



**Universidade do Minho**

UNIVERSIDADE DO MINHO

LICENCIATURA EM CIÊNCIAS DA COMPUTAÇÃO

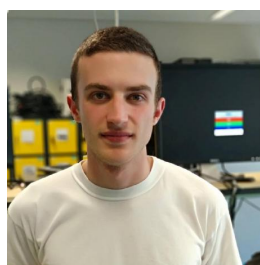
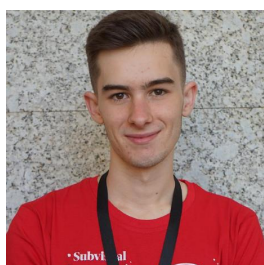
PLC - Trabalho Prático 2  
Grupo nº14

Simão Pedro Batista Caridade Quintela  
(A97444)

David José de Sousa Machado  
(A91665)

Hugo Filipe de Sá Rocha  
(A96463)

15 de Janeiro de 2023



# 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 PLC . . . . .	5
3.1.1	Declaração de variáveis . . . . .	5
3.1.2	Operadores de comparação . . . . .	5
3.1.3	Operações numéricas . . . . .	5
3.1.4	Operadores lógicos . . . . .	6
3.1.5	Instruções condicionais . . . . .	6
3.1.6	Ciclo while . . . . .	6
3.1.7	Ciclo do-while . . . . .	6
3.1.8	Input/Output . . . . .	6
3.1.9	Comentário . . . . .	6
3.1.10	Assert . . . . .	6
3.2	Símbolos . . . . .	7
3.3	Desenho da GIC . . . . .	8
3.4	Extras . . . . .	9
3.4.1	Comentários . . . . .	9
3.4.2	Erros . . . . .	10
3.4.3	Ordem de operações(simples) . . . . .	10
3.4.4	Indentação . . . . .	10
<b>4</b>	<b>Exemplos de funcionamento</b>	<b>11</b>
<b>5</b>	<b>Conclusão</b>	<b>12</b>

# Capítulo 1

## Introdução

No âmbito da disciplina de Processamento de Linguagens e Compiladores foi-nos proposto pelo docente Pedro Rangel Henriques o desenvolvimento de uma Linguagem de Programação Imperativa simples e de um compilador para reconhecer programas escritas nessa linguagem gerando o respetivo código Assembly da Máquina Virtual VM.

Começamos por tentar encontrar um nome original e atrativo para a nossa linguagem e acabou por nos surgir a ideia de colocar o nome "Python-Like-C" cuja sigla (PLC) coincide com a sigla da Unidade Curricular que integra este trabalho (Processamento de Linguagens e Compiladores).

Neste documento está apresentada a gramática da nossa linguagem, o código escrito no módulo Lexer e Yacc do Python e ainda, como foi pedido, 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 *condicionais* para controlo do fluxo de execução;
- *efetuar* instruções *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 PLC
- Os símbolos
- O desenho da gramática independente de contexto
- Extras

No nosso trabalho, acabámos por implementar *arrays* unidimensionais e subprogramas sem retorno.

### 3.1 Sintaxe da Linguagem PLC

A sintaxe da linguagem é a seguinte

#### 3.1.1 Declaração de variáveis

```
1      int x
2      int x = 10
3      int x[n]
4      int x[10] = {1,2,3,4,5,6,7,8,9,10}
5
```

#### 3.1.2 Operadores de comparação

```
1      x <= y
2      x >= y
3      x < y
4      x > y
5      x == y
6
```

#### 3.1.3 Operações numéricas

```

1      x + y
2      x - y
3      x / y
4      x * y
5      x % y
6      x ++ #(incremento)
7      y -- #(decremento)
8

```

### 3.1.4 Operadores lógicos

```

1      x and y
2      x or y
3      not x
4

```

### 3.1.5 Instruções condicionais

```

1      if (cond):
2          ...
3      elif (cond):
4          ...
5      else:
6

```

### 3.1.6 Ciclo while

```

1      while (cond):
2          ...
3

```

### 3.1.7 Ciclo do-while

```

1      do:
2          ...
3      while (cond)
4

```

### 3.1.8 Input/Output

```

1      x = input()
2      x = input("Declare the variable with the value: ")
3      print("Hello world!")
4

```

### 3.1.9 Comentário

```

1      # isto e um comentario na nossa linguagem
2

```

### 3.1.10 Assert

A linguagem tem definidos asserts que utilizam as mensagens de erro da Máquina Virtual

```
1      assert (cond)
2
```

## 3.2 Símbolos

Os símbolos da linguagem são os seguintes:

```
'INTDec',
'NUM',
'ID',
'ATRIB',
'EQUIV',
'LEQ', # <= - (less or equal)
'GEQ', # >= - (greater or equal)
'GT', # > - (greater than)
'LT', # < - (less than)
'NEQ', # /= - (not equal -> NEQ -> NECC)
'LCPARENT',
'RCPARENT',
'LSQBRACKET', # left square bracket
'RSQBRACKET', # right square bracket
'LCURLBRACKET',
'RCURLBRACKET',
'SUM',
'SUB',
'DIV',
'MULT',
'MOD',
'INC',
'DEC',
'QUOTE', # Símbolo "
'String',
'NEWLINE',
'COLON',
'WS',
'INDENT',
'DEDENT',
'ENDMARKER',
'IF',
'ELSE',
'ASSERT',
'WHILE',
'DO',
'PRINT',
```

```

'INPUT',
'AND',
'OR',
'NOT',
'DEF',
'CALL'

```

### 3.3 Desenho da GIC

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

```

Programa : Decls Corpo
          | Corpo
Decls    : Decl
          | Decls Decl
Decl     : INTDec ID
          | INTDec ID ATRIB NUM
          | INTDec ID ATRIB Input
          | INTDec ID ATRIB INPUT LCBRACKET RCBRACKET
          | INTDec ID LSQBRACKET NUM RSQBRACKET
          | INTDec ID LSQBRACKET NUM RSQBRACKET LSQBRACKET NUM RSQBRACKET
Corpo    : Proc
          | Corpo Proc
Newline  : NEWLINE
          |
Proc     : Atrib
          | Print
          | If
          | Cycle
Print    : NonFormatted
          | Formatted (not implemented)
NonFormatted : PRINT LCPARENT QUOTE Argument QUOTE RCPARENT
Formatted : ....
Argument  : String
          | Expr

If        : IF LCPARENT cond RCPARENT COLON INDENT Corpo Dedent
          | IF LCPARENT cond RCPARENT COLON INDENT Corpo Dedent ELSE COLON INDENT

Atrib    : ID ATRIB Expr
          | ID ATRIB Input
          | ....

```



```

Cond      : Expr GT Expr
          | Expr LT Expr
          | Expr GEQ Expr
          | Expr LEQ Expr
          | Expr EQUIV Expr
          | Expr NEQ Expr
          | Cond OR Cond
          | Cond AND Cond
          | NOT Cond

Expr      : Var
          | NUM
          | ID INC
          | ID DEC
          | ID SUM ATRIB Expr
          | ID SUB ATRIB Expr
          | Expr SUM Expr
          | Expr SUB Expr
          | Expr DIV Expr
          | Expr MUL Expr
          | Expr MOD Expr

Var       : ID
Input    : INPUT LCPARENT String RCPARENT
String   : QUOTE STRING QUOTE
          |

```

## 3.4 Extras

Os extras implementados foram:

- Comentários
- Erros
- Ordem de operações(simples)
- Indentação obrigatória

### 3.4.1 Comentários

Os comentários funcionam através do consumo do padrão sem retornar valor.

```

1     def t_comment(t):
2         r'\#.*'
3         pass
4

```

### 3.4.2 Erros

Definimos mensagens de erro para ajudar o utilizador a corrigir os erros do seu programa.

```

1     def p_error(p):
2         print('Syntax error!\np -> ', p)
3         parser.sucesso = False
4

```

### 3.4.3 Ordem de operações(simples)

Definimos também a precedência dos operadores aritméticos para que o cálculo de uma expressão aritmética seja feito de acordo com a precedência habitual dos operadores(não tendo em conta expressões dentro de parêntesis):

```

1     precedence = (
2         ("left", "SUM", "SUB"),
3         ("left", "MULT", "DIV")
4     )
5

```

### 3.4.4 Indentação

Para implementar a indentação obrigatória inspiramo-nos num dos exemplos da documentação do PLY chamado "GardenSanke". O processo de verificação de indentação é feito através de dois filtros que são corridos após o analisador léxico. Os tokens utilizados para o processo são o 'COLON', a 'NEWLINE', o 'WS' o 'INDENT' e o 'DEDENT'.

O primeiro filtro identifica os tokens 'COLON', 'NEWLINE' e 'WS' e utiliza-os para identificar quais os tokens que devem ser indentados e quais os tokens no início das linhas para posteriormente verificar o nível da indentação.

O segundo filtro calcula as profundidades das indentações a emitir e os tokens 'INDENT' e 'DEDENT' para criar os blocos. Nos 'WS' inicializa a profundidade da indentação e retém o token neste filtro. Nos 'NEWLINE' reinicia profundidade da indentação. Nos restantes tokens verifica se a indentação está de acordo com a respectiva profundidade e emite os seus

tokens 'INDENT' e 'DEDENT' No final criamos 'DEDENT's que estejam em falta devido à terminação do ficheiro num estado aninhado.

## Capítulo 4

# Exemplos de funcionamento

— preencher exemplos —

## Capítulo 5

# Conclusão

Fazendo uma retrospectiva referente ao trabalho prático, entendemos que os todos os objetivos do trabalho prático foram cumpridos. A realização deste trabalho foi particularmente atrativa pois ao desenvolver a nossa própria linguagem de programação somos nós quem decide toda a sua sintaxe e notação e chegar ao fim e perceber que conseguimos desenvolver a base de uma linguagem de programação é satisfatório.

A realização deste trabalho prático fez com que ficássemos bem dentro do funcionamento do módulo Lexer e do módulo Yacc, nomeadamente como funciona o reconhecimento de tokens e a implementação da nossa gramática. A geração de código Assembly foi sem dúvida também um ponto positivo deste trabalho pois permitiu-nos entender melhor a linguagem e as suas instruções.

Em suma, entendemos que todos os objetivos foram concluídos e consideramos que este trabalho foi bastante desafiador e uma excelente fonte de conhecimento para desafios futuros.