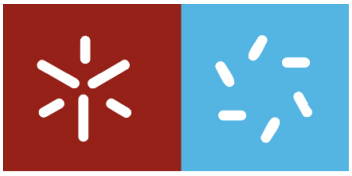

Universidade do Minho

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

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 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 atómicas 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 (ındice 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

```
x + y
x - y
x / y
x * y
x % y
x ++ #(incremento)
y -- #(decremento)

```

3.1.4 Operadores lógicos

```
x and y
x or y
not x

```

3.1.5 Instruções condicionais

```
if (cond):
    ...
else:

```

3.1.6 Ciclo while

```
while (cond):
    ...

```

3.1.7 Ciclo do-while

```
do:
    ...
while (cond)

```

3.1.8 Input/Output

```
x = input()
x = input("Declare the variable with the value: ")
print("Hello world!")

```

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 mensagems de erro da Máquina Virtual

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

3.2 Símbolos

Os simbolos 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:

```
ProgramaInit : Programa ENDMARKER
Programa : Decls Corpo
         | Corpo
Decls    : Decl Newline
         | Decls Decl Newline
Decl     : INTDec ID
         | INTDec ID ATRIB NUM
         | INTDec ID ATRIB Input
         | INTDec ID LSQBRACKET NUM RSQBRACKET ATRIB ArrayValues
         | INTDec ID LSQBRACKET NUM RSQBRACKET
         | Def
Def      : DEF ID COLON Newline INDENT Corpo DEDENT
         | DEF ID COLON Newline INDENT Decls Corpo DEDENT
ArrayValues    : LCURLBRACKET ArrayIntValues RCURLBRACKET
ArrayIntValues : ArrayIntValues ',' Expr
               | Expr
Corpo    : Proc
         | Corpo Proc
Newline  : NEWLINE
         |
Dedent   : Dedent DEDENT
         |
```

```
Proc     : Atrib
         | Print
         | If
         | Cycle
         | Call
         | Assert
Call     : CALL
Assert   : ASSERT LCPARENT Cond RCPARENT
Print    : NonFormatted
NonFormatted : PRINT LCPARENT Argument RCPARENT
Argument : String
         | Expr
If       : IF LCPARENT Cond RCPARENT COLON Newline INDENT Corpo Dedent
         | IF LCPARENT Cond RCPARENT COLON Newline INDENT Corpo Dedent ELSE C
Cycle    : While
         | DoWhile
DoWhile  : DO COLON Newline INDENT Corpo Dedent WHILE LCPARENT Cond RCPARENT N
While    : WHILE LCPARENT Cond RCPARENT COLON Newline INDENT Corpo Dedent
Atrib    : ID ATRIB Expr
         | ID ATRIB Input
         | ID INC
         | ID DEC
         | ID LSQBRACKET Expr RSQBRACKET ATRIB Expr
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
         | ExprIncDec
         | NUM
         | ID INC
         | ID DEC
         | ID SUM ATRIB Expr
         | ID SUB ATRIB Expr
         | Expr SUM Expr
         | Expr SUB Expr
         | Expr DIV Expr
         | Expr MULT Expr
         | Expr MOD Expr
```

```
Var      : ID
         | ID LSQBRACKET Expr RSQBRACKET
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 messagens 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):

```
        precedence = (
            ("left", "SUM", "SUB"),
            ("left", "MULT", "DIV")
        )

```

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 retêm o token neste filtro. Nos 'NEWLINE' reinicia profundidade da indentação. Nos restantes tokens verifica se a indentação está de acodordo 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

4.0.1 Assert

```
print("Entrei no segundo if\n")
assert(1 < 2)
print("Entrei no segundo if\n")
assert (1 > 2)

```

Código Assembly gerado:

```
START
PUSHS "Entrei no segundo if\n"
WRITES
PUSHI 1
PUSHI 2
INF
JZ label0
JUMP label0f
label0: NOP
ERR "False assertion in line 2"
label0f: NOP
PUSHS "Entrei no segundo if\n"
WRITES
PUSHI 1
PUSHI 2
SUP
JZ label1
JUMP label1f
label1: NOP
ERR "False assertion in line 4"
label1f: NOP
STOP

```

4.0.2 Bubble-Sort

```
int array[5] = {5,4,3,2,1}
int tamanho = 4
int i
int j
int temp
while(i < tamanho):
    j = 0
    while(j < tamanho):
        if (array[j] > array[j+1]):
            temp = array[j]
            array[j] = array[j+1]
            array[j+1] = temp
        j++
    i++

print(array[0])
print(array[1])
print(array[2])
print(array[3])
print(array[4])
print("Estou ordenado! :)")

```

Código Assembly gerado:

```
PUSHI 5
PUSHI 4
PUSHI 3
PUSHI 2
PUSHI 1
PUSHI 4
PUSHI 0
PUSHI 0
PUSHI 0

START
label2c: NOP
PUSHG 6
PUSHG 5
INF
JZ label2f
PUSHI 0
STOREG 7
label1c: NOP
PUSHG 7
PUSHG 5
INF
JZ label1f
PUSHGP
PUSHI 0
PADD
PUSHG 7
LOADN
```

```
PUSHGP
PUSHI 0
PADD
PUSHG 7
PUSHI 1
ADD
LOADN
SUP
JZ label0
PUSHGP
PUSHI 0
PADD
PUSHG 7
LOADN
STOREG 8
PUSHGP
PUSHI 0
PADD
PUSHG 7
PUSHGP
PUSHI 0
PADD
PUSHG 7
PUSHI 1
ADD
LOADN
STOREN
PUSHGP
PUSHI 0
PADD
PUSHG 7
PUSHI 1
ADD
PUSHG 8
STOREN
label0: NOP
PUSHG 7
PUSHI 1
ADD
STOREG 7
JUMP label1c
label1f: NOP
PUSHG 6
PUSHI 1
ADD
STOREG 6
JUMP label2c
label2f: NOP
PUSHGP
PUSHI 0
PADD
PUSHI 0
LOADN
WRITEI
```

```
WRITELN
PUSHGP
PUSHI 0
PADD
PUSHI 1
LOADN
WRITEI
WRITELN
PUSHGP
PUSHI 0
PADD
PUSHI 2
LOADN
WRITEI
WRITELN
PUSHGP
PUSHI 0
PADD
PUSHI 3
LOADN
WRITEI
WRITELN
PUSHGP
PUSHI 0
PADD
PUSHI 4
LOADN
WRITEI
WRITELN
PUSHS "Estou ordenado! :)"
WRITES
STOP

```

4.0.3 Do-While

```
int opcao
int valores = 0

do:
    print("Menu: \n")
    print("1- Ter 20 a PLC\n")
    print("2- Ir a recurso\n")
    opcao = input("Escolha a sua opcao: ")
    print("\n")
    if (opcao == 1):
        valores++
        print("Faltam estes valores para o 20: ")
        print(20-valores)

    print("\n")
while(opcao /= 2)

```

```
18
19 if (opcao == 2):
20     print("Boa sorte no recurso :(\n")
21 else:
22     print("Acabaste com estes valores: ")
23     print(valores)
24
```

Código Assembly gerado:

```
1  PUSHI 0
2  PUSHI 0
3
4  START
5  label1:
6  PUSHS "Menu: \n"
7  WRITES
8  PUSHS "1- Ter 20 a PLC\n"
9  WRITES
10 PUSHS "2- Ir a recurso\n"
11 WRITES
12 PUSHS "Escolha a sua opcao: "
13 WRITES
14 READ
15 ATOI
16 STOREG 0
17 PUSHS "\n"
18 WRITES
19 PUSHG 0
20 PUSHI 1
21 EQUAL
22 JZ label0
23 PUSHG 1
24 PUSHI 1
25 ADD
26 STOREG 1
27 PUSHS "Faltam estes valores para o 20: "
28 WRITES
29 PUSHI 20
30 PUSHG 1
31 SUB
32 WRITEI
33 WRITELN
34 label0: NOP
35 PUSHS "\n"
36 WRITES
37 PUSHG 0
38 PUSHI 2
39 EQUAL
40 NOT
41 NOT
42 JZ label1
43 PUSHG 0
```

```
PUSHI 2
EQUAL
JZ label2
PUSHS "Boa sorte no recurso :(\n"
WRITES
JUMP label2f
label2: NOP
PUSHS "Acabaste com estes valores: "
WRITES
PUSHG 1
WRITEI
WRITELN
label2f: NOP
STOP

```

4.0.4 Lógica

```
int bitUm = 0
int bitDois = 1
int bitTres = 1

if(bitUm or bitDois):
    if (bitTres and bitDois):
        print("Entrei no segundo if\n")
    else:
        print("Entrei no primeiro else\n")
else:
    print("Entrei no segundo else\n")

```

Código Assembly gerado:

```
PUSHI 0
PUSHI 1
PUSHI 1

START
PUSHG 0
PUSHG 1
ADD
PUSHI 1
SUPEQ
JZ label1
PUSHG 2
PUSHG 1
ADD
PUSHI 2
SUPEQ
JZ label0
PUSHS "Entrei no segundo if\n"
WRITES
```

```
JUMP label0f
label0: NOP
PUSHS "Entrei no primeiro else\n"
WRITES
label0f: NOP
JUMP label1f
label1: NOP
PUSHS "Entrei no segundo else\n"
WRITES
label1f: NOP
STOP

```

4.0.5 Paridade

```
int x = 1
int y = 5
int i
int array[5] = {x+y, 4, 2, y, 57}

def par:
    print("O numero e par\n")

def impar:
    print("O numero e impar\n")

while (i < 5):
    if (array[i] % 2 == 0):
        par()
    else:
        impar()
    i++

print("algo")

```

Código Assembly gerado:

```
PUSHI 1
PUSHI 5
PUSHI 0
PUSHG 0
PUSHG 1
ADD
PUSHI 4
PUSHI 2
PUSHG 1
PUSHI 57
JUMP function0Ignore
function0:
PUSHS "O numero e par\n"
WRITES
```

```
RETURN
function0Ignore:

JUMP function1Ignore
function1:
PUSHS "O numero e impar\n"
WRITES
RETURN
function1Ignore:


START
label1c: NOP
PUSHG 2
PUSHI 5
INF
JZ label1f
PUSHGP
PUSHI 3
PADD
PUSHG 2
LOADN
PUSHI 2
MOD
PUSHI 0
EQUAL
JZ label0
PUSHA function0
CALL
JUMP label0f
label0: NOP
PUSHA function1
CALL
label0f: NOP
PUSHG 2
PUSHI 1
ADD
STOREG 2
JUMP label1c
label1f: NOP
PUSHS "algo"
WRITES
STOP

```

4.0.6 Quadrado

```
int x[5] = {1,2,3,4,5}
int tamanho = 5
int i

while(i < tamanho):
    x[i] = x[i] * x[i]
```

```
    i++

print(x[0])
print(x[1])
print(x[2])
print(x[3])
print(x[4])

```

Código Assembly gerado:

```
PUSHI 1
PUSHI 2
PUSHI 3
PUSHI 4
PUSHI 5
PUSHI 5
PUSHI 0

START
label0c: NOP
PUSHG 6
PUSHG 5
INF
JZ label0f
PUSHGP
PUSHI 0
PADD
PUSHG 6
PUSHGP
PUSHI 0
PADD
PUSHG 6
LOADN
PUSHGP
PUSHI 0
PADD
PUSHG 6
LOADN
MUL
STOREN
PUSHG 6
PUSHI 1
ADD
STOREG 6
JUMP label0c
label0f: NOP
PUSHGP
PUSHI 0
PADD
PUSHI 0
LOADN
WRITEI
```

```
WRITELN
PUSHGP
PUSHI 0
PADD
PUSHI 1
LOADN
WRITEI
WRITELN
PUSHGP
PUSHI 0
PADD
PUSHI 2
LOADN
WRITEI
WRITELN
PUSHGP
PUSHI 0
PADD
PUSHI 3
LOADN
WRITEI
WRITELN
PUSHGP
PUSHI 0
PADD
PUSHI 4
LOADN
WRITEI
WRITELN
STOP

```

Capítulo 5

Conclusão

Fazendo uma retrospetiva 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.