



Universidade do Minho  
Escola de Ciências

UNIVERSIDADE DO MINHO  
LICENCIATURA EM CIÊNCIAS DA COMPUTAÇÃO

PLC - Trabalho Prático 2  
Grupo nº14

José Bernardo Moniz Fernandes  
(A102497)

Pedro Augusto Ennes de Martino Camargo  
(A102504)

4 de janeiro de 2025



# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>3</b>
<b>2</b>	<b>Enunciado</b>	<b>4</b>
<b>3</b>	<b>Concepção da Solução</b>	<b>5</b>
3.1	Sintaxe da Linguagem PMS . . . . .	5
3.1.1	Declaração de variáveis . . . . .	5
3.1.2	Operadores de comparação . . . . .	5
3.1.3	Operações numéricas . . . . .	6
3.1.4	Operadores lógicos . . . . .	6
3.1.5	Instruções condicionais . . . . .	6
3.1.6	If . . . . .	6
3.1.7	If-Else . . . . .	6
3.1.8	Ciclo while . . . . .	6
3.1.9	Ciclo do-while . . . . .	6
3.1.10	Função . . . . .	7
3.1.11	Input/Output . . . . .	7
3.2	Símbolos . . . . .	7
3.3	Desenho da GIC . . . . .	8
<b>4</b>	<b>Exemplos de funcionamento</b>	<b>11</b>
4.0.1	While . . . . .	11
4.0.2	If-Else com operadores de comparação e lógicos . . . .	11
4.0.3	Do-While . . . . .	13
4.0.4	Operações numéricas . . . . .	14
<b>5</b>	<b>Conclusão</b>	<b>17</b>

# Capítulo 1

## Introdução

No âmbito da unidade curricular de Processamento de Linguagens e Compiladores foi-nos proposto pelo docente Pedro Rangel Henriques o desenvolvimento de uma Linguagem de Programação Imperativa, ao nosso gosto, e de um compilador para reconhecer programas escritos nessa linguagem, gerando o respetivo código Assembly da Máquina Virtual VM.

Primeiramente, começámos por encontrar um nome para a nossa linguagem e acabou por nos surgir a ideia de colocar o nome "PedroMonizScript" (PMS), inspirámo-nos na linguagem de programação TypeScript, que proporciona segurança no seu desenvolvimento.

Neste documento, está apresentada a gramática e a sintaxe da nossa linguagem, bem como alguns testes com código escrito na nossa linguagem e o respetivo código Assembly gerado.

## Capítulo 2

# Enunciado

Pretende-se que comece por definir uma linguagem de programação imperativa simples, a seu gosto.

Apenas deve ter em consideração que essa linguagem terá de permitir:

- *declarar* variáveis atômicas do tipo *inteiro*, com os quais se podem realizar as habituais operações aritméticas, relacionais e lógicas.
- *efetuar* instruções algorítmicas básicas como a *atribuição do valor de expressões numéricas a variáveis*.
- *ler* do *standard input* e *escrever* no *standard output*.
- *efetuar* instruções *seleção* para controlo do fluxo de execução.
- *efetuar* instruções de repetição (*cíclicas*) para controlo do fluxo de execução, permitindo o seu aninhamento.  
Note que deve implementar pelo menos o ciclo **while-do**, **repeat-until** ou **for-do**.

Adicionalmente deve ainda suportar, à sua escolha, uma das duas funcionalidades seguintes:

- *declarar e manusear* variáveis estruturadas do tipo *array* (a 1 ou 2 dimensões) de *inteiros*, em relação aos quais é apenas permitida a operação de indexação (índice inteiro).
- *definir e invocar subprogramas* sem parâmetros mas que possam retornar um resultado do tipo inteiro.

## Capítulo 3

# Concepção da Solução

Neste capítulo vamos apresentar:

- A sintaxe da linguagem PMS
- Os símbolos
- O desenho da gramática independente de contexto

### 3.1 Sintaxe da Linguagem PMS

A sintaxe da linguagem é a seguinte:

#### 3.1.1 Declaração de variáveis

```
let a: INT = 1000;  
let b: STR = "Ola";  
let c: FLOAT = 10.9;  
let d: INT;  
let f: FLOAT;  
let arrayInt: Array<INT> = [0, 1, 2, 3];  
let arrayFloat: Array<FLOAT> = [10.9, 199.212, 322.12];  
let arrayString: Array<STR> = ["Olá!", "Tudo Bem?"];  
const inputtext: INT = console.input("Por favor dê um valor para esta variável: ");
```

#### 3.1.2 Operadores de comparação

```
x > y  
x < y  
x >= y  
x <= y  
x == y  
x != y
```

### 3.1.3 Operações numéricas

```
x + y
x - y
x * y
x / y
x % y
x ++
y --
```

### 3.1.4 Operadores lógicos

```
x && y
x || y
not x
```

### 3.1.5 Instruções condicionais

#### 3.1.6 If

```
if (Cond) {
    ...
}
```

#### 3.1.7 If-Else

```
if (Cond) {
    ...
} else {
    ...
}
```

#### 3.1.8 Ciclo while

```
while (Cond) {
    ...
}
```

#### 3.1.9 Ciclo do-while

```
do {
    ...
} while (Cond);
```

### 3.1.10 Função

```
const ID = () => {  
    ...  
}
```

### 3.1.11 Input/Output

```
const inputtext: INT = console.input("Por favor dê um valor para esta variável: ");  
console.output(inputtext);
```

## 3.2 Símbolos

Os símbolos da linguagem são os seguintes:

```
'INTVALUE',  
'FLOATVALUE',  
'STRINGVALUE',  
'INC',  
'DEC',  
'ID',  
'GEQUAL',  
'LEQUAL',  
'EQUAL',  
'DIFF',  
'INT',  
'FLOAT',  
'STR',  
'CONST',  
'LET',  
'WHILE',  
'ARRAY',  
'IF',  
'ELSE',  
'DO',  
'WHILE',  
'PRINT',  
'INPUT',  
'SEMICOLON',  
'OR',  
'AND',  
'NOT',  
'CALL',  
'RETURN',  
':',
```

```

'=' ,
'[' ,
']' ,
'<' ,
'>' ,
',' ,
'(' ,
')' ,
'<' ,
'>' ,
',' ,
'+' ,
'-' ,
'*' ,
'/' ,
'%' ,
'{' ,
'}'

```

### 3.3 Desenho da GIC

A nossa linguagem é gerada pela seguinte grámatica independente de contexto:

```

1  ProgramInit : Declarations Instructions
2                | Declarations
3                | Instructions
4
5  Declarations : Declarations IntDeclaration
6                | Declarations IntDeclarationInput
7                | Declarations StringDeclaration
8                | Declarations StringDeclarationInput
9                | Declarations FloatDeclaration
10               | Declarations FloatDeclarationInput
11               | Declarations ArrayDeclaration
12               | Declarations FunctionDeclaration
13               | Empty
14
15  Instructions : Instructions Instruction
16                | Instruction
17
18  MutationType : CONST
19                | LET
20
21  IntDeclaration : MutationType ID ':' INT '=' INTVALUE SEMICOLON
22                  | MutationType ID ':' INT SEMICOLON
23
24  IntDeclarationInput : MutationType ID ':' INT '=' Input SEMICOLON

```



```

25
26 StringDeclaration : MutationType ID ':' STR '=' STRINGVALUE SEMICOLON
27                   | MutationType ID ':' STR SEMICOLON
28
29 StringDeclarationInput : MutationType ID ':' STR '=' Input SEMICOLON
30
31 FloatDeclaration : MutationType ID ':' FLOAT '=' FLOATVALUE SEMICOLON
32                 | MutationType ID ':' FLOAT SEMICOLON
33
34 FloatDeclarationInput : MutationType ID ':' FLOAT '=' Input SEMICOLON
35
36 ArrayDeclaration : MutationType ID ':' ARRAY '<' INT '>' '=' '['
37 ArrayIntDeclaration ']' SEMICOLON
38                 | MutationType ID ':' ARRAY '<' FLOAT '>' '=' '['
39 ArrayFloatDeclaration ']' SEMICOLON
40                 | MutationType ID ':' ARRAY '<' STR '>' '=' '['
41 ArrayStringDeclaration ']' SEMICOLON
42
43 ArrayIntDeclaration : ArrayIntDeclaration ',' INTVALUE
44                   | INTVALUE
45                   | Empty
46
47 ArrayFloatDeclaration : ArrayFloatDeclaration ',' FLOATVALUE
48                   | FLOATVALUE
49                   | Empty
50
51 ArrayStringDeclaration : ArrayStringDeclaration ',' STRINGVALUE
52                   | STRINGVALUE
53                   | Empty
54
55 FunctionDeclaration : CONST ID '=' '(' ')' '=' '>' '{' Instructions '}'
56
57 Instruction : Attributions
58             | Output
59             | Call
60             | Return
61             | If
62             | Loop
63
64 Attributions : Attributions NormalAttribution SEMICOLON
65             | Attributions IncDecAttribution SEMICOLON
66             | Attributions Expression SEMICOLON
67             | Empty
68
69 NormalAttribution : ID '=' INTVALUE
70                   | ID '=' STRINGVALUE
71                   | ID '=' FLOATVALUE
72
73 IncDecAttribution : ID DEC
74                   | ID INC
75
76 Expression : ID '=' Expr
77
78 Expr : IncDecAttribution

```

```

76         | ID
77         | INTVALUE
78         | FLOATVALUE
79         | Expr '+' Expr
80         | Expr '-' Expr
81         | Expr '*' Expr
82         | Expr '/' Expr
83         | Expr '%' Expr
84
85     Cond : Expr '<' Expr
86         | Expr '>' Expr
87         | Expr GEQUAL Expr
88         | Expr LEQUAL Expr
89         | Expr EQUAL Expr
90         | Expr DIFF Expr
91         | Cond OR Cond
92         | Cond AND Cond
93         | NOT '(' Cond ')'
94
95     If : IF '(' Cond ')' '{' Instructions '}'
96         | IF '(' Cond ')' '{' Instructions '}' ELSE '{' Instructions '}'
97
98     Loop : While
99         | DoWhile
100
101     While : WHILE '(' Cond ')' '{' Instructions '}'
102
103     DoWhile : DO '{' Instructions '}' WHILE '(' Cond ')' SEMICOLON
104
105     Call : CALL SEMICOLON
106         | RETURN CALL SEMICOLON
107
108     Return : RETURN Expr SEMICOLON
109             | RETURN Call
110
111     Input : INPUT '(' STRINGVALUE ')'
112
113     Output : PRINT '(' ID ')' SEMICOLON
114             | PRINT '(' STRINGVALUE ')' SEMICOLON
115
116     Empty :
117

```

## Capítulo 4

# Exemplos de funcionamento

### 4.0.1 While

```
1 let a: INT = 3;
2
3 while (a > 0) {
4     a--;
5     console.output(a);
6     console.output("\n");
7 }
8
```

#### Código Assembly gerado:

```
1 pushi 3
2 start
3 label0c: NOP
4 PUSHG 0
5 PUSHI 0
6 SUP
7 JZ label0f
8 PUSHG 0
9 PUSHI 1
10 SUB
11 STOREG 0
12 PUSHG 0
13 WRITEI
14 PUSHS "\n"
15 WRITES
16 JUMP label0c
17 label0f: NOP
18 stop
19
```

### 4.0.2 If-Else com operadores de comparação e lógicos

```

1 const a: INT = console.input("Coloque um valor inteiro para a: \n");
2 const b: INT = console.input("Coloque um valor inteiro para b: \n");
3 const c: INT = console.input("Coloque um valor inteiro para c: \n");
4
5 if (not(a == b && b == c)) {
6     console.output("triangulo nao e equilatero\n");
7 } else {
8     console.output("triangulo e equilatero\n");
9 }
10
11 if (not(a == b || b == c || a == c)) {
12     console.output("triangulo nao e isosceles\n");
13 } else {
14     console.output("triangulo e isosceles\n");
15 }
16
17 if (a != b && b != c && a != c) {
18     console.output("triangulo e escaleno\n");
19 } else {
20     console.output("triangulo nao e escaleno\n");
21 }
22

```

### Código Assembly gerado:

```

1 PUSHS "Coloque um valor inteiro para a: \n"
2 WRITES
3 READ
4 ATOI
5 STOREG 0
6 PUSHS "Coloque um valor inteiro para b: \n"
7 WRITES
8 READ
9 ATOI
10 STOREG 1
11 PUSHS "Coloque um valor inteiro para c: \n"
12 WRITES
13 READ
14 ATOI
15 STOREG 2
16 start
17 PUSHG 0
18 PUSHG 1
19 EQUAL
20 PUSHG 1
21 PUSHG 2
22 EQUAL
23 MUL
24 NOT
25 JZ label0
26 PUSHS "triangulo nao e equilatero\n"
27 WRITES
28 JUMP label0f

```

```

29 label0: NOP
30 PUSHHS "triangulo e equilatero\n"
31 WRITES
32 label0f: NOP
33 PUSHG 0
34 PUSHG 1
35 EQUAL
36 PUSHG 1
37 PUSHG 2
38 EQUAL
39 PUSHG 0
40 PUSHG 2
41 EQUAL
42 ADD
43 ADD
44 NOT
45 JZ label1
46 PUSHHS "triangulo nao e isosceles\n"
47 WRITES
48 JUMP label1f
49 label1: NOP
50 PUSHHS "triangulo e isosceles\n"
51 WRITES
52 label1f: NOP
53 PUSHG 0
54 PUSHG 1
55 EQUAL
56 NOT
57 PUSHG 1
58 PUSHG 2
59 EQUAL
60 NOT
61 PUSHG 0
62 PUSHG 2
63 EQUAL
64 NOT
65 MUL
66 MUL
67 JZ label2
68 PUSHHS "triangulo e escaleno\n"
69 WRITES
70 JUMP label2f
71 label2: NOP
72 PUSHHS "triangulo nao e escaleno\n"
73 WRITES
74 label2f: NOP
75 stop
76

```

### 4.0.3 Do-While

```

1 let N: INT = console.input("Digite um numero: ");
2 let soma: INT = 0;

```

```

3 let i: INT = 1;
4
5 do {
6     soma = soma + i;
7     i++;
8 } while (i <= N);
9
10 console.output("\n");
11 console.output("A soma dos primeiros ");
12 console.output(N);
13 console.output(" numeros e: ");
14 console.output(soma);
15

```

### Código Assembly gerado:

```

1 PUSHS "Digite um numero: "
2 WRITES
3 READ
4 ATOI
5 STOREG 0
6 pushi 0
7 pushi 1
8 start
9 label0:
10 PUSHG 1
11 PUSHG 2
12 ADD
13 STOREG 1
14 PUSHG 2
15 PUSHI 1
16 ADD
17 STOREG 2
18 PUSHG 2
19 PUSHG 0
20 INFEQ
21 NOT
22 JZ label0
23 PUSHS "\n"
24 WRITES
25 PUSHS "A soma dos primeiros "
26 WRITES
27 PUSHG 0
28 WRITEI
29 PUSHS " numeros e: "
30 WRITES
31 PUSHG 1
32 WRITEI
33 stop
34

```

## 4.0.4 Operações numéricas

```

1 let a1: INT = 5;
2 let a2: FLOAT = 9.0;
3 let a3: INT = 2;
4 let a4: INT = 20;
5 let a5: INT = 25;
6 let b: INT = 5;
7 let c: INT = 7;
8
9 a1 = a1 + b;
10
11 a2 = a2 - 5;
12
13 a3 = a3 * b--;
14
15 a4 = a4 / b;
16
17 a5 = a5 % b;
18
19 c = a1 + a2 + 6 - 2;
20

```

### Código Assembly gerado:

```

1 pushi 5
2 pushf 9.0
3 pushi 2
4 pushi 20
5 pushi 25
6 pushi 5
7 pushi 7
8 start
9 PUSHG 0
10 PUSHG 5
11 ADD
12 STOREG 0
13 PUSHG 1
14 PUSHI 5
15 SUB
16 STOREG 1
17 PUSHG 2
18 PUSHG 5
19 PUSHI 1
20 SUB
21 STOREG 5
22 PUSHG 5
23 MUL
24 STOREG 2
25 PUSHG 3
26 PUSHG 5
27 DIV
28 STOREG 3
29 PUSHG 4
30 PUSHG 5

```

```
31 MOD
32 STOREG 4
33 PUSHG 0
34 PUSHG 1
35 PUSHI 6
36 PUSHI 2
37 SUB
38 ADD
39 ADD
40 STOREG 6
41 stop
42
```



## Capítulo 5

# Conclusão

A realização deste trabalho prático, proporcionou-nos atingir os objetivos inicialmente propostos, foi verdadeiramente um trabalho desafiador e produtivo.

Desenvolver a nossa própria linguagem de programação foi um processo que nos exigiu criatividade e decisões cuidadosas.

Além disso, chegar ao final do projeto e perceber que conseguimos projetar e implementar a base de uma linguagem é extremamente recompensador.

Este trabalho proporcionou-nos uma sólida compreensão sobre o funcionamento dos módulos Lexer e Yacc, especialmente no reconhecimento de tokens e na implementação da gramática. Além de que, a geração de código Assembly, para Máquina Virtual, revelou-se uma etapa interessante, permitindo-nos aprofundar o nosso entendimento sobre as instruções e a lógica relacionada a esta linguagem.

Para concluir, entendemos que a maior parte dos objetivos foram alcançados, consideramos, ainda que, este trabalho foi desafiante e preparou-nos para enfrentar desafios futuros.