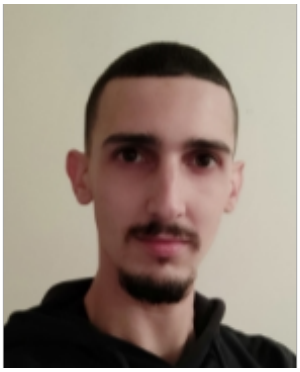
Processamento de Linguagens

MIEI - 3º ANO - 2º SEMESTRE
UNIVERSIDADE DO MINHO

COMPILADOR PARA LINGUAGEM IMPERATIVA

Grupo 58



Renato Cruzinha
A75310

Ricardo Leal
A75411

Braga, 30 de maio de 2021

Conteúdo

1. Introdução

No âmbito da unidade curricular de *Processamento de Linguagens* foi proposto que implementássemos um compilador para uma linguagem imperativa feita por nós, fazendo uso da biblioteca *PLY* da linguagem *Python*. Neste biblioteca, são importadas as componentes do *LEX* e do *YACC*.

O desenvolvimento deste projeto tem como principais objetivos aumentar a nossa capacidade e experiência em gramáticas tradutoras e de gramáticas independentes de contexto assim como o desenvolvimento de processadores de linguagens.

Por fim, de notar ainda que este relatório foi ainda desenvolvido usando LaTeX.

2. Enunciado

O enunciado dita as regras para desenvolver um *compilador de linguagem imperativa*. Para isso, foi necessário fazer a criação da sintaxe da linguagem que queremos compilar, de modo a podermos compilar para o *assembly* da máquina virtual VM.

2.1 Descrição do problema

Para o correto funcionamento da linguagem imperativa por nós definida, é necessário que esta responda aos seguintes requisitos:

- Declarar variáveis atómicas.
- Efetuar operações algorítmicas.
- Ler e escrever no standard input e output, respetivamente.
- Efetuar instruções condicionais.
- Efetuar instruções cíclicas.
- Declarar e manusear arrays de uma ou duas dimensões.
- Definir e invocar funções sem argumentos mas que retornem um valor inteiro.

3. Implementação da Solução

A primeira coisa que tivemos de fazer para a resolução deste projeto, foi a criação de uma linguagem imperativa, seguido da criação da parte léxica e da gramática que iria traduzir a nossa linguagem em linguagem assembly da máquina virtual VM.

De seguida, apresentamos a GIC da nossa linguagem imperativa.

```
g = '''
        Limp : BlocoDeclaracoes BEGIN BlocoInstrucoes

        BlocoDeclaracoes : Declaracao
                         | BlocoDeclaracoes Declaracao

        BlocoInstrucoes : Instrucao
                        | BlocoInstrucoes Instrucao

        Declaracao : DeclVar  ';'
                   | DeclArray ';'
                   | DeclArrayBi ';'
                   | DeclFun

        DeclVar : INT id '=' ExpA
                | INT id

        DeclArray : INT id [ number ]

        DeclArrayBi : INT id '[' number ']' '[' number ']'

        DeclFun : FUNCTION  id '(' ')' '{' BlocoInstrucoes RETURN ExpRel ';' '}'

        Instrucao : DUMP ';'
                  | PRINT ExpA ';'
                  | PRINTA ';'
                  | READ id ';'
                  | READ id '[' number ']' ';'
                  | Atribuicao ';'
                  | Condicional

        Atribuicao : AtrVar
                   | AtrArray
                   | AtrArrayBi
                   | AtrFun

        AtrVar : id '=' ExpA

        AtrArray : id [ number ] '=' ExpA

        AtrArrayBi : id '[' number ']' '[' number ']' '=' ExpA

        AtrFun : id '='  id '(' ')'


        Condicional : | IF '(' Condicao ')' '{' BlocoInstrucoes '}' ELSE '{'
```

```
    BlocoInstrucoes '}'
                        | IF '(' Condicao ')' '{' BlocoInstrucoes '}'
                        | REPEAT '{' BlocoInstrucoes '}' 'UNTIL' '(' Condicao ')'
                        | WHILE '(' Condicao ')' DO '{' BlocoInstrucoes '}'


        (bottom up -> coisas com mais prioridade ficam mais a baixo/direita)

        Condicao : ExpLogOr

        ExpLogOr : ExpLogAnd
                 | ExpLogOr OR ExpLogAnd

        ExpLogAnd : ExpLogNot
                  | ExpLogAnd AND ExpLogOr

        ExpLogNot : ExpEq
                  | '!' Condicao

        ExpEq : ExpRel
              | ExpEq EQ ExpRel
              | ExpEq NE ExpRel

        ExpRel : ExpA
               | ExpRel '>' ExpA
               | ExpRel '<' ExpA
               | ExpRel '>=' ExpA
               | ExpRel '<=' ExpA

        ExpA : ExpA '+' Term
             | ExpA '-' Term
             | Term

        Term : Term '*' Factor
             | Term '/' Factor
             | Factor

        Factor : id
               | number
               | '(' ExpA ')'
               | id '[' number ']'
               | id '[' number ']' '[' number ']'
               | True
               | False
    '''
```

3.1 Léxico

Para começar é importado o módulo LEX da biblioteca PLY. De seguida, são declaradas as palavras reservadas para que o compilador as consiga identificar.

```
reserved = {
    'int'      : 'INT',
    'print'    : 'PRINT',
    'printa'   : 'PRINTA',
    'read'     : 'READ',
    'dump'     : 'DUMP',
    'if'       : 'IF',
    'else'     : 'ELSE',
    'repeat'   : 'REPEAT',
    'until'    : 'UNTIL',
    'function' : 'FUNCTION',
```

```
    'return'    : 'RETURN',
    'while'     : 'WHILE',
    'do'        : 'DO',
}
```

A seguir são declarados os tokens que permitem fazer a atribuição de valores, assim como a sua comparação e valores lógicos. São ainda definidos os carateres especiais que permitem ler a linguagem de forma inequívoca.

```
# Tokens
tokens = [
    'number', 'id', 'BEGIN', 'EQ', 'NE', 'GE', 'LE', 'AND', 'NOT', 'OR', 'TRUE', '
    FALSE',
 ] + list(reserved.values())

# Literals
literals = ['+', '-', '*', '/', '(', ')', '{', '}', '?', ';', '=', '[', ']', '>', '<
    ']
t_BEGIN = r'BEGIN'
```

3.1.1 Expressões Regulares

Após estas declarações, são definidas as expressões regulares que vamos precisar, neste sentido, são as seguinte:

- **t_EQ** `r'=='` : Representa a igualdade entre expressões.
- **t_NE**: `r'!='` : Representa a diferença entre expressões.
- **t_GE**: `r'>='` : Representa a verificação se uma expressão é maior ou igual a outra.
- **t_LE**: `r'<='` : Representa a verificação se uma expressão é menor ou igual a outra.
- **t_AND**: `r'&&'` : Representa o operador lógico de conjunção.
- **t_OR**: **r'||'**: Representa o operador lógico de disjunção.
- **t_NOT**: `r'!'` : Representa a negação de uma expressão.
- **t_id**: `r'[a-z]+'`: Representa o id de uma função ou ciclo.
- **t_VARS**: `r'VARS'` : Representa uma variável.
- **t_number**: `r'\d+'` Representa um valor de um inteiro.
- **t_TRUE**: `r'True'` Representa o valor lógico 'verdade'.
- **t_FALSE**: `r'False'` Representa o valor lógico 'falso'.
- **t_newline**: `r'\n+'` Representa uma nova linha.
- **t_ignore**: `' \t'` Ignora espaços e tabs.
- **t_error**: Representa o tratamento de erros.

3.2 Gramática

A nossa gramática encontra-se na ficheiro limp_yacc.py, onde parsa a linguagem imperativa e cria o pseudo-código *Assembly*.

Inicialmente são feitas as declarações de blocos de instruções e toda a estrutura do flow da linguagem. Ou seja, são definidos que inicialmente são declaradas as variáveis e de seguida as funções auxiliares. Posto isto, dá-se o Start e começa a ser feitas as instruções do bloco de instruções.

- **p_Limp**: Limp : BlocoDeclaracoes BEGIN BlocoInstrucoes"
- **p_BlocoDeclaracoes**: "BlocoDeclaracoes : Declaracao"
- **p_BlocoDeclaracoes_list**: "BlocoDeclaracoes : BlocoDeclaracoes Declaracao"
- **p_BlocoInstrucoes**: "BlocoInstrucoes : Instrucao"
- **p_BlocoInstrucoes_list**: "BlocoInstrucoes : BlocoInstrucoes Instrucao"

Depois disto, fazemos as declarações de tudo o que possamos guardar, ou seja, declaração de variáveis, arrays (1 e 2 dimensões) e declaração de funções.

- **p_Declaracao_var**: "Declaracao : DeclVar ';'"

  Declaração de variáveis.

- **p_Declaracao_array**: "Declaracao : DeclArray ';'"

  Declaração de arrays unidimensionais.

- **p_Declaracao_array_bi**: "Declaracao : DeclArrayBi ';'"

  Declaração de arrays bidimensionais.

- **p_Declaracao_fun**: "Declaracao : DeclFun"

  Declaração de funções.

A seguir vamos fazer o tratamento de parsar a linguagem e guardar a informação para posteriormente se transformar em *Assembly* da máquina virtual VM.

- **p_DeclVar_atrib**: "DeclVar : INT id '=' ExpA"

  Declaração de variáveis com atributo.

```
    if p[2] in p.parser.registers:
        print('Erro: Vari vel  j   em uso')
        pass
    else:
        registo = {}
        registo['tipo'] = p[1]
        registo['gp'] = p.parser.gp
        p.parser.registers.update({p[2] : registo})
        p.parser.gp+=1
        p[0] = p[4]
```

- **p_DeclVar**: "DeclVar : INT id"

  Declaração de variáveis sem atributo.

```
    if p[2] in p.parser.registers:
        print('Erro: Vari vel  j   em uso')
        pass
    else:
        registo = {}
        registo['tipo'] = p[1]
        registo['gp'] = p.parser.gp
        p.parser.registers.update({p[2] : registo})
        p.parser.gp+=1
        p[0] = ' PUSHI 0 '

```

- **p_DeclArray**: "DeclArray : INT id '[' number ']'"

  Declaração de arrays unidimensionais.

```
    if p[2] in p.parser.registers:
        print('Erro: Vari vel  j   em uso')
        pass
    else:
        registo = {}
        registo['tipo'] = 'array'
        registo['gp'] = p.parser.gp
        registo['tamanho'] = p[4] + 1
        p.parser.registers.update({p[2] : registo})
        p.parser.gp+=int(p[4])+1

        p[0] = ' PUSHN ' + str(p[4]+1)

```

- **p_DeclArrayBi**: "DeclArrayBi : INT id '[' number ']' '[' number ']'"

  Declaração de arrays bidimensionais.

```
    if p[2] in p.parser.registers:
        print('Erro: Vari vel  j   em uso')
        pass
    else:
        registo = {}
        tamanho=int((p[4]+1) * (p[7]+1))
        registo['tipo'] = 'array-bi'
        registo['gp'] = p.parser.gp
        registo['tamanho'] = tamanho
        p.parser.registers.update({p[2] : registo})
        p.parser.gp+=tamanho
        p[0] = ' PUSHN ' + str(tamanho)

```

- **p_DeclFun**: "DeclFun : FUNCTION id '(' ')' '' BlocoInstrucoes RETURN ExpRel ';' ''"

  Declaração de funções.

```
    if p[2] in p.parser.registers:
        print('Erro: Vari vel  j   em uso')
        pass
    else:
        registo = {}
        tamanho=int((p[4]+1) * (p[7]+1))
        registo['tipo'] = 'array-bi'
        registo['gp'] = p.parser.gp
        registo['tamanho'] = tamanho
        p.parser.registers.update({p[2] : registo})
        p.parser.gp+=tamanho
        p[0] = ' PUSHN ' + str(tamanho)

```

A seguir temos funções que imprimem no standard output e fazem leituras do standard input, sendo que existem diferenças entre ler variáveis simples ou arrays.

- **p_Instrucao_dump**: "Instrucao : DUMP"
- **p_Instrucao_print**: "Instrucao : PRINT ExpA ';'"
- **p_Instrucao_printa**: "Instrucao : PRINTA id ';'"
- **p_Instrucao_read_var**: "Instrucao : READ id ';'"
- **p_Instrucao_read_array**: "Instrucao : READ id '[' number ']' ';'"

Posto isto, estão agora reunidas as condições necessárias para partirmos para as atribuições. Atribuições tratam de guardar espaço na stack com a memória necessária para tal acontecer, pois varia consoante o que queremos guardar. De seguida vamos então mostrar como serão armazenadas as nossas informações, dando uso ao recorrente em *assembly*.

- **p_AtrVar**: "AtrVar : id '=' ExpA"

```
    p[0] = p[3] + ' STOREG ' + str(p.parser.registers.get(p[1])['gp'])
```

- **p_AtrArray**: "AtrArray : id '[' number ']' '=' ExpA"

```
    p[0] = ' PUSHGP PUSHI ' + str(p.parser.registers.get(p[1])['gp']) + ' PADD
    PUSHI ' + str(p[3]) + p[6] + ' STOREN '
```

- **p_AtrArrayBi**: "AtrArrayBi : id '[' number ']' '[' number ']' '=' ExpA"

```
    p[0] = ' PUSHGP PUSHI ' + str(p.parser.registers.get(p[1])['gp']) + ' PADD
    PUSHI ' + str((p[3]+1) * (p[6]+1)) + p[9] + ' STOREN '
```

- **p_AtrFun**: "AtrFun : id '=' id '(' ')'"

```
    p[0] = ' PUSHA ' + p[3] + ' CALL PUSHG ' + str(p.parser.registers.get(p[3])
    ['gp-var']) + ' STOREG '+str(p.parser.registers.get(p[1])['gp'])
```

3.3 Condicionais

3.3.1 If

Esta é a parte mais lógica da linguagem imperativa, pois é aqui que são declaradas as expressões lógicas.

A primeira condição é o if, sem que haja necessidade de um else, fazendo os jump na memória da stack mediante a condição ser verdadeira.

```
def p_Condicional_if(p):
    "Condicional : IF '(' Condicao ')' '{' BlocoInstrucoes '}'"
    print('condicional if')
    label='fimif'+str(p.parser.contaIfs)
    p.parser.contaIfs += 1
    p[0] = p[3] + ' JZ ' + label + p[6] + label + ':'
```

3.3.2 If-Else

Este condicional é bastante parecido com o If, bastando apenas acrescentar o bloco de instruções a executar no caso de a condição de teste dar como resultado um valor lógico de falsidade.

```
def p_Condicional_if_else(p):
    "Condicional : IF '(' Condicao ')' '{' BlocoInstrucoes '}' ELSE '{'
    BlocoInstrucoes '}'"
    label_else='else'+str(p.parser.contaIfs)
    label_fim='fimif'+str(p.parser.contaIfs)
    p.parser.contaIfs += 1
    p[0]= p[3] + ' JZ ' + label_else + p[6] + ' JUMP ' + label_fim + ' ' +
    label_else + ':' + p[10] + label_fim + ':'
```

3.3.3 Repeat-Until

O número do nosso grupo módulo 3 representa o a condição cíclica do repeat-until. Este ciclo é o semelhante logicamente ao do-while das linguagens mais conhecidas.

Este ciclo permite executar um bloco de instruções no mínimo uma vez antes de a condição dar falsa. Para isso, é metida uma label no inicio do ciclo para que no fim, caso a condição se verifique verdadeira, a máquina virtual saiba para onde saltar para saber as instruções de início do ciclo.

```
def p_Condicional_repeat_until(p):
    "Condicional : REPEAT '{' BlocoInstrucoes '}' UNTIL '(' Condicao ')'"
    label_ciclo = 'ciclo'+ str(p.parser.contaCiclos)
    p.parser.contaCiclos += 1
    p[0] = label_ciclo + ':' + p[3]  + p[7] + ' JZ ' + label_ciclo
```

3.3.4 While-Do

Este ciclo é bastante parecido ao ciclo repeat-until, com a diferença de testar a condição no início do ciclo ao invés de no fim. Para isso, inicialmente guarda a label do início de ciclo e depois executa a verificação da condição, caso seja verdadeira, percorre o bloco de instruções e no fim volta para a label marcada anteriormente, caso contrário, efetua um JUMP para o fim do ciclo, marcado também com uma label.

```
def p_Condicional_while_do(p):
    "Condicional : WHILE '(' Condicao ')' DO '{' BlocoInstrucoes '}'"
    label_inicio_ciclo = ' iniciociclo'+ str(p.parser.contaCiclos)
    label_fim_ciclo = ' fimciclo'+ str(p.parser.contaCiclos)
    p.parser.contaCiclos += 1
    p[0] = label_inicio_ciclo + ':' + p[3] + ' JZ ' + label_fim_ciclo + p[7] + '
    JUMP ' + label_inicio_ciclo + label_fim_ciclo + ':'
```

3.4 Expressões

3.4.1 Expressões Lógicas

De modo a testar se condições de verificação lógica anteriormente apresentação tivemos de criar as expressões de condição, lou seja, os vários 'e', 'ou', 'not'.

```
def p_Condicao(p):
    "Condicao : ExpLogOr"
    p[0]=p[1]

def p_ExpLogOr(p):
    "ExpLogOr : ExpLogAnd"
    p[0]=p[1]

```

```
def p_ExpLogOr_or(p):
    "ExpLogOr : ExpLogOr OR ExpLogAnd"
    p[0] = p[1] + p[3] + ' ADD ' + p[1] + p[3] + ' MUL SUB '

def p_ExpLogAnd(p):
    "ExpLogAnd : ExpLogNot"
    p[0]=p[1]

def p_ExpLogAnd_and(p):
    "ExpLogAnd : ExpLogAnd AND ExpLogOr"
    p[0] = p[1] + p[3] + ' MUL '

def p_ExpLogNot(p):
    "ExpLogNot : ExpEq"
    p[0] = p[1]

def p_ExpLogNot_not(p):
    "ExpLogNot : NOT Condicao"
    p[0] = p[2] + ' NOT '
```

3.4.2 Expressões de Comparação

Estas expressões são responsáveis por fazer as várias comparações entre as diferentes expressões. Aqui encontram-se definidos as expressões de igualdade, e consequentemente a não igualdade, de superioridade e de inferioridade.

```
def p_ExpEq(p):
    "ExpEq : ExpRel"
    p[0] = p[1]

def p_ExpEq_eq(p):
    "ExpEq : ExpEq EQ ExpRel"
    p[0] = p[1] + p[3] + ' EQUAL '

def p_ExpEq_ne(p):
    "ExpEq : ExpEq NE ExpRel"
    p[0] = p[1] + p[3] + ' EQUAL NOT '

def p_ExpRel(p):
    "ExpRel : ExpA"
    p[0] = p[1]

def p_ExpRel_g(p):
    "ExpRel : ExpRel '>' ExpA"
    p[0] = p[1] + p[3] + ' SUP '

def p_ExpRel_l(p):
    "ExpRel : ExpRel '<' ExpA"
    p[0] = p[1] + p[3] + ' INF '

def p_ExpRel_ge(p):
    "ExpRel : ExpRel GE ExpA"
    p[0] = p[1] + p[3] + ' SUPEQ '

def p_ExpRel_le(p):
    "ExpRel : ExpRel LE ExpA"
    p[0] = p[1] + p[3] + ' INFEQ '
```

3.5 Termos

Por fim, temos os termos em si.
Nesta parte, falta então apenas as operações aritméticas mais básicas e a classificação de factors.
Sendo assim, é declarada as operações de adição, subtração, multiplicação e divisão.
Estas produções recebem os termos e parsam para a respetiva instrução em *Assembly*.
Os factores fazem a adição em memória dos nossos dados.

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

def p_Factor_id(p):
    "Factor : id"
    p[0] = ' PUSHG ' + str(p.parser.registers.get(p[1])['gp'])

def p_Factor_number(p):
    "Factor : number"
    p[0] = ' PUSHI ' + str(p[1])

def p_Factor_group(p):
    "Factor : '(' ExpA ')'"
    p[0] = p[2]

def p_Factor_array(p):
    "Factor : id '[' number ']'"
    p[0] = ' PUSHGP PUSHI ' + str(p.parser.registers.get(p[1])['gp']) + ' PADD PUSHI
     ' + str(p[3]) + ' LOADN '

def p_Factor_array_bi(p):
    "Factor : id '[' number ']' '[' number ']'"
    p[0] = ' PUSHGP PUSHI ' + str(p.parser.registers.get(p[1])['gp']) + ' PADD PUSHI
     ' + str((p[3]+1) * (p[6]+1)) + ' LOADN '

def p_Factor_true(p):
    "Factor : TRUE"
    p[0] = ' PUSHI 1 '

def p_Factor_false(p):
    "Factor : FALSE"
    p[0] = ' PUSHI 0 '
```

4. Conclusão

Com a realização deste trabalho prático da unidade curricular de *Processamento de Linguagens* conseguimos aumentar a nossa base de conhecimento em gramáticas e compiladores. Foi um trabalho deveras enriquecedor, pois fizemos uso de todo o conhecimento adquirido ao longo do semestre, assim como ao longo dos vários anos do curso, pois pusemos em prática vários conhecimentos diferentes.

Tivemos de atravessar várias barreiras e vários obstáculos para chegar ao resultado final deste projeto, mas é mesmo isso que nos faz avançar e motivar para conseguir sempre um resultado ainda melhor.

Podemos concluir que fizemos todos os requisitos propostos pelo enunciado e isso só foi possível através do conhecimento que a unidade curricular nos transmitiu!

5. Código

5.1 limp_lex.py

```
1
2  # Ricardo Leal A75411
3  # Renato Cruzinha A75310
4
5  import ply.lex as lex
6  import ply.lex as lex
7  import sys
8
9  # reserved
10 reserved = {
11     'int' : 'INT',
12     'print' : 'PRINT',
13     'printa' : 'PRINTA',
14     'read' : 'READ',
15     'dump' : 'DUMP',
16     'if'   : 'IF',
17     'else' : 'ELSE',
18     'repeat' : 'REPEAT',
19     'until'  :  'UNTIL',
20     'function' : 'FUNCTION',
21     'return'   : 'RETURN',
22     'while'    : 'WHILE',
23     'do'       : 'DO',
24 }
25
26 # tokens
27 tokens = [
28     'number', 'id', 'BEGIN', 'EQ', 'NE', 'GE', 'LE', 'AND', 'NOT', 'OR', 'TRUE', '
       FALSE',
29  ] + list(reserved.values())
30
31 # literals
32 literals = ['+', '-', '*', '/', '(', ')', '{', '}', '?', ';', '=', '[', ']', '>', '<
       ']
33 t_BEGIN = r'BEGIN'
34
35 # regular expression
36 def t_EQ(t):
37     r'=='
38     return t
39
40 def t_NE(t):
41     r'!='
42     return t
43
44 def t_GE(t):
45     r'>='
46     return t
```

```
47
48  def t_LE(t):
49      r'<='
50      return t
51
52  def t_AND(t):
53      r'&&'
54      return t
55
56  def t_OR(t):
57      r'\|\|'
58      return t
59
60  def t_NOT(t):
61      r'!'
62      return t
63
64  def t_id(t):
65      r'[a-z]+'
66      t.type = reserved.get(t.value,'id')
67      return t
68
69  def t_VARS(t):
70      r'VARS'
71      return t
72
73  def t_number(t):
74      r'\d+'
75      t.value = int(t.value)
76      t.lexer.num_count += 1
77      return t
78
79  def t_TRUE(t):
80      r'True'
81      return t
82
83  def t_FALSE(t):
84      r'False'
85      return t
86  #----------------------------------------
87
88  # track line numbers
89  def t_newline(t):
90      r'\n+'
91      t.lexer.lineno += len(t.value)
92      t.lexer.skip(1)
93
94  # string containing ignored characters (spaces and tabs)
95  t_ignore  = ' \t'
96
97  # error handling rule
98  def t_error(t):
99      print("Illegal character '%s'" % t.value[0])
100     t.lexer.skip(1)
101
102  # EOF handling rule
103 #def t_eof(t):
104   # Get more input (Example)
105   #more = input('... ')
106   #if more:
107   #    self.lexer.input(more)
108   #    return self.lexer.token()
109   #return None
```

```
110
111 #---------------------------------------
112 # Build the lexer
113 lexer = lex.lex()
114 lexer.num_count = 0
```

5.2 limp_yacc.py

```
1
2  # Ricardo Leal A75411
3  # Renato Cruzinha A75310
4
5  import sys
6  import ply.yacc as yacc
7  from limp_lex import tokens
8
9
10 def p_Limp(p):
11     "Limp : BlocoDeclaracoes BEGIN BlocoInstrucoes"
12     print(p[1], 'START', p[3], 'STOP')
13
14 def p_BlocoDeclaracoes(p):
15     "BlocoDeclaracoes : Declaracao"
16     p[0] = p[1]
17
18 def p_BlocoDeclaracoes_list(p):
19     "BlocoDeclaracoes : BlocoDeclaracoes Declaracao"
20     p[0] = p[1] + ' ' + p[2]
21
22 def p_BlocoInstrucoes(p):
23     "BlocoInstrucoes : Instrucao"
24     p[0] = p[1]
25
26 def p_BlocoInstrucoes_list(p):
27     "BlocoInstrucoes : BlocoInstrucoes Instrucao"
28     p[0] = p[1] + ' ' + p[2]
29
30 def p_Declaracao_var(p):
31     "Declaracao : DeclVar ';'"
32     p[0] = p[1]
33
34 def p_Declaracao_array(p):
35     "Declaracao : DeclArray ';'"
36     p[0] = p[1]
37
38 def p_Declaracao_array_bi(p):
39     "Declaracao : DeclArrayBi ';'"
40     p[0] = p[1]
41
42 def p_Declaracao_fun(p):
43     "Declaracao : DeclFun"
44     p[0]=p[1]
45
46 def p_DeclVar_atrib(p):
47     "DeclVar : INT id '=' ExpA"
48     if p[2] in p.parser.registers:
49         print('Erro: Vari vel  j   em uso')
50         pass
51     else:
52         registo = {}
53         registo['tipo'] = p[1]
```

```
54          registo['gp'] = p.parser.gp
55          p.parser.registers.update({p[2] : registo})
56          p.parser.gp+=1
57
58          p[0] = p[4]
59
60  def p_DeclVar(p):
61      "DeclVar : INT id"
62      if p[2] in p.parser.registers:
63          print('Erro: Vari vel  j   em uso')
64          pass
65      else:
66          registo = {}
67          registo['tipo'] = p[1]
68          registo['gp'] = p.parser.gp
69          p.parser.registers.update({p[2] : registo})
70          p.parser.gp+=1
71
72          p[0] = ' PUSHI 0 '
73
74  def p_DeclArray(p):
75      "DeclArray : INT id '[' number ']'"
76      if p[2] in p.parser.registers:
77          print('Erro: Vari vel  j   em uso')
78          pass
79      else:
80          registo = {}
81          registo['tipo'] = 'array'
82          registo['gp'] = p.parser.gp
83          registo['tamanho'] = p[4] + 1
84          p.parser.registers.update({p[2] : registo})
85          p.parser.gp+=int(p[4])+1
86
87          p[0] = ' PUSHN ' + str(p[4]+1)
88
89  def p_DeclArrayBi(p):
90      "DeclArrayBi : INT id '[' number ']' '[' number ']'"
91      if p[2] in p.parser.registers:
92          print('Erro: Vari vel  j   em uso')
93          pass
94      else:
95          registo = {}
96          tamanho=int((p[4]+1) * (p[7]+1))
97          registo['tipo'] = 'array-bi'
98          registo['gp'] = p.parser.gp
99          registo['tamanho'] = tamanho
100         p.parser.registers.update({p[2] : registo})
101         p.parser.gp+=tamanho
102
103         p[0] = ' PUSHN ' + str(tamanho)
104
105 def p_DeclFun(p):
106     "DeclFun : FUNCTION  id '(' ')' '{' BlocoInstrucoes RETURN ExpRel ';' '}'"
107     if p[2] in p.parser.registers:
108         print('Erro: Funcao  j   em uso')
109         pass
110     else:
111         registo = {}
112         registo['tipo'] = 'fun'
113         registo['gp'] = p.parser.gp + 1
114         registo['gp-var'] = p.parser.gp
115         p.parser.registers.update({p[2] : registo})
116         p.parser.gp+=1
```

```
        label_fun=p[2] + ':'
        function_end=p[2] + 'end '
    p[0] = ' PUSHI 0 JUMP '+ function_end + label_fun + p[6] + p[8] + ' STOREG ' +
    str(p.parser.registers.get(p[2])['gp-var']) + ' RETURN ' + function_end + ':'

def p_Instrucao_dump(p):
    "Instrucao : DUMP"
    print("Registers: ", p.parser.registers)

def p_Instrucao_print(p):
    "Instrucao : PRINT ExpA ';'"
    p[0] = p[2] + ' WRITEI '

def p_Instrucao_printa(p):
    "Instrucao : PRINTA id ';'"
    if p[2] in p.parser.registers:
        label_array_begin = ' ' + p[2] + 'begin'
        label_array_end = ' ' + p[2] + 'end'
        tamanho = str(p.parser.registers.get(p[2])['tamanho'])
        pos_i = str(p.parser.varsControlo.get('varprinta'))
        p.parser.gp+=1
    else:
        print('Erro: Array n o declarado/Argumento tem de ser um array')
        pass
    p[0] = ' PUSHI 0 STOREG ' + pos_i + label_array_begin + ': ' + ' PUSHG ' + pos_i
     + ' PUSHI ' + tamanho + ' INF JZ ' + label_array_end + ' PUSHGP PUSHI ' + str(p
    .parser.registers.get(p[2])['gp']) + ' PADD PUSHG ' + pos_i + ' LOADN WRITEI
    PUSHG ' + pos_i + ' PUSHI 1 ADD STOREG ' + pos_i + ' JUMP ' + label_array_begin
    + ' ' + label_array_end + ':'

def p_Instrucao_read_var(p):
    "Instrucao : READ id ';'"
    p[0] = 'READ ATOI STOREG ' + str(p.parser.registers.get(p[2])['gp'])

def p_Instrucao_read_array(p):
    "Instrucao : READ id '[' number ']' ';'"
    p[0] = 'PUSHGP PUSHI ' + str(p.parser.registers.get(p[2])['gp']) + ' PADD READ
    ATOI STOREN'

def p_Instrucao_atrib(p):
    "Instrucao : Atribuicao ';'"
    p[0] = p[1]

def p_Atribuicao_var(p):
    "Atribuicao : AtrVar"
    p[0]=p[1]

def p_Atribuicao_array(p):
    "Atribuicao : AtrArray"
    p[0]=p[1]

def p_Atribuicao_array_bi(p):
    "Atribuicao : AtrArrayBi"
    p[0]=p[1]

def p_Atribuicao_fun(p):
    "Atribuicao : AtrFun"
    p[0]=p[1]

def p_AtrVar(p):
    "AtrVar : id '=' ExpA"
    p[0] = p[3] + ' STOREG ' + str(p.parser.registers.get(p[1])['gp'])

```

```
174 def p_AtrArray(p):
175     "AtrArray : id '[' number ']' '=' ExpA"
176     p[0] = ' PUSHGP PUSHI ' + str(p.parser.registers.get(p[1])['gp']) + ' PADD PUSHI
        ' + str(p[3]) + p[6] + ' STOREN '
177
178 def p_AtrArrayBi(p):
179     "AtrArrayBi : id '[' number ']' '[' number ']' '=' ExpA"
180     p[0] = ' PUSHGP PUSHI ' + str(p.parser.registers.get(p[1])['gp']) + ' PADD PUSHI
        ' + str((p[3]+1) * (p[6]+1)) + p[9] + ' STOREN '
181
182 def p_AtrFun(p):
183     "AtrFun : id '='  id '(' ')'"
184     p[0] = ' PUSHA ' + p[3] + ' CALL PUSHG ' + str(p.parser.registers.get(p[3])['gp-
    var']) + ' STOREG '+str(p.parser.registers.get(p[1])['gp'])
185
186 def p_Instrucao_condicionais(p):
187     "Instrucao : Condicional"
188     p[0]=p[1]
189
190 def p_Condicional_if(p):
191     "Condicional : IF '(' Condicao ')' '{' BlocoInstrucoes '}'"
192     label='fimif'+str(p.parser.contaIfs)
193     p.parser.contaIfs += 1
194     p[0] = p[3] + ' JZ ' + label + p[6] + label + ':'
195
196 def p_Condicional_if_else(p):
197     "Condicional : IF '(' Condicao ')' '{' BlocoInstrucoes '}' ELSE '{'
    BlocoInstrucoes '}'"
198     label_else='else'+str(p.parser.contaIfs)
199     label_fim='fimif'+str(p.parser.contaIfs)
200     p.parser.contaIfs += 1
201     p[0]= p[3] + ' JZ ' + label_else + p[6] + ' JUMP ' + label_fim + ' ' +
    label_else + ':' + p[10] + label_fim + ':'
202
203 def p_Condicional_repeat_until(p):
204     "Condicional : REPEAT '{' BlocoInstrucoes '}' UNTIL '(' Condicao ')'"
205     label_ciclo = 'ciclo'+ str(p.parser.contaCiclos)
206     p.parser.contaCiclos += 1
207     p[0] = label_ciclo + ':' + p[3]  + p[7] + ' JZ ' + label_ciclo
208
209 def p_Condicional_while_do(p):
210     "Condicional : WHILE '(' Condicao ')' DO '{' BlocoInstrucoes '}'"
211     label_inicio_ciclo = ' iniciociclo'+ str(p.parser.contaCiclos)
212     label_fim_ciclo = ' fimciclo'+ str(p.parser.contaCiclos)
213     p.parser.contaCiclos += 1
214     p[0] = label_inicio_ciclo + ':' + p[3] + ' JZ ' + label_fim_ciclo + p[7] + '
    JUMP ' + label_inicio_ciclo + label_fim_ciclo + ':'
215
216 def p_Condicao(p):
217     "Condicao : ExpLogOr"
218     p[0]=p[1]
219
220 def p_ExpLogOr(p):
221     "ExpLogOr : ExpLogAnd"
222     p[0]=p[1]
223
224 def p_ExpLogOr_or(p):
225     "ExpLogOr : ExpLogOr OR ExpLogAnd"
226     p[0] = p[1] + p[3] + ' ADD ' + p[1] + p[3] + ' MUL SUB '
227
228 def p_ExpLogAnd(p):
229     "ExpLogAnd : ExpLogNot"
230     p[0]=p[1]
```

```
231
232 def p_ExpLogAnd_and(p):
233     "ExpLogAnd : ExpLogAnd AND ExpLogOr"
234     p[0] = p[1] + p[3] + ' MUL '
235
236 def p_ExpLogNot(p):
237     "ExpLogNot : ExpEq"
238     p[0] = p[1]
239
240 def p_ExpLogNot_not(p):
241     "ExpLogNot : NOT Condicao"
242     p[0] = p[2] + ' NOT '
243
244 def p_ExpEq(p):
245     "ExpEq : ExpRel"
246     p[0] = p[1]
247
248 def p_ExpEq_eq(p):
249     "ExpEq : ExpEq EQ ExpRel"
250     p[0] = p[1] + p[3] + ' EQUAL '
251
252 def p_ExpEq_ne(p):
253     "ExpEq : ExpEq NE ExpRel"
254     p[0] = p[1] + p[3] + ' EQUAL NOT '
255
256 def p_ExpRel(p):
257     "ExpRel : ExpA"
258     p[0] = p[1]
259
260 def p_ExpRel_g(p):
261     "ExpRel : ExpRel '>' ExpA"
262     p[0] = p[1] + p[3] + ' SUP '
263
264 def p_ExpRel_l(p):
265     "ExpRel : ExpRel '<' ExpA"
266     p[0] = p[1] + p[3] + ' INF '
267
268 def p_ExpRel_ge(p):
269     "ExpRel : ExpRel GE ExpA"
270     p[0] = p[1] + p[3] + ' SUPEQ '
271
272 def p_ExpRel_le(p):
273     "ExpRel : ExpRel LE ExpA"
274     p[0] = p[1] + p[3] + ' INFEQ '
275
276 def p_Exp_add(p):
277     "ExpA : ExpA '+' Term"
278     p[0] = p[1] + p[3] + ' ADD '
279
280 def p_Exp_sub(p):
281     "ExpA : ExpA '-' Term"
282     p[0] = p[1] + p[3] + ' SUB '
283
284 def p_Exp_term(p):
285     "ExpA : Term"
286     p[0] = p[1]
287
288 def p_Term_mul(p):
289     "Term : Term '*' Factor"
290     p[0] = p[1] + p[3] + ' MUL '
291
292 def p_Term_div(p):
293     "Term : Term '/' Factor"
```

```
294     p[0] = p[1] + p[3] + ' DIV '
295
296 def p_Term_factor(p):
297     "Term : Factor"
298     p[0] = p[1]
299
300 def p_Factor_id(p):
301     "Factor : id"
302     p[0] = ' PUSHG ' + str(p.parser.registers.get(p[1])['gp'])
303
304 def p_Factor_number(p):
305     "Factor : number"
306     p[0] = ' PUSHI ' + str(p[1])
307
308 def p_Factor_group(p):
309     "Factor : '(' ExpA ')'"
310     p[0] = p[2]
311
312 def p_Factor_array(p):
313     "Factor : id '[' number ']'"
314     p[0] = ' PUSHGP PUSHI ' + str(p.parser.registers.get(p[1])['gp']) + ' PADD PUSHI
        ' + str(p[3]) + ' LOADN '
315
316 def p_Factor_array_bi(p):
317     "Factor : id '[' number ']' '[' number ']'"
318     p[0] = ' PUSHGP PUSHI ' + str(p.parser.registers.get(p[1])['gp']) + ' PADD PUSHI
        ' + str((p[3]+1) * (p[6]+1)) + ' LOADN '
319
320 def p_Factor_true(p):
321     "Factor : TRUE"
322     p[0] = ' PUSHI 1 '
323
324 def p_Factor_false(p):
325     "Factor : FALSE"
326     p[0] = ' PUSHI 0 '
327
328 #---------------------------------------
329 def p_error(p):
330     print('Syntax error: ', p)
331     parser.success = False
332
333 #---------------------------------------
334 #inicio do Parser
335 parser = yacc.yacc()
336
337 parser.success = True
338 parser.registers = {}   #tabela de identificadores
339 parser.gp = 0           # offset em relacao ao global pointer(gp)
340 parser.contaIfs = 0
341 parser.contaCiclos = 0
342 parser.varsControlo = {'varprinta' : parser.gp}
343 parser.gp+=1
344 print('PUSHI 0')
345 for line in sys.stdin:
346     parser.parse(line)
```