Processamento de Linguagens (3º ano de Curso)

Trabalho Prático II

Relatório de Desenvolvimento

Francisco Paulino
A91666

Rafael Carvalho
A91642

16 de janeiro de 2022

Resumo

Este relatório diz respeito ao Trabalho Prático nº 2 da Unidade Curricular de Processamento de Linguanges, no qual foi proposto, o desenvolvimento de um compilador através do uso de módulos de gramáticas tradutoras e analisadores léxicos do Python.

Conteúdo

Capítulo 1

Introdução

Supervisor: Pedro Rangel Henriques

1.1 Enquandramento e contexto

Área: Processamento de Linguagens

Na Unidade Curricular de Processamento de Linguagens, foi sugerido , como segundo trabalho prático , a criação de uma linguangem imperativa simples , bem como a de um compilador capaz de interpretar essa mesma linguagem.

Através desse compilador será possivel gerar codígo máquina que será prosteriomente corrido numa máquina de stack virtual.

Estrutura do Relatório

Nos capítulos seguintes serão abordados os tópicos relevantes para o desenvolvimento do programa.

Começaremos por apresentar no Capítulo 2 uma análise ao problema proposto , bem como os dados fornecidos e os objetivos pretendidos com o mesmo.

De seguida teremos o Capítulo 3, no qual será apresentada a abordagem feita pelo grupo de trabalho , nomeadamente o desenvolvimento da linguagem e da gramática , bem como o analisador léxico e sintático.

No Capítulo 4 iremos apresentar os testes realizados para verificação do programa e os resultados obtidos.

Para finalizar, no Capítulo 5 iremos apresentar um ponto de vista geral por parte do grupo de trabalho.

Encontrear-se-á também, no final do relatório, o Apêndice , no qual consta o código produzido pelo grupo em linguagem python , que foi explicado e utilizado para responder ao proposto do trabalho prático.

Capítulo 2

Análise e Especificação

2.1 Descrição do Problema

Tal como referido anteriormente, o problema sugerido para este trabalho, passa pelo desenvolvimento de uma linguagem de programação bem como um compilador, para essa linguagem, com base na gramática indepentende de contexto (GIC) criada e com recurso aos módulos Yacc/Lex do Python. Com recurso à gramática tradutora será possivel gerar código assembly que poderá ser lido por uma máquina de stack virtual.

2.2 Especificação dos Requisitos

Com vista a responder ao problema sugerido foram definidos alguns requisitos. No que diz respeito à linguagem de programação desenvolvida, bem como a gramática, é importante que seja possivel:

- Declarar variaveis 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;
- 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);
- Efetuar a análise léxica e sintática;
- Definir regras de tradução para assembly.

2.2.1 Objetivos pretendidos

Com este trabalho pertende-se não só desenvolver um compilador através da geração de código para uma Maquina Virtual , como também:

- Aumentar a experiência em engenharia de linguagens e em programação generativa,reforçando a capacidade de escrever gramáticas, quer independentes de contexto (GIC), quer tradutoras (GT);
- Desenvolver processadores de linguagens segundo o metodo da tradução dirigida pela sintaxe,a partir de uma gramática tradutora;
- Utilizar geradores de compiladores baseados em gramáticas tradutoras, concretamente o Yacc,versão PLY do Python, completado pelo gerador de analisadores lexicos Lex, tambám versão PLY do Python;
- Criar o hábito de escrever a documentação (os relatórios dos trabalhos práticos e projectos) em LATEX

Capítulo 3

Concepção da Resolução

3.1 Gramática Independente de Contexto

Inicialmente foi desenvolvida uma gramática regular independente do contexto(gic) capaz de intrepertar a linguagem definida.

```
Programa => Declaracoes Funcoes
Funcoes => MAIN Comandos END
Declaracoes => Declaracao  Declaracoes
            | €
Declaracao => INT ID
            | INT ID '=' NUM
            | INT ID '[' NUM ']'
            |INT ID '[' NUM ']' '[' NUM ']'
            | INT ID '[' NUM ']' '=' '[' Array ']'
            | INT ID '[' NUM ']' '[' NUM ']' '=' '[' Arrays ']'
Arrays => '[' Array ']' ',' Arrays
        | €
Array => NUM ',' Array
        | €
Comandos => Comando Comandos
        | €
Comando => Comando_If
        | Comando_If_Else
        | Comando_Repeat
        | Comando_While
        | Comando_Prints
        | Comando_Read
        | Id '=' Exp
        | IdArray '=' Exp
        | IdDoubleArray '=' Exp
Comando_If => IF Condicao '{' Comandos '}'
Comando_If_Else => IF Condicao '{' Comandos '}' ELSE '{' Comandos    '}'
Comando_Repeat => REPEATE '{' Comandos '}' UNTIL Condicao
Comando_While => WHILE Condicao DO '{' Comandos '}'
```

```
Comando_Prints => PRINT Comando_Print Prints
Prints => '+' Comando_Print Prints
        | €
Comando_Print => Id
                | IdArray
                | IdDoubleArray
                |  STR
Comando_Read => READ Id
    |READ IdArray
    | READ IdDoubleArray
Condicao => '(' Cond ')'
        | NOT '(' Cond ')'
        | Condicao AND Condicao
        | Condicao OR Condicao
Cond => Exp EQUALS Exp
    | Exp GREATER Exp
    | Exp LESSER Exp
    | Exp GREATERQ Exp
    | Exp LESSERQ Exp
    | Condicao
Exp => Exp '+' Term
    |Exp '-' Term
    | Term
Term => Term '/' Factor
    | Term '*' Factor
    | Term '%' Factor
    | Factor
Factor => NUM
        | Id
        | IdArray
        | IdDoubleArray
Id => ID
IdArray => ID '[' Factor ']'
IdDoubleArray => ID '[' Factor ']' '[' Factor ']'
```

A gramática começa com o simbolo não terminal *Programa*, que pode ser dividido em dois simbolos não terminais , denomidados por *Declarações* e *Funções*. Desta forma é possivel dividir o programa de tal forma de que, no ínicio do programa sejam declaradas as variáveis e de seguida venha o restante do conteúdo necessário. No que diz respeito as declarações de variáveis , é necessario estas serem sempre declaradas, bem como , no momento da sua declaração, começarem por ser identificados com o tipo de variável, que neste trabalho serão apenas do tipo int seguido de um nome .
Para além disso, é importante referir que:

- Possibilidade de declarar variáveis inteiras, array de inteiros e um array bidimensionais de inteiros;
- Um array bidimensional , quando declarado , é necessario colocar seguido do nome , dois pares de parênteses retos , nos quais consta as dimensões do array;
- Possibilidade de fazer atribuições às variáveis, como por exemplo, atribuição de um valor inteiro a

uma variável , através do uso do simbolo "=", após o nome da variável , sendo colocado à frente do simbolo o valor pretendido;

- Possibilidade de definir os valores de um array , colocando o simbolo "=", apos o nome atribuido , sendo colocado à frente do simbolo , entre parênteses retos , os diversos valores pretendidos, separados por vírgulas;
- Possibilidade de definir os valores de um array bidimensional , colocando o símbolo "=", após o nome atribuido , sendo colocado à frente do símbolo , entre parênteses retos , os diversos valores pretendidos, separados por vírgulas.A linha é formada por parênteses retos , e entre esses , os valores númericos são separados por vírgulas.

No que diz respeito as Funções, posteriormente à declaração das variáveis é necessário escrever "MAIN"antes de iniciar a escrita de comandos que se pretendam executar, bem como no final do código produzido, ser escrito "END", de forma a marcar o fim do mesmo.
É importante realçar que , no que diz respeitos aos Comandos:

- Possibilidade de não ser definido nenhum comando;
- Possibilidade de somar, subtrair , multiplicar , dividir , fazer o mod de uma variável;
- Possibilidade de somar, subtrair um valor de um array normal ou bidimensional;
- Possibilidade de atribuir a uma variável , uma posição do array ou uma posição do array bidimensional;
- Possibilidade de mostrar no output o valor de uma variável, string ou do valor que se encontra numa posição do array normal ou bidimensional;
- Possibilidade de atribuir uma variavel, o resultado de uma expressão algébrica , através do uso do simbolo "="seguido da expressão pretendida;
- Possibilidade de realizar o cíclo Repeat Until. Neste caso , é necessário a seguir ao Repeat , colocar entre chavetas , os comandos a exectuar , sendo depois necessário complementar com um Until seguido da condiçao pretendida.
- Possibilidade de realizar o cíclo While. Neste caso , é necessário a seguir ao while , colocar a condição pretendida , sendo depois necessário complementar com Do , seguido , entre chavetas ,os comandos a exectuar .
- Possibilidade de utilização das estruturas de decisão : If e If-Else. Como nos outros casos a seguir ao If ou ao If-else , é colocada a condição ou mais do que uma condição (atraves da utilização dos operadores lógicos and e or). As condições podem basear-se em comparações de igualdade , menor , maior , menor ou igual , maior ou igual , diferente entre as expressões algébricas , variáveis ou valores de uma posição do array ou matriz.
- Possibilidade de realizar Read e Print;

3.2 Analisador Léxico

No caso do analisador léxico, começamos por definir os tokens , bem como os *iterals*, que são constituidos pelos parênteses, chavetas e outros simbolos relativos a operações logicas e algébricas. Foram também definidas algumas plavaras reservadas de forma a evitar conflitos de intrepretação de texto, uma vez que o

analisador Léxico , ao ler um conjunto de caracteres , procura por esse conjunto num dicionário de palavras , atribuindo ao token lido o tipo correspondente à chave encontrada, caso contrário o token corresponderá a uma variável. Caso o caracter lido não seja detetado pelo analisador é descartado.

```
import ply.lex as lex
import re
import sys

tokens = ['INT','STR','NUM','ID',
'MAIN','END',
'PRINT','READ',
'IF','ELSE',
'REPEATE','UNTIL','WHILE','DO',
'AND','OR','NOT',
'EQUALS','GREATER','LESSER','GREATERQ','LESSERQ'
]
literals = ['%','*', '+' ,'/', '-', '=', '(', ')', '.', '<', '>', ',', '!', '{', '}', '[', ']']

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

def t_STR(t):
    r'"[^"]+"'
    return t

def t_NUM(t):
    r'-?\d+'
    #t.value = int(t.value)
    return t

def t_MAIN(t):
    r'MAIN'
    return t

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

def t_PRINT(t):
    r'Print'
    return t

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

def t_IF(t):
```

```
    r'If'
    return t

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

def t_REPEATE(t):
    r'Repeat'
    return t

def t_UNTIL(t):
    r'Until'
    return t

def t_WHILE(t):
    r'While'
    return t

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

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

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

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

def t_GREATERQ(t):
    r'>='
    return t

def t_LESSERQ(t):
    r'<='
    return t

def t_EQUALS(t):
    r'=='
    return t

def t_GREATER(t):
```

```
    r'>'
    return t

def t_LESSER(t):
    r'<'
    return t

def t_ID(t):
    r'\w+'
    return t

t_ignore = " \t\n"
def t_error(t):
    print('Illegal character: ', t.value[0])
    t.lexer.skip(1)

lexer = lex.lex()
```

3.3 Gramática Tradutora + Compilador Yacc

Com a utilização do Yacc versão PLY da linguagem python , foi possivel criar a gramática tradutora que seria utilizada para desenvolver o processador de linguagens. Esta gramática foi baseada na gramática independente do contexto apresentada anteriormente e utilizada com base para o trabalho prático. Durante a sua criação e desenvolvimento , tivemos em atenção a criação de um paser bottom-up, de forma a ir de encontro à condição LR().
Após definida esta gramática inicalmente no ficheiro Yacc, iniciou-se o desenvolvimento de soluções, de forma a completar as definições atribuidas a todos os símbolos não terminais da gramática que permitissem criar um compilador capaz gerar código máquina , que será intrepertado através de uma Máquina Virtual.
No que diz respeito ao **Programa**:

```
def p_Programa(p):
    "Programa : Declaracoes Funcoes"
    p[0]=p[1]+p[2]
```

No que diz respeito a **Declaração**:

```
def p_Declaracoes(p):
    "Declaracoes : Declaracao  Declaracoes "
    p[0] = str(p[1]) + str(p[2])

def p_Declaracoes_vazia(p):
    "Declaracoes : "
    p[0] = ""

def p_Declaracao_Int(p):
    "Declaracao : INT ID"
    if p[2] in parser.variaveis:
        parser.exito = False
```

```
        p[0]= "ERROR"
        print("Variavel " + p[2]+" ja foi declarada")
    else:

        parser.variaveis[p[2]]=parser.count
        parser.var_valores[p[2]]=0
        p[0]="PUSHI 0\n"
        parser.count+=1

def p_Declaracao_Int_com_numero(p):
    "Declaracao : INT ID '=' NUM"
    if p[2] in parser.variaveis:
        parser.exito = False
        p[0]= "ERROR"
        print("Variavel " + p[2]+" ja foi declarada")
    else:

        parser.variaveis[p[2]]=parser.count
        parser.var_valores[p[2]]=int(p[4])
        p[0]="PUSHI "+str(p[4])+"\n"

        parser.count+=1

def p_Declaracao_Array(p):
    "Declaracao : INT ID '[' NUM ']'"
    if p[2] in parser.variaveis:
        parser.exito = False
        p[0]= "ERROR"
        print("Variavel " + p[2]+" ja foi declarada")
    else:

        parser.variaveis[p[2]]=parser.count
        parser.var_valores[p[2]]=[0]*int(p[4])
        p[0]="PUSHN "+p[4]+"\n"
        parser.count+=int(p[4])

def p_Declaracao_DoubleArray(p):
    "Declaracao : INT ID '[' NUM ']' '[' NUM ']'"
    if p[2] in parser.variaveis:
        parser.exito = False
        p[0]= "ERROR"
        print("Variavel " + p[2]+" ja foi declarada")
    else:

        parser.variaveis[p[2]]=parser.count
        parser.var_valores[p[2]]=[[0 for i in range(int(p[7]))] for j in range(int(p[4]))]
        n=int(p[4])*int(p[7])
        p[0]="PUSHN "+str(n) +"\n"
```

```
        parser.count+=n

def p_Declaracao_Array_com_numero(p):
    "Declaracao : INT ID '[' NUM ']' '=' '[' Array ']'"
    parser.var_valores[p[2]]=parser.array
    if p[2] in parser.variaveis:
        parser.exito = False
        p[0]= "ERROR"
        print("Variavel " + p[2]+" ja foi declarada")
    elif len(parser.var_valores[p[2]])!=int(p[4]):
        parser.exito = False
        print("Variavel " + p[2]+" index out of range")

        p[0]= "ERROR"
    else:
        parser.var_valores[p[2]]=parser.array
        parser.variaveis[p[2]]=parser.count
        p[0]=p[8]
        parser.count+=int(p[4])
    parser.array=[]

def p_Declaracao_DoubleArray_com_numero(p):
    "Declaracao : INT ID '[' NUM ']' '[' NUM ']' '=' '[' Arrays ']'"
    f=0
    parser.var_valores[p[2]]=[parser.array[i:i+int(p[7])] for i in range(0, len(parser.array),
    int(p[7]))]
    parser.var_valores[p[2]].reverse()
    for a in parser.var_valores[p[2]]:
        if len(a)!=int(p[7]):
            f=1
    if p[2] in parser.variaveis:
        parser.exito = False
        p[0]= "ERROR"
        print("Variavel " + p[2]+" ja foi declarada")
    elif len(parser.var_valores[p[2]])!=int(p[4]) or f==1 :
        parser.exito = False
        print("Variavel " + p[2]+" index out of range")

        p[0]= "ERROR"
    else:

        n=int(p[4])*int(p[7])
        parser.variaveis[p[2]]=parser.count
        #parser.var_valores[p[2]]=[[0 for i in range(int(p[7]))] for j in range(int(p[4]))]

        parser.count+=n
        p[0]=p[11]
    f=0
```

```
    parser.array=[]


def p_Arrays(p):
    "Arrays : '[' Array ']' ',' Arrays"
    p[0]=p[2]+p[5]

def p_Arrays_Vazio(p):
    "Arrays :"
    p[0]=""
def p_Arrays_Array(p):
    "Arrays : '[' Array ']'"
    p[0]=p[2]

def p_Array(p):
    "Array : NUM ',' Array"
    p[0]="PUSHI "+p[1]+"\n" +p[3]
    parser.array.insert(0,int(p[1]))

def p_Array_Num(p):
    "Array : NUM "
    p[0]="PUSHI "+p[1]+"\n"
    parser.array.insert(0,int(p[1]))

def p_Array_Vazio(p):
    "Array :"
    p[0]  =""
```

É importante referir que durante o desenvolvimento foi necessário ter em atenção as declarações que podem ser feitas , no que diz respeito à verificação das variáveis declaradas. Uma vez que , se ao introduzir uma nova variável for verificado que a mesma variável já tenha sido declarada anterirormente,irá ser enviado uma mensagem de erro para o utilizador. Se isto não for verificado , a variável é adicionada normalmente no dicionário onde ficará identificado o nome da variavel como também o seu endereço na stack pointer. Para além disso , no que diz respeito as variaveis do tipo Array, ficam registados os stack pointers do endereço inicial. É importante referir que para além do dicionário com os stack pointers , temos um dicionário com os valores de cada variavel, que será utilizado para ajudar na verificação de erros de index. **Exemplo:** a=10000,b[2],b[a] => Erro. Outro apesto imporante é o facto de poder declarar uma variável array com valores incluidos dentro dela, tendo sido criados os simbolos "*Arrays*", "*Arrays_vazio*","*Arrays_Array*","*Array*","*Array_Num*"e "*Array_vazio*", permitindo assim verificar que o array seja preenchido de forma correta.

No que diz respeito as **Funções**:

```
def p_Funcoes(p):
    "Funcoes : MAIN Comandos END"
    p[0] = "START\n" + str(p[2]) + "STOP\n"
```

É possivel denotar que , tal como referido anteriormente , é necessario usar START e o STOP de maneira

a delimitar o ínicio e o fim dos comandos fornecidos.

No que diz respeito aos **Comandos**:

```
def p_Comandos(p):
    "Comandos : Comando Comandos"
    p[0] = str(p[1]) + str(p[2])

def p_Comandos_vazio(p):
    "Comandos : "
    p[0] = ""


def p_Comando_If_(p):
    "Comando_If : IF Condicao '{' Comandos '}'"
    p[0]= str(p[2])+"JZ IF"+str(parser.label) +"\n" + str(p[4])+"IF"+str(parser.label)+":\n"
    parser.label+=1

def p_Comando_If_Else_(p):
    "Comando_If_Else : IF Condicao '{' Comandos '}' ELSE '{' Comandos '}'"
    p[0] = p[2]+"JZ IF"+ str(parser.label)+"\n"+p[4]+"JUMP IFEND" +  str(parser.label) +"\nIF"
    +str(parser.label) +":\n"+p[8]+"IFEND"+str(parser.label)+":\n"
    parser.label+=1

def p_Condicao(p):
    "Condicao : '(' Cond ')'"
    p[0]= p[2]

def p_Condicao_Neg(p):
    "Condicao : NOT '(' Cond ')'"
    p[0]= str(p[3])+"NOT\n"

def p_Condicao_AND(p):
    "Condicao : Condicao AND Condicao"
    p[0]=str(p[1])+str(p[3]) +"MUL\n"

def p_Condicao_OR(p):
    "Condicao : Condicao OR Condicao"
    p[0]=str(p[1])+str(p[3]) +"ADD\n"

def p_Cond_EQUALS(p):
    "Cond : Exp EQUALS Exp"
    p[0]=str(p[1][0])+str(p[3][0])+"EQUAL\n"

def p_Cond_GREATER(p):
    "Cond : Exp GREATER Exp"
    p[0] = str(p[1][0])+str(p[3][0])+"SUP\n"
```

```
def p_Cond_LESSER(p):
    "Cond : Exp LESSER Exp"
    p[0] = str(p[1][0])+str(p[3][0])+"INF\n"

def p_Cond_GREATERQ(p):
    "Cond : Exp GREATERQ Exp"
    p[0] = str(p[1][0])+str(p[3][0])+"SUPEQ\n"

def p_Cond_LESSERQ(p):
    "Cond : Exp LESSERQ Exp"
    p[0] = str(p[1][0])+str(p[3][0])+"INFEQ\n"

def p_Cond(p):
    "Cond : Condicao"
    p[0] = p[1]
```

Quanto ao comando If e If Else, foi criada uma variável onde se podessem guardar diferentes labes, de modo a estar preparado para aninhamento de comandos e utilizada a operaçao "JZ". Quando o comando em causa é If , no caso em que a condição não é verificada é realizado um salto para a lable que se encontra no final dos comandos, permitindo assim com que , nestes casos , não sejam realizados os comandos inseridos que seriam executados caso a condição se verifica-se. Quando o comando em causa é If Else , se a primeira condição não for verificada é realizado um salto para a lable que se encontra no comando Else, permitindo assim com que , nestes casos , sejam executados os comandos inseridos para quando a primeira condição não é verificada.

Quanto ao comando **Repeat**:

```
def p_Comando_Repeat_(p):
    "Comando_Repeat : REPEATE '{' Comandos '}' UNTIL Condicao"
    p[0]= "REPEAT"+ str(parser.loop)+":\n"+ p[3]+p[6]+"JZ REPEAT"+ str(parser.loop)+"\n"
    parser.loop +=1
```

Ao invés do que foi utilizado nos comandos If e If Else , aqui foi criada uma outra variável onde se podessem guardar diferentes loops, de modo a evitar conflitos no aninhamento de comandos e utilizada a operação jz. Como as comandos dentro do Repeat são várias repetições da iteração até que a condição proposta seja verificada , é utilizado mais uma vez a operação jz , que permite saltar para o loop do repeat durante as iterações que não satisfaçam a condição.
Quanto ao comando **While**:

```
def p_Comando_While_(p):
    "Comando_While : WHILE Condicao DO '{' Comandos '}'"
    p[0]="WHILE" +str(parser.loop)+":\n" + p[2] +"JZ ENDWHILE"+str(parser.loop)+"\n"+p[5]+
    "JUMP WHILE"+str(parser.loop)+"\n"+"ENDWHILE"+str(parser.loop)+":\n"
    parser.loop +=1
```

Tal como foi necessário usar nos comandos Repeat , aqui também é necessário a utilização de loops , jz e também a operação jump. Ao contrário do que acontece no Repeat , no While os comandos são iterados varias vezes até que a condição proposta não se verifique, sendo utilizado a operação jump , que permite

saltar para o início do While durante as iterações que satisfazem a condição.

Quanto ao comando **Print**:

```
def p_Comando_Prints_All(p):
    "Comando_Prints : PRINT Comando_Print Prints  "
    p[0]=str(p[2])+str(p[3])
def p_Comando_Prints(p):
    "Prints : '+' Comando_Print Prints"
    p[0]=p[2]+p[3]
def p_Comando_Prints_Vazio(p):
    "Prints :"
    p[0]=""
def p_Comando_Print_ID(p):
    "Comando_Print : Id"
    p[0]= p[1][1]+"WRITEI\n"

def p_Comando_Print_ID_Array(p):
    "Comando_Print : IdArray"
    p[0]=p[1][0]+p[1][1]+ "LOADN\n"+"WRITEI\n"
def p_Comando_Print_ID_DoubleArray(p):
    "Comando_Print : IdDoubleArray"
    p[0]=p[1][0]+p[1][1]+ "LOADN\n"+"WRITEI\n"

def p_Comando_Print_STR(p):
    "Comando_Print :  STR"
    p[0]="PUSHS " + p[1]+ "\n"+"WRITES\n"
```

Para realizar o comando de Print serão necessárias as operações de "writei"para o caso de valores inteiros e "writes"para o caso de strings. No caso de ser um array , sempre que queremos obter um determinado valor de uma posição é necessário o uso dos comandos das variaveis , juntamente com o "LOADN"e "WRITEI". É também importante referir que é possivel dar print de uma variável ou uma string separada por um símbolo "+".

Quanto ao comando **Read**:

```
def p_Comando_Read_ID(p):
    "Comando_Read : READ Id"
    p[0]="READ\nATOI\n"+ p[2][0]

def p_Comando_Read_ID_Array(p):
    "Comando_Read : READ IdArray"
    p[0]= p[1][0]+p[1][1] + "READ\nATOI\nSTOREN\n"
def p_Comando_Read_ID_DoubleArray(p):
    "Comando_Read : READ IdDoubleArray"
    p[0]= p[1][0]+p[1][1] + "READ\nATOI\nSTOREN\n"
```

Para realizar o comando Read , permitindo ler o input é necessário guardar o valor lido , tendo depois de ser convertido para um inteiro numa variável , podendo ser apenas uma letra ou várias , como também

um array de uma ou duas dimensões. Para tal sao usados os comandos das variáveis juntamente com as operações de ”READ”, ”ATOI”e ”STOREN”.
Quanto ao comando **Exp**:

```
def p_Comando_Exp(p):
    "Comando : Id '=' Exp"
    p[0]=str(p[3][0])+str(p[1][0])
    if(parser.exito):
        parser.var_valores[p[1][3]]=p[3][1]

def p_Comando_Array_Exp(p):
    "Comando : IdArray '=' Exp"
    p[0]=p[1][0]+p[1][1]+p[3][0]+"STOREN\n"
    var=p[1][3]
    index=p[1][4]
    if(parser.exito):
        parser.var_valores[var][index]=p[3][1]

def p_Comando_DoubleArray_Exp(p):
    "Comando : IdDoubleArray '=' Exp"
    p[0]=p[1][0]+p[1][1]+p[3][0]+"STOREN\n"

def p_Exp_PLUS(p):
    "Exp : Exp '+' Term"
    p[0] = (p[1][0] + p[3][0] + "ADD\n",p[1][1] + p[3][1])

def p_Exp_MINUS(p):
    "Exp : Exp '-' Term"
    p[0] = (p[1][0] + p[3][0] + "SUB\n",p[1][1] - p[3][1])

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


def p_Term_DIV(p):
    "Term : Term '/' Factor"
    p[0] = (p[1][0] + p[3][0] + "DIV\n",p[1][1] / p[3][1])

def p_Term_MULT(p):
    "Term : Term '*' Factor"
    p[0] = (p[1][0] + p[3][0] + "MUL\n",p[1][1] * p[3][1])
def p_Term_MOD(p):
    "Term : Term '%' Factor"
    p[0] = (p[1][0] + p[3][0] + "MOD\n",p[1][1] % p[3][1] )

def p_Term(p):
    "Term : Factor"
```

```
    p[0] = (p[1][0],p[1][1])


def p_Factor_NUM(p):
    "Factor : NUM"
    p[0]=("PUSHI "+p[1]+"\n",int(p[1]))

def p_Factor_ID(p):
    "Factor : Id"
    p[0] = (p[1][1],p[1][2])

def p_Factor_ID_ARRAY(p) :
    "Factor : IdArray"
    p[0]=(p[1][0]+p[1][1]+"LOADN\n",p[1][2])

def p_Factor_ID_DoubleARRAY(p) :
    "Factor : IdDoubleArray"
    p[0]=(p[1][0]+p[1][1]+"LOADN\n")
```

Para o caso do comando Exp , os valores são guardados no simbolo Exp e numa variável, aplicando-se também algumas operações da Maquina Virtudal que permitem converter as expressões algébricas em código máquina. Os Factores que são variáveis usam os comandos respetivos e "LOADN". O Factor Num usa o comando "PUSHI". Os Termos , Exp e Factores devolvem todos um tupolo , sendo o primeiro os comandos e o segundo o valor resultante da operação efetuada , em inteiro, que será usado para atualizar o dicionário de valores.

No caso do **Id**:

```
def p_Id(p):
    "Id : ID"
    if p[1] not in parser.variaveis:
        parser.exito = False
        p[0]=("ERROR\n","ERROR\n")
        print("Variavel " + p[1]+" nao foi declarada antes")
    else:
        p[0] = ("STOREG "+str(parser.variaveis[p[1]])+"\n","PUSHG "+ str(parser.variaveis[p[1]])
        +"\n",int(parser.var_valores[p[1]]),p[1])
```

Para o caso das variáveis Id , será verificada se a variável existe e caso isso se confirme é devolvido um tupolo com os comandos STOREG ou PUSHG, sendo STOREG para armazenar e PUSHG para dar "push"ao valor na pilha.

No caso do **Id Array**:

```
def p_Id_Array(p):
    "IdArray : ID '[' Factor ']'"
    if (p[1]  in parser.variaveis ) :
        fac=int(p[3][1])
```

```
        index=len(parser.var_valores[p[1]])-1
    if (p[1] not in parser.variaveis ) :
        parser.exito = False
        p[0]="ERROR\n"
        print("Variavel " + p[1]+" nao foi declarada antes")
    elif((fac>index or fac<0) ):
        parser.exito = False
        p[0]="ERROR\n"
        print("Index fora de alcance")
    else:
        fac=p[3][1]
        p[0]=("PUSHGP\nPUSHI "+ str(parser.variaveis[p[1]])+"\nPADD\n",p[3][0],
        parser.var_valores[p[1]][fac],p[1],p[3][1] )
```

Para o caso das variáveis Id_Array, será de igual forma verificada se a variável existe e se o index do array está dentro dos parâmetros, através de uma pesquisa no dicionário dos valores, retirando as medidas do array.Caso isto se confirme, será devolvido um tupolo com os comandos PUSHGP,PUSHI com o endereço do primeiro índice e PADD , os comandos do fator.

No caso do **Id DoubleArray**:

```
def p_Id_DoubleArray(p):
    "IdDoubleArray : ID '[' Factor ']' '[' Factor ']'"
    if (p[1] not in parser.variaveis ) :
        parser.exito = False
        p[0]="ERROR\n"
        print("Variavel " + p[1]+" nao foi declarada antes")
    else:
        p[0]=("PUSHGP\nPUSHI "+ str(parser.variaveis[p[1]])+"\nPADD\n",p[3][0]+
        "PUSHI "+str(len(parser.var_valores[p[1]][0]))+"\nMUL\n"+p[6][0]+ "ADD\n" )
```

Para o caso das variáveis Id_DoubleArray, será usado o mesmo método utilizado no Id_Array , mas no caso índice do array é efetuada uma expressão (Coluna*Colunas Totais) + Linhas , para saber o endereço desse index.

Flags: É importante referir alguns dos possiveis erros que o compilador será capaz de verificar, ao longo das operações efetuadas. Estes são:

- Existência de repetição de nome de variáveis nas declarações;
- Uso de uma variável sem ser declarada;
- Caso um array ou double array seja declarado, o número de valores introduzidos, separados devidamente por virgulas, tem de ser igual ao valor declarado no index;
- Uso da variavel array com um index maior do que o index que foi delcarado nas declarações;
- Existência de algum erro de sintaxe.

Capítulo 4

Teste

4.1 Testes Realizados e Resultados

Conforme pedido no guião do trabalho pratico , foram executados alguns testes que serão de seguida apresentados com os seus respetivos resultados.Cada exemplo será apresentado primeiro com o código não compilado , seguido do código compilado.

4.1.1 Exemplo 1

```
Int a
Int b
Int c
MAIN
Read a
Read b
Read c
If (((a == 0) OR (b == 0) OR (c == 0)) OR ((a + b <= c) OR (a + c <= b) OR (b + c <= a)))
{
    Print "NAO E TRIANGULO \n"
}
Else{
    Print "TRANGULO \n"
}
END
```

```
PUSHI 0
PUSHI 0
PUSHI 0
START
READ
ATOI
STOREG 0
READ
ATOI
```

```
STOREG 1
READ
ATOI
STOREG 2
PUSHG 0
PUSHI 0
EQUAL
PUSHG 1
PUSHI 0
EQUAL
PUSHG 2
PUSHI 0
EQUAL
ADD
ADD
PUSHG 0
PUSHG 1
ADD
PUSHG 2
INFEQ
PUSHG 0
PUSHG 2
ADD
PUSHG 1
INFEQ
PUSHG 1
PUSHG 2
ADD
PUSHG 0
INFEQ
ADD
ADD
ADD
JZ IF0
PUSHS "NAO E TRIANGULO \n"
WRITES
JUMP IFEND0
IF0:
PUSHS "TRANGULO \n"
WRITES
IFEND0:
STOP
```

4.1.2 Exemplo 2

```
Int N
Int counter
Int menor
```

```
Int input
MAIN
Read N
Read input
menor=input
counter=counter+1
While(counter<N)
Do{
    Read input
    If(input<menor){
        menor=input
    }
    counter=counter+1
}
Print menor +"\n"
END

PUSHI 0
PUSHI 0
PUSHI 0
PUSHI 0
START
READ
ATOI
STOREG 0
READ
ATOI
STOREG 3
PUSHG 3
STOREG 2
PUSHG 1
PUSHI 1
ADD
STOREG 1
WHILE0:
PUSHG 1
PUSHG 0
INF
JZ ENDWHILE0
READ
ATOI
STOREG 3
PUSHG 3
PUSHG 2
INF
JZ IF0
PUSHG 3
STOREG 2
```

```
IF0:
PUSHG 1
PUSHI 1
ADD
STOREG 1
JUMP WHILE0
ENDWHILE0:
PUSHG 2
WRITEI
PUSHS "\n"
WRITES
STOP
```

4.1.3 Exemplo 3

```
Int N=4
Int counter
Int r
Int input
MAIN
Read input
r=input
counter=counter+1
While(counter<N)
Do{
    Read input
    r=r*input
    counter=counter+1
}
Print r +"\n"
END

PUSHI 4
PUSHI 0
PUSHI 0
PUSHI 0
START
READ
ATOI
STOREG 3
PUSHG 3
STOREG 2
PUSHG 1
PUSHI 1
ADD
STOREG 1
WHILE0:
PUSHG 1
```

```
PUSHG 0
INF
JZ ENDWHILE0
READ
ATOI
STOREG 3
PUSHG 2
PUSHG 3
MUL
STOREG 2
PUSHG 1
PUSHI 1
ADD
STOREG 1
JUMP WHILE0
ENDWHILE0:
PUSHG 2
WRITEI
PUSHS "\n"
WRITES
STOP
```

4.1.4 Exemplo 4

```
Int N
Int count
Int i
Int input
MAIN
Read N
While(i<N)
Do{
    Read input
    If!(input % 2 == 0){
        Print "IMPAR: " + input + "\n"
        count=count+1
    }
    i=i+1
}
Print "COUNTER = "+ count+ " \n"
END
```

```
PUSHI 0
PUSHI 0
PUSHI 0
PUSHI 0
START
```

```
READ
ATOI
STOREG 0
WHILE0:
PUSHG 2
PUSHG 0
INF
JZ ENDWHILE0
READ
ATOI
STOREG 3
PUSHG 3
PUSHI 2
MOD
PUSHI 0
EQUAL
NOT
JZ IF0
PUSHS "IMPAR: "
WRITES
PUSHG 3
WRITEI
PUSHS "\n"
WRITES
PUSHG 1
PUSHI 1
ADD
STOREG 1
IF0:
PUSHG 2
PUSHI 1
ADD
STOREG 2
JUMP WHILE0
ENDWHILE0:
PUSHS "COUNTER = "
WRITES
PUSHG 1
WRITEI
PUSHS " \n"
WRITES
STOP
```

4.1.5 Exemplo 5

```
Int N=5
Int a[5]
Int i
```

```
Int input
MAIN
While(i<N)
Do{
    Read input
    a[i]=input
    i=i + 1
}
i=4
While (i>=0)
Do{
    Print a[i]+"\n"
    i=i - 1
}

END

PUSHI 5
PUSHN 5
PUSHI 0
PUSHI 0
START
WHILE0:
PUSHG 6
PUSHG 0
INF
JZ ENDWHILE0
READ
ATOI
STOREG 7
PUSHGP
PUSHI 1
PADD
PUSHG 6
PUSHG 7
STOREN
PUSHG 6
PUSHI 1
ADD
STOREG 6
JUMP WHILE0
ENDWHILE0:
PUSHI 4
STOREG 6
WHILE1:
PUSHG 6
PUSHI 0
SUPEQ
```

```
JZ ENDWHILE1
PUSHGP
PUSHI 1
PADD
PUSHG 6
LOADN
WRITEI
PUSHS "\n"
WRITES
PUSHG 6
PUSHI 1
SUB
STOREG 6
JUMP WHILE1
ENDWHILE1:
STOP
```

Capítulo 5

Conclusão

A nosso ver , este trabalho permitiu consolidar e aplicar a matéria lecionada ao longo das aulas , tanto no que diz respeito ao analisador lexico como às gramáticas independentes do contexto e tradutoras. Em geral , consideramos que o trabalho foi bem desenvolvido , tentando responder da melhor forma aos requesitos e obejtivos proprostos pela equipa docente.Tendo sido também uma ótima forma de aprofundar os nossos conhecimentos em Python , engenharia de linguagens e programação generativa (gramatical).

Apêndice A

Código do Programa

Lista-se a seguir o código do programa que foi desenvolvido.

A.1 Analisador lexico

```
import ply.lex as lex
import re
import sys

tokens = ['INT','STR','NUM','ID',
'MAIN','END',
'PRINT','READ',
'IF','ELSE',
'REPEATE','UNTIL','WHILE','DO',
'AND','OR','NOT',
'EQUALS','GREATER','LESSER','GREATERQ','LESSERQ'
]
literals = ['%','*', '+' ,'/', '-', '=', '(', ')', '.', '<', '>', ',', '!', '{', '}', '[', ']']

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

def t_STR(t):
    r'"[^"]+"'
    return t

def t_NUM(t):
    r'-?\d+'
    #t.value = int(t.value)
    return t

def t_MAIN(t):
    r'MAIN'
    return t
```

```
def t_END(t):
    r'END'
    return t

def t_PRINT(t):
    r'Print'
    return t

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

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

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

def t_REPEATE(t):
    r'Repeat'
    return t

def t_UNTIL(t):
    r'Until'
    return t

def t_WHILE(t):
    r'While'
    return t

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

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

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

def t_NOT(t):
    r'!'
    return t
```

```
def t_GREATERQ(t):
    r'>='
    return t

def t_LESSERQ(t):
    r'<='
    return t

def t_EQUALS(t):
    r'=='
    return t

def t_GREATER(t):
    r'>'
    return t
def t_LESSER(t):
    r'<'
    return t

def t_ID(t):
    r'\w+'
    return t

t_ignore = " \t\n"
def t_error(t):
    print('Illegal character: ', t.value[0])
    t.lexer.skip(1)

lexer = lex.lex()
```

A.2 Compilador Yacc

```
from os import write
import ply.yacc as yacc
import os
import sys
from tp2_lex import tokens



def p_Programa(p):
    "Programa : Declaracoes Funcoes"
    p[0]=p[1]+p[2]
    #print(str(p[1])+str(p[2]))
```

```
def p_Funcoes(p):
    "Funcoes : MAIN Comandos END"
    p[0] = "START\n" + str(p[2]) + "STOP\n"


##DECLARACOES
def p_Declaracoes(p):
    "Declaracoes : Declaracao  Declaracoes "
    p[0] = str(p[1]) + str(p[2])

def p_Declaracoes_vazia(p):
    "Declaracoes : "
    p[0] = ""

def p_Declaracao_Int(p):
    "Declaracao : INT ID"
    if p[2] in parser.variaveis:
        parser.exito = False
        p[0]= "ERROR"
        print("Variavel " + p[2]+" ja foi declarada")
    else:

        parser.variaveis[p[2]]=parser.count
        parser.var_valores[p[2]]=0
        p[0]="PUSHI 0\n"
        parser.count+=1

def p_Declaracao_Int_com_numero(p):
    "Declaracao : INT ID '=' NUM"
    if p[2] in parser.variaveis:
        parser.exito = False
        p[0]= "ERROR"
        print("Variavel " + p[2]+" ja foi declarada")
    else:

        parser.variaveis[p[2]]=parser.count
        parser.var_valores[p[2]]=int(p[4])
        p[0]="PUSHI "+str(p[4])+"\n"

        parser.count+=1
def p_Declaracao_Array(p):
    "Declaracao : INT ID '[' NUM ']'"
    if p[2] in parser.variaveis:
        parser.exito = False
        p[0]= "ERROR"
        print("Variavel " + p[2]+" ja foi declarada")
    else:
```

```
        parser.variaveis[p[2]]=parser.count
        parser.var_valores[p[2]]=[0]*int(p[4])
        p[0]="PUSHN "+p[4]+"\n"
        parser.count+=int(p[4])
def p_Declaracao_DoubleArray(p):
    "Declaracao : INT ID '[' NUM ']' '[' NUM ']'"
    if p[2] in parser.variaveis:
        parser.exito = False
        p[0]= "ERROR"
        print("Variavel " + p[2]+" ja foi declarada")
    else:

        parser.variaveis[p[2]]=parser.count
        parser.var_valores[p[2]]=[[0 for i in range(int(p[7]))] for j in range(int(p[4]))]
        n=int(p[4])*int(p[7])
        p[0]="PUSHN "+str(n) +"\n"
        parser.count+=n
def p_Declaracao_Array_com_numero(p):
    "Declaracao : INT ID '[' NUM ']' '=' '[' Array ']'"
    parser.var_valores[p[2]]=parser.array
    if p[2] in parser.variaveis:
        parser.exito = False
        p[0]= "ERROR"
        print("Variavel " + p[2]+" ja foi declarada")
    elif len(parser.var_valores[p[2]])!=int(p[4]):
        parser.exito = False
        print("Variavel " + p[2]+" index out of range")

        p[0]= "ERROR"
    else:
        parser.var_valores[p[2]]=parser.array
        parser.variaveis[p[2]]=parser.count
        p[0]=p[8]
        parser.count+=int(p[4])
    parser.array=[]

def p_Declaracao_DoubleArray_com_numero(p):
    "Declaracao : INT ID '[' NUM ']' '[' NUM ']' '=' '[' Arrays ']'"
    f=0
    parser.var_valores[p[2]]=[parser.array[i:i+int(p[7])] for i in range(0, len(parser.array),
    int(p[7]))]
    parser.var_valores[p[2]].reverse()
    for a in parser.var_valores[p[2]]:
        if len(a)!=int(p[7]):
            f=1
    if p[2] in parser.variaveis:
        parser.exito = False
```

```
        p[0]= "ERROR"
        print("Variavel " + p[2]+" ja foi declarada")
    elif len(parser.var_valores[p[2]])!=int(p[4]) or f==1 :
        parser.exito = False
        print("Variavel " + p[2]+" index out of range")

        p[0]= "ERROR"
    else:

        n=int(p[4])*int(p[7])
        parser.variaveis[p[2]]=parser.count
        #parser.var_valores[p[2]]=[[0 for i in range(int(p[7]))] for j in range(int(p[4]))]

        parser.count+=n
        p[0]=p[11]
    f=0
    parser.array=[]
##ARRAY
def p_Arrays(p):
    "Arrays : '[' Array ']' ',' Arrays"
    p[0]=p[2]+p[5]

def p_Arrays_Vazio(p):
    "Arrays :"
    p[0]=""
def p_Arrays_Array(p):
    "Arrays : '[' Array ']'"
    p[0]=p[2]

def p_Array(p):
    "Array : NUM ',' Array"
    p[0]="PUSHI "+p[1]+"\n" +p[3]
    parser.array.insert(0,int(p[1]))

def p_Array_Num(p):
    "Array : NUM "
    p[0]="PUSHI "+p[1]+"\n"
    parser.array.insert(0,int(p[1]))

def p_Array_Vazio(p):
    "Array :"
    p[0]  =""


##COMANDOS
def p_Comandos(p):
    "Comandos : Comando Comandos"
    p[0] = str(p[1]) + str(p[2])
```

```
def p_Comandos_vazio(p):
    "Comandos : "
    p[0] = ""

def p_Comando_If(p):
    "Comando : Comando_If"
    p[0] = p[1]

def p_Comando_If_Else(p):
    "Comando : Comando_If_Else"
    p[0] = p[1]

def p_Comando_Repeat(p):
    "Comando : Comando_Repeat"
    p[0] = p[1]
def p_Comando_While(p):
    "Comando : Comando_While"
    p[0]=p[1]

def p_Comando_Print(p):
    "Comando : Comando_Prints"
    p[0] = p[1]

def p_Comando_Read(p):
    "Comando : Comando_Read"
    p[0] = p[1]

def p_Comando_Exp(p):
    "Comando : Id '=' Exp"
    p[0]=str(p[3][0])+str(p[1][0])
    if(parser.exito):
        parser.var_valores[p[1][3]]=p[3][1]


def p_Comando_Array_Exp(p):
    "Comando : IdArray '=' Exp"
    p[0]=p[1][0]+p[1][1]+p[3][0]+"STOREN\n"
    var=p[1][3]
    index=p[1][4]
    if(parser.exito):
        parser.var_valores[var][index]=p[3][1]
def p_Comando_DoubleArray_Exp(p):
    "Comando : IdDoubleArray '=' Exp"
    p[0]=p[1][0]+p[1][1]+p[3][0]+"STOREN\n"
```

```
#
def p_Comando_If_(p):
    "Comando_If : IF Condicao '{' Comandos '}'"
    p[0]= str(p[2])+"JZ IF"+str(parser.label) +"\n" + str(p[4])+"IF"+str(parser.label)+":\n"
    parser.label+=1

def p_Comando_If_Else_(p):
    "Comando_If_Else : IF Condicao '{' Comandos '}' ELSE '{' Comandos '}'"
    p[0] = p[2]+"JZ IF"+ str(parser.label)+"\n"+p[4]+"JUMP IFEND" +  str(parser.label) +"\nIF"
    +str(parser.label) +":\n"+p[8]+"IFEND"+str(parser.label)+":\n"
    parser.label+=1
def p_Comando_Repeat_(p):
    "Comando_Repeat : REPEATE '{' Comandos '}' UNTIL Condicao"
    p[0]= "REPEAT"+ str(parser.loop)+":\n"+ p[3]+p[6]+"JZ REPEAT"+ str(parser.loop)+"\n"
    parser.loop +=1
def p_Comando_While_(p):
    "Comando_While : WHILE Condicao DO '{' Comandos '}'"
    p[0]="WHILE" +str(parser.loop)+":\n" + p[2] +"JZ ENDWHILE"+str(parser.loop)+"\n"+p[5]+
    "JUMP WHILE"+str(parser.loop)+"\n"+"ENDWHILE"+str(parser.loop)+":\n"
    parser.loop +=1
#
def p_Comando_Prints_All(p):
    "Comando_Prints : PRINT Comando_Print Prints  "
    p[0]=str(p[2])+str(p[3])
def p_Comando_Prints(p):
    "Prints : '+' Comando_Print Prints"
    p[0]=p[2]+p[3]
def p_Comando_Prints_Vazio(p):
    "Prints :"
    p[0]=""
def p_Comando_Print_ID(p):
    "Comando_Print : Id"
    p[0]= p[1][1]+"WRITEI\n"

def p_Comando_Print_ID_Array(p):
    "Comando_Print : IdArray"
    p[0]=p[1][0]+p[1][1]+ "LOADN\n"+"WRITEI\n"
def p_Comando_Print_ID_DoubleArray(p):
    "Comando_Print : IdDoubleArray"
    p[0]=p[1][0]+p[1][1]+ "LOADN\n"+"WRITEI\n"

def p_Comando_Print_STR(p):
    "Comando_Print :  STR"
    p[0]="PUSHS " + p[1]+ "\n"+"WRITES\n"
#
def p_Comando_Read_ID(p):
    "Comando_Read : READ Id"
    p[0]="READ\nATOI\n"+ p[2][0]
```

```
def p_Comando_Read_ID_Array(p):
    "Comando_Read : READ IdArray"
    p[0]= p[1][0]+p[1][1] + "READ\nATOI\nSTOREN\n"
def p_Comando_Read_ID_DoubleArray(p):
    "Comando_Read : READ IdDoubleArray"
    p[0]= p[1][0]+p[1][1] + "READ\nATOI\nSTOREN\n"
#CONDICOES
def p_Condicao(p):
    "Condicao : '(' Cond ')'"
    p[0]= p[2]
def p_Condicao_Neg(p):
    "Condicao : NOT '(' Cond ')'"
    p[0]= str(p[3])+"NOT\n"
def p_Condicao_AND(p):
    "Condicao : Condicao AND Condicao"
    p[0]=str(p[1])+str(p[3]) +"MUL\n"
def p_Condicao_OR(p):
    "Condicao : Condicao OR Condicao"
    p[0]=str(p[1])+str(p[3]) +"ADD\n"

def p_Cond_EQUALS(p):
    "Cond : Exp EQUALS Exp"
    p[0]=str(p[1][0])+str(p[3][0])+"EQUAL\n"

def p_Cond_GREATER(p):
    "Cond : Exp GREATER Exp"
    p[0] = str(p[1][0])+str(p[3][0])+"SUP\n"

def p_Cond_LESSER(p):
    "Cond : Exp LESSER Exp"
    p[0] = str(p[1][0])+str(p[3][0])+"INF\n"

def p_Cond_GREATERQ(p):
    "Cond : Exp GREATERQ Exp"
    p[0] = str(p[1][0])+str(p[3][0])+"SUPEQ\n"

def p_Cond_LESSERQ(p):
    "Cond : Exp LESSERQ Exp"
    p[0] = str(p[1][0])+str(p[3][0])+"INFEQ\n"

def p_Cond(p):
    "Cond : Condicao"
    p[0] = p[1]

#EXPRECOES
def p_Exp_PLUS(p):
    "Exp : Exp '+' Term"
```

```
    p[0] = (p[1][0] + p[3][0] + "ADD\n",p[1][1] + p[3][1])

def p_Exp_MINUS(p):
    "Exp : Exp '-' Term"
    p[0] = (p[1][0] + p[3][0] + "SUB\n",p[1][1] - p[3][1])

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

#TERMOS
def p_Term_DIV(p):
    "Term : Term '/' Factor"
    p[0] = (p[1][0] + p[3][0] + "DIV\n",p[1][1] / p[3][1])

def p_Term_MULT(p):
    "Term : Term '*' Factor"
    p[0] = (p[1][0] + p[3][0] + "MUL\n",p[1][1] * p[3][1])
def p_Term_MOD(p):
    "Term : Term '%' Factor"
    p[0] = (p[1][0] + p[3][0] + "MOD\n",p[1][1] % p[3][1] )

def p_Term(p):
    "Term : Factor"
    p[0] = (p[1][0],p[1][1])

#FATORES
def p_Factor_NUM(p):
    "Factor : NUM"
    p[0]=("PUSHI "+p[1]+"\n",int(p[1]))
def p_Factor_ID(p):
    "Factor : Id"
    p[0] = (p[1][1],p[1][2])
def p_Factor_ID_ARRAY(p) :
    "Factor : IdArray"
    p[0]=(p[1][0]+p[1][1]+"LOADN\n",p[1][2])
def p_Factor_ID_DoubleARRAY(p) :
    "Factor : IdDoubleArray"
    p[0]=(p[1][0]+p[1][1]+"LOADN\n")
#ID
def p_Id(p):
    "Id : ID"
    if p[1] not in parser.variaveis:
        parser.exito = False
        p[0]=("ERROR\n","ERROR\n")
        print("Variavel " + p[1]+" nao foi declarada antes")
    else:
        p[0] = ("STOREG "+str(parser.variaveis[p[1]])+"\n",
```

```
        "PUSHG "+ str(parser.variaveis[p[1]])+"\n",int(parser.var_valores[p[1]]),p[1])

def p_Id_Array(p):
    "IdArray : ID '[' Factor ']'"
    if (p[1]  in parser.variaveis ) :
        fac=int(p[3][1])
        index=len(parser.var_valores[p[1]])-1
    if (p[1] not in parser.variaveis ) :
        parser.exito = False
        p[0]="ERROR\n"
        print("Variavel " + p[1]+" nao foi declarada antes")
    elif((fac>index or fac<0) ):
        parser.exito = False
        p[0]="ERROR\n"
        print("Index fora de alcance")
    else:
        fac=p[3][1]
        p[0]=("PUSHGP\nPUSHI "+ str(parser.variaveis[p[1]])+"\nPADD\n",p[3][0],
        parser.var_valores[p[1]][fac],p[1],p[3][1] )


def p_Id_DoubleArray(p):
    "IdDoubleArray : ID '[' Factor ']' '[' Factor ']'"
    if (p[1] not in parser.variaveis ) :
        parser.exito = False
        p[0]="ERROR\n"
        print("Variavel " + p[1]+" nao foi declarada antes")
    else:
        p[0]=("PUSHGP\nPUSHI "+ str(parser.variaveis[p[1]])+"\nPADD\n",p[3][0]+
        "PUSHI "+str(len(parser.var_valores[p[1]][0]))+"\nMUL\n"+p[6][0]+ "ADD\n" )




def p_error(p):
    print("Syntax error!")
    parser.exito = False

parser = yacc.yacc()
parser.variaveis = {}
parser.var_valores={}
parser.count=0
parser.loop=0
parser.label=0
parser.exito = True
parser.array=[]
```

```
fIn= input('FileInput: ')
if not os.path.exists(fIn):
    print("Nao encontrado")
else:
    fOut = input('FileOutput: ')
    with  open(fIn,'r') as file:
        code = file.read()
    out=parser.parse(code)
    if (parser.exito==True):
        print ("Parsing teminou com sucesso!")
        with open(fOut,'w') as output:
            output.write(str(out))
    else:
        print("Parsing nao terminou com sucesso.......")

#print(parser.variaveis)
#print(parser.var_valores)
```