

Processamento de Linguagens e Compiladores (3º ano de Curso)

**Trabalho Prático 2**

Relatório de Desenvolvimento

Breno Fernando Guerra Marrão  
A97768

Tales André Rovaris Machado  
A96314

Tiago Passos Rodrigues  
A96414

3 de janeiro de 2023

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>2</b>
<b>2</b>	<b>Enunciado</b>	<b>3</b>
<b>3</b>	<b>Escolhas Feitas</b>	<b>4</b>
3.1	Desenho da Linguagem . . . . .	4
3.1.1	Declarar variáveis . . . . .	4
3.1.2	Operações . . . . .	4
3.1.3	Ler do standard input e escrever no standard output . . . . .	4
3.1.4	Instruções Condicionais . . . . .	5
3.1.5	Instruções Cíclicas . . . . .	5
3.2	Desenho da gramática . . . . .	5
<b>4</b>	<b>Regras de tradução para Assembly</b>	<b>7</b>
<b>5</b>	<b>Exemplos de Utilização</b>	<b>8</b>
5.1	Calcular fatorial . . . . .	8
5.2	Calcular multiplicação . . . . .	9
5.3	Calcular funções sem return . . . . .	11
5.4	Calcular Fibonacci . . . . .	13
5.5	Calcular Exponenciação . . . . .	15
5.6	Calcular Área Retângulo . . . . .	17
<b>6</b>	<b>Conclusão</b>	<b>19</b>
<b>A</b>	<b>Código do Programa</b>	<b>20</b>
A.1	Analisador Léxico . . . . .	20
A.2	Analisador Sintático . . . . .	22

# Capítulo 1

## Introdução

Este relatório no âmbito de processamento de linguagens e compiladores do segundo projeto teve como o objetivo a criação de uma simples linguagem de programação que seja capaz de criar variáveis, atribuir valores a essas variáveis, ler e escrever no terminal, escrever ciclos e efetuar instruções de seleção. Na execução deste projeto utilizamos os módulos `ply.lex` para a análise léxica e `yacc` para a gramática na linguagem python.

## 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 de seleção para controlo do fluxo de execução.
- efetuar instruções de repetição (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.

Como é da praxe neste tipo de linguagens, as variáveis deverão ser declaradas no início do programa e não pode haver re-declarações, nem utilizações sem declaração prévia. Se nada for explicitado, o valor da variável após a declaração é 0 (zero).

Desenvolva, então, um compilador para essa linguagem com base na GIC criada acima e com recurso aos módulos Yacc/ Lex do PLY/Python.

O compilador deve gerar pseudo-código, Assembly da Máquina Virtual VM.

Muito Importante:

Para a entrega do TP deve preparar um conjunto de testes (programas-fonte escritos na sua linguagem) e mostrar o código Assembly gerado bem como o programa a correr na máquina virtual VM.

## Capítulo 3

# Escolhas Feitas

### 3.1 Desenho da Linguagem

Na nossa linguagem optamos por usar notações case sensitive, tais como os token if, while, etc.. Onde é posicionado o habitual ponto e vírgula das outras linguagens de programação nós escolhemos pôr um ponto final para delimitar as linhas de código.

#### 3.1.1 Declarar variáveis

Decidimos que a declaração de variáveis seria feita no início do programa e é feita com a seguinte notação, segue exemplos :

```
int x . # declaração da variável sem argumentos
int x = 2 . #declaração da variável com atribuição de valor
int y = x + 2 . # declaração da variável com outra variável e com uma operação de soma
int y = x + func() . # declaração da variável com uma variável e com uma operação de soma com uma função
↪ arbitrária
```

#### 3.1.2 Operações

Decidimos manter as nossas operações como é costume na maior partes das linguagens, com os seus símbolos, segue exemplos :

```
x = x + 1 .# operação de soma entre variável e inteiro
x = x + y .# operação de soma entre variáveis
x = x - y .# operação de subtração entre variáveis
x = y * z .# operação de multiplicação entre variáveis
x = x / y .# operação de divisão entre variáveis
x = (x - 1 ) * 2 .# operação com prioridade
x = x + 2 + 3 + y * z - 10 / 5 .# Varias operações juntas.
```

#### 3.1.3 Ler do standard input e escrever no standard output

Para escrever no standard output decidimos que deveria começar por "print" e deveria ser seguido por frases entre aspas e se quisesse colocar números fechava as aspas e escrevia os números, variáveis ou expressões, segue exemplos :

```
int x = 20 .
int y = 10 .
```

```

print x . # exemplo de escrever com variável .
print "Trabalho de PLC" # exemplo de escrever com uma frase
print "Este é nosso segundo projeto de PLC e merecemos " x " como nota " . # exemplo de escrever com frases
↪ e uma variável
print "A soma de variável x e y é " x + y . # exemplo de escrever com frases e uma operação

```

Para ler no standard input dizemos para qual variável que queremos atribuir o valor (que só pode ser inteiro) lido e alguma frase (opcional).

```

int x .
x = input " Qual valor para x ?" .

```

### 3.1.4 Instruções Condicionais

Nas instruções condicionais a estrutura é tal que devemos primeiro escrever "if" uma condição lógica, "then" as instruções que queremos que seja realizada e o mesmo para "else", segue exemplos:

```

int x = 2 .
if x == 2 then x = x + 1 . else x = x - 1 . end if .

```

### 3.1.5 Instruções Cíclicas

A instrução cíclica que fizemos foi 'do while' em que a estrutura é 'while' seguido da condição do ciclo e depois um 'do'. Após isso vem as instruções que queremos executar, e terminamos com um 'while'.

```

int x = 2 .
while x > 0 do x = x - 1 . end while .

```

## 3.2 Desenho da gramática

Literais : ['+', '-', '\*', '/', '=', '(', ')', ',', '!', '%']

Tokens : 'IF', 'TRUE', 'FALSE', 'EQUAL', 'INF', 'INFEQ', 'SUP', 'SUPEQ', 'INPUT', 'PRINT', 'INT', 'OU', 'E', 'THEN', 'ELSE', 'WHILE', 'begin', 'return', 'end', 'DO', 'NUM', 'ID', 'FRASE', 'MOD'

A gramática implementada é constituída pelas seguintes regras de derivação.

```

Programa : Vars Funcs Cod
Vars : Var Vars | €
Funcs : Funcs Func | €
Var : INT ID '.' | INT ID '=' expr '.'
Func : ID begin Cod end return expr '.' | ID begin Cod end '.'
Cod : Linha Cod | €
Linha : Escrever | atr | ID '(' ')' '.' | Ler | cond | SE | Ciclo
SE : IF cond THEN Cod ELSE Cod end IF "."
Ler : ID '=' INPUT FRASE '.'
Escrever : PRINT corpoescreve '.'
corpoescreve : alter corpoescreve | €
alter : FRASE | expr
Ciclo : WHILE cond DO Cod end WHILE '.'
atr : ID '=' expr '.'
cond : bool | expr | expr oprelacao expr | cond E cond | cond OU cond

```

```
oprelacao : INF | EQUAL | DIFF | INFEQ | SUP | SUPEQ
bool : TRUE | FALSE
expr : termo | expr '+' termo | expr '-' termo
termo : fator | termo '*' fator | termo '/' fator | termo '%' fator
fator : NUM | ID | '(' expr ')' | ID '(' ' ' )'
```

## Capítulo 4

# Regras de tradução para Assembly

As regras de tradução para Assembly estão apresentadas no Apêndice B.



## Capítulo 5

# Exemplos de Utilização

Aqui estão alguns exemplos de programas conhecidos representados na nossa linguagem e o respectivo código assembly.

### 5.1 Calcular fatorial

Simples exemplo de um programa que calcula fatorial

```
int x = 10 .
int aux = x - 1 .
while aux > 0 do
x = x * aux .
aux = aux - 1 .
end while .
print x .
```

O código assembly gerado foi o seguinte:

```
PUSHI 10
PUSHG 0
PUSHI 1
SUB
comeco0:
PUSHG 1
PUSHI 0
sup
JZ terminar0
PUSHG 0
PUSHG 1
MUL
storeg 0
PUSHG 1
PUSHI 1
SUB
storeg 1
JUMP comeco0
terminar0:
```

```

PUSHG 0
WRITEI
STOP

```

The screenshot shows the Virtual Machine EWVM interface. On the left, there is a text area for writing code, with a 'Clear' button. Below it is a 'Browse...' button and a 'Run' button. The main area displays the assembly code being executed, with line numbers 1 through 39. The code includes instructions like START, PUSHI, PUSHG, SUB, começo0, JZ, terminar0, and WRITEI. On the right, there is a memory stack visualization showing the Call Stack, Operand Stack (with values 0 and 3628800), and Struct Heap. Below the stack, there are pointers: Global Pointer: 0, Frame Pointer: 0, and Stack Pointer: 2. At the bottom, there is an 'Output:' section showing the value 3628800.

## 5.2 Calcular multiplicação

Simples exemplo de um programa que calcula multiplicação

```

int a = 27 .
int b = 15 .
int z = 0 .
while b > 0 do
if b % 2 then
z = z + a .
b = b - 1 .
else
z = z * 2 .
b = b / 2 .
end if .
end while .
print z .

```

O código assembly gerado foi o seguinte:

```

PUSHI 27
PUSHI 15
PUSHI 0
começo1:
PUSHG 1
PUSHI 0
sup
JZ terminar1
PUSHG 1

```

```

PUSHI 2
MOD
pushi 0
sup
JZ then0
PUSHG 2
PUSHG 0
ADD
storeg 2
  PUSHG 1
PUSHI 1
SUB
storeg 1
  JUMP final0
then0:
PUSHG 2
PUSHI 2
MUL
storeg 2
  PUSHG 1
PUSHI 2
DIV
storeg 1
  final0:
  JUMP comeco1
terminar1:
  PUSHG 2
WRITEI
STOP

```

### Virtual Machine EWVM

[Examples](#)
[Documentation](#)

Write your code in the text area. Clear

```

1  START
2  PUSHI 27
3  PUSHI 15
4  PUSHI 0
5  comeco1:
6  PUSHG 1
7  PUSHI 0
8  sup
9  JZ terminar1
10 PUSHG 1
11 PUSHI 2
12 MOD
13 pushi 0
14 sup
15 JZ then0
16 PUSHG 2
17 PUSHG 0
18 ADD
19 storeg 2
20 PUSHG 1
21 PUSHI 1
22 SUB
23 storeg 1
24 JUMP final0
25 then0:
26 PUSHG 2
27 PUSHI 2
28 MUL
29 storeg 2
30 PUSHG 1
31 PUSHI 2
32 DIV
33 storeg 1
34 final0:
35 JUMP comeco1
36 terminar1:
37 PUSHG 2
38 WRITEI
39 STOP

```

Call Stack

Operand Stack

405

0

27

String Heap

Struct Heap

<< < 148 > >>
Global Pointer: 0 Frame Pointer: 0 Stack Pointer: 3

Output:

405

Browse... No file selected.

Run

## 5.3 Calcular funções sem return

Simple exemplo de um programa que utiliza funções sem return

```
int a = 99 .
int b = 50 .
int c = 10 .
int res .
amenosb begin
c = a - b .
end .
bmenosc begin
a = b - c .
end .
cmenosa begin
b = c - a .
end .
print a "\n" b "\n" c "\n" .
amenosb() .
print a "\n" b "\n" c "\n" .
bmenosc() .
print a "\n" b "\n" c "\n" .
cmenosa() .
print a "\n" b "\n" c "\n" .
```

O código assembly gerado foi o seguinte:

```
PUSHI 99
PUSHI 50
PUSHI 10
PUSHI 0
PUSHG 0
WRITEI
pushs "\n"
WRITES
PUSHG 1
WRITEI
pushs "\n"
WRITES
PUSHG 2
WRITEI
pushs "\n"
WRITES
pusha amenosb
CALL
PUSHG 0
WRITEI
pushs "\n"
WRITES
PUSHG 1
WRITEI
pushs "\n"
WRITES
```

```

PUSHG 2
WRITEI
pushs "\n"
WRITES
pusha bmenosc
CALL
PUSHG 0
WRITEI
pushs "\n"
WRITES
PUSHG 1
WRITEI
pushs "\n"
WRITES
PUSHG 2
WRITEI
pushs "\n"
WRITES
pusha cmenos
CALL
PUSHG 0
WRITEI
pushs "\n"
WRITES
PUSHG 1
WRITEI
pushs "\n"
WRITES
PUSHG 2
WRITEI
pushs "\n"
WRITES
STOP

```

```

amenosb:
PUSHG 0
PUSHG 1
SUB
storeg 2

```

```

return

```

```

bmenosc:
PUSHG 1
PUSHG 2
SUB
storeg 0

```

```
return
```

```
cmenosa:  
PUSHG 2  
PUSHG 0  
SUB  
storeg 1
```

```
return
```

Virtual Machine EWVM

Write your code in the text area. [Clear](#)

```
46 pusha cmenosa  
47 CALL  
48 PUSHG 0  
49 WRITEI  
50 pushs "\n"  
51 WRITES  
52 PUSHG 1  
53 WRITEI  
54 pushs "\n"  
55 WRITES  
56 PUSHG 2  
57 WRITEI  
58 pushs "\n"  
59 WRITES  
60 STOP  
61  
62 amenosb:  
63 PUSHG 0  
64 PUSHG 1  
65 SUB  
66 storeg 2  
67  
68 return  
69  
70 bmenosb:  
71 PUSHG 1  
72 PUSHG 2  
73 SUB  
74 storeg 0  
75  
76 return  
77  
78 cmenosa:  
79 PUSHG 2  
80 PUSHG 0  
81 SUB  
82 storeg 1  
83  
84 return
```

[Browse...](#) No file selected. [Run](#) [Copy](#)

Call Stack

Operand Stack

String Heap

Struct Heap

Global Pointer: 0 Frame Pointer: 0 Stack Pointer: 4

Output:

```
1  
48  
49
```

## 5.4 Calcular Fibonacci

Função que calcula Fibonacci

```
int res .  
int fib .  
int a .  
int b .  
int c .  
int i .  
fibonacci begin  
a = 0 .  
b = 1 .  
if fib == 0 then  
b = 0 .  
else  
i = 2 .  
while i <= fib do  
c = a + b .  
a = b .  
b = c .  
i = i + 1 .  
end while .  
end if .
```

```

end return b .
fib = 5 .
res = fibonacci() .
fib = 15 .
print res "\n" .
res = fibonacci() .
print res "\n" .

```

O código assembly gerado foi o seguinte:

```

PUSHI 0
PUSHI 0
PUSHI 0
PUSHI 0
PUSHI 0
PUSHI 0
PUSHI 5
storeg 1
pusha fibonacci
CALLstoreg 0
PUSHI 15
storeg 1
PUSHG 0
WRITEI
pushs "\n"
WRITES
pusha fibonacci
CALLstoreg 0
PUSHG 0
WRITEI
pushs "\n"
WRITES
STOP

fibonacci:
PUSHI 0
storeg 2
PUSHI 1
storeg 3
PUSHG 1
PUSHI 0
EQUAL

JZ then1
PUSHI 0
storeg 3
JUMP final1
then1:
PUSHI 2
storeg 5

```

```

    comeco0:
PUSHG 5
PUSHG 1
infeq
JZ terminar0
PUSHG 2
PUSHG 3
ADD
storeg 4
    PUSHG 3
storeg 2
    PUSHG 4
storeg 3
    PUSHG 5
PUSHI 1
ADD
storeg 5
    JUMP comeco0
terminar0:
final1:

PUSHG 3
return

```

**Virtual Machine EWVM** Examples Documentation

Write your code in the text area. Clear

```

24 STOP
25
26 fibonacci:
27 PUSHI 0
28 storeg 2
29 PUSHI 1
30 storeg 3
31 PUSHG 1
32 PUSHI 0
33 EQUAL
34
35 JZ then1
36 PUSHI 0
37 storeg 3
38 JUMP final1
39 then1:
40 PUSHI 2
41 storeg 5
42 comeco0:
43 PUSHG 5
44 PUSHG 1
45 infeq
46 JZ terminar0
47 PUSHG 2
48 PUSHG 3
49 ADD
50 storeg 4
51 PUSHG 3
52 storeg 2
53 PUSHG 4
54 storeg 3
55 PUSHG 5
56 PUSHI 1
57 ADD
58 storeg 5
59 JUMP comeco0
60 terminar0:
61 final1:
62
63

```

Browse... No file selected. Run 🔍

Call Stack

Operand Stack

Struct Heap

Global Pointer: 0 Frame Pointer: 0 Stack Pointer: 6

Output:  
5  
610

## 5.5 Calcular Exponenciação

Função que calcula Exponenciação recursivamente

```

int a .
int res .

```



```

int aux .
exp begin
if aux == 1 then res = res .
else
aux = aux - 1.
res = res * exp() .
end if .
end return res .
res = input "diga qual numero queres exponenciar \n" .
aux =input "diga quantas veze queres multiplicar \n " .
a = exp() .
print "resultado " a .

```

O código assembly gerado foi o seguinte:

```

PUSHI 0
PUSHI 0
PUSHI 0
pushs "diga qual numero queres exponenciar \n"
WRITES
read
atoi
storeg 1
  pushs "diga quantas veze queres multiplicar \n "
WRITES
read
atoi
storeg 2
  pusha exp
CALLstoreg 0
  pushs "resultado "
WRITES
PUSHG 0
WRITEI
  STOP

exp:
PUSHG 2
PUSHI 1
EQUAL

JZ then0
PUSHG 1
storeg 1
  JUMP final0
then0:
PUSHG 2
PUSHI 1
SUB
storeg 2
  PUSHG 1

```

```

pusha exp
CALLMUL
storeg 1
final0:

```

```

PUSHG 1
return

```

Virtual Machine EWVM

Examples Documentation

Write your code in the text area. Clear

```

1 START
2 PUSHI 0
3 PUSHI 0
4 PUSHI 0
5 pushs "diga qual numero queres exponenciar \n"
6 WRITES
7 read
8 atoi
9 storeg 1
10 pushs "diga quantas veze queres multiplicar \n "
11 WRITES
12 read
13 atoi
14 storeg 2
15 pusha exp
16 CALLstoreg 0
17 pushs "resultado "
18 WRITES
19 PUSHG 0
20 WRITEI
21 STOP
22
23 exp:
24 PUSHG 2
25 PUSHI 1
26 EQUAL
27
28 JZ then0
29 PUSHG 1
30 storeg 1
31 JUMP final0
32 then0:
33 PUSHG 2
34 PUSHI 1
35 SUB
36 storeg 2
37 PUSHG 1
38 pusha exp
39 CALLMUL

```

Browse... No file selected. Run

Call Stack

Operand Stack

Struct Heap

Global Pointer: 0 Frame Pointer: 0 Stack Pointer: 3

Output:

diga qual numero queres exponenciar
diga quantas veze queres multiplicar
resultado 1000

## 5.6 Calcular Área Retângulo

Função que calcula Área do retângulo recebendo inputs

```

int l1 .
int l2 .
int res .
arearet begin
l1 = l1 * l2 .
end return l1 .
l1 = input "diga o lado do retangulo \n" .
l2 = input "diga lado do segundo lado \n" .
res = arearet() .
print "area do retangulo e " res .

```

O código assembly gerado foi o seguinte:

```

PUSHI 0
PUSHI 0
PUSHI 0
pushs "diga o lado do retangulo \n"
WRITES
read
atoi

```

```

storeg 0
  pushs "diga lado do segundo lado \n"
WRITES
read
atoi
storeg 1
  pusha arearet
CALLstoreg 2
  pushs "area do retangulo e   "
WRITES
PUSHG 2
WRITEI
STOP

arearet:
PUSHG 0
PUSHG 1
MUL
storeg 0

PUSHG 0
return

```

## Virtual Machine EWVM

[Examples](#)
[Documentation](#)

Write your code in the text area. [Clear](#)

```

1  START
2  PUSHI 0
3  PUSHI 0
4  PUSHI 0
5  pushs "diga o lado do retangulo \n"
6  WRITES
7  read
8  atoi
9  storeg 0
10 pushs "diga lado do segundo lado \n"
11 WRITES
12 read
13 atoi
14 storeg 1
15 pusha arearet
16 CALLstoreg 2
17 pushs "area do retangulo e   "
18 WRITES
19 PUSHG 2
20 WRITEI
21 STOP
22
23 arearet:
24 PUSHG 0
25 PUSHG 1
26 MUL
27 storeg 0
28
29 PUSHG 0
30 return
31
32
33
34
35
36
37
38
39

```

Browse... No file selected.
Run

Call Stack

Operand Stack

2139
31
2139

Struct Heap

area do retangulo e

31

diga lado do segundo lado \n

69

diga o lado do retangulo \n

<< < 28 > >>
Global Pointer: 0
Frame Pointer: 0
Stack Pointer: 3

Output:
diga o lado do retangulo
diga lado do segundo lado
area do retangulo e 2139

## Capítulo 6

# Conclusão

Este projeto foi do mais importante para a introdução às técnicas na área do processamento de linguagens e compiladores. Permitiu-nos aplicar em casos concretos os nossos conhecimentos de análise sintática e léxica pelo módulo `yacc` e `ply.lex` do python com o uso de tokens e da definição de gramáticas através da criação de uma linguagem e a sintaxe da própria, o assembly gerado pela mesma e tratamento de erros.

Consideramos este projeto de grande importância para nós como alunos de Ciência da Computação pois agora temos uma melhor compreensão de como funcionam as linguagens de programação algo que usamos diariamente e como elas tratam o código.

# Apêndice A

## Código do Programa

### A.1 Analisador Léxico

```
import re
import ply.lex as lex

literals = ['+', '-', '*', '/', '=', '(', ')', '.', '!', '%']

tokens = (
    'IF',
    'TRUE',
    'FALSE',
    'EQUAL',
    'INF',
    'INFEQ',
    'SUP',
    'SUPEQ',
    'DIFF',
    'INPUT',
    'PRINT',
    'INT',
    'OU',
    'E',
    'THEN',
    'ELSE',
    'WHILE',
    'begin',
    'return',
    'end',
    'DO',
    'NUM',
    'ID',
    'FRASE',
)

t_TRUE = r'true'

t_FALSE = r'false'

t_EQUAL = r'\=\='

t_DIFF = r'\!\='
```

```

t_INF = r'\<'
t_INFEQ = r'\<\'
t_SUP = r'\>'
t_SUPEQ = r'\>\'
t_INPUT = r'input'
t_PRINT = r'print'
t_INT = r'int'
t_OU = r'\|'
t_E = r'\^'
t_THEN = r'then'
t_ELSE = r'else'
t_WHILE = r'while'
t_begin = r'begin'
t_return = r'return'
t_end = r'end'
t_ID = r'\w+'
t_NUM = r'[0-9]+'
t_FRASE = r'\("[^"]*"*)'
t_ANY_ignore = ' \n\t'

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

def t_DO(t):
    r'do'
    return t

def t_ANY_error(t):
    print('Illegal character: %s', t.value[0])

lexer = lex.lex()

```

## A.2 Analisador Sintático

```
import ply.yacc as yacc
import sys

from linguagem_lex import tokens

def p_Programa(p):
    "Programa : Vars Funcs Cod"
    parser.assembly = f'START\n{p[1]}\n{p[3]}\nSTOP\n{p[2]}'

def p_Escrever(p):
    "Escrever : PRINT corpoescreve '.'"
    p[0] = f"{p[2]}"

def p_corpoescreve_null(p):
    "corpoescreve : "
    p[0] = f''

def p_corpoescreve_alter(p):
    "corpoescreve : alter corpoescreve"
    p[0] = f'{p[1]}\n{p[2]}'

def p_alter_frase(p):
    "alter : FRASE"
    p[0] = f'pushs {p[1]}\nWRITES'

def p_alter_expr(p):
    "alter : expr"
    p[0] = f'{p[1]}WRITEI'

def p_Vars_Empty(p):
    "Vars : "
    p[0] = f''

def p_Vars_Var(p):
    "Vars : var Vars"
    p[0] = f'{p[1]}\n{p[2]}'

def p_Funcs_Empty(p):
    "Funcs : "
    p[0] = f''

def p_Funcs_Func(p):
    "Funcs : Funcs Func"
    p[0] = f'{p[1]}\n{p[2]}'

def p_Func_comRETURN(p):
    "Func : ID begin Cod end return expr '.'"
    p[0] = f'{p[1]}:\n{p[3]}\n{p[6]}return\n'

def p_Func(p):
    "Func : ID begin Cod end '.'"
    p[0] = f'{p[1]}:\n{p[3]}\nreturn\n'

def p_Cod_Empty(p):
    "Cod : "
    p[0] = f''
```

```

def p_var_tipoID(p):
    "var : INT ID '.'"
    var = p[2]
    p.parser.table[var] = parser.pc
    parser.pc += 1
    p[0] = "PUSHI 0\n"

def p_var_atribuicao(p):
    "var : INT ID '=' expr '.'"
    var = p[2]
    p.parser.table[var] = parser.pc
    parser.pc += 1
    p[0] = f"{p[4]}"

def p_Cod_linhas(p):
    "Cod : Linha Cod"
    p[0] = f'{p[1]} {p[2]}'

def p_Linha_Escrever(p):
    "Linha : Escrever"
    p[0] = p[1]

def p_Linha_atr(p):
    "Linha : atr"
    p[0] = p[1]

def p_linha_func(p):
    "Linha : ID '(' ')' '.'"
    p[0] = f'pusha {p[1]}\nCALL\n'

def p_Linha_Ler(p):
    "Linha : Ler"
    p[0] = p[1]

def p_Linha_Cond(p):
    "Linha : cond"
    p[0] = f'{p[1]}\n'

def p_Linha_Se(p):
    "Linha : SE"
    p[0] = p[1]

def p_se_else(p):
    "SE : IF cond THEN Cod ELSE Cod end IF '.'"
    p[0] = f'{p[2]}\nJZ then{p.parser.labels}\n{p[4]}JUMP
    ↪ final{p.parser.labels}\nthen{p.parser.labels}:\n{p[6]}final{p.parser.labels}:\n'
    p.parser.labels += 1

def p_ler(p):
    "Ler : ID '=' INPUT FRASE '.'"
    p[0] = f"pushs {p[4]}\nWRITES\nread\natoi\nstoreg {p.parser.table[p[1]]}\n"

def p_Linha_Ciclo(p):
    "Linha : Ciclo"
    p[0] = p[1]

def p_ciclo(p):
    "Ciclo : WHILE cond DO Cod end WHILE '.'"

```



```

p[0] = f'comeco{p.parser.labels}: \n{p[2]}\nJZ terminar{p.parser.labels}\n{p[4]}JUMP
↪  comeco{p.parser.labels}\nterminar{p.parser.labels}: \n'
p.parser.labels += 1

def p_atr(p):
    "atr : ID '=' expr '.'"
    p[0] = f'{p[3]}storeg {parser.table[p[1]]}\n'

def p_bool_true(p):
    "bool : TRUE"
    p[0] = f'PUSHI 1'

def p_bool_false(p):
    "bool : FALSE"
    p[0] = f'PUSHI 0'

def p_cond_bool(p):
    "cond : bool"
    p[0] = f'{p[1]}'

def p_cond_expr(p):
    "cond : expr"
    p[0] = f'{p[1]}pushi 0\nsup'

def p_oprelacao_inf(p):
    "oprelacao : INF"
    p[0] = 'inf'

def p_oprelacao_EQUALS(p):
    "oprelacao : EQUAL"
    p[0] = 'EQUAL\n'

def p_oprelacao_DIFF(p):
    "oprelacao : DIFF"
    p[0] = 'EQUAL\nNOT\n'

def p_oprelacao_infeq(p):
    "oprelacao : INFEQ"
    p[0] = 'infeq'

def p_oprelacao_sup(p):
    "oprelacao : SUP"
    p[0] = 'sup'

def p_oprelacao_supeq(p):
    "oprelacao : SUPEQ"
    p[0] = 'supeq'

def p_cond_oprelacao(p):
    "cond : expr oprelacao expr"
    p[0] = f'{p[1]}.{p[3]}.{p[2]}'

def p_cond_e(p):
    "cond : cond E cond"
    p[0] = f'{p[1]}\n{p[3]}\nadd\npushi 2\nequal'

def p_cond_ou(p):
    "cond : cond OU cond"
    p[0] = f'{p[1]}\n{p[3]}\nadd\npushi 0\nsup'

```

```

def p_expr_add(p):
    "expr : expr '+' termo"
    p[0] = p[1] + p[3] + "ADD\n"

def p_expr_sub(p):
    "expr : expr '-' termo"
    p[0] = p[1] + p[3] + "SUB\n"

def p_expr_termo(p):
    "expr : termo"
    p[0] = p[1]

def p_termo_mul(p):
    "termo : termo '*' fator"
    p[0] = p[1] + p[3] + "MUL\n"

def p_termo_div(p):
    "termo : termo '/' fator"
    p[0] = p[1] + p[3] + "DIV\n"

def p_termo_mod(p):
    "termo : termo '%' fator"
    p[0] = p[1] + p[3] + "MOD\n"

def p_termo_fator(p):
    "termo : fator"
    p[0] = p[1]

def p_fator_NUM(p):
    "fator : NUM"
    p[0] = f"PUSHI {p[1]}\n"

def p_fator_func(p):
    "fator : ID '(' ' ')' "
    p[0] = f"pusha {p[1]}\nCALL"

def p_fator_ID(p):
    "fator : ID"
    p[0] = f"PUSHG {parser.table[p[1]]}\n"

def p_fator_expr(p):
    "fator : '(' expr ')'"
    p[0] = p[2]

def p_error(p):
    print("Syntax error:", p)
    parser.sucesso = False

#inicio do Parser e do Processamento
parser = yacc.yacc()
parser.table = {}
parser.pc = 0

parser.sucesso = True
parser.assembly = ""
parser.labels = 0

```

```
fonte = ""  
for linha in sys.stdin:  
    fonte += linha  
  
parser.parse(fonte)  
  
print(parser.table)  
print(parser.assembly)
```