label_num}"
9
10     p[0] = (
11         f"{start_label}:\n" +
12         cond_code +
13         f"jz {end_label}\n" +
14         body_code +
15         f"jump {start_label}\n" +
16         f"{end_label}:\n"
17     )

```

A função *p_while_statement* traduz o ciclo “WHILE” em *Pascal* num conjunto de saltos condicionais. São usados labels para delimitar o bloco do ciclo para garantir uma execução sem conflitos com ciclos diferentes.

3. Exemplos de utilização

3.1. Olá, Mundo!

Input:

```
1 program HelloWorld;
2 begin
3     writeln('Ola, Mundo!');
4 end.
5
```

Ouput:

```
1 start
2 pushes "Ola, Mundo!"
3 writes
4 writeln
5
6 stop
```

3.2. Maior de 3

Input:

```
1 program Maior3;
2 var
3     num1, num2, num3, maior: Integer;
4 begin
5     { Ler 3 números }
6     Write('Introduza o primeiro número: \n');
7     ReadLn(num1);
8     Write('Introduza o segundo número: \n');
9     ReadLn(num2);
10    Write('Introduza o terceiro número: \n');
11    ReadLn(num3);
12    { Calcular o maior }
13    if num1 > num2 then
14        if num1 > num3 then maior := num1
15        else maior := num3
16    else
17        if num2 > num3 then maior := num2
18        else maior := num3;
19    { Escrever o resultado }
20    WriteLn('O maior é: ', maior)
21 end.
```

Ouput:

```
1 pushn 4
2 start
3 pushes "Introduza o primeiro número: \n"
4 writes
5 read
6 atoi
```

```
7 storeg 0
8 pushs "Introduza o segundo número: \n"
9 writes
10 read
11 atoi
12 storeg 1
13 pushs "Introduza o terceiro número: \n"
14 writes
15 read
16 atoi
17 storeg 2
18 pushg 0
19 pushg 1
20 sup
21 jz ifelse3
22 pushg 0
23 pushg 2
24 sup
25 jz ifelse1
26 pushg 0
27 storeg 3
28 jump ifend1
29 ifelse1:
30 pushg 2
31 storeg 3
32 ifend1:
33 jump ifend3
34 ifelse3:
35 pushg 1
36 pushg 2
37 sup
38 jz ifelse2
39 pushg 1
40 storeg 3
41 jump ifend2
42 ifelse2:
43 pushg 2
44 storeg 3
45 ifend2:
46 ifend3:
47 pushs "O maior é: "
48 writes
49 pushg 3
50 writei
51 writeln
52
53 stop
```

3.3. Fatorial

Input:

```
1 program Fatorial;
2 var
3     n, i, fat: integer;
4 begin
5     writeln('Introduza um número inteiro positivo:');
6     readln(n);
7     fat := 1;
8     for i := 1 to n do
9         fat := fat * i;
10    writeln('Fatorial de ', n, ': ', fat);
11 end.
```

Ouput:

```
1 pushn 4
2 start
3 pushes "Introduza um número inteiro positivo:"
4 writes
5 writeln
6 read
7 atoi
8 storeg 0
9 pushi 1
10 storeg 2
11 pushi 1
12 storeg 1
13 pushg 0
14 storeg 3
15 forloop1:
16 pushg 1
17 pushg 3
18 infeq
19 jz forend1
20 pushg 2
21 pushg 1
22 mul
23 storeg 2
24 pushg 1
25 pushi 1
26 add
27 storeg 1
28 jump forloop1
29 forend1:
30 pushes "Fatorial de "
31 writes
32 pushg 0
33 writei
34 pushes ": "
35 writes
36 pushg 2
37 writei
```

```
38 writeln
39
40 stop
```

3.4. Verificação de Número Primo

Input:

```
1 program NumeroPrimo;
2 var
3     num, i: integer;
4     primo: boolean;
5 begin
6     writeln('Introduza um número inteiro positivo:');
7     readln(num);
8     primo := true;
9     i := 2;
10    while (i <= (num div 2)) and primo do
11        begin
12            if (num mod i) = 0 then
13                primo := false;
14            i := i + 1;
15        end;
16    if primo then
17        writeln(num, ' é um número primo')
18    else
19        writeln(num, ' não é um número primo')
20 end.
```

Ouput:

```
1 pushn 3
2 start
3 pushes "Introduza um número inteiro positivo:"
4 writes
5 writeln
6 read
7 atoi
8 storeg 0
9 pushi 1
10 storeg 2
11 pushi 2
12 storeg 1
13 whilestart2:
14 pushg 1
15 pushg 0
16 pushi 2
17 div
18 infeq
19 pushg 2
20 AND
21 jz whileend2
22 pushg 0
23 pushg 1
24 mod
```



```
25 pushi 0
26 equal
27 jz ifend1
28 pushi 0
29 storeg 2
30 ifend1:
31 pushg 1
32 pushi 1
33 add
34 storeg 1
35 jump whilestart2
36 whileend2:
37 pushg 2
38 jz ifelse3
39 pushg 0
40 writei
41 pushs " é um número primo"
42 writes
43 writeln
44 jump ifend3
45 ifelse3:
46 pushg 0
47 writei
48 pushs " não é um número primo"
49 writes
50 writeln
51 ifend3:
52
53 stop
54
```

3.5. Soma de uma lista de inteiros

Input:

```
1 program SomaArray;
2 var
3     numeros: array[1..5] of integer;
4     i, soma: integer;
5 begin
6     soma := 0;
7     writeln('Introduza 5 números inteiros:');
8     for i := 1 to 5 do
9         begin
10             readln(numeros[i]);
11             soma := soma + numeros[i];
12         end;
13     writeln('A soma dos números é: ', soma);
14 end.
```

Output:

```
1 pushn 8
2 start
3 pushi 0
```

```
4 storeg 6
5 pushs "Introduza 5 números inteiros:"
6 writes
7 writeln
8 pushi 1
9 storeg 5
10 pushi 5
11 storeg 7
12 forloop1:
13 pushg 5
14 pushg 7
15 infeq
16 jz forend1
17 pushgp
18 pushi 0
19 padd
20 pushg 5
21 pushi 1
22 sub
23 read
24 atoi
25 storen
26 pushg 6
27 pushgp
28 pushi 0
29 padd
30 pushg 5
31 pushi 1
32 sub
33 loadn
34 add
35 storeg 6
36 pushg 5
37 pushi 1
38 add
39 storeg 5
40 jump forloop1
41 forend1:
42 pushs "A soma dos números é: "
43 writes
44 pushg 6
45 writei
46 writeln
47
48 stop
```

3.6. Conversão binário-decimal

Input:

```
1 program BinarioParaInteiro;
2 var
3     bin: string;
4     i, valor, potencia: integer;
5 begin
6     writeln('Introduza uma string binária:');
7     readln(bin);
8     valor := 0;
9     potencia := 1;
10    for i := length(bin) downto 1 do
11    begin
12        if bin[i] = '1' then
13            valor := valor + potencia;
14        potencia := potencia * 2;
15    end;
16    writeln('O valor inteiro correspondente é: ', valor);
17 end.
```

Ouput:

```
1 pushn 5
2 start
3 pushs "Introduza uma string binária:"
4 writes
5 writeln
6 read
7 storeg 0
8 pushi 0
9 storeg 2
10 pushi 1
11 storeg 3
12 pushg 0
13 STRLEN
14 storeg 1
15 pushi 1
16 storeg 4
17 forloop2:
18 pushg 1
19 pushg 4
20 supeq
21 jz forend2
22 pushg 0
23 pushg 1
24 pushi 1
25 sub
26 CHARAT
27 pushs "1"
28 CHRCODE
29 equal
30 jz ifend1
31 pushg 2
```

```
32 pushg 3
33 add
34 storeg 2
35 ifend1:
36 pushg 3
37 pushi 2
38 mul
39 storeg 3
40 pushg 1
41 pushi 1
42 sub
43 storeg 1
44 jump forloop2
45 forend2:
46 pushes "0 valor inteiro correspondente é: "
47 writes
48 pushg 2
49 writei
50 writeln
51
52 stop
```

4. Conclusão e Trabalho Futuro

Este projeto permitiu-nos consolidar os conhecimentos adquiridos ao longo da unidade curricular de Processamento de Linguagens, através da implementação de um compilador para *Pascal* direcionado a uma máquina virtual stack-based. Ao longo do desenvolvimento, tivemos contacto direto com conceitos como gramáticas tradutoras, parsing bottom-up e geração de código.

Conseguimos implementar os principais componentes do compilador — analisador léxico e sintático — utilizando a biblioteca *PLY* em *Python*, suportando várias construções do *Pascal*, como ciclos, condicionais, expressões e entrada/saída. Apesar de alguns desafios durante a fase de teste e integração, o resultado final foi um compilador funcional, capaz de traduzir corretamente programas escritos em Pascal para a linguagem da máquina virtual fornecida.

Reconhecemos, no entanto, que muitas das nossas regras são bastante extensas e concentram múltiplas responsabilidades numa só função. Esta abordagem torna mais difícil a leitura e a manutenção do código. Sendo uma possível melhoria futura, seria importante dividir melhor essas regras para tornar o compilador mais organizado e escalável.