



Processamento de Linguagens

Trabalho Prático 2
30 de Maio de 2021
Grupo 28



António Santos, A83700



Jorge Vieira, A84240



Pedro Fernandes, A84313

Contents

1	Introdução	4
1.1	Contexto	4
1.2	Problema	4
1.3	Objetivos	5
1.4	Estrutura do documento	5
2	A linguagem de programação	6
2.1	Condições	6
2.2	Ciclos	6
2.3	Array	6
2.4	Parser	7
2.5	Tokens	7
2.6	Regras da gramática	8
3	Testes	10
3.1	Ler 4 numeros e dizer se podem ser os lados de um quadrado	10
3.1.1	Pseudo-código	10
3.1.2	Código VM	10
3.1.3	Execução	11
3.2	Ler um inteiro N, depois ler N numeros e escrever o menor deles	12
3.2.1	Pseudo-código	12
3.2.2	Código VM	12
3.2.3	Execução	13
3.3	Ler N (constante do programa) numeros e calcular e imprimir o seu produtorio.	13
3.3.1	Pseudo-código	13
3.3.2	Código VM	13
3.3.3	Execução	14
3.4	Contar e imprimir os numeros impares de uma sequencia de numeros naturais.	15
3.4.1	Pseudo-código	15
3.4.2	Código VM	15
3.4.3	Execução	16
3.5	Ler e armazenar N numeros num array; imprimir os valores por ordem inversa.	17
3.5.1	Pseudo-código	17

3.5.2	Código VM	17
3.5.3	Execução	18
3.6	[EXTRA] Ler dois numeros e imprimir os divisores em comum de ambos	18
3.6.1	Pseudo-código	18
3.6.2	Código VM	18
3.6.3	Execução	20
4	Conclusão	21
A	Lexer	22
B	Compilador	24

Chapter 1

Introdução

1.1 Contexto

No contexto da Unidade Curricular de Processamento de Linguagens, foi proposto ao grupo o desenvolvimento de uma linguagem imperativa, a gosto e um compilador para esta linguagem que gere pseudo-código Assembly da Máquina Virtual VM.

Esta linguagem de programação deveria permitir também certas funcionalidades como:

- declarar variáveis atômicas do tipo inteiro com as quais se podem realizar operações aritméticas, lógicas e relacionais.
- efetuar instruções algorítmicas básicas
- ler do standard input e escrever no standard output.
- efetuar instruções condicionais para controlo de fluxo de execução
- efetuar instruções cíclicas

Por escolha do grupo:

- declarar e manusear variáveis do tipo *array* de inteiros, em relação aos é apenas permitida a operação de indexação.

1.2 Problema

Neste trabalho prático o desafio foi conseguir desenvolver uma linguagem imperativa utilizando os módulos *Yacc* e *Lex* do *PLY* que tivesse todas as funcionalidades pedidas e que de certa forma fosse de fácil utilização e compreensão. Para além disso, a parte do compilador e da Virtual Machine foi também trabalhosa pois o grupo teve que aprender como a VM funciona e as suas instruções para que fosse possível que gerar o pseudo-código Assembly corretamente através da nossa linguagem.

1.3 Objetivos

Os principais objetivos do grupo ao realizar o trabalho foram:

1. Desenvolver uma linguagem que cumprisse com todos os requisitos do enunciado e que fosse de fácil utilização e compreensão.
2. Conseguir produzir o pseudo-código Assembly através do compilador corretamente.
3. Ter a certeza que a linguagem passava nos testes explicitados no enunciado.

1.4 Estrutura do documento

Este relatório está dividido em 4 capítulos sendo o primeiro a Introdução ao projeto, o segundo um resumo da implementação de cada funcionalidade da linguagem desenvolvida, o terceiro apresenta o resultado dos testes realizados (pedidos no enunciado). E o quarto uma reflexão sobre o desenvolvimento do programa no geral. Também está incluído em apêndice com todo o código necessário para o funcionamento do programa.

Chapter 2

A linguagem de programação

O grupo inspirou-se na syntax relaxada do python para a estrutura, com esta lingua o utilizador tem liberdade de estruturar o código como quiser sem ser necessário indentação para o funcionamento dos programas. Na maior parte dos casos não é necessário a declaração prévia das variáveis com a exceção dos arrays que necessitam de uma declaração com o tamanho desejado.

2.1 Condições

A lingua de programação suporta if statements como statement condicional, suportando comparações de maior (ou igual), menor (ou igual), diferentes ou iguais. É possível juntar várias condições com o uso do OR ou AND.

A estrutura dos statements é o seguinte:

```
if(condition){instructions}
```

Se usarmos o OR ficará:

```
if(condition or conditions){instructions}
```

2.2 Ciclos

A lingua de programação apenas suporta "repeat-until" para fazer ciclos, as condições de paragem suportam as mesmas comparações que o if statement sendo bastante flexíveis para o utilizador.

A estrutura dos ciclos é a seguinte:

```
repeat{instructions}until(conditions)
```

2.3 Array

Os arrays suportados pelo projeto são muito básicos sendo apenas possível armazenar numeros inteiros. Para utilizar um array é necessário declarar previamente referenciando o tamanho. Para declarar um array b de tamanho 5 fazemos o seguinte:

```
array(b,5)
```

A unica maneira suportada de adicionar elementos ao array é a seguinte:

```
b[indice] = numero_inteiro
```

2.4 Parser

Este projeto é compilado por um *parser* LR capaz de analisar todo o código em tempo linear, dessa forma, todas as derivações presentes no projeto recorrem á recursividade pela esquerda. O parser é capaz de interpretar o pseudo-código e posteriormente converter em instruções assembly compatíveis com a máquina virtual fornecida pela equipa docente.

2.5 Tokens

Os tokens presentes são os seguintes:

```
'int', 'id', 'float', 'string', 'and', 'or', 'if', 'else', 'array', 'repeat', 'until', 'read'
```

As expressões regulares pertencentes aos tokens são os seguintes:

```
def t_float(t):  
    r'\d+\.\d+'  
    t.value = float(t.value)  
    return t  
  
def t_and(t):  
    r'and'  
    return t  
  
def t_if(t):  
    r'if'  
    return t  
  
def t_else(t):  
    r'else'  
    return t  
  
def t_or(t):  
    r'or'  
    return t  
  
def t_array(t):  
    r'array'  
    return t  
  
def t_repeat(t):  
    r'repeat'  
    return t
```

```

def t_until(t):
    r'until'
    return t

def t_read(t):
    r'read'
    return t

def t_int(t):
    r'\d+'
    t.value = int(t.value)
    return t

def t_string(t):
    r'"[\w ]{2,}"'
    t.value = t.value[1:-1]
    return t

def t_id(t):
    r'[a-z]+'
    return t

```

2.6 Regras da gramática

As regras usadas na produção são as seguintes:

```

"Main : Instrucoes"

"Instrucoes : Instrucoes Instrucao"

"Instrucoes : Instrucao"

"Instrucao : Atr"

"Instrucao : repeat '{' Instrucoes '}' until '(' Conditions ')'"

"Instrucao : if '(' Conditions ')' '{' Instrucoes '}'"

"Instrucao : if '(' Conditions ')' '{' Instrucoes '}' else '{' Instrucoes '}'"

"Conditions : Cond"

"Conditions : Conditions or Cond"

"Condition : Exp '<' Exp"

"Condition : Exp '>' Exp"

"Condition : Exp '=' Exp"

"Condition : Exp '<' '=' Exp"

```



```

"Condition : Exp '>' '=' Exp"
"Condition : Exp '!' '=' Exp"
"Cond : Cond and Cond2"
"Cond : Cond2"
"Cond2 : '!' Cond"
"Cond2 : Condition"
"Cond2 : '(' Conditions ')'"
"Atr : id '=' Exp"
"Atr : array '(' id ',' int ')'"
"Atr : id '[' Exp ']'" '=' Exp"
  "Instrucao : '$' string"
  "Instrucao : '$' Exp"
  "Atr : read '(' id ')'"
"Exp : Exp '+' Term"
"Term : Term '-' Factor"
"Term : Term '%' Factor"
"Exp : Term"
"Term : Term '*' Factor"
"Term : Term '/' Factor"
"Term : Factor"
"Factor : id"
"Factor : int"
"Factor : float"
"Factor : id '[' Exp ']'"

```

Chapter 3

Testes

No enunciado são pedidos alguns exemplos:

3.1 Ler 4 numeros e dizer se podem ser os lados de um quadrado

3.1.1 Pseudo-código

```
a=0 b=0 c=0 d=0
$"Lado A"
read(a)
$"Lado B"
read(b)
$"Lado C"
read(c)
$"Lado D"
read(d)
if(a==b and b==c and c==d){$"Sou um quadrado"}else{ $"Nao sou um quadrado"}
```

3.1.2 Código VM

```
START
PUSHI 0
PUSHI 0
PUSHI 0
PUSHI 0
PUSHS "Lado A\n"
WRITES
READ
ATOI
STOREG 0
PUSHS "Lado B\n"
WRITES
READ
ATOI
STOREG 1
PUSHS "Lado C\n"
```

```


WRITES
READ
ATOI
STOREG 2
PUSHS "Lado D\n"
WRITES
READ
ATOI
STOREG 3
PUSHG 0
PUSHG 1
EQUAL
PUSHG 1
PUSHG 2
EQUAL
MUL
PUSHG 2
PUSHG 3
EQUAL
MUL
JZ ELSE0
PUSHS "Sou um quadrado\n"
WRITES

JUMP ENDO
ELSE0:
PUSHS "Nao sou um quadrado\n"
WRITES

END0:
STOP

```

3.1.3 Execução



```

Lado A
3
Lado B
3
Lado C
3
Lado D
3
Sou um quadrado

```

Figure 3.1: Execução exercicio 1

3.2 Ler um inteiro N, depois ler N numeros e escrever o menor deles

3.2.1 Pseudo-código

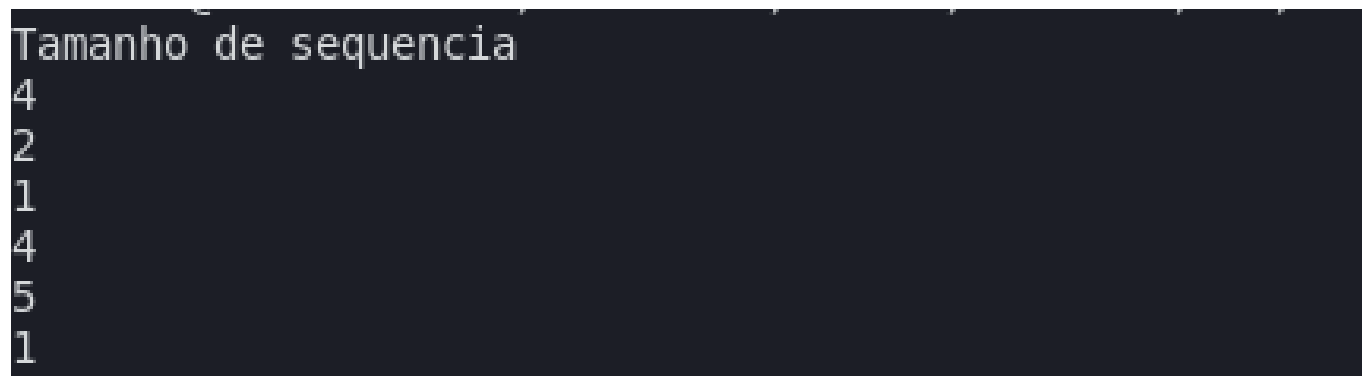
```
s=0
m=0
c=0
$"Tamanho de sequencia"
read(s)
read(m)
s=s-1
repeat{s=s-1 read(c) if(m>c){m = c}}until(s==0)
$m
```

3.2.2 Código VM

```
START
PUSHI 0
PUSHI 0
PUSHI 0
PUSHS "Tamanho de sequencia\n"
WRITES
READ
ATOI
STOREG 0
READ
ATOI
STOREG 1
PUSHG 0
PUSHI 1
SUB
STOREG 0
REPEAT1:
PUSHG 0
PUSHI 0
EQUAL
PUSHI 0
EQUAL
JZ END1
PUSHG 0
PUSHI 1
SUB
STOREG 0
READ
ATOI
STOREG 2
PUSHG 1
PUSHG 2
SUP
JZ END0
PUSHG 2
STOREG 1
```

```
END0:  
JUMP REPEAT1  
END1:  
PUSHG 1  
WRITEI  
PUSHS "\n"  
WRITES  
STOP
```

3.2.3 Execução



```
Tamanho de sequencia  
4  
2  
1  
4  
5  
1
```

Figure 3.2: Execução exercicio 2

3.3 Ler N (constante do programa) numeros e calcular e imprimir o seu produtorio.

3.3.1 Pseudo-código

```
s=10  
m=0  
r=1  
repeat{s=s-1 read(m) r = r*m}until(s==0)  
$r
```


3.3.2 Código VM

```
START  
PUSHI 10  
PUSHI 0  
PUSHI 1  
REPEAT0:  
PUSHG 0  
PUSHI 0
```

```
EQUAL
PUSHI 0
EQUAL
JZ ENDO
PUSHG 0
PUSHI 1
SUB
STOREG 0
READ
ATOI
STOREG 1
PUSHG 2
PUSHG 1
MUL
STOREG 2

JUMP REPEATO
END0:
PUSHG 2
WRITEI
PUSHS "\n"
WRITES
STOP
```

3.3.3 Execução



```
3
1
2
5
4
5
8
4
3
2
115200
```

Figure 3.3: Execução exercicio 3

3.4 Contar e imprimir os numeros impares de uma sequencia de numeros naturais.

3.4.1 Pseudo-código

```
a = 0
b = 0
c = 0
$"Limite inferior"
read(a)
$"Limite superior"
read(b)

repeat{a=a+1 if(a%2==0){$a c = c + 1}}until(a==b)

$"Numeros impares"
$c
```

3.4.2 Código VM

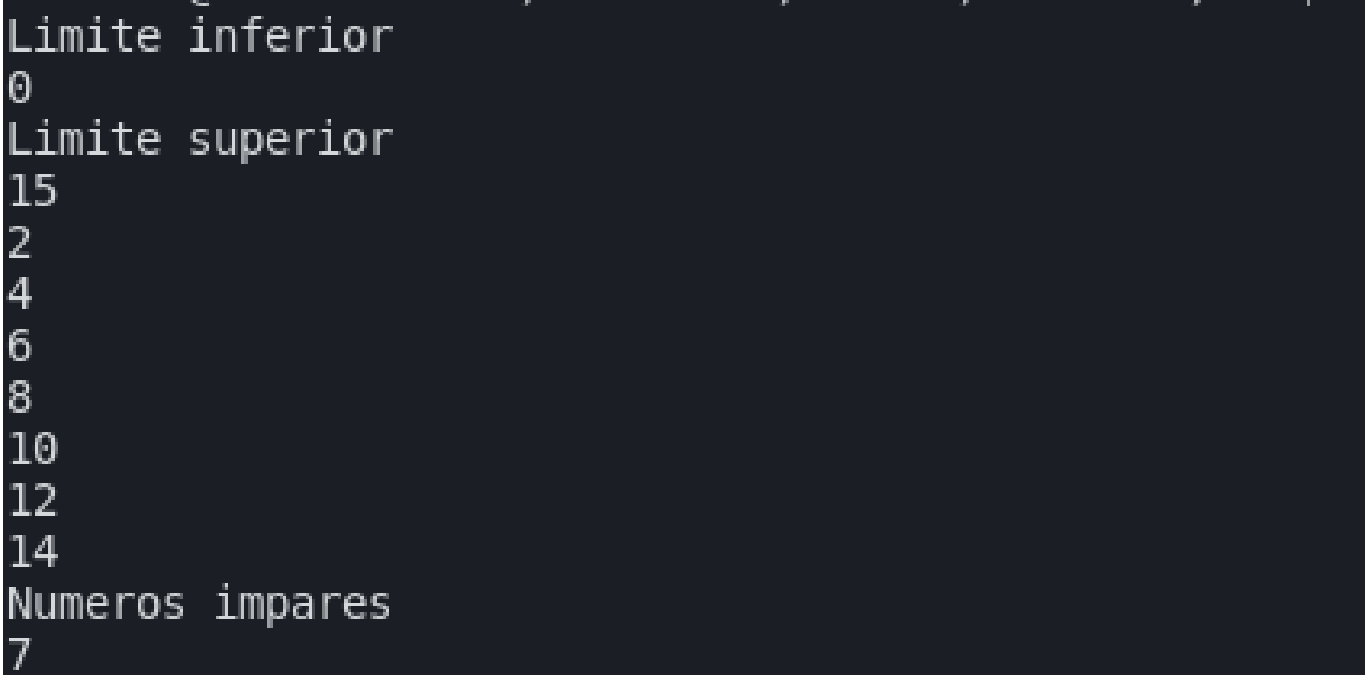
```
START
PUSHI 0
PUSHI 0
PUSHI 0
PUSHS "Limite inferior\n"
WRITES
READ
ATOI
STOREG 0
PUSHS "Limite superior\n"
WRITES
READ
ATOI
STOREG 1
REPEAT1:
PUSHG 0
PUSHG 1
EQUAL
PUSHI 0
EQUAL
JZ END1
PUSHG 0
PUSHI 1
ADD
STOREG 0
PUSHG 0
PUSHI 2
MOD
PUSHI 0
EQUAL
JZ ENDO
```

```
PUSHG 0
WRITEI
PUSHS "\n"
WRITES
PUSHG 2
PUSHI 1
ADD
STOREG 2

END0:
JUMP REPEAT1
END1:
PUSHS "Numeros impares\n"
WRITES
PUSHG 2
WRITEI
PUSHS "\n"
WRITES

STOP
```

3.4.3 Execução



```
Limite inferior
0
Limite superior
15
2
4
6
8
10
12
14
Numeros impares
7
```

Figure 3.4: Execução exercicio 4

3.5 Ler e armazenar N numeros num array; imprimir os valores por ordem inversa.

3.5.1 Pseudo-código

```
array(a,3)
c=3
a[0] = 1
a[1] = 3
a[2] = 6

repeat{c = c - 1 $a[c]}until(c==0)
```

3.5.2 Código VM

```
START
PUSHN 3
PUSHI 3
PUSHGP
PUSHI 0
PADD
PUSHI 0
PUSHI 1
STOREN
PUSHGP
PUSHI 0
PADD
PUSHI 1
PUSHI 3
STOREN
PUSHGP
PUSHI 0
PADD
PUSHI 2
PUSHI 6
STOREN
REPEATO:
PUSHG 3
PUSHI 0
EQUAL
PUSHI 0
EQUAL
JZ ENDO
PUSHG 3
PUSHI 1
SUB
STOREG 3
PUSHGP
PUSHI 0
PADD
PUSHG 3
```

```
LOADN
WRITEI
PUSHS "\n"
WRITES
JUMP REPEATO
ENDO:
STOP
```

3.5.3 Execução

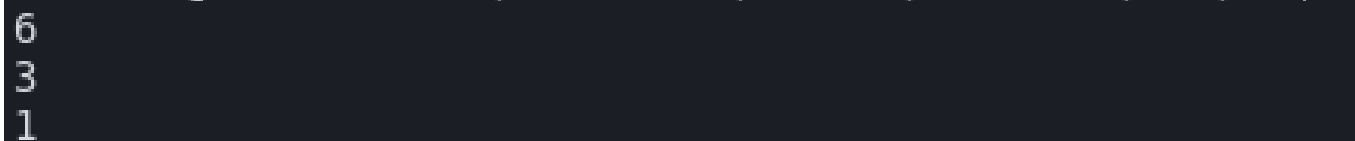


Figure 3.5: Execução exercício 5

3.6 [EXTRA] Ler dois numeros e imprimir os divisores em comum de ambos

3.6.1 Pseudo-código

```
a=0
b=0
c=0
m=0
$"Primeiro numero"
read(a)
$"Segundo numero"
read(b)

if(a<=b){m=a}else{m=b}

$"Divisores comuns"
repeat{c=c+1 if(a%c==0 and b%c==0){$c}}until(c==m)
```

3.6.2 Código VM

```
START
PUSHI 0
PUSHI 0
PUSHI 0
PUSHI 0
PUSHS "Primeiro numero\n"
WRITES
READ
```

```

ATOI
STOREG 0
PUSHS "Segundo numero\n"
WRITES
READ
ATOI
STOREG 1
PUSHG 0
PUSHG 1
INFEQ
JZ ELSE0
PUSHG 0
STOREG 3

JUMP ENDO
ELSE0:
PUSHG 1
STOREG 3

END0:PUSHS "Divisores comuns\n"
WRITES
REPEAT2:
PUSHG 2
PUSHG 3
EQUAL
PUSHI 0
EQUAL
JZ END2
PUSHG 2
PUSHI 1
ADD
STOREG 2
PUSHG 0
PUSHG 2
MOD
PUSHI 0
EQUAL
PUSHG 1
PUSHG 2
MOD
PUSHI 0
EQUAL
MUL
JZ END1
PUSHG 2
WRITEI
PUSHS "\n"
WRITES

END1:
JUMP REPEAT2
END2:

STOP

```

3.6.3 Execução

```
Primeiro numero
36
Segundo numero
60
Divisores comuns
1
2
3
4
6
12
```

Figure 3.6: Execução exercicio extra

Chapter 4

Conclusão

Em jeito de conclusão, o grupo acredita que cumpriu com todos os objetivos principais do trabalho e conseguiu produzir uma linguagem coerente e de fácil utilização e um compilador eficiente que gera corretamente pseudo-código da VM a partir da linguagem criada, cumprindo assim os requisitos do enunciado.

Com este trabalho o grupo aprendeu que tinha em sua disposição uma ferramenta potente e flexível para criar linguagens e compilá-las, o que faz com que estes módulos do *PLY/Python*, *Yacc* e *Lex* possam ser uma mais valia para qualquer tipo de projeto no futuro.

Em termos pedagógicos, todos os elementos concluem que o trabalho foi bastante enriquecedor e que aprofundou o nosso conhecimento na área da UC.

Appendix A

Lexer

```
import ply.lex as lex
import sys

tokens = (
    'int', 'id', 'float', 'string', 'and', 'or', 'if', 'else', 'array', 'repeat', 'until', 'read'
)
# Literals
literals = ['+', '-', '*', '/', '(', ')', '?', '!', '<', '>', '[', ']', ',', '{', '}', '=', '$', '%']

def t_float(t):
    r'\d+\.\d+'
    t.value = float(t.value)
    return t

def t_and(t):
    r'and'
    return t

def t_if(t):
    r'if'
    return t

def t_else(t):
    r'else'
    return t

def t_or(t):
    r'or'
    return t

def t_array(t):
    r'array'
    return t

def t_repeat(t):
    r'repeat'
    return t
```

```

def t_until(t):
    r'until'
    return t

def t_read(t):
    r'read'
    return t

def t_int(t):
    r'\d+'
    t.value = int(t.value)
    return t

def t_string(t):
    r'"[\w ]{2,}"'
    t.value = t.value[1:-1]
    return t

def t_id(t):
    r'[a-z]+'
    return t

def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

t_ignore = ' \t'

def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)

#-----
# INICIALIZAR LEXER #
lexer = lex.lex()

```

Appendix B

Compilador

```
import ply.yacc as yacc
import sys
from tp2_lex import tokens
from tp2_lex import literals

#Inicializacao das estruturas de dados#
varDic = dict({})
arrayDic = dict({})
stackPos = 0
label = 0

#INICIALIZACAO#

def p_Main(p):
    "Main : Instrucoes"
    p[0] = "START\n{0}\nSTOP\n".format(p[1])

#INSTRUCOES#

def p_Instrucoes_Instrucoes(p):
    "Instrucoes : Instrucoes Instrucao"
    p[0] = p[1] + p[2]

def p_Instrucoes_Instrucao(p):
    "Instrucoes : Instrucao"
    p[0] = p[1]

def p_Instrucao_Atrib(p):
    "Instrucao : Atr"
    p[0] = p[1]

def p_Instrucao_loop(p):
    "Instrucao : repeat '{' Instrucoes '}' until '(' Conditions ')'"
    global label
    p[0] = "REPEAT{0}:\n{1}\nPUSHI 0\nEQUAL\nJZ END{0}\n{2}\nJUMP\nREPEAT{0}\nEND{0}:\n".format(label,p[7],p[3])
    label+=1
```



```

def p_Instrucao_Condition(p):
    "Instrucao : if '(' Conditions ')' '{' Instrucoes '}"
    global stackPos
    global label

    p[0] = "{0}\nJZ END{1}\n{2}\nEND{1}:".format(p[3],label,p[6])
    label+=1

def p_Instrucao_Condition_else(p):
    "Instrucao : if '(' Conditions ')' '{' Instrucoes '}' else '{' Instrucoes '}'"
    global stackPos
    global label

    p[0] = "{0}\nJZ ELSE{1}\n{2}\nJUMP\nEND{1}\nELSE{1}:\n{3}\nEND{1}:".format(p[3],label,p[6],p[10])
    label+=1

#CONDICOES#

def p_Conditions_Cond(p):
    "Conditions : Cond"
    p[0] = p[1]

def p_Conditions_or_Cond(p):
    "Conditions : Conditions or Cond"
    p[0] = "{0}\n{1}\nADD\n{0}\n{1}\nMUL\nSUB".format(p[1],p[3])

def p_Condition_less(p):
    "Condition : Exp '<' Exp"
    global stackPos
    p[0] = "{0}\n{1}\nINF".format(str(p[1]),str(p[3]))
    stackPos-=2

def p_Condition_more(p):
    "Condition : Exp '>' Exp"
    global stackPos
    p[0] = "{0}\n{1}\nSUP".format(str(p[1]),str(p[3]))
    stackPos-=2

def p_Condition_equals(p):
    "Condition : Exp '=' Exp"
    global stackPos
    p[0] = "{0}\n{1}\nEQUAL".format(str(p[1]),str(p[4]))
    stackPos-=2

def p_Condition_less_equals(p):
    "Condition : Exp '<' '=' Exp"
    global stackPos
    p[0] = "{0}\n{1}\nINFEQ".format(str(p[1]),str(p[4]))
    stackPos-=2

def p_Condition_more_equals(p):
    "Condition : Exp '>' '=' Exp"
    global stackPos

```

```

p[0] = "{0}\n{1}\nSUPEQ".format(str(p[1]),str(p[4]))
stackPos-=2

def p_Condition_different(p):
    "Condition : Exp '!' '=' Exp"
    global stackPos
    p[0] = "{0}\n{1}\nEQUAL\nNOT".format(str(p[1]),str(p[4]))
    stackPos-=2

def p_Cond_Cond_and(p):
    "Cond : Cond and Cond2"
    p[0] = "{0}\n{1}\nMUL".format(p[1],p[3])

def p_Cond_Cond2(p):
    "Cond : Cond2"
    p[0] = p[1]

def p_Cond2_Not(p):
    "Cond2 : '!' Cond"
    p[0] = "{0}\nPUSHI 0\nEQUAL".format(p[2])

def p_Cond2_Condition(p):
    "Cond2 : Condition"
    p[0] = p[1]

def p_Cond2_Conditions(p):
    "Cond2 : '(' Conditions ')'"
    p[0] = p[1]

#ATRIBUICOES#

def p_Atr_id(p):
    "Atr : id '=' Exp"
    global stackPos
    if p[1] not in varDic:
        varDic[p[1]] = stackPos-1
        p[0] = str(p[3])+"\n"
    else:
        p[0] = str(p[3])+"\nSTOREG "+str(varDic[p[1]])+"\n"
        stackPos-=1

def p_decl_Array(p):
    "Atr : array '(' id ',' int ')'"
    global stackPos
    p[0] = "PUSHN {0}\n".format(p[5])
    arrayDic[p[3]] = [p[5],stackPos]
    stackPos+= int(p[5])

def p_Atr_int_Array(p):
    "Atr : id '[' Exp ']' '=' Exp"
    global stackPos
    if p[1] in arrayDic:
        p[0] = "PUSHGP\nPUSHI {1}\nPADD\n{0}\n{2}\nSTOREN\n".format(p[3],arrayDic[p[1]][1],p[6])

```

```

        stackPos-=1
    else:
        pass

def p_Atr_print_str(p):
    "Instrucao : '$' string"
    global stackPos
    p[0] = 'PUSHS "{0}\\n"\\n{1}\\n'.format(p[2], "WRITES")

def p_Atr_print_exp(p):
    "Instrucao : '$' Exp"
    global stackPos
    p[0] = '{0}\\n{1}\\nPUSHS "\\n"\\nWRITES\\n'.format(p[2], "WRITEI")

def p_Atr_read(p):
    "Atr : read '(' id ')'"
    global stackPos
    if p[3] in varDic:
        p[0] = "{0}\\n{1}\\nSTOREG {2}\\n".format("READ", "ATOI", varDic[p[3]])
        stackPos-=1
    else:
        varDic[p[3]] = stackPos
        p[0] = "{0}\\n{1}\\n".format("READ", "ATOI")

#OPERACOES ARITMETICAS#
def p_Exp_add(p):
    "Exp : Exp '+' Term"
    global stackPos
    p[0] = "{0}\\n{1}\\n{2}".format(p[1], p[3], "ADD")
    stackPos-=1

def p_Exp_sub(p):
    "Term : Term '-' Factor"
    global stackPos
    p[0] = "{0}\\n{1}\\n{2}".format(p[1], p[3], "SUB")
    stackPos-=1

def p_Exp_mod(p):
    "Term : Term '%' Factor"
    global stackPos
    p[0] = "{0}\\n{1}\\n{2}".format(p[1], p[3], "MOD")
    stackPos-=1

def p_Exp_term(p):
    "Exp : Term"
    p[0] = p[1]

#TERMOS#
def p_Term_mul(p):
    "Term : Term '*' Factor"
    global stackPos
    p[0] = "{0}\\n{1}\\n{2}".format(p[1], p[3], "MUL")
    stackPos-=1

```

```

def p_Term_div(p):
    "Term : Term '/' Factor"
    global stackPos
    p[0] = "{0}\n{1}\n{2}".format(p[1],p[3],"DIV")
    stackPos-=1

def p_Term_factor(p):
    "Term : Factor"
    global stackPos
    p[0] = p[1]

#FATORES#
def p_Factor_id(p):
    "Factor : id"
    global stackPos
    p[0] = "PUSHG {0}".format(varDic[p[1]])
    stackPos+=1

def p_Factor_int(p):
    "Factor : int"
    global stackPos
    p[0] = "PUSHI {0}".format(p[1])
    stackPos+=1

def p_Factor_float(p):
    "Factor : float"
    global stackPos
    p[0] = "PUSHF {0}".format(p[1])
    stackPos+=1

def p_Factor_Array(p):
    "Factor : id '[' Exp ']' "
    global stackPos
    p[0] = "PUSHGP\nPUSHI {0}\nPADD\n{1}\nLOADN".format(str(arrayDic[p[1]][1]),p[3])
    stackPos -=1

#-----
def p_error(p):
    print('Syntax error: ', p)
    parser.success = False
#-----

#INICIALIZACAO DO PARSER#
parser = yacc.yacc()

parser.success = True

if(sys.argv[1] == 'i'):
    for line in sys.stdin:
        print(parser.parse(line))
else:
    inp = open(sys.argv[2], 'r')
    out = open(sys.argv[2].replace("txt","vm"), 'w')

```

```
out.writelines(parser.parse(" ".join(list(inp.readlines()))))
```
