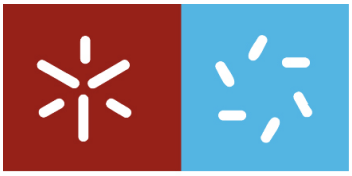

Universidade do Minho

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

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

```

3.1.5 Instruções condicionais

```
if (cond):
    ...
elif (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

```
# isto e um comentario na nossa linguagem

```

3.2 Símbolos

Os simbolos da linguagem são os seguintes:

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

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 INDE

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)
- Asserts?
- 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):

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

3.4.4 Indentação

– colocar indentação

Capítulo 4

Exemplos de funcionamento

— preencher exemplos –

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.