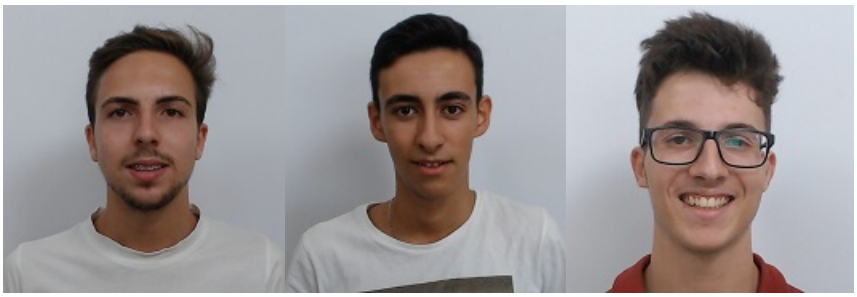
Universidade do Minho
Escola de Engenharia

Processamento de Linguagens

Trabalho Prático 2

Grupo 56
Guilherme Martins a89532
João Pereira a89607
Tiago Freitas a89570



A89532 a89607 A89570

Conteúdo

1 Introdução

Como 2.º trabalho prático da Unidade Curricular de Processamento de Linguagens, foi proposto o desenvolvimento de um compilador de uma linguagem à nossa escolha, e que gere pseudo-código para uma máquina de stack virtual, a *Virtual Machine*, no nosso caso.

Neste relatório, iremos apresentar todos os passos e decisões que fomos tomando ao longo do desenvolvimento do projeto.

1.1 Objetivos

Este projeto será importante para aumentar a experiência em engenharia de linguagens e em programação generativa, aprofundando os temas já lecionados durantes as aulas, as gramáticas independentes de contexto (GIC) e as gramáticas tradutoras (GT).

O principal objetivo do trabalho prático será desenvolver um compilador gerando código para uma máquina de stack virtual (VM - Virtual Machine).

Iremos também utilizar e examinar ainda mais geradores de compiladores baseados em gramáticas tradutoras, o *YACC*, que será completado pelo gerador de análises léxicos, o *LEX*.

Com este projeto a nossa equipa pretende ser capaz de resolver qualquer problema que envolva as GIC e ,no contexto do que é abordado neste trabalho, fazer o *parsing* do máximo de funcionalidades que a nossa linguagem pode ter, e poder implementar as suas instruções na máquina virtual.

2 Descrição do problema

Para inicializar o problema, foi-nos pedido que começássemos por definir uma linguagem de programação imperativa simples, à qual terá de permitir:

- declarar varáveis 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 utilizamos o **for-do**, visto que $56\%3 = 2$.
- por fim, das duas funcionalidades disponíveis, optamos por declarar e manusear variáveis estruturadas do tipo array (1 dimensão) de inteiros, em relação aos quais é apenas permitida a operação de indexação (índice inteiro).

De seguida, teremos que desenvolver um compilador para essa linguagem com basa na GIC já criada e com recurso aos módulos YACC e LEX.

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

3 Implementação da solução

3.1 Definição da nossa linguagem

Para abordar a questão da linguagem, começamos por analisar os requisitos que esta devia cumprir: ser uma linguagem imperativa segundo a qual fosse possível cumprir todas as funcionalidades acima descritas.

Decidimos criar a nossa própria linguagem, de forma a simplificar algumas funcionalidades e fazer algo exclusivo da equipa.

Utilizamos uma mistura da sintaxe do C com o Haskell, e definimos as seguintes funcionalidades assim:

- **1. Declarar variaveis atomicas do tipo inteiro, com os quais se podem realizar as habituais operacoes aritmeticas, relacionais e logicas.**

```
-> int nome
-> int i = 0
```

- **2. Operações aritméticas**

```
'+' -> add (a b)
'-' -> sub (a b)
'*' -> mul (a b)
'/' -> div (a b)
'%' -> mod (a b)
```

- **3. Operações relacionais**

```
'==' -> eq (a b)
'!=' -> diff (a b)
'>'  -> grt (a b)
'>=' -> geq (a b)
'<'  -> lwr (a b)
'<=' -> leq (a b)
```

- **4. Operações lógicas**

```
'&&' -> and (a b)
'||' -> or (a b)
'!'  -> not (a)
```

- **5. Atribuições**

```
'=' -> atrib (var (exp))
```

- **6. Input e output**

```
read (var)
write (exp)
```

- **7. Instruções condicionais**

```
if (cond) then {
    action1
} else {
    action2
}

if (cond) then {
    action
}
```

- **8. Instruções cíclicas**

```
for ( Atrib ; Cond ; Atrib ) {
    action
}
```

- **9. Arrays**

```
int lista[10]

atrib( lista[2] (exp) )
write(lista[4])
```

A nossa linguagem ficou assim formada por estas 9 divisões. Tínhamos mais objetivos em mente como manipular **strings**, **floats**, e **arrays** de 2 dimensões, caso tivéssemos mais tempo para continuar o desenvolvimento do projeto, e para extender a nossa GIC.

3.2 Lexer

Depois de termos a linguagem definida facilmente conseguimos pensar nos *tokens* necessários na nossa linguagem, para fazer a análise léxica do código fonte. O desenvolvimento do **compilador_lex** foi progressivo, e em paralelo com o **compilador_yacc**, no entanto, o nosso primeiro passo foi definir os *tokens* mais essenciais antes de começar a desenvolver a gramática.

À medida que fomos avançando na implementação da solução adicionamos novos *tokens* ao ficheiro **lex**.

O nosso conjunto de *tokens* e *literals* final é o seguinte:

```
# Token declarations
tokens = [
    'START', 'END',
    'INT', 'NUM',
    'ID',
    'ATRIB',
    'ADD', 'SUB', 'MUL', 'DIV', 'MOD',
    'EQ', 'DIFF', 'GRT', 'GEQ', 'LWR', 'LEQ',
    'AND', 'OR', 'NOT',
    'READ', 'WRITE',
    'IF', 'THEN', 'ELSE',
    'FOR'
]

literals = ['(',')','{','}','=','[',']',';','-']
```

Inicialmente utilizamos as expressões regulares dos *tokens* sem lhes associar nenhuma ação, mas mais tarde percebemos que nos dava problemas, porque por exemplo, a palavra reservada **int** podia ser reconhecida como um **id**. Deste modo, decidimos associar uma ação a todos os *tokens*, retornando-os. E colocamos os mais prioritários em cima, ou seja, as palavras reservadas. Um exemplo é o seguinte (corresponde ao ID que fica em baixo porque só deve ser reconhecido se nenhuma palavra reservada for):

```
def t_ID(t):
    r'\_?[a-zA-Z]+\d*'
    return t
```

3.3 GIC e utilização do yacc

Depois de ter uma base dos *tokens* a utilizar, começamos a desenvolver a nossa gramática para a linguagem, em que cada produção tem uma ação semântica associada para converter o código fonte em **pseudo-código** da VM. Utilizamos uma ordem específica para as produções da gramática, sempre tendo em conta o algoritmo **Bottom-up LR**.

A ordem que seguimos para desenvolver a GIC foi esta:

1. Operações aritméticas
2. Declaração de variáveis do tipo inteiro
3. Operações relacionais
4. Operações lógicas
5. Read e write para inteiros e expressões
6. Atribuições
7. Estruturas condicionais (if)
8. Estruturas cíclicas (for)
9. Declaração de arrays de inteiros
10. Atribuição e leitura de valores de arrays
11. Read e write com arrays

No final obtivemos este conjunto representativo da nossa gramática:

```
#
# T = {
#       'START', 'END',
#       'INT', 'NUM', 'ID', 'ATRIB',
#       'ADD', 'SUB', 'MUL', 'DIV', 'MOD', 'EQ', 'DIFF', 'GRT', 'GEQ', 'LWR', 'LEQ',
#       'AND', 'OR', 'NOT', 'READ', 'WRITE', 'IF', 'THEN', 'ELSE', 'FOR',
#       '(', ')', '{', '}', ';', '[', ']', '-'
# }
#
#
# N = {
#       Linguagem, Decls, Instrs, Decl, DeclAtrib, DeclArray,
#       CabecaInstrs, CaudaInstrs,
#       ReadContent, WriteContent, ReadCRest, Value,
#       Atrib, AtribArray,
#       Logic, LogicNot, Relac, Exp, Termo, Factor
# }
#
```

```
#
#
#  Linguagem --> Decls START Instrs END
#
#  Decls --> Decl Decls
#          |
#
#  Decl --> INT ID DeclAtrib
#         | DeclArray
#
#
#  DeclAtrib --> '=' Logic
#              |
#
#
#  DeclArray --> INT ID '[' NUM ']'
#
#
#
#
#  Instrs --> CabecaInstrs CaudaInstrs
#
#  CabecaInstrs --> READ '(' ReadContent ')'
#                 | WRITE '(' WriteContent ')'
#                 | IF '(' Logic ')' THEN '{' Instrs '}'
#                 | IF '(' Logic ')' THEN '{' Instrs '}' ELSE '{' Intrs '}'
#                 | FOR '(' Atrib ';' Logic ';' Atrib ')' '{' Instrs '}'
#                 | Atrib
#                 | AtribArray
#
#
#
#  ReadContent --> ID ReadCRest
#
#  ReadCRest --> '[' Value ']'
#              |
#
#
#
#  WriteContent --> Logic
#                 | ID '[' Value ']'
#
#
#
#
#
#  Atrib --> ATRIB '(' ID '(' Logic ')' ')'
#
#
```

```
#  AtribArray --> ATRIB '(' ID '[' Value ']' '(' Logic ')' ')'
#
#
#
#  Value --> ID
#          | NUM
#
#
#
#
#
#  CaudaInstrs --> CabecaInstrs CaudaInstrs
#                |
#
#
#  Logic --> AND '(' Logic LogicNot ')'
#          | OR  '(' Logic LogicNot ')'
#          | LogicNot
#
#
#  LogicNot --> NOT '(' Logic ')'
#             | Relac
#
#
#  Relac -->  EQ '(' Logic Exp ')'
#          |  DIFF '(' Logic Exp ')'
#          |  GRT '(' Logic Exp ')'
#          |  GEQ '(' Logic Exp ')'
#          |  LWR '(' Logic Exp ')'
#          |  LEQ '(' Logic Exp ')'
#          |  Exp
#
#
#  Exp --> ADD '(' Exp Termo ')'
#        |  SUB '(' Exp Termo ')'
#        |  Termo
#
#
#  Termo --> MUL '(' Exp Termo ')'
#          |  DIV '(' Exp Termo ')'
#          |  MOD '(' Exp Termo ')'
#          |  Factor
#
#  Factor -->  '(' Logic ')'
#            |  NUM
#            |  ID
#            |  '-' NUM
```

3.4 Estruturas utilizadas

Ao longo do desenvolvimento do projeto surgiu-nos a necessidade de utilizar variáveis globais para controlar os registos das variáveis do código fonte, assim como do *gp* e os contadores dos ifs e fors.

Temos uma tabela de registos para as variáveis do tipo **int**, guardando lá os eu tipo e o *offset* na stack. Também temos uma idêntica para as variáveis do tipo **arrayInt**, onde guardamos o tipo e também o *offset* de início na stack.

Sempre que é declarada uma variável a tabela de registos é atualizada e o *gp* é incrementado. Para ter controlo das *labels* geradas para os ciclos e as estruturas condicionais utilizamos contadores, de modo a nunca repetir as *labels*.

```
parser.registers = {}
parser.arrays = {}
parser.gp = 0
parser.if_counter = 0
parser.for_counter = 0
```

4 Resultados obtidos

Como resultado final conseguimos produzir código para a máquina virtual que nos permite fazer todo o tipo de operações (lógicas, relacionais e aritméticas), declarar variáveis do tipo inteiro, e do tipo array de inteiro (unidimensional), efetuar atribuições às variáveis existentes, aplicar condicionais, instruções cíclicas (for do C) e por fim instruções de input e output envolvendo tudo o que foi mencionado anteriormente.

Temos vários ficheiros de teste, que testam todas as funcionalidades progressivamente, no entanto vamos apresentar apenas no relatório aqueles requeridos no enunciado do projeto:

4.1 Ler 4 números e dizer se podem ser os lados de um quadrado

Ficheiro fonte

```
int input
int i
int acc
int final = 1

start
    for( atrib(i (1)) ; and( leq(i 4) final ) ; atrib(i ( add(i 1) ) ) ){
        read(input)
        if( grt(i 1) ) then {
            if( diff(acc input) ) then {
                atrib(final (0))
            }
        } else{
            atrib(acc (input))
        }
    }
    write(final)
end
```

Ficheiro vm

```
PUSHI 0
PUSHI 0
PUSHI 0
PUSHI 1

START

PUSHI 1
STOREG 1
BEGINFOR1:

PUSHG 1
PUSHI 4
INFEQ
PUSHG 3
ADD
PUSHI 2
EQUAL
JZ ENDFOR1

READ
ATOI
STOREG 0
PUSHG 1
```

```
PUSHI 1
SUP
JZ ELSE2

PUSHG 2
PUSHG 0
EQUAL
NOT
JZ ENDIF1

PUSHI 0
STOREG 3
ENDIF1:

JUMP ENDIF2

ELSE2:

PUSHG 0
STOREG 2
ENDIF2:

PUSHG 1
PUSHI 1
ADD
STOREG 1
JUMP BEGINFOR1

ENDFOR1:

PUSHG 3
WRITEI

STOP
```

Output na VM

Figura 1: Verificação com sucesso

Figura 2: Lado inválido

4.2 Ler um inteiro N, depois ler N números e escrever o menor deles

Ficheiro fonte

```
int min
int n
int i
int input

start
    read(n)
    for( atrib(i (1)) ; leq(i n) ; atrib(i ( add(i 1) ) ) ){
        read(input)
        if( eq(i 1) ) then {
            atrib(min (input))
        } else{
            if ( lwr(input min) ) then {
                atrib(min (input))
            }
        }
    }

    write(min)
end
```

Ficheiro vm

```
PUSHI 0
PUSHI 0
PUSHI 0
PUSHI 0

START

READ
ATOI
```

```
STOREG 1
PUSHI 1
STOREG 2
BEGINFOR1:

PUSHG 2
PUSHG 1
INFEQ
JZ ENDFOR1

READ
ATOI
STOREG 3
PUSHG 2
PUSHI 1
EQUAL
JZ ELSE2

PUSHG 3
STOREG 0
JUMP ENDIF2

ELSE2:

PUSHG 3
PUSHG 0
INF
JZ ENDIF1

PUSHG 3
STOREG 0
ENDIF1:

ENDIF2:

PUSHG 2
PUSHI 1
ADD
STOREG 2
JUMP BEGINFOR1

ENDFOR1:

PUSHG 0
WRITEI

STOP
```

Output na VM

Figura 3: Menor

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

Ficheiro fonte

```
int n = 3
int prod = 1
int i
int input

start
    for ( atrib(i (1)) ; leq(i n) ; atrib(i ( add(i 1)) ) ){
        read(input)
        atrib(
            prod
            ( mul(prod input) )
        )
    }
    write(prod)
end
```

Ficheiro vm

```
PUSHI 3
PUSHI 1
PUSHI 0
PUSHI 0

START
```

```
PUSHI 1
STOREG 2
BEGINFOR1:

PUSHG 2
PUSHG 0
INFEQ
JZ ENDFOR1

READ
ATOI
STOREG 3
PUSHG 1
PUSHG 3
MUL
STOREG 1
PUSHG 2
PUSHI 1
ADD
STOREG 2
JUMP BEGINFOR1

ENDFOR1:

PUSHG 1
WRITEI

STOP
```

Output na VM

Figura 4: Produtório

4.4 Contar e imprimir os números impares de uma sequência de números naturais

Ficheiro fonte

```
int minIntervalo
int maxIntervalo
int sum
int i

start

    read(minIntervalo)
    read(maxIntervalo)
    for( atrib(i (minIntervalo)) ; leq(i maxIntervalo) ; atrib(i (add(i 1))) ){
        if ( mod(i 2) ) then {
            write(i)
            atrib (sum (add(sum 1)))
        }
    }

    write(sum)

end
```

Ficheiro vm

```
PUSHI 0
PUSHI 0
PUSHI 0
PUSHI 0

START

READ
ATOI
STOREG 0
READ
ATOI
STOREG 1
PUSHG 0
STOREG 3
BEGINFOR1:

PUSHG 3
PUSHG 1
INFEQ
JZ ENDFOR1

PUSHG 3
```

```
PUSHI 2
MOD
JZ ENDIF1

PUSHG 3
WRITEI
PUSHG 2
PUSHI 1
ADD
STOREG 2
ENDIF1:

PUSHG 3
PUSHI 1
ADD
STOREG 3
JUMP BEGINFOR1

ENDFOR1:

PUSHG 2
WRITEI

STOP
```

Output na VM

Figura 5: Números ímpares

4.5 Ler e armazenar N números num array; imprimir os valores por ordem inversa

Ficheiro fonte

```
int array[5]
int i
int input
int index

start

    for( atrib(i (1)) ; leq(i 5) ; atrib(i (add(i 1))) ) {
        read(input)
        atrib( index ( sub(i 1) ) )
        atrib( array[index] (input) )
    }

    for( atrib(i (5)) ; grt(i 0) ; atrib(i (sub(i 1))) ) {
        atrib(index ( sub(i 1) ) )
        write(array[index])
    }

end
```

Ficheiro vm

```
PUSHN 5
PUSHI 0
PUSHI 0
PUSHI 0

START

PUSHI 1
STOREG 5
BEGINFOR1:

PUSHG 5
PUSHI 5
INFEQ
JZ ENDFOR1

READ
ATOI
STOREG 6
PUSHG 5
PUSHI 1
SUB
STOREG 7
```

```
PUSHGP
PUSHI 0
PADD
PUSHG 7
PUSHG 6
STOREN
PUSHG 5
PUSHI 1
ADD
STOREG 5
JUMP BEGINFOR1

ENDFOR1:

PUSHI 5
STOREG 5
BEGINFOR2:

PUSHG 5
PUSHI 0
SUP
JZ ENDFOR2

PUSHG 5
PUSHI 1
SUB
STOREG 7
PUSHGP
PUSHI 0
PADD
PUSHG 7
LOADN
WRITEI
PUSHG 5
PUSHI 1
SUB
STOREG 5
JUMP BEGINFOR2

ENDFOR2:


STOP
```

Output na VM

Figura 6: Ordem inversa

5 Conclusões e trabalho futuro

Este trabalho prático foi bastante importante para o grupo consolidar todos os conhecimentos adquiridos ao longo do semestre no contexto de *Processamento de Linguagens*, visto que nos obrigou a entender melhor geradores de compiladores e gerador de análises léxicos, o *YACC* e o *LEX*.

O grupo sente que foi capaz de planear bem a implementação do problema e soube corresponder ao que era pedido, no entanto, sentimos que há aspetos que poderíam ter sido melhorados se tivéssemos mais tempo dedicado ao trabalho, o que foi complicado, uma vez que nos encontramos numa altura agitada do nosso semestre.

6 Código

6.1 Lexer

```
import ply.lex as lex # importar s   a parte l xica do ply
import sys

# Token declarations
tokens = [
    'START', 'END',
    'INT', 'NUM',
    'ID',
    'ATRIB',
    'ADD', 'SUB', 'MUL', 'DIV', 'MOD',
    'EQ', 'DIFF', 'GRT', 'GEQ', 'LWR', 'LEQ',
    'AND', 'OR', 'NOT',
    'READ', 'WRITE',
    'IF', 'THEN', 'ELSE',
    'FOR'
]

literals = ['(',')','{','}','=','[',']',';','-']

# Token regex

# Tokens with some action code
def t_NUM(t):
    r'\d+'
    return t

def t_INT(t):
    r'int'
    return t

def t_ADD(t):
    r'add'
    return t

def t_SUB(t):
    r'sub'
    return t

def t_DIV(t):
    r'div'
    return t

def t_MUL(t):
    r'mul'
    return t
```

```
def t_MOD(t):
    r'mod'
    return t

def t_EQ(t):
    r'eq'
    return t

def t_DIFF(t):
    r'diff'
    return t

def t_GRT(t):
    r'grt'
    return t

def t_GEQ(t):
    r'geq'
    return t

def t_LWR(t):
    r'lwr'
    return t

def t_LEQ(t):
    r'leq'
    return t

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

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

def t_NOT(t):
    r'not'
    return t

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

def t_WRITE(t):
    r'write'
    return t
```

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

def t_THEN(t):
    r'then'
    return t

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

def t_FOR(t):
    r'for'
    return t

def t_ATRIB(t):
    r'atrib'
    return t

def t_START(t):
    r'start|START'
    return t

def t_END(t):
    r'end|END'
    return t

def t_ID(t):
    r'\_?[a-zA-Z]+\d*'
    return t

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

# Characters to be ignored
t_ignore = " \t\n"

# Errors
def t_error(t):
    print(f"Car ter errado {t.value[0]}")
    t.lexer.skip(1)

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

6.2 Yacc

```
import ply.yacc as yacc
import sys

from compilador_lex import tokens

#Produ  o da linguagem
def p_Linguagem(p):
    "Linguagem : Decls START Instrs END"
    p[0] = p[1] + '\n\nSTART\n' + p[3] + '\n\nSTOP\n'

#Produ  es Decls
def p_Decls(p):
    "Decls : Decl Decls"
    p[0] = p[1] + p[2]

def p_Decls_empty(p):
    "Decls : "
    p[0] = ''

#Produ  es Decl
def p_Decl(p):
    "Decl : INT ID DeclAtrib"
    if(p[3] == ""):
        p[0] = '\nPUSHI 0'
    else:
        p[0] = p[3]
    p.parser.registers.update({p[2]: (p[1],p.parser.gp)})
    p.parser.gp += 1

def p_Decl_array(p):
    "Decl : DeclArray"
    p[0] = p[1]


#Produ  es DeclArray
def p_DeclArray(p):
    "DeclArray : INT ID '[' NUM ']'"
    p[0] = '\nPUSHN ' + p[4]
    p.parser.arrays.update({p[2] : (p[1],p.parser.gp)})
    p.parser.gp += int(p[4])


#Produ  es DeclAtrib
def p_DeclAtrib_logic(p):
    "DeclAtrib : '=' Logic"
    p[0] = p[2]

def p_DeclAtrib_empty(p):
    "DeclAtrib : "
    p[0] = ''


#Produ  es das instru  es
def p_Instrs(p):
    "Instrs : CabecaInstrs CaudaInstrs"
    p[0] = p[1] + p[2]

#Produ  es de uma instru  o
def p_CabecaInstrs_Read(p):
    "CabecaInstrs : READ '(' ReadContent ')'"
    p[0] = p[3]

def p_CabecaInstrs_Write(p):
    "CabecaInstrs : WRITE '(' WriteContent ')'"
    p[0] = p[3] + '\nWRITEI'

def p_CabecaInstrs_IfT(p):
    "CabecaInstrs : IF '(' Logic ')' THEN '{' Instrs '}'"
    p.parser.if_counter += 1
    counter = str(p.parser.if_counter)
    p[0] = p[3] + "\nJZ ENDIF" + counter + "\n" + p[7] + "\nENDIF" + counter + ":\n"

def p_CabecaInstrs_IfTE(p):
    "CabecaInstrs : IF '(' Logic ')' THEN '{' Instrs '}' ELSE '{' Instrs '}'"
    p.parser.if_counter += 1
    counter = str(p.parser.if_counter)
    p[0] = p[3] + "\nJZ ELSE" + counter + "\n" + p[7] + "\nJUMP ENDIF" + counter + "\n" + "\nELSE" + counter

def p_CabecaInstrs_For(p):
    "CabecaInstrs : FOR '(' Atrib ';' Logic ';' Atrib ')' '{' Instrs '}'"
```

```
    p.parser.for_counter += 1
    counter = str(p.parser.for_counter)
    p[0] = p[3] + "\nBEGINFOR" + counter + ":\n" + p[5] + "\nJZ ENDFOR" + counter + "\n" + p[10] + p[7] + "\n

def p_CabecaInstrs_Atrib(p):
    "CabecaInstrs : Atrib"
    p[0] = p[1]

def p_CabecaInstrs_AtribArray(p):
    "CabecaInstrs : AtribArray"
    p[0] = p[1]


#Produ  es readContent
def p_ReadContent(p):
    "ReadContent : ID ReadCRest"
    if(p[2] == ''):
        (_,offset) = p.parser.registers.get(p[1])
        p[0] = '\nREAD\nATOI\nSTOREG ' + str(offset)
    else:
        (_,offset) = p.parser.arrays.get(p[1])
        p[0] = '\nPUSHGP\nPUSHI ' + str(offset) + '\nPADD' + p[2] + '\nREAD\nATOI\nSTOREN'


#Produ  es ReadCRest
def p_ReadCRest_Array(p):
    "ReadCRest : '[' Value ']'"
    p[0] = p[2]

def p_ReadCRest_empty(p):
    "ReadCRest : "
    p[0] = ''



#Produ  es WriteContent
def p_WriteContent_Logic(p):
    "WriteContent : Logic"
    p[0] = p[1]

def p_WriteContent_Array(p):
    "WriteContent : ID '[' Value ']'"
    (_,offset) = p.parser.arrays.get(p[1])
    p[0] = '\nPUSHGP\nPUSHI ' + str(offset) + '\nPADD' + p[3] + '\nLOADN'


#Produ  es atribui  o
def p_Atrib(p):
    "Atrib : ATRIB '(' ID '(' Logic ')' ')'"
    (_,offset) = p.parser.registers.get(p[3])
    p[0] = p[5] + '\nSTOREG ' + str(offset)



#Produ  es atribArray
def p_AtribArray(p):
    "AtribArray : ATRIB '(' ID '[' Value ']' '(' Logic ')' ')'"
    (_,offset) = p.parser.arrays.get(p[3])
    p[0] = '\nPUSHGP ' + '\nPUSHI ' + str(offset) + '\nPADD' + p[5] + p[8] + '\nSTOREN'



#Produ  es Value
def p_Value_ID(p):
    "Value : ID"
    (_,offset) = p.parser.registers.get(p[1])
    p[0] = '\nPUSHG ' + str(offset)

def p_Value_NUM(p):
    "Value : NUM"
    p[0] = '\nPUSHI ' + p[1]



#Produ  es cauda de instru  es
def p_CaudaInstrs_Instrs(p):
    "CaudaInstrs : CabecaInstrs CaudaInstrs"
```

```
    p[0] = p[1] + p[2]

def p_CaudaInstrs_empty(p):
    "CaudaInstrs : "
    p[0] = ''


#Produ es das opera es l gicas
def p_Logic_AND(p):
    "Logic : AND '(' Logic LogicNot ')'"
    p[0] = p[3] + p[4] + '\nADD\nPUSHI 2\nEQUAL'

def p_Logic_OR(p):
    "Logic : OR '(' Logic LogicNot ')'"
    p[0] = p[3] + p[4] + '\nADD\nPUSHI 0\nEQUAL\nNOT'

def p_Logic_LogicNot(p):
    "Logic : LogicNot"
    p[0] = p[1]



#Produ es para o not
def p_LogicNot_not(p):
    "LogicNot : NOT '(' Logic ')'"
    p[0] = p[3] + '\nNOT'

def p_LogicNot_Relac(p):
    "LogicNot : Relac"
    p[0] = p[1]



#Produ es das opera es relacionais
def p_Relac_EQ(p):
    "Relac : EQ '(' Logic Exp ')'"
    p[0] = p[3] + p[4] + '\nEQUAL'

def p_Relac_DIFF(p):
    "Relac : DIFF '(' Logic Exp ')'"
    p[0] = p[3] + p[4] + '\nEQUAL\nNOT'

def p_Relac_GRT(p):
    "Relac : GRT '(' Logic Exp ')'"
    p[0] = p[3] + p[4] + '\nSUP'

def p_Relac_GEQ(p):
    "Relac : GEQ '(' Logic Exp ')'"
    p[0] = p[3] + p[4] + '\nSUPEQ'

def p_Relac_LWR(p):
    "Relac : LWR '(' Logic Exp ')'"
    p[0] = p[3] + p[4] + '\nINF'

def p_Relac_LEQ(p):
    "Relac : LEQ '(' Logic Exp ')'"
    p[0] = p[3] + p[4] + '\nINFEQ'

def p_Relac_Exp(p):
    "Relac : Exp"
    p[0] = p[1]



#Produ es Exp
def p_Exp_add(p):
    "Exp : ADD '(' Exp Termo ')'"
    p[0] = p[3] + p[4] + '\nADD'

def p_Exp_sub(p):
    "Exp : SUB '(' Exp Termo ')'"
    p[0] = p[3] + p[4] + '\nSUB'

def p_Exp_Termo(p):
    "Exp : Termo"
    p[0] = p[1]

#Produ es Termo
def p_Termo_mul(p):
    "Termo : MUL '(' Exp Termo ')'"
    p[0] = p[3] + p[4] + '\nMUL'

def p_Termo_div(p):
    "Termo : DIV '(' Exp Termo ')'"
```

```
    p[0] = p[3] + p[4] + '\nDIV'

def p_Termo_mod(p):
    "Termo : MOD '(' Exp Termo ')'"
    p[0] = p[3] + p[4] + '\nMOD'

def p_Termo_factor(p):
    "Termo : Factor"
    p[0] = p[1]

#Produ  es Factor
def p_Factor_group(p):
    "Factor : '(' Logic ')'"
    p[0] = p[2]

def p_Factor_num(p):
    "Factor : NUM"
    p[0] = '\nPUSHI ' + p[1]

def p_Factor_ID(p):
    "Factor : ID"
    (_, offset) = p.parser.registers.get(p[1])
    p[0] = '\nPUSHG ' + str(offset)

def p_Factor_Negativos(p):
    "Factor : '-' NUM"
    p[0] = '\nPUSHI ' + str(-1 * int(p[2]))



# Error rule for syntax errors
def p_error(p):
    print('Syntax error in input: ', p)

# Build the parser
parser = yacc.yacc()

# Creating the model
parser.registers = {}
parser.arrays = {}
parser.gp = 0
parser.if_counter = 0
parser.for_counter = 0


path = 'testesLinguagem/Final/'
print("Ficheiro para ler: ")
i = input()
pathI = path + i
file = open(pathI,"r")

cont = ''
for linha in file:
    cont += linha

print("Output: ")
o = input()
pathO = path + o

f = open(pathO,"w")
result = parser.parse(cont)

f.write(result)
```