Processamento de Linguagens

Trabalho Prático Nº2
GIC/GT + Compiladores

Joana Afonso Gomes
(a84912)

Susana Marques
(a84167)

8 de março de 2022

Resumo

No âmbito da Unidade Curricular de Processamento de Linguagens, foi-nos proposta a realização deste segundo trabalho prático com vista a aprofundar conhecimentos no que toca a engenharia de linguagens e programação generativa (gramatical), reforçando a capacidade de escrever gramáticas, quer independentes de contexto (GIC), quer tradutoras (GT);

Este projeto foca-se, especificamente, no desenvolvimento de processadores de linguagens segundo o método da tradução dirigida pela sintaxe, a partir de uma gramática tradutora.

Foi-nos pedido o desenvolvimento de um compilador, gerando código para uma máquina de stack virtual.

Conteúdo

Capítulo 1

Introdução

Área: Processamento de Linguagens

1.1 Enquadramento & Contexto

Este segundo trabalho prático tem como objetivo aprofundar conhecimentos no que toca a engenharia de linguagens e programação generativa (gramatical), reforçando a capacidade de **escrever gramáticas**, quer independentes de contexto (**GIC**), quer tradutoras (**GT**);

Com tal fim em vista, este projeto abrange o desenvolvimento de processadores de linguagens segundo o método da tradução dirigida pela sintaxe, a partir de uma gramática tradutora.

É pedido o desenvolvimento de um compilador, gerando código para uma máquina de stack virtual (**VM**, ***Virtual Machine***).

Para este fim, recorreu-se a geradores de compiladores baseados em gramáticas tradutoras, concretamente o **Yacc** (versão PLY do Python) e o gerador de analisadores léxicos **Lex** (versão PLY do Python).

1.2 Problema & Objetivo

O objetivo deste projeto é definir uma linguagem de programação imperativa básica que permita:

- 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 (no caso do nosso grupo, implementamos um ciclo **for-do**).

É expectável também a implementação de uma funcionalidade extra, que o nosso grupo optou por ser a declaração e o manuseamento de 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).

O objetivo passa então pelo desenvolvimento de um compilador para essa linguage GIC criada, com recurso aos módulos previamente referidos (Yacc/Lex do PLY/Python), que deve gerar pseudo-código, Assembly da Máquina Virtual VM.

1.3 Estrutura do documento

Neste relatório apresentamos a nossa abordagem ao enunciado que nos foi atribuído. O mesmo está divido nos seguintes capítulos:

No presente capítulo 1, **Introdução**, é feito o enquadramento a contextualização do problema, sendo referido simultaneamente quais os objetivos pretendidos no projeto e os requisitos para a sua correta resolução.

Logo depois, no capítulo 2, **Concepção da Resolução**, são apresentadas as estruturas de dados usadas e os algoritmos implementados no desenvolvimento do projeto.

De seguida, no capítulo 3, **Testes**, apresentamos os testes realizados, bem como o código Assembly gerado por eles e uma demonstração do programa a correr na máquina virtual.

Terminamos o relatório com o capítulo 4, **Conclusão**, analisando e sintetizando aquilo que foi apreendido com o desenvolvimento deste projeto.

No final do documento, no apêndice A, é apresentado todo **código do programa** desenvolvido.

Capítulo 2

Concepção da Resolução

2.1 Estruturas de Dados

Neste trabalho cujo objetivo é implementar um compilador, as estruturas de dados adotadas foram dicionários de *Python*:

- parser.registers - Guarda como chave o *id* correspondente e como valor a posição da *stack da VM* que o contém.
- parser.registers_linhas - Guarda como chave o *id* da matriz e como valor a posição da linha da matriz, para auxiliar ao cálculo da posição na *Stack* Exemplo: x[10] –> parser.registers_linhas {x:10}

2.2 Lex.py

Para o vocabulário usado pelo compilador, no nosso *Lexer* usou-se palavras reservadas, *tokens* e literais. As palavras reservadas servem como palavras que serão usadas para que a linguagem imperativa desenvolvida consiga abordar tudo o que foi pedido no enunciado. Assim o grupo escolheu a palavra *PRINT* para que a linguagem tenha uma forma de imprimir *strings* e usou *WRITE* para que tenha forma de imprimir valores das variáveis.
Usou-se *START* e *STOP* para definir quando a VM deve começar, depois de todas as declarações serem feitas e quando deve parar respetivamente.
Para lermos variáveis usou-se o *READ* e finalmente usou-se determinadas palavras reservadas para informar o compilador de intruções condicionais e cíclicas:

- CONDIÇÃO CONDICIONAL-> *IF ... THEN... ELSE ... FI*
- CONDIÇÃO CÍCLICA-> *FOR ... ROF*

As palavras *IF* E *FOR* servem para que com a linguagem se possa efetuar o início da instrução correspondente e as palavras *FI* e *ROF* que se possa efetuar o final das mesmas.
Os *tokens* usados são: *num, id, frase* e a lista de palavras reservadas.
O *num* é todo o conjunto de números possíveis, com regex:

```
\textit{r'\d+'}
```

O *id* é todo o conjunto de variáveis, com regex:

```
\textit{r'[a-z][a-zA-Z_0-9]*'}
```

A *frase* é todo o conjunto de palavras separadas por um espaço, sem acentos ou pontuação que comecem e terminem por aspas (servem para representar *strings*), com regex:

```
\textit{r'\"[a-zA-Z_0-9 \\]+\"'}
```

Finalmente como literais, tem-se todos os símbolos que serão usados pela linguagem:

```
['(',')','!','>','<','+','-','*','/',';','=',']','[']
```

2.3 Yacc.py

Para implementar a linguagem, usando o *Yacc* começamos por definir as regras de produção.
Estas regras de produção podem ser desenhadas da seguinte forma:

```
Comandos-> Comandos Comando
          | Comando

Comando -> Printing
          | Reading
          | Writing
          | Start
          | Exp
          | Atrib
          | Condition
          | Cicle
          | End

Printing -> PRINT frase

Reading  -> READ id

Writing  -> WRITE id
         |  WRITE id '[' Factor ']'

Start    -> START

Exp      -> Exp '+' Termo
         |  Exp '-' Termo
         |  Termo

Termo    -> Termo '*' Factor
         |  Termo '/' Factor
         |  Termo '%' Factor
         |  Factor
```

```
Factor   -> id
         | num
         | '(' Exp ')'
         | Array
         | Matriz

Atrib    -> id '=' Exp
         |  id '[' Factor ']' '=' Factor
         |  id '[' Factor ']' '[' Factor ']' '=' Factor

Condition -> IF '(' Exp '>' Termo ')' THEN Comandos ELSE Comandos FI
           | IF '(' Exp '<' Termo ')' THEN Comandos ELSE Comandos FI
           | IF '(' Exp '>' '=' Termo ')' THEN Comandos ELSE Comandos FI
           | IF '(' Exp '<' '=' Termo ')' THEN Comandos ELSE Comandos FI
           | IF '(' Exp '=' '=' Termo ')' THEN Comandos ELSE Comandos FI
           | IF '(' Exp '!' '=' Termo ')' THEN Comandos ELSE Comandos FI

Cicle    ->  FOR '(' Atrib ';' Factor '<' Factor ';' Atrib ')' Comandos ROF
           | FOR '(' Atrib ';' Factor '>' Factor ';' Atrib ')' Comandos ROF
           | FOR '(' Atrib ';' Factor '<' '=' Factor ';' Atrib ')' Comandos ROF
           | FOR '(' Atrib ';' Factor '>' '=' Factor ';' Atrib ')' Comandos ROF
           | FOR '(' Atrib ';' Factor '=' '=' Factor ';' Atrib ')' Comandos ROF
           | FOR '(' Atrib ';' Factor '!' '=' Factor ';' Atrib ')' Comandos ROF

Array     -> id '[' Factor ']'

Matriz    -> id '[' Factor ']' '[' Factor ']

End       -> END
```

Para gerar código para a *VM*, a nossa linguagem consiste em definir o vocabulário certo para gerar esse código de acordo com as regras de produção. Assim, começando pelo início a primeira regra apenas indica que poderão ser usados vários *Comandos* ou apenas um, mas terá que se usar um comando uma vez que a VM não irá correr com nada se não existir código gerado.
Para cada comando, usou-se métodos e lógica para traduzir para o código Assembly adequado:

- **Printing**: É usado para se conseguir imprimir uma *string*/frase na VM. Usa-se a instrução *PUSHS* seguido da frase pretendida e finalmente a instrução *WRITES*.
- : Usado para ler do *standard input* usando as instruções *READ, ATOI e STOREG*.
- **Writing**: É usado para escrever o valor de uma variável (id) ou o valor de uma determinada posição do array. Depende de qual pretendemos usar, as instruções dadas podem varias com *PUSHG* para a variável e *PUSHGP, PUSHI , PADD* para o array. No final o valor é imprimido usando a instrução *WRITEI*.
- **Start**: Usada para informar a VM que o programa irá ser iniciado e que não poderá haver redeclarações.
- **Exp**: Usado para indicar operações entre inteiros: usando o *ADD e o SUB* da VM.

- **Termo**: Usado para realizar outras operações sobre inteiros (*MULT, DIV e MOD* introduzindo o conceito de *Factor*.
- **Factor**: Infere o conceito de *id* guardando no dicionário de *python* a variável e a sua posição respetiva na *Stack*. E dependendo se a variável for declarada ou não poderá usar instruções para a colocar no início da *Stack (usando como início o campo mais recente desta*. Também infere o conceito de *num*, de *Array* e de *Matriz*.
- **Atrib**: Poderá ser usado para atribuir uma expressão a uma variável, ou um factor (número, variavel, ou expressão) a um *Array, ou Matriz.*
- **Condition**: Usado para efetuar instruções condicionais com o auxílio de *SUP, INF, NOT, EQUAL, SUPEQ e INFEQ*, de instruções para "saltar"para o ciclo (*JUMP, JZ*) e de declarar o fim e o *else* que deve ser sempre diferente, no nosso caso juntou-se ao nome do círculo, o número do contador de *ifs e de elses*.
- **Condition**: Usado para efetuar instruções condicionais com o auxílio de *SUP, INF, NOT, EQUAL, SUPEQ e INFEQ*, de instruções para "saltar"para o ciclo (*JUMP, JZ*) e de declarar o fim e o *else* que deve ser sempre diferente, no nosso caso juntou-se ao nome do círculo, o número do contador de *ifs* e de *elses*.
- **Cicle**: Tal e qual como o *condition*, mas agora serve para efetuar instruções cíclicas. Assim subtitui-se o *else* na declaração por um *cicle*.
- **Array**: Usado para inferir um *Array* usando a instrução *PUSHN*.
- **Matriz** Usado para inferir uma matriz usando IGUALMENTE O *PUSHN* e para guardar o valor do número de linhas que a matriz contém para posterior uso no dicionário de *Python* já criado previamente [1].

OBSERVAÇÃO [1]: Guarda-se o número de linhas da matriz ao inicializá-la de forma a quando se realizar uma atribuição com a mesma se possa chegar à posição correta da *Stack* da VM obtida com a fórmula: **posição (k[x][y]) = posição de x * número de elementos (obtidos pelo dicionário de *Python*) + posição de y.**

```
Por exemplo:
w[5][7];
posiçao(w[4][3]) = 4 * 5 + 3
```

Capítulo 3

Testes

1) Ler 4 números e dizer se podem ser os lados de um quadrado.

Como proposto, o grupo preparou um conjunto de testes (programas-fonte escritos na nossa linguagem) de forma a testar a correta geração de código Assembly.

De seguida são apresentados os testes realizados com o seu respetivo código fonte, o código Assembly gerado e ainda uma demonstração do programa a correr na máquina virtual VM.

- **Teste (programa fonte)**

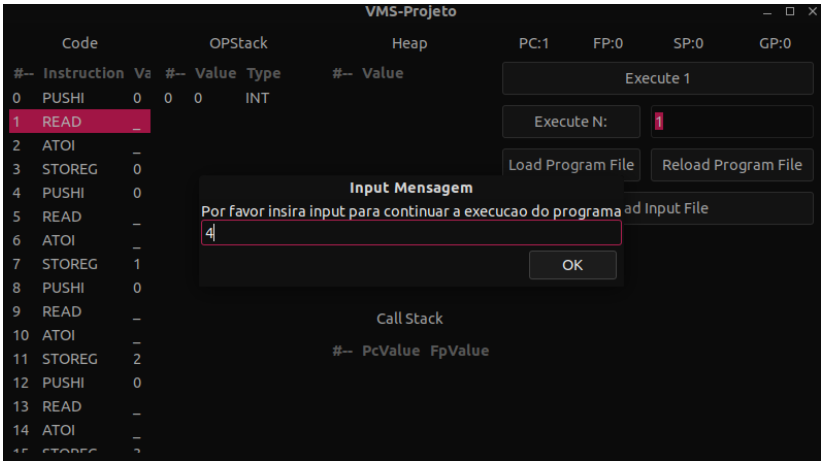
```
read x
read y
read z
read w
k=0
start
if (x==y) then if (y==z) then if (z==w) then k=1 else 0 fi else 0 fi else 0 fi
if(k==1) then print "Os numeros lidos podem ser os lados de um quadrado" else print "Os numeros
↪  lidos NAO sao lados de um quadrado" fi
end
```

- **Código Assembly gerado**

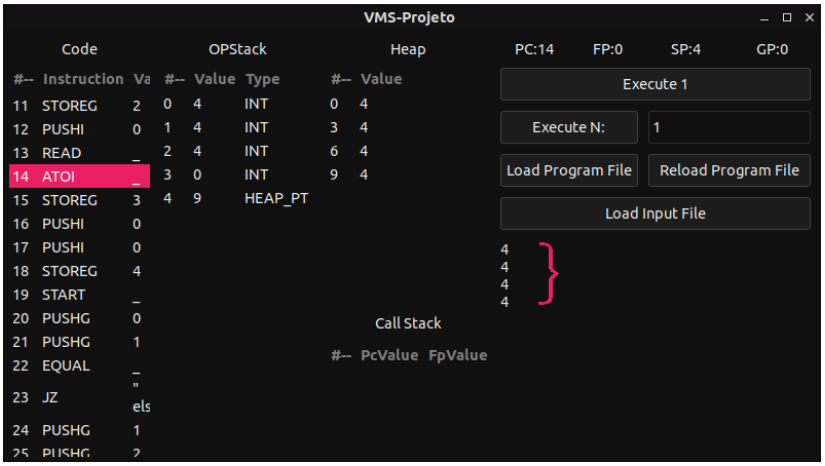
```
PUSHI 0
READ
ATOI
STOREG 0
PUSHI 0
READ
ATOI
STOREG 1
PUSHI 0
READ
ATOI
STOREG 2
PUSHI 0
READ
ATOI
STOREG 3
PUSHI 0
PUSHI 0
STOREG 4
START
PUSHG 0
PUSHG 1
EQUAL
```

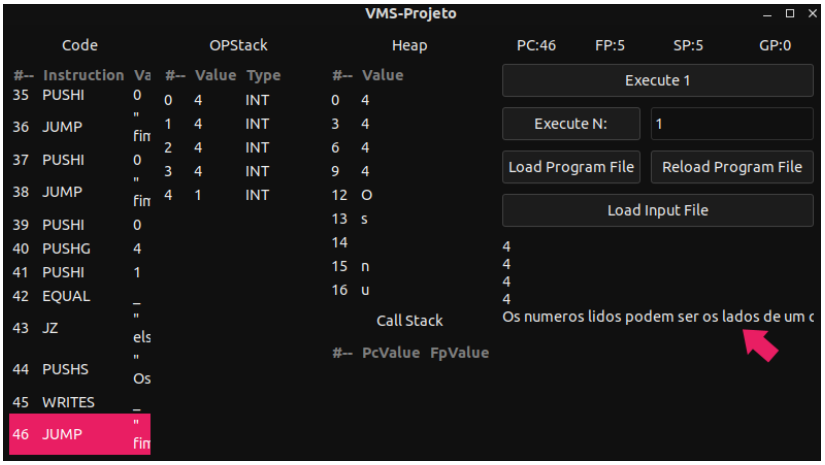
```
JZ else4
PUSHG 1
PUSHG 2
EQUAL
JZ else2
PUSHG 2
PUSHG 3
EQUAL
JZ else0
PUSHI 1
STOREG 4
JUMP fim1
else0:
PUSHI 0
fim1:
JUMP fim3
else2:
PUSHI 0
fim3:
JUMP fim5
else4:
PUSHI 0
fim5:
PUSHG 4
PUSHI 1
EQUAL
JZ else6
PUSHS "Os numeros lidos podem ser os lados de um quadrado"
WRITES
JUMP fim7
else6:
PUSHS "Os numeros lidos NAO sao lados de um quadrado"
WRITES
fim7:
STOP
```

- **Demonstração na VM**

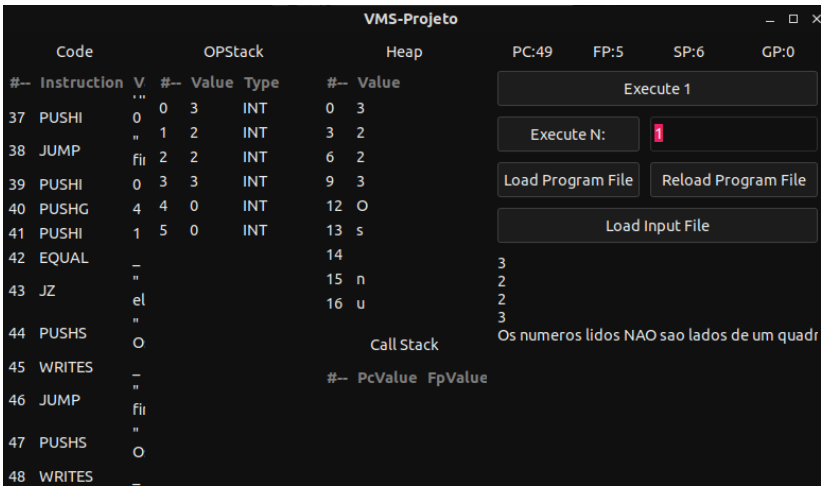


É pedido o *input* de 4 números.





No caso de serem válidos, aparece a confirmação de que podem ser os lados de um quadrado.



Caso contrário, é apresentado o aviso acima.

2) Ler um inteiro N, depois ler N números e escrever o menor deles.

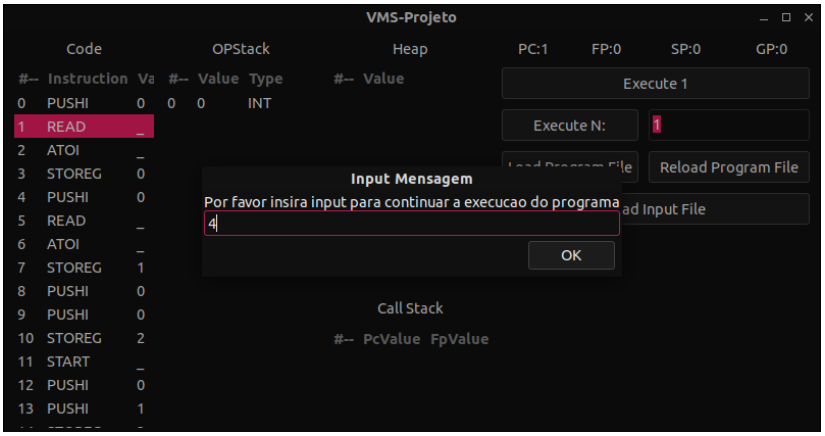
- **Teste (programa fonte)**

```
read n
read c
a=0
start
for(x=1; x<n; x=x+1) read a if (a<c) then c=a else 0 fi rof
write c
end
```

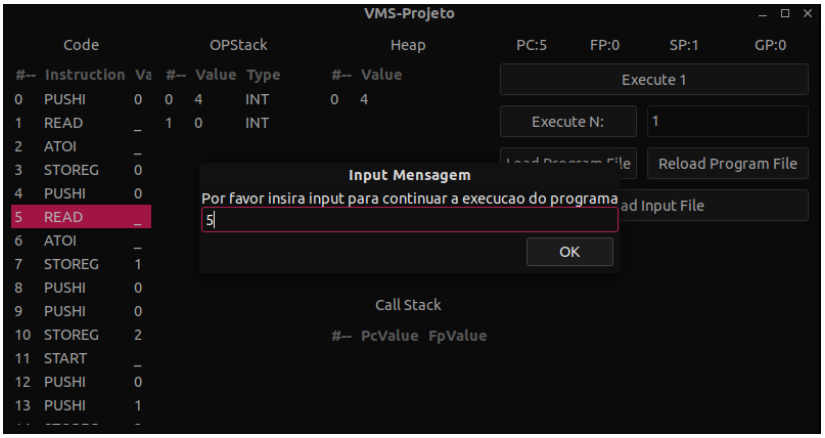
- **Código Assembly gerado**

```
PUSHI 0
READ
ATOI
STOREG 0
PUSHI 0
READ
ATOI
STOREG 1
PUSHI 0
PUSHI 0
STOREG 2
START
PUSHI 0
PUSHI 1
STOREG 3
ciclo2:
PUSHG 3
PUSHG 0
INF
JZ fim3
READ
ATOI
STOREG 2
PUSHG 2
PUSHG 1
INF
JZ else0
PUSHG 2
STOREG 1
JUMP fim1
else0:
PUSHI 0
fim1:
PUSHG 3
PUSHI 1
ADD
STOREG 3
JUMP ciclo2
fim3:
PUSHG 1
WRITEI
STOP
```

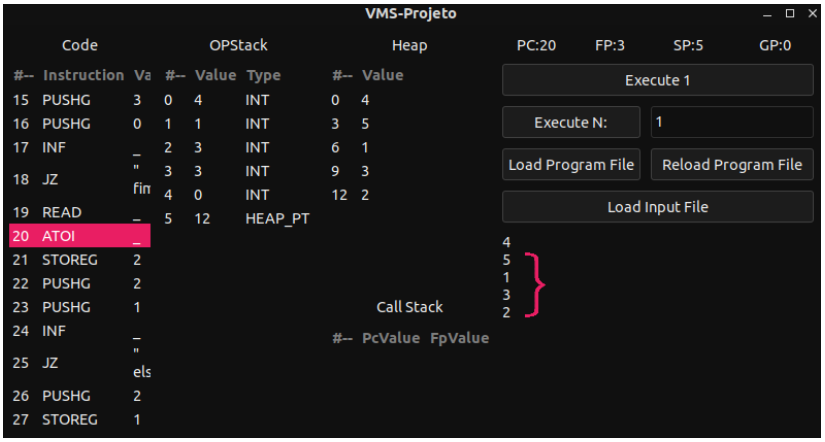
- **Demonstração na VM**

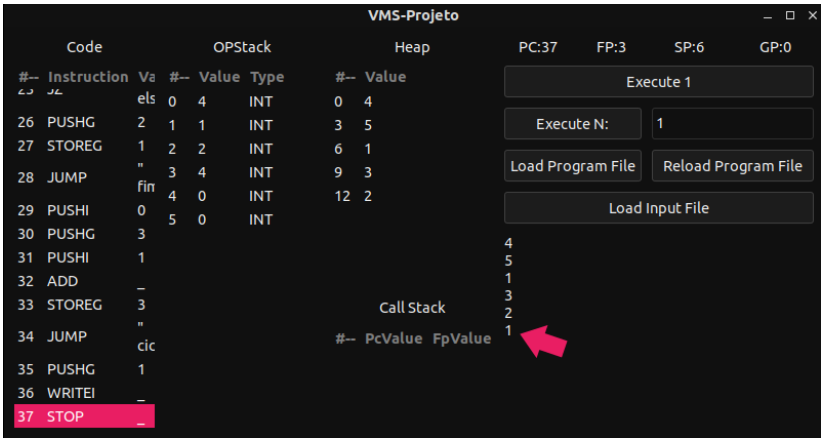


É pedido o *input* de N (quantidade de números que serão pedidos).



É pedido o *input* dos N números.





É apresentado o menor desses N números.

---

3) Ler N (constante do programa) números e calcular e imprimir o seu produtório.

- **Teste (programa fonte)**

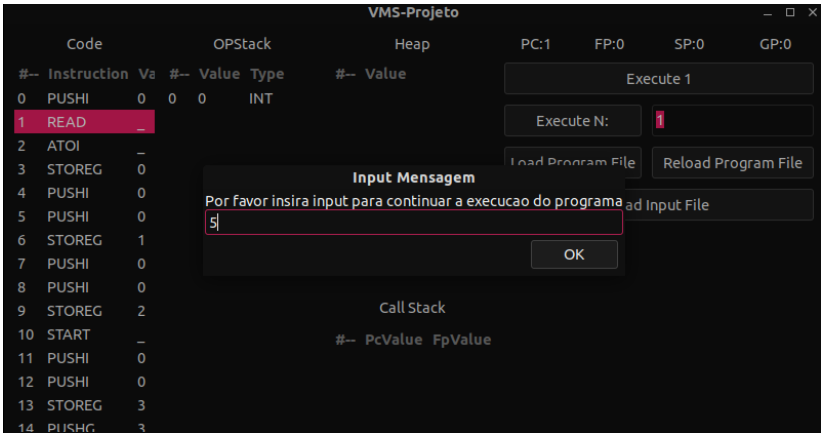
```
read n
a=0
p=0
start
for(x=0; x<n; x=x+1) read a if (x==0) then p=a else p=p*a fi rof
write p
end
```

- **Código Assembly gerado**

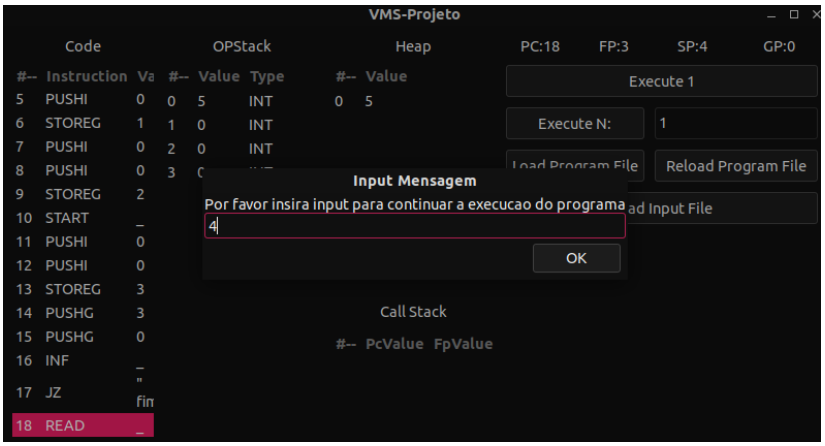
```
PUSHI 0
READ
ATOI
STOREG 0
PUSHI 0
PUSHI 0
STOREG 1
PUSHI 0
PUSHI 0
STOREG 2
START
PUSHI 0
PUSHI 0
STOREG 3
ciclo2:
PUSHG 3
PUSHG 0
INF
JZ fim3
READ
ATOI
STOREG 1
PUSHG 3
PUSHI 0
EQUAL
JZ else0
PUSHG 1
STOREG 2
JUMP fim1
else0:
PUSHG 2
PUSHG 1
MUL
```

```
STOREG 2
fim1:
PUSHG 3
PUSHI 1
ADD
STOREG 3
JUMP ciclo2
fim3:
PUSHG 2
WRITEI
STOP
```

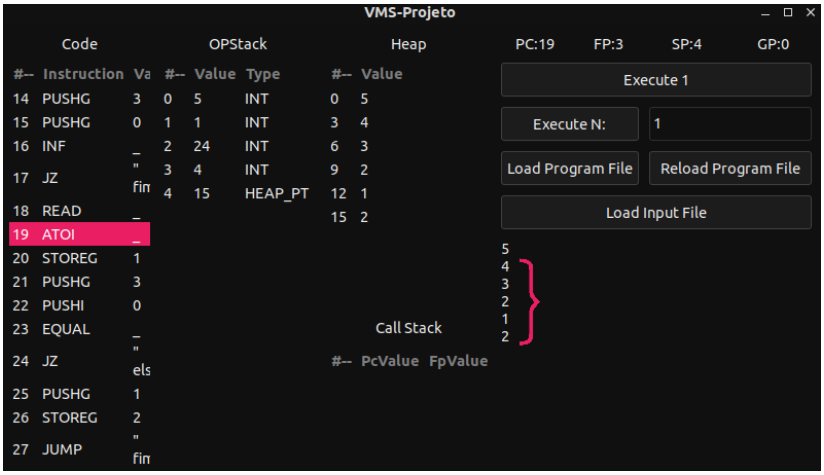
- **Demonstração na VM**

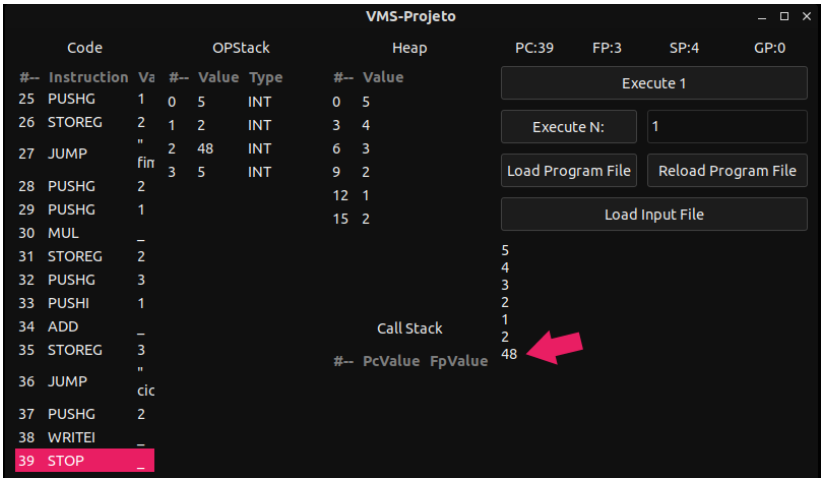


É pedido o *input* de N (quantidades de números para calcular o produtório).



É pedido o *input* desses N números.





É apresentado o produtório desses N números.

---

4) Contar e imprimir os números impares de uma sequência de números naturais.

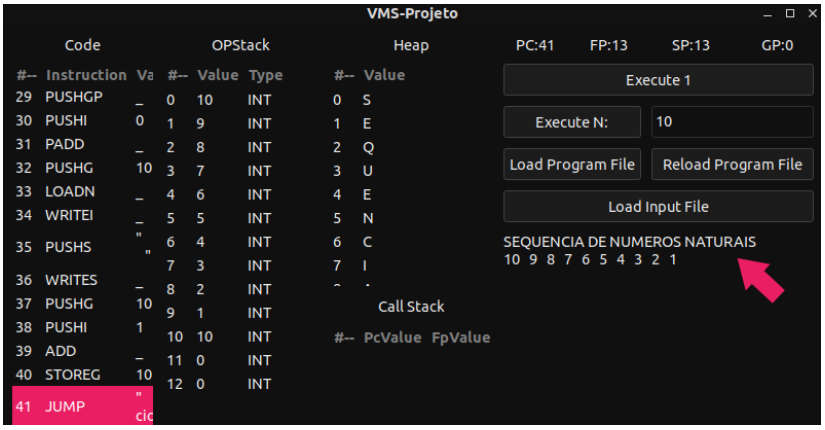
- **Teste (programa fonte)**

```
x[10]
i=0
k=0
s=10
start
print "SEQUENCIA DE NUMEROS NATURAIS\n"
for(i=0; i<10; i=i+1) x[i]=s s=s-1 write x[i] print "   " rof
print "\n\nIMPARES \n"
for(i=0; i<10; i=i+1) if ((x[i]%2)==0) then 0 else k=k+1 write x[i] print "   " fi rof
print "\n\nCOUNT IMPARES\n"
write k
end
```

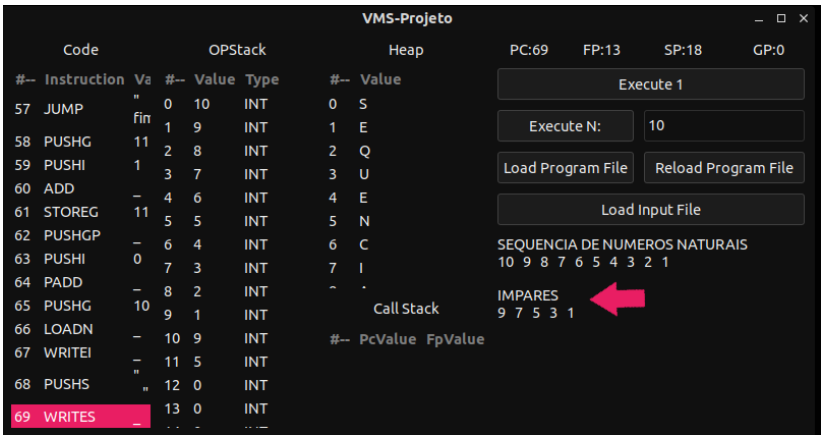
- **Código Assembly gerado**

```
PUSHN 10
PUSHI 0
PUSHI 0
STOREG 10
PUSHI 0
PUSHI 0
STOREG 11
PUSHI 0
PUSHI 10
STOREG 12
START
PUSHS "SEQUENCIA DE NUMEROS NATURAIS\n"
WRITES
PUSHI 0
STOREG 10
ciclo0:
PUSHG 10
PUSHI 10
INF
JZ fim1
PUSHGP
PUSHI 0
PADD
PUSHG 10
PUSHG 12
STOREN
PUSHG 12
PUSHI 1
SUB
STOREG 12
PUSHGP
PUSHI 0
PADD
PUSHG 10
LOADN
WRITEI
PUSHS "   "
WRITES
PUSHG 10
PUSHI 1
ADD
STOREG 10
JUMP ciclo0
fim1:
PUSHS "\n\nIMPARES \n"
WRITES
PUSHI 0
STOREG 10
ciclo4:
PUSHG 10
PUSHI 10
INF
JZ fim5
PUSHG 10
PUSHI 2
MOD
PUSHI 0
EQUAL
JZ else2
PUSHI 0
JUMP fim3
else2:
PUSHG 11
PUSHI 1
ADD
STOREG 11
PUSHGP
PUSHI 0
PADD
PUSHG 10
LOADN
WRITEI
PUSHS "   "
WRITES
fim3:
PUSHG 10
PUSHI 1
ADD
STOREG 10
JUMP ciclo4
fim5:
PUSHS "\n\nCOUNT IMPARES\n"
WRITES
PUSHG 11
WRITEI
STOP
```

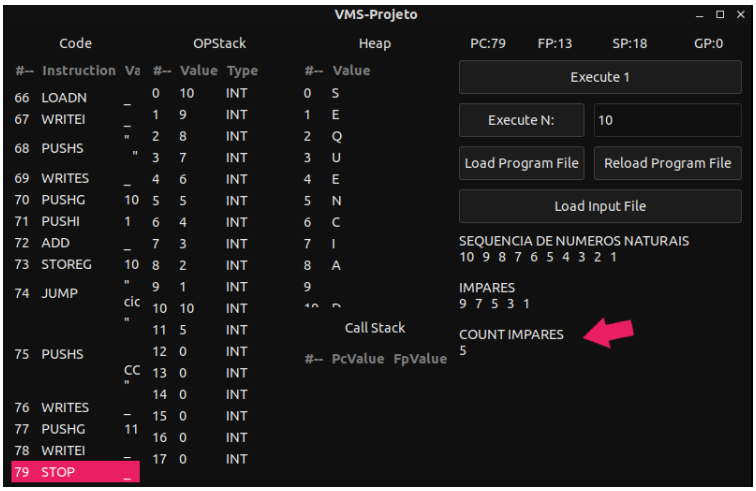
- **Demonstração na VM**



Há uma sequência de números naturais.



É averiguado quais destes são números ímpares.



É apresentado quantos números na sequência inicial são ímpares.

5) Ler e armazenar N números num array; imprimir os valores por ordem inversa.

- **Teste (programa fonte)**

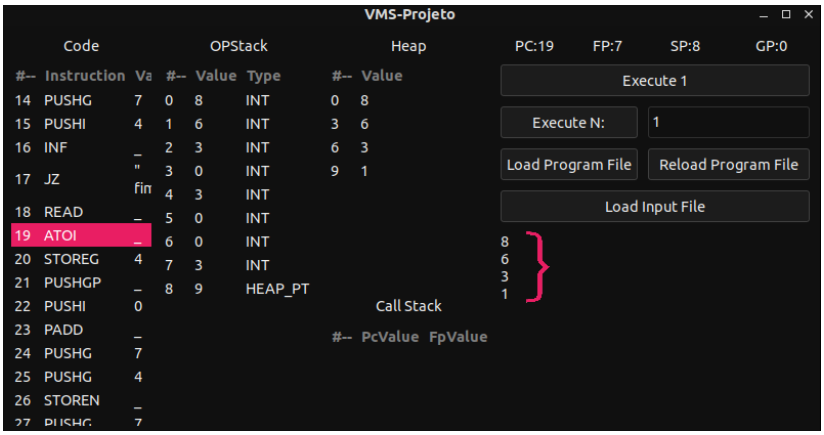
```
x[4]
a=0
l=0
p=0
start
for(i=0; i<4; i=i+1) read a x[i]=a rof
l=4-1
print "ORDEM INVERSA \n"
for(j=l; j>=0; j=j-1) write x[j] print "  " rof
end
```

- **Código Assembly gerado**

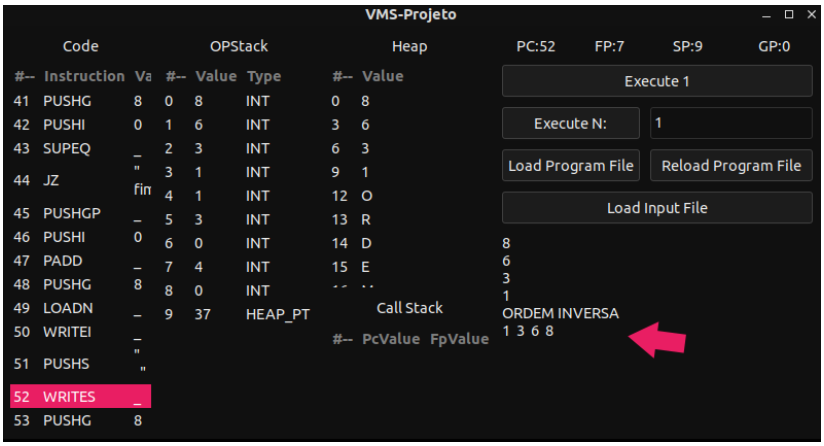
```
PUSHN 4
PUSHI 0
PUSHI 0
STOREG 4
PUSHI 0
PUSHI 0
STOREG 5
PUSHI 0
PUSHI 0
STOREG 6
START
PUSHI 0
PUSHI 0
STOREG 7
ciclo0:
PUSHG 7
PUSHI 4
INF
JZ fim1
READ
ATOI
STOREG 4
PUSHGP
PUSHI 0
PADD
PUSHG 7
PUSHG 4
STOREN
PUSHG 7
PUSHI 1
ADD
STOREG 7
JUMP ciclo0
fim1:
PUSHI 4
PUSHI 1
SUB
STOREG 5
PUSHS "ORDEM INVERSA \n"
WRITES
PUSHI 0
PUSHG 5
STOREG 8
ciclo2:
PUSHG 8
PUSHI 0
SUPEQ
JZ fim3
PUSHGP
PUSHI 0
PADD
PUSHG 8
LOADN
WRITEI
PUSHS "  "
WRITES
PUSHG 8
PUSHI 1
```

```
SUB
STOREG 8
JUMP ciclo2
fim3:
STOP
```

- **Demonstração na VM**



São lidos 4 números (tamanho do array) que são armazenados no array.



Apresenta-se a ordem inversa dos números do array.

Capítulo 4

Conclusão

Este trabalho contribuiu amplamente para a aprendizagem do grupo no que toca a engenharia de linguagens e programação gramatical, consolidando o conhecimento prático e teórico adquirido relativo à escrita gramáticas.

Simultaneamente, concedeu-nos uma familiarização com os módulos *Yacc* e *Lex*, e consideramos desafiante e interessante a abordagem de um compilador que gera código para uma máquina virtual.

Tendo em conta o enunciado atribuído, consideramos que foram alcançados os objetivos propostos, julgando, portanto, na sua globalidade, um desfecho positivo na resolução do problema em questão neste segundo trabalho prático.

Apêndice A

Código do Programa

- **Gerador_lex.py**

```
import ply.lex as lex

reserved = {
    'print' : 'PRINT',
    'if' : 'IF',
    'then' : 'THEN',
    'else' : 'ELSE',
    'for' : 'FOR',
    'fi' : 'FI',
    'rof' : 'ROF',
    'write' : 'WRITE',
    'read'  :'READ',
    'start' :'START',
    'end' : 'END',
 }
tokens = ['num','id','frase'] + list(reserved.values())
literals = ['(',')','!','>','<','+','-','*','/',';','=',']','[','%']

t_num = r'\d+'
def t_id(t):
     r'[a-z][a-zA-Z_0-9]*'
     t.type = reserved.get(t.value,'id')    # Check for reserved words
     return t
def t_frase(t):
     r'\"[a-zA-Z_0-9 \\]+\"'
     t.type = reserved.get(t.value,'frase')    # Check for reserved words
     return t

t_ignore = "\t\n"
def t_error(t):
    #print("Carater ilegal: ", t.value[0])
    t.lexer.skip(1)

#build the lexer
lexer = lex.lex()
```

- **Gerador_yacc.py**

```
import ply.yacc as yacc
import fileinput
import sys
from gerador_lex import tokens


#Production rules
def p_Comandos(p):
    "Comandos : Comandos Comando"
    p[0] = p[1] + p[2]
def p_Comando_unico(p):
    "Comandos : Comando"
    p[0] = p[1]

def p_Comando(p):
    """
    Comando :  Printing
            |  Reading
            |  Writing
            |  Start
            |  Exp
            |  Atrib
            |  Condition
            |  Cicle
            |  End
    """
    p[0] = p[1]


def p_Printing(p):
    "Printing : PRINT frase"
    p[0] = 'PUSHS ' +p[2] +'\n' + 'WRITES\n'


def p_Reading(p):
    "Reading : READ id"
    global i
    flag=1
    for key in p.parser.registers.keys():
        if key == p[2]:
            p[0] = 'READ\nATOI\n'+ 'STOREG '+ str(p.parser.registers.get(key)) +'\n'
            flag=0
    if(flag==1):
        p.parser.registers.update({p[2]: i})
        p[0] = 'PUSHI 0\nREAD\nATOI\n'+ 'STOREG '+ str(i) +'\n'
        i= i+1

def p_Writing(p):
    "Writing : WRITE id"
    indice= p.parser.registers.get(p[2])
    p[0]= 'PUSHG ' + str(indice) +'\n'+ 'WRITEI\n'
```

```
def p_Writing_Array(p):
    "Writing : WRITE id '[' Factor ']'"
    indice = p.parser.registers.get(p[2])
    p[0] = 'PUSHGP\n' + 'PUSHI ' + str(indice) +'\n' + 'PADD\n' +  p[4] + 'LOADN\n' + 'WRITEI\n'



def p_Start(p):
    "Start : START"
    global f
    p[0]='START\n'
    f=1

def p_Exp_add(p):
    "Exp : Exp '+' Termo"
    p[0] = p[1]+ p[3]+ 'ADD\n'
    # file_vm.write(p[0])

def p_Exp_sub(p):
    "Exp : Exp '-' Termo"
    p[0] = p[1]+ p[3]+ 'SUB\n'
    # file_vm.write(p[0])

def p_Exp_termo(p):
    "Exp : Termo"
    p[0] = p[1]
    # file_vm.write(p[0])

def p_Termo_mul(p):
    "Termo : Termo '*' Factor"
    p[0] = p[1]+ p[3] + 'MUL\n'
    # file_vm.write(p[0])

def p_Termo_div(p):
    "Termo : Termo '/' Factor"
    if(p[3] != 0):
        p[0] = p[1]+ p[3]+ 'DIV\n'
        # file_vm.write(p[0])
    else:
        print ("Erro: divisao por 0, a continuar com 0...)")
        p[0] = str(0)

def p_Termo_mod(p):
    "Termo : Termo '%' Factor"
    p[0] = p[1]+ p[3] + 'MOD\n'
    # file_vm.write(p[0])


def p_Termo_factor(p):
    "Termo : Factor"
    p[0] = p[1]
    # file_vm.write(p[0])

def p_Factor_id(p):
```

```
    "Factor : id"
    global i
    indice = p.parser.registers.get(p[1])
    if(str(indice) == 'None'):
        if(f==1):
            sys.exit("Erro: Impossivel declarar variavel")
        else:
            p.parser.registers.update({p[1]: i})
            p[0]= 'PUSHI 0\n' + 'PUSHG ' + str(i) + '\n'
            i=i+1
    else:
        p[0] = 'PUSHG ' + str(indice) + '\n'

def p_Factor_num(p):
    "Factor : num"
    p[0] = 'PUSHI ' + p[1] + '\n'
    # file_vm.write(p[0])


def p_Factor_group(p):
    "Factor : '(' Exp ')'"
    p[0] = p[2]
    # file_vm.write(p[0])

def p_Factor_Array(p):
    "Factor : Array"
    p[0] = p[1]

def p_Factor_Matriz(p):
    "Factor : Matriz"
    p[0] = p[1]

def p_End(p):
    "End : END"
    p[0]='STOP'
    file_vm.write(p[0])
    file_vm.close()
    sys.exit("Programa compilado com sucesso!")


def p_Atrib(p):
    "Atrib : id '=' Exp"
    global i
    flag=1
    for key in p.parser.registers.keys():
            if key == p[1]:
                p[0] = p[3] + 'STOREG '+ str(p.parser.registers.get(key)) +'\n'
                flag=0
                # file_vm.write(p[0])
    if(flag==1):
            p.parser.registers.update({p[1]: i})
            p[0] = 'PUSHI 0\n' +p[3]+ 'STOREG '+ str(i) +'\n'
            i= i+1
            # file_vm.write(p[0])
```

```
def p_Condition_maior(p):
    "Condition : IF '(' Exp '>' Termo ')' THEN Comandos ELSE Comandos FI"
    global count
    p[0] = p[3] + p[5] +'SUP\n' +'JZ else' + str(count)+'\n' + (p[8])+ 'JUMP ' +
    ↪  'fim'+str((count+1))+'\n' + 'else'+str(count)+':\n' + (p[10])+
    ↪  'fim'+str((count+1))+':\n'
    count=count+2


def p_Condition_menor(p):
    "Condition : IF '(' Exp '<' Termo ')' THEN Comandos ELSE Comandos FI"
    global count
    p[0] = p[3] + p[5] +'INF\n' +'JZ else' + str(count)+'\n' + (p[8])+ 'JUMP ' +
    ↪  'fim'+str((count+1))+'\n' + 'else'+str(count)+':\n' + (p[10])+
    ↪  'fim'+str((count+1))+':\n'
    count=count+2

def p_Condition_maiorigual(p):
    "Condition : IF '(' Exp '>' '=' Termo ')' THEN Comandos ELSE Comandos FI"
    global count
    p[0] = p[3] + p[6]+ 'SUPEQ\n' +'JZ else' + str(count)+'\n' + (p[9])+ 'JUMP ' +
    ↪  'fim'+str((count+1))+'\n' + 'else'+str(count)+':\n' + (p[11])+
    ↪  'fim'+str((count+1))+':\n'
    count=count+2

def p_Condition_menorigual(p):
    "Condition : IF '(' Exp '<' '=' Termo ')' THEN Comandos ELSE Comandos FI"
    global count
    p[0] = p[3] + p[6] +'INFEQ\n' +'JZ else' + str(count)+'\n' + (p[9])+ 'JUMP ' +
    ↪  'fim'+str((count+1))+'\n' + 'else'+str(count)+':\n' + (p[11])+
    ↪  'fim'+str((count+1))+':\n'
    count=count+2

def p_Condition_igual(p):
    "Condition : IF '(' Exp '=' '=' Termo ')' THEN Comandos ELSE Comandos FI"
    global count
    p[0] = p[3] + p[6] +'EQUAL\n' +'JZ else' + str(count)+'\n' + (p[9])+ 'JUMP ' +
    ↪  'fim'+str((count+1))+'\n' + 'else'+str(count)+':\n' + (p[11])+
    ↪  'fim'+str((count+1))+':\n'
    count=count+2

def p_Condition_not_igual(p):
    "Condition : IF '(' Exp '!' '=' Termo ')' THEN Comandos ELSE Comandos FI"
    global count
    p[0] = p[3] + p[6] +'EQUAL\n'+'NOT\n' +'JZ else' + str(count)+'\n' + (p[9])+ 'JUMP ' +
    ↪  'fim'+str((count+1))+'\n' + 'else'+str(count)+':\n' + (p[11])+
    ↪  'fim'+str((count+1))+':\n'
    count=count+2

def p_Cicle_menor(p):
    "Cicle : FOR '(' Atrib ';' Factor '<' Factor ';' Atrib ')' Comandos ROF"
    global count
```

```
    p[0] = p[3] + 'ciclo'+str(count)+':\n' + p[5] + p[7] + 'INF\n'  +'JZ fim'+str((count+1))
    ↪  +'\n' + p[11] + p[9]  + 'JUMP ciclo'+str(count)+'\n' + 'fim'+str((count+1))+':\n'
    count=count+2

def p_Cicle_maior(p):
    "Cicle : FOR '(' Atrib ';' Factor '>' Factor ';' Atrib ')' Comandos ROF"
    global count
    p[0] = p[3] + 'ciclo'+str(count)+':\n' + p[5] + p[7] + 'SUP\n'  +'JZ fim'+str((count+1))
    ↪  +'\n' + p[11] + p[9]  + 'JUMP ciclo'+str(count)+'\n' + 'fim'+str((count+1))+':\n'
    count=count+2

def p_Cicle_menor_igual(p):
    "Cicle : FOR '(' Atrib ';' Factor '<' '=' Factor ';' Atrib ')' Comandos ROF"
    global count
    p[0] = p[3] + 'ciclo'+str(count)+':\n' + p[5] + p[8] + 'INFEQ\n'  +'JZ fim'+str((count+1))
    ↪  +'\n' + p[12] + p[10]  + 'JUMP ciclo'+str(count)+'\n' + 'fim'+str((count+1))+':\n'
    count=count+2

def p_Cicle_maior_igual(p):
    "Cicle : FOR '(' Atrib ';' Factor '>' '=' Factor ';' Atrib ')' Comandos ROF"
    global count
    p[0] = p[3] + 'ciclo'+str(count)+':\n' + p[5] + p[8] + 'SUPEQ\n'  +'JZ fim'+str((count+1))
    ↪  +'\n' + p[12] + p[10]  + 'JUMP ciclo'+str(count)+'\n' + 'fim'+str((count+1))+':\n'
    count=count+2

def p_Cicle_igual(p):
    "Cicle : FOR '(' Atrib ';' Factor '=' '=' Factor ';' Atrib ')' Comandos ROF"
    global count
    p[0] = p[3] + 'ciclo'+str(count)+':\n' + p[5] + p[8] + 'EQUAL\n'  +'JZ fim'+str((count+1))
    ↪  +'\n' + p[12] + p[10]  + 'JUMP ciclo'+str(count)+'\n' + 'fim'+str((count+1))+':\n'
    count=count+2

def p_Cicle_not_igual(p):
    "Cicle : FOR '(' Atrib ';' Factor '!' '=' Factor ';' Atrib ')' Comandos ROF"
    global count
    p[0] = p[3] + 'ciclo'+str(count)+':\n' + p[5] + p[8] + 'EQUAL\n'+'NOT\n'  +'JZ
    ↪  fim'+str((count+1)) +'\n' + p[12] + p[10]  + 'JUMP ciclo'+str(count)+'\n' +
    ↪  'fim'+str((count+1))+':\n'
    count=count+2

def p_Array(p):
    "Array : id '[' Factor ']'"
    indice = p.parser.registers.get(p[1])
    global i
    if(str(indice)=='None' ):
        p.parser.registers.update({p[1]: i})
        string = p[3].split()
        print(string)
        p[0] = 'PUSHN ' +string[1]+ '\n'
        i= i+ int(string[1])
    else:
        p[0] = p[3]

def p_Atrib_Array(p):
```

```
    "Atrib : id '[' Factor ']' '=' Factor"
    indice = p.parser.registers.get(p[1])
    if(str(indice)=='None' ):
        string = "ERRO: Variavel " +p[1] +" por declarar!"
        sys.exit(string)
    else:
        p[0] = 'PUSHGP\n' + 'PUSHI '+str(indice) +'\n' +'PADD\n' + str(p[3])+ str(p[6])
        ↪  +'STOREN\n'

def p_Matriz(p):
    "Matriz : id '[' Factor ']' '[' Factor ']'"
    global i
    indice = p.parser.registers.get(p[1])
    # apenas para saber quantas celulas da matriz existem ao todo
    if(str(indice)=='None'):
        s1 = p[3].split()
        s2 = p[6].split()
        m = int(int(s1[1])) *  int(int(s2[1]))
        p.parser.registers.update({p[1]: i})
        p.parser.registers_linhas.update({p[1]: int(s1[1])})
        p[0] = 'PUSHN ' +str(m)+ '\n'
        i= i+ m
    else:
        p[0] = p[3] + p[6]

def p_Atrib_Matriz(p):
    "Atrib : id '[' Factor ']' '[' Factor ']' '=' Factor"
    # apenas para saber qual o indice maior que se pretende
    # para depois se comparar com o numero de celulas
    # subtrai-se um porque e de 0 a n-1
    s1 = p[3].split()
    s2 = p[6].split()
    m = (int(s1[1])+1 *  int(s2[1])+1) -1
    indice = p.parser.registers.get(p[1])
    if(str(indice) == 'None'):
        string = "ERRO: Variavel " +p[1] +" por declarar!"
        sys.exit(string)
    else:
        elementos_linha= p.parser.registers_linhas.get(p[1])
        p[0] = 'PUSHGP\n' + 'PUSHI '+str(indice) +'\n' 'PADD\n' +str(p[3])+ '\n'+'PUSHI '+
        ↪  str(elementos_linha)+'\n'+'MUL\n'+str(p[6]) +'ADD\n' + str(p[9]) +'STOREN\n'


#Error value for syntax errors
def p_error(token):
    if token is not None:
        print ("Illegal token: %s" % (token.value))
    else:
        print('Unexpected end of input')
#Build the parser
parser =  yacc.yacc()
# my state
# dicionario inicializado a vazio
parser.registers = {}
```

```
parser.registers_linhas ={}
#GENERATE file.vm
global i
global count
i=0
f=0
count=0
file = 'file.vm'
file_vm = open(file, "w+")
# reading input
if(len(sys.argv)>1):
    for linha in fileinput.input(files= sys.argv[1]):
        result = parser.parse(linha)
        file_vm.write(str(result))
else:
    for linha in fileinput.input():
        result = parser.parse(linha)
        file_vm.write(str(result))
```